PostGIS 3.8.0dev □□□□□□

DEV (Sun Nov 15 09:26:18 AM UTC 2026 rev. 58fb588)

Contents

1 □□ 1
1.1 What Is PostGIS? . . . 1
1.2 What Is in the PostGIS Distribution? . . . 1
1.3 The PostGIS Project Family . . . 2
1.4 How to Use This Manual . . . 2
1.5 Standards Support . . . 3
1.6 Project and Community . . . 3

2 PostGIS □□ 4
2.1 □□□□ . . . 4
2.2 □□□□□□□□□□□□ . . . 4
2.2.1 □□□□ . . . 5
2.2.2 □□□□□□ . . . 5
2.2.3 □□ . . . 7
2.2.4 □□ . . . 8
2.2.5 PostGIS Extensions □□□□□ . . . 9
2.2.6 □□□ . . . 11
2.2.7 □□ . . . 14
2.3 □□□□□□□□□□□□□□ . . . 14

3 PostGIS Administration 16
3.1 Performance Tuning . . . 16
3.1.1 Startup . . . 16
3.1.2 Runtime . . . 17
3.2 Configuring raster support . . . 17
3.3 □□□□□□□□□□ . . . 18
3.3.1 Spatially enable database using EXTENSION . . . 18
3.3.2 Spatially enable database without using EXTENSION (discouraged) . . . 19
3.4 Upgrading spatial databases . . . 19
3.4.1 Soft upgrade . . . 20
3.4.1.1 Soft upgrade using extensions . . . 20
3.4.1.2 Soft upgrade without extensions . . . 21
3.4.1.3 Hard upgrade . . . 21

4 Data Management **24**
4.1 GIS (□□) □□□□□ . . . 24
4.1.1 OGC Geometry . . . 24
4.1.1.1 Point . . . 25
4.1.1.2 LineString . . . 25
4.1.1.3 LinearRing . . . 25
4.1.1.4 Polygon . . . 25
4.1.1.5 MultiPoint . . . 26
4.1.1.6 MultiLineString . . . 26
4.1.1.7 MultiPolygon . . . 26
4.1.1.8 GeometryCollection . . . 26
4.1.1.9 PolyhedralSurface . . . 26
4.1.1.10Triangle . . . 27
4.1.1.11TIN . . . 27
4.1.2 SQL-MM Part 3 . . . 27
4.1.2.1 CircularString . . . 27
4.1.2.2 CompoundCurve . . . 27
4.1.2.3 CurvePolygon . . . 28
4.1.2.4 MultiCurve . . . 28
4.1.2.5 MultiSurface . . . 28
4.1.3 OpenGIS WKB □ WKT . . . 28
4.2 Geometry Data Type . . . 30
4.2.1 OpenGIS WKB □ WKT . . . 30
4.3 PostGIS □□□□□ . . . 32
4.3.1 □□□□□□□ . . . 32
4.3.2 PostGIS □□□□□ . . . 33
4.3.3 □□□□□□□□□□□□□□□□□□□□□□□□□ . . . 35
4.3.4 □□□□□ FAQ . . . 36
4.4 Geometry Validation . . . 36
4.4.1 Simple Geometry . . . 36
4.4.2 Valid Geometry . . . 38
4.4.3 Managing Validity . . . 43
4.5 SPATIAL_REF_SYS □□□□□□□□□□□ . . . 45
4.5.1 SPATIAL_REF_SYS Table . . . 46
4.5.2 SPATIAL_REF_SYS □□□□□□□□□□ . . . 47
4.6 □□□□□□□ . . . 48
4.6.1 □□□□□□□ . . . 48
4.6.2 The GEOMETRY_COLUMNS VIEW . . . 49
4.6.3 geometry_columns □□□□□□□□□□ . . . 49

4.7 GIS (□□) □□□□□ . . . 52
4.7.1 SQL □□□□□□□□□□□ . . . 52
4.7.2 shp2pgsql: ESRI shapefile □□□□□□ . . . 53
4.8 □□□□□□□ . . . 55
4.8.1 SQL □□□□□□□□□□□ . . . 55
4.8.2 □□□□□□ . . . 56
4.9 □□□□□□□ . . . 56
4.9.1 GiST □□□ . . . 57
4.9.2 GiST □□□ . . . 58
4.9.3 GiST □□□ . . . 60
4.9.4 □□□□□□□ . . . 60

5 Spatial Queries **62**
5.1 Determining Spatial Relationships . . . 62
5.1.1 Dimensionally Extended 9-Intersection Model . . . 62
5.1.2 Named Spatial Relationships . . . 66
5.1.3 General Spatial Relationships . . . 66
5.2 Using Spatial Indexes . . . 68
5.3 Examples of Spatial SQL . . . 69

6 □□□□□□ **72**
6.1 □□□□□□□□□□□□□□ . . . 72
6.1.1 □□□□□ . . . 72
6.1.2 □□□□ . . . 72
6.2 □□□□□□□□□□□□□□ . . . 73
6.3 □□□□□□□ . . . 73

7 PostGIS Reference **74**
7.1 PostgreSQL PostGIS Geometry/Geography/Box □□ . . . 74
7.1.1 box2d . . . 74
7.1.2 box3d . . . 75
7.1.3 geometry . . . 75
7.1.4 geometry_dump . . . 76
7.1.5 geography . . . 76
7.2 □□□□ . . . 77
7.2.1 AddGeometryColumn . . . 77
7.2.2 DropGeometryColumn . . . 79
7.2.3 DropGeometryTable . . . 80
7.2.4 Find_SRID . . . 81
7.2.5 Populate_Geometry_Columns . . . 81

7.2.6 UpdateGeometrySRID . . . 83
7.3 □□□□□ (constructor) . . . 85
7.3.1 ST_Collect . . . 85
7.3.2 ST_LineFromMultiPoint . . . 89
7.3.3 ST_MakeEnvelope . . . 90
7.3.4 ST_MakeLine . . . 91
7.3.5 ST_MakePoint . . . 94
7.3.6 ST_MakePointM . . . 96
7.3.7 ST_MakePolygon . . . 98
7.3.8 ST_Point . . . 102
7.3.9 ST_PointZ . . . 103
7.3.10 ST_PointM . . . 104
7.3.11 ST_PointZM . . . 104
7.3.12 ST_Polygon . . . 105
7.3.13 ST_TileEnvelope . . . 107
7.3.14 ST_HexagonGrid . . . 109
7.3.15 ST_Hexagon . . . 114
7.3.16 ST_SquareGrid . . . 116
7.3.17 ST_Square . . . 117
7.3.18 ST_Letters . . . 118
7.3.19 ST_MakeNurbsCurve . . . 120
7.4 □□□□□ (accessor) . . . 123
7.4.1 GeometryType . . . 123
7.4.2 ST_Boundary . . . 125
7.4.3 ST_BoundingDiagonal . . . 131
7.4.4 ST_CoordDim . . . 132
7.4.5 ST_Dimension . . . 133
7.4.6 ST_Dump . . . 135
7.4.7 ST_DumpPoints . . . 139
7.4.8 ST_DumpSegments . . . 144
7.4.9 ST_DumpRings . . . 147
7.4.10 ST_EndPoint . . . 149
7.4.11 ST_Envelope . . . 153
7.4.12 ST_ExteriorRing . . . 157
7.4.13 ST_GeometryN . . . 160
7.4.14 ST_GeometryType . . . 164
7.4.15 ST_HasArc . . . 165
7.4.16 ST_InteriorRingN . . . 166
7.4.17 ST_NumCurves . . . 168

7.4.18ST_CurveN . . . 169
7.4.19ST_IsClosed . . . 171
7.4.20ST_IsCollection . . . 175
7.4.21ST_IsEmpty . . . 177
7.4.22ST_IsPolygonCCW . . . 179
7.4.23ST_IsPolygonCW . . . 180
7.4.24ST_IsRing . . . 181
7.4.25ST_IsSimple . . . 183
7.4.26ST_IsSimpleDetail . . . 185
7.4.27ST_M . . . 187
7.4.28ST_MemSize . . . 188
7.4.29ST_NDims . . . 189
7.4.30ST_NPoints . . . 190
7.4.31ST_NRings . . . 191
7.4.32ST_NumGeometries . . . 192
7.4.33ST_NumInteriorRings . . . 193
7.4.34ST_NumInteriorRing . . . 194
7.4.35ST_NumPatches . . . 195
7.4.36ST_NumPoints . . . 195
7.4.37ST_PatchN . . . 196
7.4.38ST_PointN . . . 198
7.4.39ST_Points . . . 201
7.4.40ST_StartPoint . . . 202
7.4.41ST_StartM . . . 207
7.4.42ST_EndM . . . 210
7.4.43ST_SetStartM . . . 213
7.4.44ST_SetEndM . . . 216
7.4.45ST_Summary . . . 219
7.4.46ST_X . . . 221
7.4.47ST_Y . . . 221
7.4.48ST_Z . . . 222
7.4.49ST_ZMFlag . . . 223
7.4.50ST_HasZ . . . 224
7.4.51ST_HasM . . . 225
7.4.52ST_ControlPoints . . . 227
7.4.53ST_Degree . . . 228
7.4.54ST_Weights . . . 230
7.4.55ST_Knots . . . 232
7.4.56ST_NumControlPoints . . . 234

7.4.57ST_NurbsCurveIsRational . . . 235
7.4.58ST_IsNurbsCurve . . . 237
7.4.59ST_Evaluate . . . 239
7.4.60ST_NurbsToLineString . . . 240
7.5 □□□□□ (editor) . . . 242
7.5.1 ST_AddPoint . . . 242
7.5.2 ST_CollectionExtract . . . 244
7.5.3 ST_CollectionHomogenize . . . 247
7.5.4 ST_CurveToLine . . . 251
7.5.5 ST_Scroll . . . 255
7.5.6 ST_FlipCoordinates . . . 256
7.5.7 ST_Force2D . . . 257
7.5.8 ST_Force3D . . . 259
7.5.9 ST_Force3DZ . . . 261
7.5.10ST_Force3DM . . . 263
7.5.11ST_Force4D . . . 265
7.5.12ST_ForceCollection . . . 267
7.5.13ST_ForceCurve . . . 270
7.5.14ST_ForcePolygonCCW . . . 271
7.5.15ST_ForcePolygonCW . . . 272
7.5.16ST_ForceSFS . . . 274
7.5.17ST_ForceRHR . . . 275
7.5.18ST_LineExtend . . . 276
7.5.19ST_LineToCurve . . . 277
7.5.20ST_Multi . . . 279
7.5.21ST_Normalize . . . 280
7.5.22ST_Project . . . 281
7.5.23ST_QuantizeCoordinates . . . 283
7.5.24ST_RemovePoint . . . 285
7.5.25ST_RemoveRepeatedPoints . . . 286
7.5.26ST_RemoveIrrelevantPointsForView . . . 289
7.5.27ST_RemoveSmallParts . . . 294
7.5.28ST_Reverse . . . 296
7.5.29ST_Segmentize . . . 297
7.5.30ST_SetPoint . . . 300
7.5.31ST_ShiftLongitude . . . 303
7.5.32ST_WrapX . . . 306
7.5.33ST_SnapToGrid . . . 306
7.5.34ST_Snap . . . 310

7.5.35 ST_SwapOrdinates . . . 314
7.6 Geometry Validation . . . 315
7.6.1 ST_IsValid . . . 315
7.6.2 ST_IsValidDetail . . . 318
7.6.3 ST_IsValidReason . . . 320
7.6.4 ST_MakeValid . . . 322
7.7 Spatial Reference System Functions . . . 326
7.7.1 ST_InverseTransformPipeline . . . 326
7.7.2 ST_SetSRID . . . 328
7.7.3 ST_SRID . . . 330
7.7.4 ST_Transform . . . 331
7.7.5 ST_TransformPipeline . . . 335
7.7.6 postgis_srs_codes . . . 338
7.7.7 postgis_srs . . . 339
7.7.8 postgis_srs_all . . . 340
7.7.9 postgis_srs_search . . . 341
7.8 Geometry Input . . . 342
7.8.1 Well-Known Text (WKT) . . . 342
7.8.1.1 ST_BdPolyFromText . . . 342
7.8.1.2 ST_BdMPolyFromText . . . 343
7.8.1.3 ST_GeogFromText . . . 344
7.8.1.4 ST_GeographyFromText . . . 345
7.8.1.5 ST_GeomCollFromText . . . 345
7.8.1.6 ST_GeomFromEWKT . . . 346
7.8.1.7 ST_GeomFromMARC21 . . . 349
7.8.1.8 ST_GeometryFromText . . . 353
7.8.1.9 ST_GeomFromText . . . 354
7.8.1.10 ST_LineFromText . . . 359
7.8.1.11 ST_MLineFromText . . . 360
7.8.1.12 ST_MPointFromText . . . 361
7.8.1.13 ST_MPolyFromText . . . 362
7.8.1.14 ST_PointFromText . . . 364
7.8.1.15 ST_PolygonFromText . . . 366
7.8.1.16 ST_WKTToSQL . . . 368
7.8.2 Well-Known Binary (WKB) . . . 368
7.8.2.1 ST_GeogFromWKB . . . 368
7.8.2.2 ST_GeomFromEWKB . . . 369
7.8.2.3 ST_GeomFromWKB . . . 371
7.8.2.4 ST_LineFromWKB . . . 373

7.8.2.5 ST_LinestringFromWKB . . . 375
7.8.2.6 ST_PointFromWKB . . . 377
7.8.2.7 ST_WKBToSQL . . . 378
7.8.3 Other Formats . . . 379
7.8.3.1 ST_Box2dFromGeoHash . . . 379
7.8.3.2 ST_GeomFromGeoHash . . . 379
7.8.3.3 ST_GeomFromGML . . . 381
7.8.3.4 ST_GeomFromGeoJSON . . . 385
7.8.3.5 ST_GeomFromKML . . . 387
7.8.3.6 ST_GeomFromTWKB . . . 388
7.8.3.7 ST_GMLToSQL . . . 390
7.8.3.8 ST_LineFromEncodedPolyline . . . 391
7.8.3.9 ST_PointFromGeoHash . . . 392
7.8.3.10 ST_FromFlatGeobufToTable . . . 395
7.8.3.11 ST_FromFlatGeobuf . . . 395
7.9 Geometry Output . . . 395
7.9.1 Well-Known Text (WKT) . . . 395
7.9.1.1 ST_AsEWKT . . . 395
7.9.1.2 ST_AsText . . . 398
7.9.2 Well-Known Binary (WKB) . . . 400
7.9.2.1 ST_AsBinary . . . 400
7.9.2.2 ST_AsEWKB . . . 402
7.9.2.3 ST_AsHEXEWKB . . . 403
7.9.3 Other Formats . . . 404
7.9.3.1 ST_AsEncodedPolyline . . . 404
7.9.3.2 ST_AsFlatGeobuf . . . 406
7.9.3.3 ST_AsGeobuf . . . 406
7.9.3.4 ST_AsGeoJSON . . . 407
7.9.3.5 ST_AsGML . . . 410
7.9.3.6 ST_AsKML . . . 413
7.9.3.7 ST_AsLatLonText . . . 414
7.9.3.8 ST_AsMARC21 . . . 416
7.9.3.9 ST_AsMVTGeom . . . 419
7.9.3.10 ST_AsMVT . . . 420
7.9.3.11 ST_AsSVG . . . 422
7.9.3.12 ST_AsTWKB . . . 425
7.9.3.13 ST_AsX3D . . . 426
7.9.3.14 ST_GeoHash . . . 430
7.10 ☒☒☒ (operator) . . . 431

7.10.1 Bounding Box Operators . . . 431
7.10.1.1 && . . . 431
7.10.1.2 &&(geometry,box2df) . . . 433
7.10.1.3 &&(box2df,geometry) . . . 434
7.10.1.4 &&(box2df,box2df) . . . 435
7.10.1.5 &&& . . . 436
7.10.1.6 &&&(geometry,gidx) . . . 438
7.10.1.7 &&&(gidx,geometry) . . . 439
7.10.1.8 &&&(gidx,gidx) . . . 440
7.10.1.9 &< . . . 441
7.10.1.10 &<| . . . 442
7.10.1.11 &> . . . 443
7.10.1.12 << . . . 445
7.10.1.13 <<| . . . 446
7.10.1.14 = . . . 447
7.10.1.15 >> . . . 450
7.10.1.16 @ . . . 452
7.10.1.17 @(geometry,box2df) . . . 453
7.10.1.18 @(box2df,geometry) . . . 454
7.10.1.19 @(box2df,box2df) . . . 455
7.10.1.20 |&> . . . 456
7.10.1.21 |>> . . . 457
7.10.1.22 ~ . . . 458
7.10.1.23 ~(geometry,box2df) . . . 460
7.10.1.24 ~(box2df,geometry) . . . 460
7.10.1.25 ~(box2df,box2df) . . . 461
7.10.1.26 ~= . . . 462
7.10.2 □□□ (operator) . . . 463
7.10.2.1 <-> . . . 463
7.10.2.2 |=| . . . 465
7.10.2.3 <#> . . . 467
7.10.2.4 <<->> . . . 468
7.11 Spatial Relationships . . . 468
7.11.1 Topological Relationships . . . 468
7.11.1.1 ST_3DIntersects . . . 468
7.11.1.2 ST_Contains . . . 471
7.11.1.3 ST_ContainsProperly . . . 482
7.11.1.4 ST_CoveredBy . . . 485
7.11.1.5 ST_Covers . . . 487

7.11.1.6 ST_Crosses . . . 489
7.11.1.7 ST_Disjoint . . . 493
7.11.1.8 ST_Equals . . . 495
7.11.1.9 ST_Intersects . . . 497
7.11.1.10 ST_LineCrossingDirection . . . 500
7.11.1.11 ST_OrderingEquals . . . 503
7.11.1.12 ST_Overlaps . . . 506
7.11.1.13 ST_Relate . . . 512
7.11.1.14 ST_RelateMatch . . . 516
7.11.1.15 ST_Touches . . . 517
7.11.1.16 ST_Within . . . 525
7.11.2 Distance Relationships . . . 528
7.11.2.1 ST_3DDWithin . . . 528
7.11.2.2 ST_3DDFullyWithin . . . 529
7.11.2.3 ST_DFullyWithin . . . 531
7.11.2.4 ST_DWithin . . . 532
7.11.2.5 ST_PointInsideCircle . . . 535
7.12 Measurement Functions . . . 536
7.12.1 ST_Area . . . 536
7.12.2 ST_Azimuth . . . 539
7.12.3 ST_Angle . . . 542
7.12.4 ST_ClosestPoint . . . 545
7.12.5 ST_3DClosestPoint . . . 547
7.12.6 ST_Distance . . . 550
7.12.7 ST_3DDistance . . . 552
7.12.8 ST_DistanceSphere . . . 554
7.12.9 ST_DistanceSpheroid . . . 555
7.12.10 ST_FrechetDistance . . . 557
7.12.11 ST_HausdorffDistance . . . 559
7.12.12 ST_Length . . . 561
7.12.13 ST_Length2D . . . 564
7.12.14 ST_3DLength . . . 564
7.12.15 ST_LengthSpheroid . . . 566
7.12.16 ST_LongestLine . . . 568
7.12.17 ST_3DLongestLine . . . 571
7.12.18 ST_MaxDistance . . . 574
7.12.19 ST_3DMaxDistance . . . 577
7.12.20 ST_MinimumClearance . . . 578
7.12.21 ST_MinimumClearanceLine . . . 579

7.12.23 ST_Perimeter . . . 580
7.12.24 ST_Perimeter2D . . . 584
7.12.25 ST_3DPerimeter . . . 585
7.12.26 ST_ShortestLine . . . 586
7.12.27 ST_3DShortestLine . . . 588
7.13 Overlay Functions . . . 591
7.13.1 ST_ClipByBox2D . . . 591
7.13.2 ST_Difference . . . 592
7.13.3 ST_Intersection . . . 594
7.13.4 ST_MemUnion . . . 598
7.13.5 ST_Node . . . 598
7.13.6 ST_Split . . . 600
7.13.7 ST_Subdivide . . . 603
7.13.8 ST_SymDifference . . . 607
7.13.9 ST_UnaryUnion . . . 609
7.13.10 ST_Union . . . 609
7.14 □□□□□□ . . . 615
7.14.1 ST_Buffer . . . 615
7.14.2 ST_BuildArea . . . 624
7.14.3 ST_Centroid . . . 627
7.14.4 ST_ChaikinSmoothing . . . 633
7.14.5 ST_CatmullRomSmoothing . . . 636
7.14.6 ST_ConcaveHull . . . 639
7.14.7 ST_ConvexHull . . . 643
7.14.8 ST_DelaunayTriangles . . . 645
7.14.9 ST_FilterByM . . . 648
7.14.10 ST_GeneratePoints . . . 650
7.14.11 ST_GeometricMedian . . . 652
7.14.12 ST_LineMerge . . . 653
7.14.13 ST_MaximumInscribedCircle . . . 658
7.14.14 ST_LargestEmptyCircle . . . 660
7.14.15 ST_MinimumBoundingCircle . . . 662
7.14.16 ST_MinimumBoundingRadius . . . 664
7.14.17 ST_OrientedEnvelope . . . 665
7.14.18 ST_OffsetCurve . . . 667
7.14.19 ST_PointOnSurface . . . 674
7.14.20 ST_Polygonize . . . 679
7.14.21 ST_ReducePrecision . . . 682
7.14.22 ST_SharedPaths . . . 686

7.14.23 ST_Simplify . . . 688
7.14.24 ST_SimplifyPreserveTopology . . . 691
7.14.25 ST_SimplifyPolygonHull . . . 694
7.14.26 ST_SimplifyVW . . . 697
7.14.27 ST_SetEffectiveArea . . . 701
7.14.28 ST_TriangulatePolygon . . . 702
7.14.29 ST_VoronoiLines . . . 705
7.14.30 ST_VoronoiPolygons . . . 706
7.15 Coverages . . . 708
7.15.1 ST_CoverageInvalidEdges . . . 708
7.15.2 ST_CoverageClean . . . 710
7.15.3 ST_CoverageSimplify . . . 712
7.15.4 ST_CoverageEdges . . . 713
7.15.5 ST_CoverageUnion . . . 714
7.16 Affine Transformations . . . 716
7.16.1 ST_Affine . . . 716
7.16.2 ST_Rotate . . . 718
7.16.3 ST_RotateX . . . 721
7.16.4 ST_RotateY . . . 722
7.16.5 ST_RotateZ . . . 723
7.16.6 ST_Scale . . . 725
7.16.7 ST_Translate . . . 729
7.16.8 ST_TransScale . . . 732
7.17 Clustering Functions . . . 734
7.17.1 ST_ClusterDBSCAN . . . 734
7.17.2 ST_ClusterIntersecting . . . 736
7.17.3 ST_ClusterIntersectingWin . . . 737
7.17.4 ST_ClusterRelateWin . . . 739
7.17.5 ST_MinimumSpanningTree . . . 740
7.17.6 ST_ClusterKMeans . . . 742
7.17.7 ST_ClusterWithin . . . 746
7.17.8 ST_ClusterWithinWin . . . 747
7.18 Bounding Box Functions . . . 749
7.18.1 Box2D . . . 749
7.18.2 Box3D . . . 751
7.18.3 ST_EstimatedExtent . . . 753
7.18.4 ST_Expand . . . 754
7.18.5 ST_Extent . . . 757
7.18.6 ST_3DExtent . . . 758

7.18.7 ST_MakeBox2D . . . 760
7.18.8 ST_3DMakeBox . . . 760
7.18.9 ST_XMax . . . 761
7.18.10 ST_XMin . . . 763
7.18.11 ST_YMax . . . 765
7.18.12 ST_YMin . . . 768
7.18.13 ST_ZMax . . . 770
7.18.14 ST_ZMin . . . 772
7.18.15 ST_XSize . . . 775
7.18.16 ST_YSize . . . 776
7.18.17 ST_ZSize . . . 778
7.18.18 ST_MMin . . . 779
7.18.19 ST_MMax . . . 780
7.18.20 ST_MSize . . . 780
7.19 □□□□ (Linear Referencing) . . . 781
7.19.1 ST_LineInterpolatePoint . . . 781
7.19.2 ST_3DLineInterpolatePoint . . . 785
7.19.3 ST_LineInterpolatePoints . . . 786
7.19.4 ST_LineLocatePoint . . . 788
7.19.5 ST_LineSubstring . . . 791
7.19.6 ST_LocateAlong . . . 797
7.19.7 ST_LocateBetween . . . 798
7.19.8 ST_LocateBetweenElevations . . . 801
7.19.9 ST_InterpolatePoint . . . 803
7.19.10 ST_3DInterpolatePoint . . . 804
7.19.11 ST_AddMeasure . . . 806
7.20 Trajectory Functions . . . 809
7.20.1 ST_IsValidTrajectory . . . 809
7.20.2 ST_ClosestPointOfApproach . . . 810
7.20.3 ST_DistanceCPA . . . 812
7.20.4 ST_CPAWithin . . . 813
7.21 Version Functions . . . 814
7.21.1 PostGIS_Extensions_Upgrade . . . 814
7.21.2 PostGIS_Full_Version . . . 815
7.21.3 PostGIS_GEOS_Version . . . 816
7.21.4 PostGIS_GEOS_Compiled_Version . . . 816
7.21.5 PostGIS_Liblwgeom_Version . . . 817
7.21.6 PostGIS_LibXML_Version . . . 818
7.21.7 PostGIS_LibJSON_Version . . . 818

7.21.8 PostGIS_Lib_Build_Date . . . 819
7.21.9 PostGIS_Lib_Version . . . 819
7.21.10 PostGIS_PROJ_Version . . . 820
7.21.11 PostGIS_PROJ_Compiled_Version . . . 820
7.21.12 PostGIS_Wagyu_Version . . . 821
7.21.13 PostGIS_Scripts_Build_Date . . . 822
7.21.14 PostGIS_Scripts_Installed . . . 822
7.21.15 PostGIS_Scripts_Released . . . 823
7.21.16 PostGIS_Version . . . 823
7.22 PostGIS GUC(Grand Unified Custom Variable) . . . 824
7.22.1 postgis.gdal_datapath . . . 824
7.22.2 postgis.gdal_enabled_drivers . . . 825
7.22.3 postgis.enable_outdb_rasters . . . 826
7.22.4 postgis.gdal_vsi_options . . . 828
7.22.5 postgis.gdal_cpl_debug . . . 828
7.23 Troubleshooting Functions . . . 829
7.23.1 PostGIS_AddBBox . . . 829
7.23.2 PostGIS_DropBBox . . . 829
7.23.3 PostGIS_HasBBox . . . 830

8 SFCGAL Functions Reference 832
8.1 SFCGAL Management Functions . . . 832
8.1.1 postgis_sfcgal_version . . . 832
8.1.2 postgis_sfcgal_full_version . . . 832
8.2 SFCGAL Accessors and Setters . . . 833
8.2.1 CG_ForceLHR . . . 833
8.2.2 CG_IsPlanar . . . 834
8.2.3 CG_IsValid . . . 835
8.2.4 CG_IsValidDetail . . . 836
8.2.5 CG_IsSimple . . . 837
8.2.6 CG_IsSimpleDetail . . . 838
8.2.7 CG_IsSolid . . . 839
8.2.8 CG_MakeSolid . . . 841
8.2.9 CG_Orientation . . . 841
8.2.10 CG_Area . . . 841
8.2.11 CG_3DArea . . . 843
8.2.12 CG_Volume . . . 843
8.3 SFCGAL Processing and Relationship Functions . . . 845
8.3.1 CG_Intersection . . . 845

8.3.2 CG_Intersects . . . 846
8.3.3 CG_3DIntersects . . . 848
8.3.4 CG_Difference . . . 849
8.3.5 CG_3DDifference . . . 850
8.3.6 CG_Distance . . . 852
8.3.7 CG_3DDistance . . . 853
8.3.8 CG_3DConvexHull . . . 854
8.3.9 CG_3DIntersection . . . 857
8.3.10 CG_Union . . . 860
8.3.11 CG_3DUnion . . . 861
8.3.12 CG_AlphaShape . . . 863
8.3.13 CG_ApproxConvexPartition . . . 866
8.3.14 CG_ApproximateMedialAxis . . . 868
8.3.15 CG_GenerateFlatRoof . . . 870
8.3.16 CG_GenerateHippedRoof . . . 871
8.3.17 CG_GenerateGableRoof . . . 872
8.3.18 CG_GenerateSkillionRoof . . . 873
8.3.19 CG_GenerateRoof . . . 875
8.3.20 CG_PolygonRepair . . . 876
8.3.21 CG_ConstrainedDelaunayTriangles . . . 879
8.3.22 CG_Extrude . . . 881
8.3.23 CG_ExtrudeStraightSkeleton . . . 883
8.3.24 CG_GreeneApproxConvexPartition . . . 885
8.3.25 CG_MinkowskiSum . . . 886
8.3.26 CG_OptimalAlphaShape . . . 888
8.3.27 CG_OptimalConvexPartition . . . 890
8.3.28 CG_StraightSkeleton . . . 892
8.3.29 CG_Tesselate . . . 894
8.3.30 CG_Triangulate . . . 896
8.3.31 CG_Visibility . . . 898
8.3.32 CG_YMonotonePartition . . . 900
8.3.33 CG_StraightSkeletonPartition . . . 901
8.3.34 CG_3DBuffer . . . 903
8.3.35 CG_Rotate . . . 906
8.3.36 CG_2DRotate . . . 907
8.3.37 CG_3DRotate . . . 908
8.3.38 CG_RotateX . . . 909
8.3.39 CG_RotateY . . . 910
8.3.40 CG_RotateZ . . . 911

8.3.41CG_Scale . . . 912
8.3.42CG_3DScale . . . 913
8.3.43CG_3DScaleAroundCenter . . . 914
8.3.44CG_Translate . . . 915
8.3.45CG_3DTranslate . . . 916
8.3.46CG_Simplify . . . 917
8.3.47CG_3DAlphaWrapping . . . 922
8.3.48CG_NurbsCurveInterpolate . . . 923
8.3.49CG_NurbsCurveApproximate . . . 925
8.3.50CG_NurbsCurveDerivative . . . 928
8.4 Deprecated SFCGAL Function Aliases . . . 931
8.4.1 ST_3DIntersection . . . 931
8.4.2 ST_3DDifference . . . 932
8.4.3 ST_3DUnion . . . 932
8.4.4 ST_Tesselate . . . 932
8.4.5 ST_3DArea . . . 933
8.4.6 ST_Extrude . . . 933
8.4.7 ST_ForceLHR . . . 934
8.4.8 ST_Orientation . . . 934
8.4.9 ST_MinkowskiSum . . . 935
8.4.10ST_StraightSkeleton . . . 935
8.4.11ST_ApproximateMedialAxis . . . 935
8.4.12ST_IsPlanar . . . 936
8.4.13ST_Volume . . . 936
8.4.14ST_MakeSolid . . . 937
8.4.15ST_IsSolid . . . 937
8.4.16ST_ConstrainedDelaunayTriangles . . . 938
8.4.17ST_3DConvexHull . . . 938
8.4.18ST_AlphaShape . . . 938
8.4.19ST_OptimalAlphaShape . . . 939

9 □□ (topology) 940
9.1 Building Faces from Linework . . . 940
9.2 Topology Primitive Tables . . . 942
9.3 □□□□ . . . 942
9.3.1 getfaceedges_returntype . . . 942
9.3.2 TopoGeometry . . . 943
9.3.3 validatetopology_returntype . . . 943
9.4 □□□□□ . . . 944

9.4.1 TopoElement . . . 944
9.4.2 TopoElementArray . . . 945
9.5 □□□ TopoGeometry □□ . . . 946
9.5.1 AddTopoGeometryColumn . . . 946
9.5.2 RenameTopoGeometryColumn . . . 947
9.5.3 DropTopology . . . 948
9.5.4 RenameTopology . . . 948
9.5.5 DropTopoGeometryColumn . . . 949
9.5.6 FixCorruptTopoGeometryColumn . . . 949
9.5.7 Populate_Topology_Layer . . . 950
9.5.8 TopologySummary . . . 952
9.5.9 ValidateTopology . . . 952
9.5.10 ValidateTopologyRelation . . . 955
9.5.11 ValidateTopologyPrecision . . . 955
9.5.12 MakeTopologyPrecise . . . 956
9.5.13 FindTopology . . . 957
9.5.14 FindLayer . . . 958
9.5.15 TotalTopologySize . . . 958
9.5.16 UpgradeTopology . . . 959
9.6 Topology Statistics Management . . . 960
9.7 □□□□□ . . . 960
9.7.1 CreateTopology . . . 960
9.7.2 CopyTopology . . . 961
9.7.3 ST_InitTopoGeo . . . 962
9.7.4 ST_CreateTopoGeo . . . 962
9.7.5 TopoGeo_AddPoint . . . 964
9.7.6 TopoGeo_AddLineString . . . 964
9.7.7 TopoGeo_AddPolygon . . . 965
9.7.8 TopoGeo_LoadGeometry . . . 965
9.8 □□□□□ . . . 966
9.8.1 ST_AddIsoNode . . . 966
9.8.2 ST_AddIsoEdge . . . 967
9.8.3 ST_AddEdgeNewFaces . . . 967
9.8.4 ST_AddEdgeModFace . . . 968
9.8.5 ST_RemEdgeNewFace . . . 968
9.8.6 ST_RemEdgeModFace . . . 969
9.8.7 ST_ChangeEdgeGeom . . . 970
9.8.8 ST_ModEdgeSplit . . . 971
9.8.9 ST_ModEdgeHeal . . . 972

9.8.10ST_NewEdgeHeal . . . 972
9.8.11ST_MoveIsoNode . . . 973
9.8.12ST_NewEdgesSplit . . . 974
9.8.13ST_RemoveIsoNode . . . 974
9.8.14ST_RemoveIsoEdge . . . 975
9.9 □□□□□ . . . 976
9.9.1 GetEdgeByPoint . . . 976
9.9.2 GetFaceByPoint . . . 977
9.9.3 GetFaceContainingPoint . . . 978
9.9.4 GetNodeByPoint . . . 978
9.9.5 GetTopologyID . . . 979
9.9.6 GetTopologySRID . . . 980
9.9.7 GetTopologyName . . . 981
9.9.8 ST_GetFaceEdges . . . 981
9.9.9 ST_GetFaceGeometry . . . 982
9.9.10GetRingEdges . . . 983
9.9.11GetNodeEdges . . . 984
9.9.12FindVertexSegmentPairsBelowDistance . . . 984
9.10□□□□□□ . . . 985
9.10.1Polygonize . . . 985
9.10.2AddNode . . . 985
9.10.3AddEdge . . . 986
9.10.4AddFace . . . 988
9.10.5ST_Simplify . . . 989
9.10.6RemoveUnusedPrimitives . . . 990
9.11TopoGeometry □□□ . . . 990
9.11.1CreateTopoGeom . . . 990
9.11.2toTopoGeom . . . 993
9.11.3TopoElementArray_Agg . . . 995
9.11.4TopoElement . . . 995
9.12TopoGeometry □□□ . . . 996
9.12.1clearTopoGeom . . . 996
9.12.2TopoGeom_addElement . . . 996
9.12.3TopoGeom_remElement . . . 997
9.12.4TopoGeom_addTopoGeom . . . 998
9.12.5toTopoGeom . . . 998
9.13TopoGeometry □□□ . . . 998
9.13.1GetTopoGeomElementArray . . . 998
9.13.2GetTopoGeomElements . . . 999

9.13.3ST_SRID . . . 999
9.14TopoGeometry □□□ . . . 1000
9.14.1AsGML . . . 1000
9.14.2AsTopoJSON . . . 1002
9.15□□□□□□□ . . . 1004
9.15.1Equals . . . 1004
9.15.2Intersects . . . 1005
9.16Importing and exporting Topologies . . . 1006
9.16.1Using the Topology exporter . . . 1006
9.16.2Using the Topology importer . . . 1006

10□□□□□□□□□, □□□□□ 1008
10.1□□□□□□□□ . . . 1008
10.1.1raster2pgsql □□□□□□□□□□□□ . . . 1008
10.1.1.1Example Usage . . . 1008
10.1.1.2raster2pgsql options . . . 1009
10.1.2PostGIS □□□□□□□□□□□□□□□ . . . 1011
10.1.3Using ”out db” cloud rasters . . . 1011
10.2Raster Coverages and Raster/Vector Analysis . . . 1012
10.3□□□□□□□ . . . 1014
10.3.1□□□□□□□□ . . . 1014
10.3.2□□□□□□ . . . 1015
10.4PostGIS □□□□□□□□□□□□□□□□□□□□□□ . . . 1016
10.4.1Rendering Map Algebra and Reclass Results . . . 1016
10.4.2PL/pgSQL Map Algebra Callback Example . . . 1017
10.4.3□□□□□□□□□□ ST_AsPNG □□□□□ PHP □□□□□□□ . . . 1017
10.4.4□□□□□□□□□□ ST_AsPNG □□□□□ ASP.NET C# □□□□□□□ . . . 1018
10.4.5□□□□□□□□□□□□□□□□□ Java □□□□□□□□ . . . 1020
10.4.6PLPython □□□□□ SQL □□□□□□□□□□□ . . . 1021
10.4.7PSQL □□□□□□□□□□□□ . . . 1022

11□□□□□□□ 1024
11.1□□□□□□□□□ . . . 1025
11.1.1geomval . . . 1025
11.1.2addbandarg . . . 1025
11.1.3rastbandarg . . . 1025
11.1.4raster . . . 1026
11.1.5reclassarg . . . 1026
11.1.6summarystats . . . 1027

11.1.7unionarg . . . 1028
11.2□□□□□ . . . 1028
11.2.1AddRasterConstraints . . . 1028
11.2.2DropRasterConstraints . . . 1030
11.2.3AddOverviewConstraints . . . 1031
11.2.4DropOverviewConstraints . . . 1032
11.2.5PostGIS_GDAL_Version . . . 1033
11.2.6PostGIS_Raster_Lib_Build_Date . . . 1033
11.2.7PostGIS_Raster_Lib_Version . . . 1034
11.2.8ST_GDALDrivers . . . 1035
11.2.9UpdateRasterSRID . . . 1038
11.2.10ST_CreateOverview . . . 1038
11.3□□□□□□ (constructor) . . . 1039
11.3.1ST_AddBand . . . 1039
11.3.2ST_AsRaster . . . 1041
11.3.3ST_AsRasterAgg . . . 1045
11.3.4ST_Band . . . 1046
11.3.5ST_MakeEmptyCoverage . . . 1050
11.3.6ST_MakeEmptyRaster . . . 1052
11.3.7ST_Tile . . . 1053
11.3.8ST_Retile . . . 1055
11.3.9ST_FromGDALRaster . . . 1056
11.4□□□□□□ (accessor) . . . 1057
11.4.1ST_GeoReference . . . 1057
11.4.2ST_Height . . . 1058
11.4.3ST_IsEmpty . . . 1059
11.4.4ST_MemSize . . . 1059
11.4.5ST_MetaData . . . 1060
11.4.6ST_NumBands . . . 1061
11.4.7ST_PixelHeight . . . 1062
11.4.8ST_PixelWidth . . . 1063
11.4.9ST_ScaleX . . . 1065
11.4.10ST_ScaleY . . . 1065
11.4.11ST_RasterToWorldCoord . . . 1066
11.4.12ST_RasterToWorldCoordX . . . 1067
11.4.13ST_RasterToWorldCoordY . . . 1068
11.4.14ST_Rotation . . . 1069
11.4.15ST_SkewX . . . 1070
11.4.16ST_SkewY . . . 1071

11.4.17 ST_SRID . . . 1072
11.4.18 ST_Summary . . . 1073
11.4.19 ST_UpperLeftX . . . 1073
11.4.20 ST_UpperLeftY . . . 1074
11.4.21 ST_Width . . . 1075
11.4.22 ST_WorldToRasterCoord . . . 1075
11.4.23 ST_WorldToRasterCoordX . . . 1076
11.4.24 ST_WorldToRasterCoordY . . . 1077
11.5 □□□□□□□□ . . . 1078
11.5.1 ST_BandMetaData . . . 1078
11.5.2 ST_BandNoDataValue . . . 1079
11.5.3 ST_BandIsNoData . . . 1080
11.5.4 ST_BandPath . . . 1081
11.5.5 ST_BandFileSize . . . 1082
11.5.6 ST_BandFileTimestamp . . . 1083
11.5.7 ST_BandPixelType . . . 1083
11.5.8 ST_MinPossibleValue . . . 1084
11.5.9 ST_HasNoBand . . . 1085
11.6 □□□□□□□□□□□□ (setter) . . . 1086
11.6.1 ST_PixelAsPolygon . . . 1086
11.6.2 ST_PixelAsPolygons . . . 1087
11.6.3 ST_PixelAsPoint . . . 1088
11.6.4 ST_PixelAsPoints . . . 1089
11.6.5 ST_PixelAsCentroid . . . 1091
11.6.6 ST_PixelAsCentroids . . . 1092
11.6.7 ST_Value . . . 1094
11.6.8 ST_NearestValue . . . 1097
11.6.9 ST_SetZ . . . 1099
11.6.10 ST_SetM . . . 1101
11.6.11 ST_Neighborhood . . . 1103
11.6.12 ST_SetValue . . . 1105
11.6.13 ST_SetValues . . . 1106
11.6.14 ST_DumpValues . . . 1114
11.6.15 ST_PixelOfValue . . . 1115
11.7 □□□□□□ . . . 1116
11.7.1 ST_SetGeoReference . . . 1116
11.7.2 ST_SetRotation . . . 1118
11.7.3 ST_SetScale . . . 1119
11.7.4 ST_SetSkew . . . 1120

11.7.5 ST_SetSRID . . . 1122
11.7.6 ST_SetUpperLeft . . . 1122
11.7.7 ST_Resample . . . 1123
11.7.8 ST_Rescale . . . 1124
11.7.9 ST_Reskew . . . 1126
11.7.10 ST_SnapToGrid . . . 1127
11.7.11 ST_Resize . . . 1129
11.7.12 ST_Transform . . . 1130
11.8 □□□□□□□□ . . . 1134
11.8.1 ST_SetBandNoDataValue . . . 1134
11.8.2 ST_SetBandIsNoData . . . 1135
11.8.3 ST_SetBandPath . . . 1137
11.8.4 ST_SetBandIndex . . . 1138
11.9 □□□□□□□□□□ . . . 1140
11.9.1 ST_Count . . . 1140
11.9.2 ST_CountAgg . . . 1141
11.9.3 ST_Histogram . . . 1142
11.9.4 ST_Quantile . . . 1144
11.9.5 ST_SummaryStats . . . 1146
11.9.6 ST_SummaryStatsAgg . . . 1148
11.9.7 ST_ValueCount . . . 1149
11.10 Raster Inputs . . . 1152
11.10.1 ST_RastFromWKB . . . 1152
11.10.2 ST_RastFromHexWKB . . . 1153
11.11 □□□□□ . . . 1154
11.11.1 ST_AsBinary/ST_AsWKB . . . 1154
11.11.2 ST_AsHexWKB . . . 1154
11.11.3 ST_AsGDALRaster . . . 1155
11.11.4 ST_AsJPEG . . . 1157
11.11.5 ST_AsPNG . . . 1158
11.11.6 ST_AsTIFF . . . 1159
11.12 □□□□□□□ . . . 1160
11.12.1 ST_Clip . . . 1160
11.12.2 ST_ColorMap . . . 1165
11.12.3 ST_Grayscale . . . 1168
11.12.4 ST_Intersection . . . 1172
11.12.5 ST_MapAlgebra (callback function version) . . . 1175
11.12.6 ST_MapAlgebra (expression version) . . . 1183
11.12.7 ST_MapAlgebraExpr . . . 1185

11.12.8 ST_MapAlgebraExpr . . . 1189
11.12.9 ST_MapAlgebraFct . . . 1196
11.12.10 ST_MapAlgebraFct . . . 1200
11.12.11 ST_MapAlgebraFctNgb . . . 1206
11.12.12 ST_Reclass . . . 1210
11.12.13 ST_ReclassExact . . . 1212
11.12.14 ST_Union . . . 1213
11.13 □□□□□□□□□ . . . 1215
11.13.1 ST_Distinct4ma . . . 1215
11.13.2 ST_InvDistWeight4ma . . . 1216
11.13.3 ST_Max4ma . . . 1217
11.13.4 ST_Mean4ma . . . 1218
11.13.5 ST_Min4ma . . . 1220
11.13.6 ST_MinDist4ma . . . 1221
11.13.7 ST_Range4ma . . . 1222
11.13.8 ST_StdDev4ma . . . 1223
11.13.9 ST_Sum4ma . . . 1224
11.14 □□□□□□□ . . . 1225
11.14.1 ST_Aspect . . . 1225
11.14.2 ST_HillShade . . . 1228
11.14.3 ST_Roughness . . . 1230
11.14.4 ST_Slope . . . 1230
11.14.5 ST_TPI . . . 1232
11.14.6 ST_TRI . . . 1233
11.14.7 ST_InterpolateRaster . . . 1234
11.14.8 ST_Contour . . . 1235
11.15 □□□□□□□□ . . . 1236
11.15.1 Box3D . . . 1236
11.15.2 ST_ConvexHull . . . 1237
11.15.3 ST_DumpAsPolygons . . . 1238
11.15.4 ST_Envelope . . . 1239
11.15.5 ST_MinConvexHull . . . 1240
11.15.6 ST_Polygon . . . 1242
11.15.7 ST_IntersectionFractions . . . 1244
11.16 □□□□□□ . . . 1245
11.16.1 && . . . 1245
11.16.2 &< . . . 1245
11.16.3 &> . . . 1246
11.16.4 = . . . 1247

11.16.5 @ . . . 1247
11.16.6 ~= . . . 1248
11.16.7 ~ . . . 1249
11.17 □□□□□□□□□□□□□□□ . . . 1249
11.17.1 ST_Contains . . . 1249
11.17.2 ST_ContainsProperly . . . 1250
11.17.3 ST_Covers . . . 1251
11.17.4 ST_CoveredBy . . . 1252
11.17.5 ST_Disjoint . . . 1253
11.17.6 ST_Intersects . . . 1254
11.17.7 ST_Overlaps . . . 1255
11.17.8 ST_Touches . . . 1256
11.17.9 ST_SameAlignment . . . 1257
11.17.10 ST_NotSameAlignmentReason . . . 1258
11.17.11 ST_Within . . . 1259
11.17.12 ST_DWithin . . . 1260
11.17.13 ST_DFullyWithin . . . 1261
11.18 Raster Tips . . . 1262
11.18.1 Out-DB Rasters . . . 1262
11.18.1.1 Directory containing many files . . . 1262
11.18.1.2 Maximum Number of Open Files . . . 1263
11.18.1.2.1 Maximum number of open files for the entire system . . . 1263
11.18.1.2.2 Maximum number of open files per process . . . 1264

12 PostGIS Extras **1266**

13 PostGIS Special Functions Index **1267**
13.1 PostGIS Aggregate Functions . . . 1267
13.2 PostGIS Window Functions . . . 1268
13.3 PostGIS SQL-MM Compliant Functions . . . 1269
13.4 PostGIS Geography Support Functions . . . 1272
13.5 PostGIS Raster Support Functions . . . 1274
13.6 PostGIS Geometry / Geography / Raster Dump Functions . . . 1280
13.7 PostGIS Box Functions . . . 1280
13.8 PostGIS Functions that support 3D . . . 1281
13.9 PostGIS Curved Geometry Support Functions . . . 1287
13.10 PostGIS Polyhedral Surface Support Functions . . . 1291
13.11 PostGIS Function Support Matrix . . . 1294
13.12 New, Enhanced or changed PostGIS Functions . . . 1313

13.12.1 PostGIS Functions new or enhanced in 3.7 . . . 1313
13.12.2 PostGIS Functions new or enhanced in 3.6 . . . 1315
13.12.3 PostGIS Functions new or enhanced in 3.5 . . . 1317
13.12.4 PostGIS Functions new or enhanced in 3.4 . . . 1319
13.12.5 PostGIS Functions new or enhanced in 3.3 . . . 1320
13.12.6 PostGIS Functions new or enhanced in 3.2 . . . 1321
13.12.7 PostGIS Functions new or enhanced in 3.1 . . . 1322
13.12.8 PostGIS Functions new or enhanced in 3.0 . . . 1324
13.12.9 PostGIS Functions new or enhanced in 2.5 . . . 1326
13.12.10 PostGIS Functions new or enhanced in 2.4 . . . 1327
13.12.11 PostGIS Functions new or enhanced in 2.3 . . . 1328
13.12.12 PostGIS Functions new or enhanced in 2.2 . . . 1330
13.12.13 PostGIS Functions new or enhanced in 2.1 . . . 1333
13.12.14 PostGIS Functions new or enhanced in 2.0 . . . 1338
13.12.15 PostGIS Functions new or enhanced in 1.5 . . . 1349
13.12.16 PostGIS Functions new or enhanced in 1.4 . . . 1350
13.12.17 PostGIS Functions new or enhanced in 1.3 . . . 1351

14 Reporting Problems **1352**
14.1 Reporting Software Bugs . . . 1352
14.2 Reporting Documentation Issues . . . 1352

15 Credits and Acknowledgments **1354**
15.1 Project Steering Committee . . . 1354
15.2 Core Contributors Present . . . 1354
15.3 Core Contributors Past . . . 1355
15.4 PostGIS Raster Origins . . . 1355
15.5 Other Contributors . . . 1356

A Release Notes **1360**
A.1 PostGIS 3.7.0 . . . 1360
A.1.1 Breaking Changes . . . 1360
A.1.2 New Features . . . 1360
A.1.3 Enhancements . . . 1361
A.1.4 Bug Fixes . . . 1362

Abstract

PostGIS ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ PostgreSQL ⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠ GIS(⊠⊠⊠⊠⊠⊠⊠) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. PostGIS ⊠ GiST ⊠⊠ R-Tree ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, GIS ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠⊠⊠ 3.8.0dev ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠-⊠⊠⊠⊠⊠⊠⊠⊠ 3.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠ PostGIS ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ https://postgis.net⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

Chapter 1

☒☒

1.1 What Is PostGIS?

PostGIS extends PostgreSQL so that geospatial data can be stored, indexed, queried, and processed in the database. It keeps location data beside ordinary relational data, so a spatial query can use the same SQL, transactions, constraints, access controls, backup tools, and application connections as the rest of a PostgreSQL database.

The main `postgis` extension adds:

- `geometry` for planar coordinate systems and `geography` for geodetic coordinates on the earth.
- Spatial indexes that let the PostgreSQL planner avoid scanning every row when objects can be filtered by location.
- Functions for measurement, relationships, overlays, geometry construction and editing, coordinate transformation, clustering, and spatial data exchange.
- Support for common geospatial encodings and formats, including WKT, WKB, GeoJSON, GML, KML, FlatGeobuf, Geobuf, and Mapbox Vector Tiles.

This makes PostgreSQL a spatial database: applications can ask both ordinary questions and location-aware questions without moving data to a separate GIS processing system. Desktop GIS, map servers, ETL tools, and application frameworks can connect through normal PostgreSQL interfaces.

1.2 What Is in the PostGIS Distribution?

This manual documents the main PostGIS source distribution. Depending on how PostGIS was built and packaged, it can provide several PostgreSQL extensions and command-line tools:

`postgis` The core geometry and geography types, spatial indexes, and most spatial functions.

`postgis_raster` Raster storage, analysis, loading, and export. Raster support uses GDAL and can be omitted from a build.

`postgis_topology` SQL/MM topology types and functions for models based on shared nodes, edges, and faces.

`postgis_sfcgal` Additional 2D and 3D processing functions backed by the optional SFCGAL library.

Loaders and utilities **shp2pgsql**, **pgsql2shp**, and **raster2pgsql** move data between common file formats and PostGIS. Other utilities support upgrades, profiling, and development.

The source tree also contains internal libraries such as `liblwgeom` and `libpgcommon`. They are implementation layers shared by PostGIS components, not separately released PostgreSQL extensions. Most users start by enabling the core extension with `CREATE EXTENSION postgis`; see 2□ for available components and build options.

1.3 The PostGIS Project Family

This manual covers the main `postgis/postgis` distribution. The official PostGIS source organization also hosts or mirrors complementary projects. These projects have their own release cycles, installation instructions, and documentation; installing the core extension does not install them.

- Address Standardizer parses and normalizes postal addresses. It moved from the main PostGIS source tree to a separate repository for the PostGIS 3.7 development cycle.
- PostGIS TIGER Geocoder provides United States address geocoding and reverse geocoding using Census TIGER data. It is also maintained separately from the main distribution.
- H3-pg provides PostgreSQL bindings for the H3 hierarchical hexagonal indexing system and interoperates with PostGIS.
- PostGIS Java provides Java bindings for PostGIS types.
- Introduction to PostGIS is the hands-on workshop maintained by the PostGIS project.

Other PostgreSQL spatial extensions can complement PostGIS while remaining independent projects. For example, pgRouting adds routing and graph analysis. Use each project's own manual for its functions and support policy.

1.4 How to Use This Manual

The manual is both a guide to the PostGIS data model and the complete reference for the SQL interface. Choose a starting point based on what you are trying to do:

If you are new to spatial SQL Start with the Introduction to PostGIS workshop for a tutorial, then use 4□ to understand spatial types, coordinate systems, validity, loading, and indexes.

If you are installing or upgrading PostGIS Read 2□ and 3□. Package-specific instructions and supported release information are maintained on the PostGIS website.

If you are writing spatial queries Read 5□ for spatial relationships and index-aware query patterns, and 6□ for common performance problems.

If you are connecting a desktop GIS Use the client's current database documentation for its user interface and supported geometry types. For example, the QGIS DB Manager lesson covers PostgreSQL connections, table layers, SQL query layers, import, and export. A query layer should return a stable unique identifier and a geometry column with a known SRID and geometry type. Expensive read-only queries can be exposed through a view or materialized view and indexed as appropriate.

If you need a particular function Use 7□ or 13□. Function pages state accepted types, dimensional support, standards conformance, dependency requirements, and version availability.

If you use an optional component See 9□, 10□, 11□, or 8□.

If something goes wrong Collect the versions reported by PostGIS_Full_Version and follow 14□. Check □□ A when behavior differs between release lines.

1.5 Standards Support

This manual marks individual functions as compliant with spatial standards so users can tell which behavior is portable and which behavior is a PostGIS extension. PostGIS implements the OGC Simple Features for SQL model, including the geometry type, Well-Known Text (WKT), Well-Known Binary (WKB), and the common spatial accessors, constructors, predicates, and processing functions. PostGIS also implements many SQL/MM Spatial types and functions.

PostGIS supports geometry families defined by later standards, including SQL/MM curved geometry and OGC Simple Features Access 1.2 PolyhedralSurface, Triangle, and TIN. Support for these geometry families is function-specific, and each function page documents the supported input types and conformance level.

PostGIS also includes widely used extensions and integrations. Examples of PostGIS extensions include Extended WKT (EWKT), Extended WKB (EWKB), SRID metadata in those extended formats, geography, and raster. PostGIS also supports input and output exchange formats defined by other standards, such as GeoJSON, GML, and KML. These features are documented individually and should not be read as OGC Simple Features or SQL/MM requirements unless the specific function or format says so.

When the implementation intentionally differs from a standard, or when a standard leaves behavior ambiguous, the relevant function page describes the PostGIS behavior. 13☒ summarizes SQL/MM compliance and geometry-type support across functions.

1.6 Project and Community

PostGIS began in 2001 as a spatial database research project at Refractions Research. It is now an OSGeo project developed by contributors and organizations around the world. The Project Steering Committee coordinates releases, project policy, and the general direction of development. The complete contributor and sponsor record is in 15☒ at the end of this manual.

The manual is the authoritative reference for the released SQL interface. The project website and source tree provide the surrounding material:

- Documentation routes readers to getting-started instructions, versioned manuals, tips, training material, and the FAQ.
- Community lists mailing lists, chat, events, videos, and other support channels.
- Developer documentation covers development environments, testing, coding style, internals, maintenance, governance, releases, and website work.

Chapter 2

PostGIS ⊠⊠

⊠⊠⊠⊠⊠ PostGIS ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

2.1 ⊠⊠⊠⊠

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠:

Code

```
tar -xvzf postgis-3.8.0dev.tar.gz
cd postgis-3.8.0dev
./configure
make
make install
```

PostGIS ⊠⊠⊠⊠⊠, PostGIS ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (3.3⊠) ⊠⊠⊠⊠⊠ (3.4⊠) ⊠⊠⊠⊠⊠.

2.2 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠

> ⊠⊠
> ⊠⊠⊠⊠ OS ⊠⊠⊠⊠⊠ PostgreSQL/PostGIS ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
> This section includes general compilation instructions. If you are compiling for Windows or another operating system, see the PostGIS getting started guides and the development environment guides.
> Pre-Built Packages for various OS are listed in the PostGIS getting started guides
> ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ Stackbuilder ⊠⊠ PostGIS Windows download site ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠ 1~2 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ very bleeding-edge windows experimental builds ⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ PostGIS ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

The PostGIS module is an extension to the PostgreSQL backend server. As such, PostGIS 3.8.0dev *requires* full PostgreSQL server headers access in order to compile. It can be built against PostgreSQL versions 14 - 19. Earlier versions of PostgreSQL are *not* supported.

Refer to the PostgreSQL installation guides if you haven't already installed PostgreSQL. https://www.postgre
.

⊠⊠

GEOS ⊠⊠⊠⊠⊠⊠ PostgreSQL ⊠⊠⊠⊠⊠ C++ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

Code

```
LDFLAGS=-lstdc++ ./configure [YOUR OPTIONS HERE]
```

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ C++ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠) ⊠⊠⊠⊠⊠ PostgreSQL ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠⊠ PostGIS ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

2.2.1 ⊠⊠⊠⊠

PostGIS ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ https://postgis.net/stuff/postgis-3.8.0dev.tar.gz ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

Code

```
wget https://postgis.net/stuff/postgis-3.8.0dev.tar.gz
tar -xvzf postgis-3.8.0dev.tar.gz
cd postgis-3.8.0dev
```

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ postgis-3.8.0dev (⊠) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠⊠⊠⊠⊠⊠, svn ⊠⊠⊠ http://svn.osgeo.org/postgis/trunk/ ⊠⊠⊠⊠⊠⊠⊠⊠⊠ (checkout) ⊠⊠⊠⊠⊠⊠.

Code

```
git clone https://git.osgeo.org/gitea/postgis/postgis.git postgis
cd postgis
sh autogen.sh
```

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ postgis-3.8.0dev ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

Code

```
./configure
```

2.2.2 ⊠⊠⊠⊠⊠⊠

PostGIS ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠⊠⊠

- PostgreSQL 14 - 19. A complete installation of PostgreSQL (including server headers) is required. PostgreSQL is available from https://www.postgresql.org 18 .

  For current release-line support and end-of-life policy, see https://postgis.net/development/versions_eol/-

- GNU C ⊠⊠⊠⊠ (gcc). PostGIS ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ANSI C ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ gcc ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

- GNU Make(`gmake` ⊠⊠ `make`). ⊠⊠⊠⊠⊠⊠⊠⊠⊠ GNU `make` ⊠ make ⊠⊠⊠⊠⊠⊠⊠⊠⊠. `make -v` ⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠ make ⊠ PostGIS `Makefile` ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

- Proj reprojection library. Proj 6.1 or above is required. The Proj library is used to provide coordinate reprojection support within PostGIS. Proj is available for download from https://proj.org/ .

- GEOS geometry library, version 3.10.0 or greater, but GEOS 3.15+ is required to take full advantage of all the new functions and features. GEOS is available for download from https://libgeos.org .

- LibXML2. LibXML2 is currently used in some imports functions (ST_GeomFromGML and ST_GeomFromKI LibXML2 is available for download from https://gitlab.gnome.org/GNOME/libxml2/-/releases.

- Perl is required to build PostGIS. Perl is available for download from https://www.perl.org/.

- The iconv library is required by the shp2pgsql loader for encoding conversion. On GNU/Linux systems it is typically provided by libc; on other platforms you may need a separate libiconv package.

- The GNU Multiple Precision Arithmetic Library (libgmp). Used to control precision of math operations in PostGIS Topology. GMP is available for download from https://gmplib.org/. It is detected via **pkg-config** and has no fallback, so **pkg-config** must be available for a default build.

- GDAL, version 2.4 or higher, is required for raster support. GDAL 3+ is preferred. https://gdal.org/download.html.

- ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠ PostgreSQL ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠ http://trac.osgeo.org/postgis/ticket/635 ⊠⊠⊠⊠⊠⊠⊠.

⊠⊠⊠⊠

- ⊠ 2.1⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

- JSON-C. JSON-C is currently used to import GeoJSON via the function ST_GeomFromGeoJson. JSON-C is available for download from https://github.com/json-c/json-c/releases/. It is enabled by default but optional; if it is not found configure only warns and the build continues. Disable with `--without-json`.

- GNU gettext for loader and documentation translation support. It may be disabled with `--with-gettext=n` https://www.gnu.org/software/gettext/.

- shapefile ⊠⊠ shp2pgsql-gui ⊠⊠⊠⊠⊠⊠⊠⊠ GTK(GTK+2.0, 2.8+ ⊠⊠). http://www.gtk.org/.

- SFCGAL, 1.3.1 or higher is required, 2.3+ is needed to be able to use all functionality. SFCGAL can be used to provide additional 2D and 3D advanced analysis functions to PostGIS cf 8⊠. https://sfcgal.gitlab.io/sfcgal/.

- To enable ST_AsMVT protobuf-c library 1.1.0 or higher (for usage) and the protoc-c compiler (for building) are required. Also, pkg-config is required to verify the correct minimum version of protobuf-c. See protobuf-c. By default, Postgis will use Wagyu to validate MVT polygons faster which requires a c++11 compiler. It will use CXXFLAGS and the same compiler as the PostgreSQL installation. To disable this and use GEOS instead use the `--without-wagyu` during the configure step.

- CUnit(`CUnit`). ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. http://cunit.sourceforge.net/

- DocBook(`xsltproc`) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. DocBook ⊠ http://www.docbook.org/ ⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠.

- Python 3 is required, when configured with **--with-visual-examples** (off by default), to run the manual example tester that generates the visual examples which the HTML and PDF documentation embed. The examples are executed against a live PostgreSQL server running the staged PostGIS build via **make -C regress visual-examples**; no Python database driver is required, the tooling talks to the server through the **psql** client. Python 3 is available from https://www.python.org/ .

- DBLatex(`dblatex`) ⊠ PDF ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. DBLatex ⊠ http://dblatex.sourceforge.net/- ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- PDF documentation needs an SVG rasterizer for its high-resolution figure fallbacks. The build prefers `rsvg-convert`, and can also use GraphicsMagick (`gm convert`) or ImageMagick (`magick` or `convert`).

2.2.3 ⊠⊠

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ Makefile ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ Makefile ⊠⊠⊠⊠⊠⊠⊠.

Code

```
./configure
```

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ PostGIS ⊠⊠⊠. **./configure** ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠.

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠ **--help** ⊠⊠ **--help=short** ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

`--with-library-minor-version` Starting with PostGIS 3.0, the library files generated by default will no longer have the minor version as part of the file name. This means all PostGIS 3 libs will end in `postgis-3`. This was done to make pg_upgrade easier, with downside that you can only install one version PostGIS 3 series in your server. To get the old behavior of file including the minor version: e.g. `postgis-3.0` add this switch to your configure statement.

`--prefix=PREFIX` This is the installation prefix for PostGIS command-line utilities. By default, utilities are installed in the executable directory reported by the detected PostgreSQL installation.

> ⊠⊠
> PostgreSQL extension files are installed in the directories reported by **pg_config**, so the server can load them. This includes the PostGIS shared library and SQL extension files. Use `--with-pgconfig=FILE` to choose the PostgreSQL installation that PostGIS builds and installs against.

`--with-pgconfig=FILE` PostgreSQL ⊠ PostGIS ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ PostgreSQL ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ **pg_config** ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. PostGIS ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ PostgreSQL ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (**--with-pgconfig=/path/to/pg_config**) ⊠⊠⊠⊠⊠⊠⊠⊠.

`--with-gdalconfig=FILE` GDAL ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, GDAL ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ **gdal-config** ⊠⊠⊠⊠⊠⊠⊠. PostGIS ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ GDAL ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (**--with-gdalconfig=/path/to/gdal-config**) ⊠⊠⊠⊠⊠⊠⊠⊠.

`--with-geosconfig=FILE` GEOS ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, GEOS ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ **geos-config** ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. PostGIS ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ GEOS ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (**--with-geosconfig=/path/to/geos-config**) ⊠⊠⊠⊠⊠⊠⊠.

`--with-xml2config=FILE` LibXML is the library required for doing GeomFromKML/GML processes. It normally is found if you have libxml installed, but if not or you want a specific version used, you'll need to point PostGIS at a specific `xml2-config` confi file to enable software installations to locate the LibXML installation directory. Use this parameter (**>--with-xml2config=/path/to/xml2-config**) to manually specify a particular LibXML installation that PostGIS will build against.

--with-projdir=DIR Proj4 ⊠ PostGIS ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. PostGIS ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ Proj4 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (**--with-projdir=/path/to/projdir**) ⊠⊠⊠⊠⊠⊠⊠⊠.

--with-libiconv=DIR iconv ⊠⊠⊠⊠

--with-jsondir=DIR JSON-C ⊠ MIT-⊠⊠⊠⊠⊠ JSON ⊠⊠⊠⊠⊠⊠, PostGIS ⊠ ST_GeomFromJSON ⊠⊠⊠⊠⊠⊠⊠⊠. PostGIS ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ JSON-C ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (**--with-jsondir=/path/to/jsondir**) ⊠⊠⊠⊠⊠⊠⊠.

--with-gui ⊠⊠⊠⊠⊠⊠ GUI ⊠⊠⊠ (GTK+2.0 ⊠⊠). shp2pgsql-gui ⊠ shp2pgsql ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

--without-raster ⊠⊠⊠⊠⊠⊠⊠

--without-topology Compile without topology support.

--with-gettext=no ⊠⊠⊠⊠⊠ PostGIS ⊠ gettext ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ gettext ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ http://trac.osgeo.org/postgis/ticket/748 ⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠: gettext ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. gettext ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ GUI ⊠⊠⊠⊠⊠⊠⊠⊠⊠/⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

--with-sfcgal=PATH ⊠⊠⊠⊠⊠ PostGIS ⊠⊠⊠⊠⊠⊠⊠⊠⊠ sfcgal ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. `PATH` ⊠ sfcgal-config ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

--without-phony-revision Disable updating postgis_revision.h to match current HEAD of the git repository.

⊠⊠
If you obtained PostGIS from the code repository , the first step is really to run the script

---

Code

```
./autogen.sh
```

⊠⊠⊠⊠⊠⊠ **configure** ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠ PostGIS ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠ tar ⊠⊠⊠⊠⊠ PostGIS ⊠⊠⊠⊠⊠⊠⊠⊠ **configure** ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ **./autogen.sh** ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

2.2.4 ⊠⊠

⊠⊠ Makefile ⊠⊠⊠⊠⊠ PostGIS ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

Code

```
make
```

Output

```
[last line of output]
PostGIS was built successfully. Ready to install.
```

All the functions have comments generated from the documentation. If you wish to install these comments into your spatial databases later, run the command which requires docbook. The postgis_comments.sql and other package comment files are also packaged in the tar.gz distribution in the doc folder so no need to make comments if installing from the tar ball. Comments are also included as part of the CREATE EXTENSION install.

Code

```
make comments
```

The **make cheatsheets** target generates html cheat sheets suitable for quick reference or for student handouts. This requires xsltproc to build and will generate 4 files in doc folder `topology_cheatsheet.html`, `raster_cheatsheet.html`, `postgis_cheatsheet.html`

html ⊠ pdf ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ PostGIS / PostgreSQL Study Guides ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

Code

```
make cheatsheets
```

2.2.5 PostGIS Extensions ⊠⊠⊠⊠⊠

The PostGIS extensions are built and installed automatically when PostgreSQL extension support is available.

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠ function descriptions ⊠⊠⊠⊠⊠⊠⊠⊠⊠. docbook ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠:

Code

```
make comments
```

⊠⊠⊠⊠⊠⊠ tar ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ tar ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ comments ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠ PostgreSQL 9.1 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ extensions ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠ extensions ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

Set PGUSER if you need to override the psql connection user, then test before installing. After installation, run the extension regression tests.

Code

```
cd extensions
cd postgis
make clean
make
export PGUSER=postgres
make check
make install
make check RUNTESTFLAGS=--extension
```

> **Note** ⊠⊠
>
> make check uses psql to run tests and as such can use psql environment variables. Common ones useful to override are PGUSER,PGPORT, and PGHOST. Refer to psql environment variables

extension ⊠⊠⊠ OS ⊠⊠⊠⊠⊠ PostGIS ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠ PostGIS binaries ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠ extension ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ PostgreSQL ⊠⊠⊠⊠⊠ `PostgreSQL / share / extension` ⊠⊠⊠⊠⊠ extensions ⊠⊠⊠⊠⊠⊠⊠⊠⊠ PostGIS ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

- □□□□□□□□□□□□□□□□□□□ extension □□□□□□□□□□□□□□□□□□. `postgis.control, postgis_topology.control`.
- □ extension □ /sql □□□□□□□□□□□□□. □□□□□□ postgreSQL □ share/extension □□□□ □□□□□□□□□□□□□□□□. `extensions/postgis/sql/*.sql`, `extensions/postgis_topology/sql/*.sql`

Once you do that, you should see `postgis`, `postgis_topology` as available extensions in PgAdmin -> extensions.

□□ psql □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□.

Code

```
SELECT name, default_version, installed_version
FROM pg_available_extensions
WHERE name LIKE 'postgis%'
ORDER BY name;
```

Output

```
+-----------------+-----------------+-------------------+
| name            | default_version | installed_version |
+-----------------+-----------------+-------------------+
| postgis         | 3.8.0dev        | 3.8.0dev        |
| postgis_raster  | 3.8.0dev        | 3.8.0dev        |
| postgis_sfcgal  | 3.8.0dev        |                   |
| postgis_topology | 3.8.0dev        |                   |
+-----------------+-----------------+-------------------+
```

□□□□□□□□□□□□□□□□□□ extension □□□□□□□□□, □□□□ `installed_version` □□□□ □□□□□□□□□□. □□□□□□□□□□□□□□□□ postgis extension □□□□□□□□□□□□□□□□□. PgAdmin III 1.14 □□□□□□□□□□□□□□□□□□□ `extensions` □□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□.

extension □□□□□□□□□□□ pgAdmin extension □□□□□□□□□□□ sql □□□□□□□□□□□□□□ □□□□□□□□ postgis extension □□□□□□□□□□:

Code

```
CREATE EXTENSION postgis;
CREATE EXTENSION postgis_raster;
CREATE EXTENSION postgis_sfcgal;
CREATE EXTENSION postgis_topology;
```

PSQL □□□□□□□□□□□□□□□□□□□□□□□□□□□, □□□□□□□□□□□□□□□□□□□.

Code

```
\connect mygisdb
\x
\dx postgis*
```

Output

```
List of installed extensions
+-[ RECORD 1 ]-------------------------------------------+
| Name        | postgis                                  |
| Version     | 3.8.0dev                                   |
| Schema      | public                                   |
| Description | PostGIS geometry, geography, and raster...  |
+-[ RECORD 2 ]-------------------------------------------+
| Name        | postgis_raster                           |
| Version     | 3.8.0dev                                   |
| Schema      | public                                   |
| Description | PostGIS raster types and functions       |
+-[ RECORD 3 ]-------------------------------------------+
| Name        | postgis_topology                         |
```

```
| Version     | 3.8.0dev                                |
| Schema      | topology                              |
| Description | PostGIS topology spatial types and functions  |
+-------------------------------------------------------------+
```

⊠⊠

Extension tables `spatial_ref_sys`, `layer`, `topology` can not be explicitly backed up. They can only be backed up when the respective `postgis` or `postgis_topology` extension is backed up, which only happens when you back up the whole database. Only srid records not packaged with PostGIS are captured in backups, so don't change the entries we ship and expect the modifications to persist. Put in a ticket if you find an issue. The structures of extension tables are never backed up since they are created with `CREATE EXTENSION` and assumed to be the same for a given version of an extension. These behaviors are built into the current PostgreSQL extension model.

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ 3.8.0dev ⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠: `postgis_upgrade_22_minor.sql`,`raster_upgrade_22_minor.sql`,`topology_upgrade_22_minor.sql`.

Code

```
CREATE EXTENSION postgis FROM unpackaged;
CREATE EXTENSION postgis_raster FROM unpackaged;
CREATE EXTENSION postgis_topology FROM unpackaged;
```

2.2.6 ⊠⊠⊠

⊠⊠ PostGIS ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠.

Code

```
make check
```

⊠⊠⊠⊠⊠⊠⊠ PostgreSQL ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠.

⊠⊠

PostgreSQL, GEOS, ⊠⊠ Proj4 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, LD_LIBRARY_PATH ⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

⊠⊠, **make check** ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ PATH ⊠ PGPORT ⊠⊠⊠⊠⊠⊠⊠⊠⊠. PostgreSQL ⊠⊠⊠ ⊠⊠⊠⊠⊠ **--with-pgconfig** ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠. ⊠⊠⊠, PostgreSQL ⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ PATH ⊠⊠⊠⊠⊠⊠⊠⊠.



□□
Sandboxed build accounts that are not PostgreSQL superusers can delegate database ownership during the regression cycle by exporting POSTGIS_REGRESS_DB_OWNER. The harness will create the temporary regression database owned by the nominated role while continuing to connect using the less privileged account. Combine this with POSTGIS_REGRESS_ROLE_EXT_CREATOR when the extension creation role must differ from the database owner.
These variables allow automated environments to exercise the full upgrade and extension install paths without promoting the calling account to superuser, provided the target PostgreSQL instance permits extension installation by those delegate roles.

If successful, make check will produce the output of almost 500 tests. The results will look similar to the following (numerous lines omitted below):

Code

```
CUnit - A unit testing framework for C - Version 2.1-3
     http://cunit.sourceforge.net/

  .
  .
  .

Run Summary:    Type  Total    Ran Passed Failed Inactive
              suites     44     44    n/a      0        0
               tests    300    300    300      0        0
             asserts   4215   4215   4215      0      n/a
Elapsed time =    0.229 seconds

  .
  .
  .

Running tests

  .
  .
  .

Run tests: 134
Failed: 0


-- if you build with SFCGAL

  .
  .
  .

Running tests

  .
  .
  .

Run tests: 13
Failed: 0

-- if you built with raster support
```

```
  .
  .
  .

Run Summary:    Type  Total    Ran Passed Failed Inactive
              suites     12     12    n/a      0        0
               tests     65     65     65      0        0
             asserts  45896  45896  45896      0      n/a


  .
  .
  .

Running tests

  .
  .
  .

Run tests: 101
Failed: 0

-- topology regress

.
.
.

Running tests

  .
  .
  .

Run tests: 51
Failed: 0

-- if you built --with-gui, you should see this too

     CUnit - A unit testing framework for C - Version 2.1-2
     http://cunit.sourceforge.net/

  .
  .
  .

Run Summary:    Type  Total    Ran Passed Failed Inactive
              suites      2      2    n/a      0        0
               tests      4      4      4      0        0
             asserts      4      4      4      0      n/a
```

□□□□□□□□□□□□□□□□:

Output

```
============== dropping database "contrib_regression" ==============
DROP DATABASE
============== creating database "contrib_regression" ==============
CREATE DATABASE
ALTER DATABASE
============== installing fuzzystrmatch               ==============
CREATE EXTENSION
============== installing postgis                     ==============
```

```
CREATE EXTENSION
============== running regression test queries        ==============
test test-normalize_address   ... ok

=====================
All 2 tests passed.
=====================
```

2.2.7 ⊠⊠

PostGIS ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

Code

```
make install
```

⊠⊠⊠ **--prefix** ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ PostGIS ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

- ⊠⊠ (loader) ⊠⊠⊠⊠⊠⊠⊠⊠⊠ `[prefix]/bin` ⊠⊠⊠⊠⊠⊠.
- `postgis.sql` ⊠⊠⊠ SQL ⊠⊠⊠⊠ `[prefix]/share/contrib` ⊠⊠⊠⊠⊠⊠.
- PostGIS ⊠⊠⊠⊠⊠⊠⊠ `[prefix]/lib` ⊠⊠⊠⊠⊠⊠.

⊠⊠⊠⊠⊠ `postgis_comments.sql`, `raster_comments.sql` ⊠⊠⊠⊠⊠⊠⊠⊠⊠ **make comments** ⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠ sql ⊠⊠⊠⊠⊠⊠⊠⊠⊠.

Code

```
make comments-install
```

> **Note!**
>
> ⊠⊠
>
> **xsltproc** ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ `postgis_comments.sql`, `raster_comments.sql`, `topology_comments.sql` ⊠⊠⊠⊠⊠⊠⊠⊠.

2.3 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

PostgreSQL version mismatch.

PostgreSQL 14 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠ PostgreSQL ⊠⊠⊠⊠ ⊠⊠⊠ PostgreSQL ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. (Linux) ⊠⊠⊠⊠⊠⊠ PostgreSQL ⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. PostGIS ⊠ PostgreSQL 14 ⊠⊠. ⊠⊠ PostgreSQL ⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ psql ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠:

Code

```sql
SELECT version();
```

If you are running an RPM based distribution, you can check for the existence of pre-installed packages using the **rpm** command as follows:

Code

```
rpm -qa | grep postgresql
```

Restoring an upgrade into a database without PostGIS.

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ PostGIS ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

Code

```
SELECT PostGIS_Full_Version();
```

⊠⊠ PostreSQL, Proj4 ⊠⊠⊠⊠⊠⊠ GEOS ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠⊠ `postgis_config.hh` ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. `POSTGIS_PGSQL_VERSION`, `POSTGIS_PROJ_VERSION` and `POSTGIS_GEOS_VERSION` ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

Chapter 3

PostGIS Administration

3.1 Performance Tuning

Tuning for PostGIS performance is much like tuning for any PostgreSQL workload. The only additional consideration is that geometries and rasters are usually large, so memory-related optimizations generally have more of an impact on PostGIS than other types of PostgreSQL queries.

For general details about optimizing PostgreSQL, refer to Tuning your PostgreSQL Server.

PostgreSQL configuration can be set at the server level without touching `postgresql.conf` or `postgresql.a` by using the `ALTER SYSTEM` command.

Set `work_mem`, reload the configuration so new connections use the change, and display the current value. Use **SHOW ALL** to list all settings.

Code

```
ALTER SYSTEM SET work_mem = '256MB';
SELECT pg_reload_conf();
SHOW work_mem;
```

In addition to the Postgres settings, PostGIS has some custom settings which are listed in 7.22□.

3.1.1 Startup

These settings are configured in `postgresql.conf`:

constraint_exclusion

- Default: partition
- This is generally used for table partitioning. The default of ”partition” forces the planner to only analyze tables for constraint consideration if they are in an inherited hierarchy and avoids paying the planner penalty otherwise.

shared_buffers

- Default: ~128MB
- Set to about 25% to 40% of available RAM. On windows you may not be able to set as high.

max_worker_processes controls the maximum number of background processes available to the server and therefore also bounds how many workers can participate in parallel queries.

- Default: 8
- Sets the maximum number of background processes that the system can support. This parameter can only be set at server start.

3.1.2 Runtime

work_mem - sets the size of memory used for sort operations and complex queries

- Default: 1-4MB
- Adjust up for large dbs, complex queries, lots of RAM
- Adjust down for many concurrent users or low RAM.
- If you have lots of RAM and few developers:

Code

```
SET work_mem TO '256MB';
```

maintenance_work_mem - the memory size used for VACUUM, CREATE INDEX, etc.

- Default: 16-64MB
- Generally too low - ties up I/O, locks objects while swapping memory
- Recommend 32MB to 1GB on production servers w/lots of RAM, but depends on the # of concurrent users. If you have lots of RAM and few developers:

Code

```
SET maintenance_work_mem TO '1GB';
```

max_parallel_workers_per_gather

Parallel query support in modern PostgreSQL and PostGIS uses this setting to decide how many workers a plan may request. If set to higher than 0, then some queries such as those involving relation functions like `ST_Intersects` can use multiple processes and can run more than twice as fast when doing so. If you have a lot of processors to spare, you should change the value of this to as many processors as you have. Also make sure to bump up `max_worker_processes` to at least as high as this number.

- Default: 0
- Sets the maximum number of workers that can be started by a single `Gather` node. Parallel workers are taken from the pool of processes established by `max_worker_processes`. Note that the requested number of workers may not actually be available at run time. If this occurs, the plan will run with fewer workers than expected, which may be inefficient. Setting this value to 0, which is the default, disables parallel query execution.

3.2 Configuring raster support

If you enabled raster support you may want to read below how to properly configure it.

Out-of-db rasters and all raster drivers are disabled by default. In order to re-enable these, you need to set the following environment variables `POSTGIS_GDAL_ENABLED_DRIVERS` and `POSTGIS_ENABLE_OUTDB_RAST` in the server environment. You can also use the more cross-platform approach of setting the corresponding 7.22☒.

If you want to enable offline raster:

Code

```
POSTGIS_ENABLE_OUTDB_RASTERS=1
```

Any other setting or no setting at all will disable out of db rasters.

In order to enable all GDAL drivers available in your GDAL install, set this environment variable as follows

Code

```
POSTGIS_GDAL_ENABLED_DRIVERS=ENABLE_ALL
```

If you want to only enable specific drivers, set your environment variable as follows:

Code

```
POSTGIS_GDAL_ENABLED_DRIVERS="GTiff PNG JPEG GIF XYZ"
```



☒☒
If you are on windows, do not quote the driver list

Setting environment variables varies depending on OS. For PostgreSQL installed on Ubuntu or Debian via apt-postgresql, the preferred way is to edit `/etc/postgresql/`*`MAJOR`*`/`*`CLUSTER`*`/environment` where the placeholders refer to the PostgreSQL major version and cluster name.

On windows, if you are running as a service, you can set via System variables which for Windows 7 you can get to by right-clicking on Computer->Properties Advanced System Settings or in explorer navigating to `Control Panel\All Control Panel Items\System`. Then clicking *Advanced System Settings ->Advanced->Environment Variables* and adding new system variables.

After you set the environment variables, you'll need to restart your PostgreSQL service for the changes to take effect.

3.3 ☒☒☒☒☒☒☒☒☒

3.3.1 Spatially enable database using EXTENSION

If you have compiled and installed the extensions/postgis modules, you can turn a database into a spatial one using the EXTENSION mechanism.

Core postgis extension includes geometry, geography, spatial_ref_sys and all the functions and comments. Raster and topology are packaged as a separate extension.

Run the following SQL snippet in the database you want to enable spatially:

Code

```
CREATE EXTENSION IF NOT EXISTS plpgsql;
CREATE EXTENSION postgis;
CREATE EXTENSION postgis_raster; -- OPTIONAL
CREATE EXTENSION postgis_topology; -- OPTIONAL
```

3.3.2 Spatially enable database without using EXTENSION (discouraged)



□□
This is generally only needed if you cannot or don't want to get PostGIS installed in the PostgreSQL extension directory (for example during testing, development or in a restricted environment).

Adding PostGIS objects and function definitions into your database is done by loading the various sql files located in `[prefix]/share/contrib` as specified during the build phase.

The core PostGIS objects (geometry and geography types, and their support functions) are in the `postgis.sql` script. Raster objects are in the `rtpostgis.sql` script. Topology objects are in the `topology.sql` script.

For a complete set of EPSG coordinate system definition identifiers, you can also load the `spatial_ref_sys.sql` definitions file and populate the `spatial_ref_sys` table. This will permit you to perform ST_Transform() operations on geometries.

If you wish to add comments to the PostGIS functions, you can find them in the `postgis_comments.sql` script. Comments can be viewed by simply typing **\dd [function_name]** from a **psql** terminal window.

Run the following Shell commands in your terminal:

Code

```
DB=[yourdatabase]
SCRIPTSDIR=`pg_config --sharedir`/contrib/postgis-3.6/

# Core objects
psql -d ${DB} -f ${SCRIPTSDIR}/postgis.sql
psql -d ${DB} -f ${SCRIPTSDIR}/spatial_ref_sys.sql
psql -d ${DB} -f ${SCRIPTSDIR}/postgis_comments.sql # OPTIONAL

# Raster support (OPTIONAL)
psql -d ${DB} -f ${SCRIPTSDIR}/rtpostgis.sql
psql -d ${DB} -f ${SCRIPTSDIR}/raster_comments.sql # OPTIONAL

# Topology support (OPTIONAL)
psql -d ${DB} -f ${SCRIPTSDIR}/topology.sql
psql -d ${DB} -f ${SCRIPTSDIR}/topology_comments.sql # OPTIONAL
```

3.4 Upgrading spatial databases

Upgrading existing spatial databases can be tricky as it requires replacement or introduction of new PostGIS object definitions.

Unfortunately not all definitions can be easily replaced in a live database, so sometimes your best bet is a dump/reload process.

PostGIS provides a SOFT UPGRADE procedure for minor or bugfix releases, and a HARD UPGRADE procedure for major releases.

Before attempting to upgrade PostGIS, it is always worth to backup your data. If you use the -Fc flag to pg_dump you will always be able to restore the dump with a HARD UPGRADE.

3.4.1 Soft upgrade

If you installed your database using extensions, you'll need to upgrade using the extension model as well. If you installed using the old sql script way, you are advised to switch your install to extensions because the script way is no longer supported.

3.4.1.1 Soft upgrade using extensions

If you originally installed PostGIS with extensions, then you need to upgrade using extensions as well. Doing a minor upgrade with extensions is fairly painless.

Use the PostGIS_Extensions_Upgrade function to upgrade to the latest version you have installed.

Code

```
SELECT postgis_extensions_upgrade();
```

If you are upgrading a database that still runs PostGIS 2.5, issue an **ALTER EXTENSION** followed by `postgis_extensions_upgrade()` to make sure the legacy `postgis_raster` objects are folded back into the main extension.

Code

```
ALTER EXTENSION postgis UPDATE;
SELECT postgis_extensions_upgrade();
```

This second call is needed to rebundle the `postgis_raster` extension.

Code

```
SELECT postgis_extensions_upgrade();
```

If you have multiple versions of PostGIS installed, and you don't want to upgrade to the latest, you can explicitly specify the version as follows:

Code

```
ALTER EXTENSION postgis UPDATE TO "3.8.0dev";
ALTER EXTENSION postgis_topology UPDATE TO "3.8.0dev";
```

If you get an error notice something like:

Output

```
No migration path defined for b''…b'' to 3.8.0dev
```

Then you'll need to backup your database, create a fresh one as described in 3.3.1☒ and then restore your backup on top of this new database.

If you get a notice message like:

Output

```
Version "3.8.0dev" of extension "postgis" is already installed
```

Then everything is already up to date and you can safely ignore it. **UNLESS** you're attempting to upgrade from an development version to the next (which doesn't get a new version number); in that case you can append "next" to the version string, and next time you'll need to drop the "next" suffix again:

Code

```
ALTER EXTENSION postgis UPDATE TO "3.8.0devnext";
        ALTER EXTENSION postgis_topology UPDATE TO "3.8.0devnext";
```



☒☒

If you installed PostGIS originally without a version specified, you can often skip the reinstallation of postgis extension before restoring since the backup just has `CREATE EXTENSION postgis` and thus picks up the newest latest version during restore.



☒☒

If you are upgrading PostGIS extension from a version prior to 3.0.0, you will have a new extension *postgis_raster* which you can safely drop, if you don't need raster support.

Code

```
DROP EXTENSION postgis_raster;
```

3.4.1.2 Soft upgrade without extensions

Extensions have been the supported installation method for many releases. If you still rely on an unpackaged installation, perform a HARD UPGRADE into a database created with extensions and migrate your data there.

3.4.1.3 Hard upgrade

By HARD UPGRADE we mean full dump/reload of postgis-enabled databases. You need a HARD UPGRADE when PostGIS objects' internal storage changes or when SOFT UPGRADE is not possible. The Release Notes appendix reports for each version whether you need a dump/reload (HARD UPGRADE) to upgrade.

The dump/reload process is assisted by the **postgis_restore** script which takes care of skipping from the dump all definitions which belong to PostGIS (including old ones), allowing you to restore your schemas and data into a database with PostGIS installed without getting duplicate symbol errors or bringing forward deprecated objects.

Supplementary instructions for windows users are available in the PostGIS Windows installation guide.

The Procedure is as follows:

1. Create a "custom-format" dump of the database you want to upgrade (let's call it `olddb`) include binary blobs (-b) and verbose (-v) output. The user can be the owner of the db, need not be postgres super account.

   Code

   ```
   pg_dump -h localhost -p 5432 -U postgres -Fc -b -v -f "/somepath/olddb.backup" olddb
   ```

2. Do a fresh install of PostGIS in a new database -- we'll refer to this database as `newdb`. Please refer to 3.3.2☒ and 3.3.1☒ for instructions on how to do this.

   The spatial_ref_sys entries found in your dump will be restored, but they will not override existing ones in spatial_ref_sys. This is to ensure that fixes in the official set will be properly propagated to restored databases. If for any reason you really want your own overrides of standard entries just don't load the spatial_ref_sys.sql file when creating the new db.

   If your database is really old or you know you've been using long deprecated functions in your views and functions, you might need to load `legacy.sql` for all your functions and views etc. to properly come back. Only do this if _really_ needed. Consider upgrading your views and functions before dumping instead, if possible. The deprecated functions can be later removed by loading `uninstall_legacy.sql`.

3. Restore your backup into your fresh `newdb` database using postgis_restore. Unexpected errors, if any, will be printed to the standard error stream by psql. Keep a log of those.

   Code

   ```
   postgis_restore "/somepath/olddb.backup" | psql -h localhost -p 5432 -U postgres newdb ↩
       2> errors.txt
   ```

Errors may arise in the following cases:

1. Some of your views or functions make use of deprecated PostGIS objects. In order to fix this you may try loading `legacy.sql` script prior to restore or you'll have to restore to a version of PostGIS which still contains those objects and try a migration again after porting your code. If the `legacy.sql` way works for you, don't forget to fix your code to stop using deprecated functions and drop them loading `uninstall_legacy.sql`.

2. Some custom records of spatial_ref_sys in dump file have an invalid SRID value. Valid SRID values are bigger than 0 and smaller than 999000. Values in the 999000.999999 range are reserved for internal use while values > 999999 can't be used at all. All your custom records with invalid SRIDs will be retained, with those > 999999 moved into the reserved range, but the spatial_ref_sys table would lose a check constraint guarding for that invariant to hold and possibly also its primary key (when multiple invalid SRIDS get converted to the same reserved SRID value).

   In order to fix this you should copy your custom SRS to a SRID with a valid value (maybe in the 910000..910999 range), convert all your tables to the new srid (see UpdateGeometrySRID), delete the invalid entry from spatial_ref_sys and re-construct the check(s) with:

   Code

   ```
   ALTER TABLE spatial_ref_sys ADD CONSTRAINT spatial_ref_sys_srid_check check (srid
   > 0 AND srid < 999000 );
   ```

   Code

   ```
   ALTER TABLE spatial_ref_sys ADD PRIMARY KEY(srid));
   ```

   If you are upgrading an old database containing french IGN cartography, you will have probably SRIDs out of range and you will see, when importing your database, issues like this :

   Code

   ```
   WARNING: SRID 310642222 converted to 999175 (in reserved zone)
   ```

   In this case, you can try following steps : first throw out completely the IGN from the sql which is resulting from postgis_restore. So, after having run :

Code

```
postgis_restore "/somepath/olddb.backup" > olddb.sql
```

run this command :

Code

```
grep -v IGNF olddb.sql > olddb-without-IGN.sql
```

Create your newdb, activate the required PostGIS extensions, and insert the French IGN spatial reference definitions using this script. After these operations, import your data:

Code

```
psql -h localhost -p 5432 -U postgres -d newdb -f olddb-without-IGN.sql  2> errors.txt
```

Chapter 4

Data Management

4.1 GIS (□□) □□□□□

4.1.1 OGC Geometry

The Open Geospatial Consortium (OGC) developed the *Simple Features Access* standard (SFA) to provide a model for geospatial data. It defines the fundamental spatial type of **Geometry**, along with operations which manipulate and transform geometry values to perform spatial analysis tasks. PostGIS implements the OGC Geometry model as the PostgreSQL data types geometry and geography.

Geometry is an *abstract* type. Geometry values belong to one of its *concrete* subtypes which represent various kinds and dimensions of geometric shapes. These include the **atomic** types Point, LineString, LinearRing and Polygon, and the **collection** types MultiPoint, MultiLineString, MultiPolygon and GeometryCollection. The *Simple Features Access - Part 1: Common architecture v1.2.1* adds subtypes for the structures PolyhedralSurface, Triangle and TIN.

Geometry models shapes in the 2-dimensional Cartesian plane. The PolyhedralSurface, Triangle, and TIN types can also represent shapes in 3-dimensional space. The size and location of shapes are specified by their **coordinates**. Each coordinate has a X and Y **ordinate** value determining its location in the plane. Shapes are constructed from points or line segments, with points specified by a single coordinate, and line segments by two coordinates.

Coordinates may contain optional Z and M ordinate values. The Z ordinate is often used to represent elevation. The M ordinate contains a measure value, which may represent time or distance. If Z or M values are present in a geometry value, they must be defined for each point in the geometry. If a geometry has Z or M ordinates the **coordinate dimension** is 3D; if it has both Z and M the coordinate dimension is 4D. The coordinate dimension is at least 2, because every geometry has at least X and Y coordinates.

Geometry values are associated with a **spatial reference system** indicating the coordinate system in which it is embedded. The spatial reference system is identified by the geometry SRID number. The units of the X and Y axes are determined by the spatial reference system. In **planar** reference systems the X and Y coordinates typically represent easting and northing, while in **geodetic** systems they represent longitude and latitude. SRID 0 represents an infinite Cartesian plane with no units assigned to its axes. See 4.5□.

The geometry **dimension** is a property of geometry types. Point types have dimension 0, linear types have dimension 1, polygonal types have dimension 2, and solid types have dimension 3. Collections have the dimension of the maximum element dimension.

A geometry value may be **empty**. Empty values contain no vertices (for atomic geometry types) or no elements (for collections).

An important property of geometry values is their spatial **extent** or **bounding box**, which the OGC model calls **envelope**. This is the 2 or 3-dimensional box which encloses the coordinates of a geometry. It is an efficient way to represent a geometry's extent in coordinate space and to check whether two geometries interact.

The geometry model allows evaluating topological spatial relationships as described in 5.1.1□. To support this the concepts of **interior**, **boundary** and **exterior** are defined for each geometry type. Geometries are topologically closed, so they always contain their boundary. The boundary is a geometry of dimension one less than that of the geometry itself. For example, the boundary of a Polygon is its rings (LineStrings), the boundary of a LineString is its endpoints (Points), and the boundary of a Point is empty.

The OGC geometry model defines validity rules for each geometry type. These rules ensure that geometry values represents realistic situations (e.g. it is possible to specify a polygon with a hole lying outside its shell, but this makes no sense geometrically and is thus invalid). PostGIS also allows storing and manipulating invalid geometry values. This allows detecting and fixing them if needed. See 4.4□

4.1.1.1 Point

A Point is a 0-dimensional geometry that represents a single location in coordinate space.

Code

```
POINT (1 2)
POINT Z (1 2 3)
POINT ZM (1 2 3 4)
```

4.1.1.2 LineString

A LineString is a 1-dimensional line formed by a contiguous sequence of line segments. Each line segment is defined by two points, with the end point of one segment forming the start point of the next segment. An OGC-valid LineString has either zero or two or more points, but PostGIS also allows single-point LineStrings. LineStrings may cross themselves (self-intersect). A LineString is **closed** if the start and end points are the same. A LineString is **simple** if it does not self-intersect.

Code

```
LINESTRING (1 2, 3 4, 5 6)
```

4.1.1.3 LinearRing

A LinearRing is a LineString which is both closed and simple. The first and last points must be equal, and the line must not self-intersect.

Code

```
LINEARRING (0 0 0, 4 0 0, 4 4 0, 0 4 0, 0 0 0)
```

4.1.1.4 Polygon

A Polygon is a 2-dimensional planar region, delimited by an exterior boundary (the shell) and zero or more interior boundaries (holes). Each boundary is a LinearRing.

Code

```
POLYGON ((0 0 0, 4 0 0, 4 4 0, 0 4 0, 0 0 0), (1 1 0, 2 1 0, 2 2 0, 1 2 0, 1 1 0))
```

4.1.1.5 MultiPoint

A MultiPoint is a collection of Points.

Code

```
MULTIPOINT ( (0 0), (1 2) )
```

4.1.1.6 MultiLineString

A MultiLineString is a collection of LineStrings. A MultiLineString is closed if each of its elements is closed.

Code

```
MULTILINESTRING ( (0 0, 1 1, 1 2), (2 3, 3 2, 5 4) )
```

4.1.1.7 MultiPolygon

A MultiPolygon is a collection of non-overlapping, non-adjacent Polygons. Polygons in the collection may touch only at a finite number of points. (Two polygons are adjacent if they share an edge. They touch if they share only points or edges on their boundaries). For more details on MultiPolygon validity, see 4.4☒.

Code

```
MULTIPOLYGON (((1 5, 5 5, 5 1, 1 1, 1 5)), ((6 5, 9 1, 6 1, 6 5)))
```

4.1.1.8 GeometryCollection

A GeometryCollection is a heterogeneous (mixed) collection of geometries.

Code

```
GEOMETRYCOLLECTION ( POINT(2 3), LINESTRING(2 3, 3 4))
```

4.1.1.9 PolyhedralSurface

A PolyhedralSurface is a contiguous collection of patches or facets which share some edges. Each patch is a planar Polygon. If the Polygon coordinates have Z ordinates then the surface is 3-dimensional.

Code

```
POLYHEDRALSURFACE Z (
((0 0 0, 0 0 1, 0 1 1, 0 1 0, 0 0 0)),
((0 0 0, 0 1 0, 1 1 0, 1 0 0, 0 0 0)),
((0 0 0, 1 0 0, 1 0 1, 0 0 1, 0 0 0)),
((1 1 0, 1 1 1, 1 0 1, 1 0 0, 1 1 0)),
((0 1 0, 0 1 1, 1 1 1, 1 1 0, 0 1 0)),
((0 0 1, 1 0 1, 1 1 1, 0 1 1, 0 0 1)) )
```

4.1.1.10 Triangle

A Triangle is a polygon defined by three distinct non-collinear vertices. Because a Triangle is a polygon it is specified by four coordinates, with the first and fourth being equal.

Code

```
TRIANGLE ((0 0, 0 9, 9 0, 0 0))
```

4.1.1.11 TIN

A TIN is a collection of non-overlapping Triangles representing a Triangulated Irregular Network.

Code

```
TIN Z ( ((0 0 0, 0 0 1, 0 1 0, 0 0 0)), ((0 0 0, 0 1 0, 1 1 0, 0 0 0)) )
```

4.1.2 SQL-MM Part 3

The *ISO/IEC 13249-3 SQL Multimedia - Spatial* standard (SQL/MM) extends the OGC SFA to define Geometry subtypes containing curves with circular arcs. The SQL/MM types support 3DM, 3DZ and 4D coordinates.

> **Note!** ⊠⊠
> SQL-MM ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠ 1E-8 ⊠⊠⊠.

4.1.2.1 CircularString

CIRCULARSTRING ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠ LINESTRING ⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (⊠⊠⊠⊠⊠⊠⊠⊠) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, LINESTRING ⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ 1 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

Code

```
CIRCULARSTRING(0 0, 1 1, 1 0)

CIRCULARSTRING(0 0, 4 0, 4 4, 0 4, 0 0)
```

4.1.2.2 CompoundCurve

⊠⊠⊠⊠ (compound curve) ⊠⊠⊠⊠ (⊠⊠) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, (⊠⊠⊠⊠⊠⊠⊠⊠⊠) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

Code

```
COMPOUNDCURVE( CIRCULARSTRING(0 0, 1 1, 1 0), (1 0, 0 1))
```

4.1.2.3 CurvePolygon

CURVEPOLYGON ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠ 0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

PostGIS ⊠ 1.4 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

Code

```
CURVEPOLYGON(
CIRCULARSTRING(0 0, 4 0, 4 4, 0 4, 0 0),
(1 1, 3 3, 3 1, 1 1) )
```

Example: A CurvePolygon with the shell defined by a CompoundCurve containing a CircularString and a LineString, and a hole defined by a CircularString

Code

```
CURVEPOLYGON(
COMPOUNDCURVE( CIRCULARSTRING(0 0, 2 0, 2 1, 2 3, 4 3),
               (4 3, 4 5, 1 4, 0 0)),
CIRCULARSTRING(1.7 1, 1.4 0.4, 1.6 0.4, 1.6 0.5, 1.7 1) )
```

4.1.2.4 MultiCurve

MULTICURVE ⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

Code

```
MULTICURVE( (0 0, 5 5), CIRCULARSTRING(4 0, 4 4, 8 4))
```

4.1.2.5 MultiSurface

MULTISURFACE ⊠⊠⊠⊠⊠⊠, (⊠⊠) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

Code

```
MULTISURFACE(
CURVEPOLYGON(
  CIRCULARSTRING( 0 0, 4 0, 4 4, 0 4, 0 0),
  (1 1, 3 3, 3 1, 1 1)),
((10 10, 14 12, 11 10, 10 10), (11 11, 11.5 11, 11 11.5, 11 11)))
```

4.1.3 OpenGIS WKB ⊠ WKT

OpenGIS ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠: Well-Known Text (WKT) ⊠⊠⊠ Well-Known Binary (WKB) ⊠⊠. WKT ⊠ WKB ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠⊠⊠⊠⊠⊠⊠ WKT(Well-Known Text) ⊠⊠⊠⊠⊠⊠. ⊠⊠⊠ WKT SRS ⊠⊠⊠⊠⊠⊠⊠⊠:

- POINT(0 0)
- POINT(0 0)
- POINT(0 0)
- POINT EMPTY
- LINESTRING(0 0,1 1,1 2)
- LINESTRING
- POLYGON((0 0,4 0,4 4,0 4,0 0),(1 1, 2 1, 2 2, 1 2,1 1))
- MULTIPOINT((0 0),(1 2))
- MULTIPOINT((0 0),(1 2))
- MULTIPOINT
- MULTILINESTRING((0 0,1 1,1 2),(2 3,3 2,5 4))
- MULTIPOLYGON(((0 0,4 0,4 4,0 4,0 0),(1 1,2 1,2 2,1 2,1 1)), ((-1 -1,-1 -2,-2 -2,-2 -1,-1 -1)))
- GEOMETRYCOLLECTION(POINT(2 3), LINESTRING(2 3,3 4))
- GEOMETRYCOLLECTION

Input and output of WKT is provided by the functions ST_AsText and ST_GeomFromText:

Code

```
text WKT = geometry;
geometry = ST_GeomFromText(text WKT, SRID);
```

For example, a statement to create and insert a spatial object from WKT with SRID 312 is:

Code

```
INSERT INTO geotable ( geom, name )
VALUES ( ST_GeomFromText('POINT(-126.4 45.32)', 312), 'A Place');
```

Well-Known Binary (WKB) provides a portable, full-precision representation of spatial data as binary data (arrays of bytes). Examples of the WKB representations of spatial objects are:

- POINT(0 0)

  WKB: 0101000000000000000000F03F000000000000F03
- LINESTRING(0 0,1 1,1 2)

  WKB: 010200000002000000000000000000000000400000000000000040000000000000022400000000000000002

Input and output of WKB is provided by the functions ST_AsBinary and ST_GeomFromWKB:

Code

```
bytea WKB = ST_AsBinary(geometry);
geometry = ST_GeomFromWKB(bytea WKB, SRID);
```

□□□□ OGC □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□:

Code

```
INSERT INTO geotable ( geom, name )
VALUES ( ST_GeomFromWKB('\x0101000000000000000000f03f000000000000f03f', 312), 'A Place');
```

4.2 Geometry Data Type

PostGIS implements the OGC Simple Features model by defining a PostgreSQL data type called `geometry`. It represents all of the geometry subtypes by using an internal type code (see GeometryType and ST_GeometryType). This allows modelling spatial features as rows of tables defined with a column of type `geometry`.

The `geometry` data type is *opaque*, which means that all access is done via invoking functions on geometry values. Functions allow creating geometry objects, accessing or updating all internal fields, and computing new geometry values. PostGIS supports all the functions specified in the OGC *Simple feature access - Part 2: SQL option* (SFS) specification, as well many others. See 7□ for the full list of functions.



□□
PostGIS follows the SFA standard by prefixing spatial functions with "ST_". This was intended to stand for "Spatial and Temporal", but the temporal part of the standard was never developed. Instead it can be interpreted as "Spatial Type".

OpenGIS □□□□□□□□□□□□□□□□□□□□□□□□□□□□□ (SRID) □□□□□□□□□□□□. □□□□ □□□□□□□□□□□□□□□ SRID □□□□□□□.

To make querying geometry efficient PostGIS defines various kinds of spatial indexes, and spatial operators to use them. See 4.9□ and 5.2□ for details.

4.2.1 OpenGIS WKB □ WKT

OGC SFA specifications initially supported only 2D geometries, and the geometry SRID is not included in the input/output representations. The OGC SFA specification 1.2.1 (which aligns with the ISO 19125 standard) adds support for 3D (ZYZ) and measured (XYM and XYZM) coordinates, but still does not include the SRID value.

Because of these limitations PostGIS defined extended EWKB and EWKT formats. They provide 3D (XYZ and XYM) and 4D (XYZM) coordinate support and include SRID information. Including all geometry information allows PostGIS to use EWKB as the format of record (e.g. in DUMP files).

EWKB and EWKT are used for the "canonical forms" of PostGIS data objects. For input, the canonical form for binary data is EWKB, and for text data either EWKB or EWKT is accepted. This allows geometry values to be created by casting a text value in either HEXEWKB or EWKT to a geometry value using `::geometry`. For output, the canonical form for binary is EWKB, and for text it is HEXEWKB (hex-encoded EWKB).

For example this statement creates a geometry by casting from an EWKT text value, and outputs it using the canonical form of HEXEWKB:

Code

```
SELECT 'SRID=4;POINT(0 0)'::geometry;
```

Output

```
geometry
----------------------------------------------------
0101000020040000000000000000000000000000000000000000
```

PostGIS EWKT output has a few differences to OGC WKT:

- For 3DZ geometries the Z qualifier is omitted:
  POINT(0 0)
  POINT(0 0)
- For 3DM geometries the M qualifier is included:
  POINT(0 0)
  POINT(0 0)
- For 4D geometries the ZM qualifier is omitted:
  POINT(0 0)
  POINT(0 0)

EWKT avoids over-specifying dimensionality and the inconsistencies that can occur with the OGC/ISO format, such as:

- POINT(0 0)
- POINT(0 0)
- POINT(0 0)



□□
PostGIS □□□ OGC □□□□□□□□□□□□□□□□□□ (□□□□□ WKB/WKT □□□□ EWKB/EWKT □□□□). □□□□□□□□□□□□□□□. □□ OGC □ PostGIS □□□□□□□□□□□□□□□□□□ □□□□□□□□□□. □□□□□□□□□□□□□□□□□□□!

□□□□□□□□□□□□□□□□□□□□□□ (WKT) □□□□□□□□□□□□□□□□□□:

- POINT(0 0 0) -- XYZ
- SRID=32632;POINT(0 0) -- SRID □□ XY
- POINTM(0 0 0) -- XYM
- POINT(0 0 0 0) -- XYZM
- SRID=4326;MULTIPOINTM(0 0 0,1 2 1) -- SRID □□ XYM
- MULTILINESTRING((0 0 0,1 1 0,1 2 1),(2 3 1,3 2 1,5 4 1))
- POLYGON((0 0 0,4 0 0,4 4 0,0 4 0,0 0 0),(1 1 0,2 1 0,2 2 0,1 2 0,1 1 0))
- MULTIPOLYGON(((0 0 0,4 0 0,4 4 0,0 4 0,0 0 0),(1 1 0,2 1 0,2 2 0,1 2 0,1 1 0)),((-1 -1 0,-1 -2 0,-2 -2 0,-2 -1 0,-1 -1 0)))
- GEOMETRYCOLLECTIONM(POINTM(2 3 9), LINESTRINGM(2 3 4, 3 4 5))
- MULTICURVE((0 0, 5 5), CIRCULARSTRING(4 0, 4 4, 8 4))
- POLYHEDRALSURFACE(((0 0 0, 0 0 1, 0 1 1, 0 1 0, 0 0 0)), ((0 0 0, 0 1 0, 1 1 0, 1 0 0, 0 0 0)), ((0 0 0, 1 0 0, 1 0 1, 0 0 1, 0 0 0)), ((1 1 0, 1 1 1, 1 0 1, 1 0 0, 1 1 0)), ((0 1 0, 0 1 1, 1 1 1, 1 1 0, 0 1 0)), ((0 0 1, 1 0 1, 1 1 1, 0 1 1, 0 0 1)))
- TRIANGLE ((0 0, 0 9, 9 0, 0 0))
- TIN(((0 0 0, 0 0 1, 0 1 0, 0 0 0)), ((0 0 0, 0 1 0, 1 1 0, 0 0 0)))

□□□□□□□□□□□□□□□□□□□□□□/□□□□□□□□.

Code

```
bytea EWKB = ST_AsEWKB(geometry);
text EWKT = geometry;
geometry = ST_GeomFromEWKB(bytea EWKB);
geometry = ST_GeomFromEWKT(text EWKT);
```

⊠⊠⊠⊠ PostGIS ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠:

Code

```
INSERT INTO geotable ( geom, name )
VALUES ( ST_GeomFromEWKT('SRID=312;POINTM(-126.4 45.32 15)'), 'A Place' )
```

4.3 PostGIS ⊠⊠⊠⊠⊠

⊠⊠⊠⊠⊠⊠ (⊠⊠” ⊠⊠” ⊠⊠, ⊠⊠” ⊠⊠/⊠⊠” ⊠” ⊠⊠/⊠⊠” ⊠⊠⊠⊠⊠) ” ⊠⊠” ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠ (⊠) ⊠⊠⊠⊠⊠⊠⊠ (球面) ⊠⊠⊠⊠⊠.

PostGIS ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (⊠⊠, ⊠⊠, ⊠⊠, ⊠⊠⊠⊠) ⊠⊠⊠⊠⊠⊠⊠⊠.

PostGIS ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (大圈; great circle arc) ⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠ (⊠⊠, ⊠⊠, ⊠⊠, ⊠⊠⊠⊠) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (spheroidal shape) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

Like the geometry data type, geography data is associated with a spatial reference system via a spatial reference system identifier (SRID). Any geodetic (long/lat based) spatial reference system defined in the `spatial_ref_sys` table can be used. You can add your own custom geodetic spatial reference system as described in 4.5.2⊠.

For all spatial reference systems the units returned by measurement functions (e.g. ST_Distance, ST_Length, ST_Perimeter, ST_Area) and for the distance argument of ST_DWithin are in meters.

4.3.1 ⊠⊠⊠⊠⊠⊠⊠

You can create a table to store geography data using the CREATE TABLE SQL statement with a column of type `geography`. The following example creates a table with a geography column storing 2D LineStrings in the WGS84 geodetic coordinate system (SRID 4326):

Code

```
CREATE TABLE global_points (
  id SERIAL PRIMARY KEY,
  name VARCHAR(64),
  location geography(POINT, 4326)
);
```

The geography type supports two optional type modifiers:

- ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. POINT, LINESTRING, POLYGON, MULTIPOINT, MULTI-LINESTRING, MULTIPOLYGON. ⊠⊠⊠⊠⊠⊠ Z, M ⊠ ZM ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠, ⊠⊠⊠⊠⊠’LINESTRINGM’ ⊠⊠⊠⊠⊠ 3 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠’POINTZM’ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

- the SRID modifier restricts the spatial reference system SRID to a particular number. If omitted, the SRID defaults to 4326 (WGS84 geodetic), and all calculations are performed using WGS84.

Examples of creating tables with geography columns:

POINT: 2D ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠:

Code

```
CREATE TABLE ptgeogwgs(gid serial PRIMARY KEY, geog geography(POINT));
```

POINT: 2D ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠:

Code

```
CREATE TABLE ptgeognad83(gid serial PRIMARY KEY, geog geography(POINT, 4269));
```

Create a table with 3D (XYZ) POINTs and an explicit SRID of 4326:

Code

```
CREATE TABLE ptzgeogwgs84(gid serial PRIMARY KEY, geog geography(POINTZ, 4326));
```

Create a table with 2D LINESTRING geography with the default SRID 4326:

Code

```
CREATE TABLE lgeog(gid serial PRIMARY KEY, geog geography(LINESTRING));
```

POINT: 2D ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠:

Code

```
CREATE TABLE lgeognad27(gid serial PRIMARY KEY, geog geography(POLYGON, 4267));
```

Geography fields are registered in the `geography_columns` system view. You can query the `geography_colum` view and see that the table is listed:

Code

```
SELECT * FROM geography_columns;
```

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. PostGIS ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

Code

```
CREATE INDEX global_points_gix ON global_points USING GIST ( location );
```

4.3.2 PostGIS ⊠⊠⊠⊠⊠

You can insert data into geography tables in the same way as geometry. Geometry data will autocast to the geography type if it has SRID 4326. The EWKT and EWKB formats can also be used to specify geography values.

Code

```
INSERT INTO global_points (name, location) VALUES ('Town', 'SRID=4326;POINT(-110 30)');
INSERT INTO global_points (name, location) VALUES ('Forest', 'SRID=4326;POINT(-109 29)');
INSERT INTO global_points (name, location) VALUES ('London', 'SRID=4326;POINT(0 49)');
```

Any geodetic (long/lat) spatial reference system listed in `spatial_ref_sys` table may be specified as a geography SRID. Non-geodetic coordinate systems raise an error if used.

A NAD83 longitude/latitude geography value:

Code

```
SELECT 'SRID=4269;POINT(-123 34)'::geography;
```

Output

```
geography
----------------------------------------------------
 0101000020AD1000000000000000C05EC00000000000004140
```

A NAD27 longitude/latitude geography value:

Code

```
SELECT 'SRID=4267;POINT(-123 34)'::geography;
```

Output

```
geography
----------------------------------------------------
 0101000020AB1000000000000000C05EC00000000000004140
```

NAD83 UTM zone meters fails because it is a meter-based planar projection.

Code

```
SELECT 'SRID=26910;POINT(-123 34)'::geography;
```

Output

```
ERROR:  Only lon/lat coordinate systems are supported in geography.
```

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠ (⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠) ⊠⊠⊠⊠⊠⊠⊠⊠.

This query finds rows within a 1000 km tolerance:

Code

```
SELECT name FROM global_points WHERE ST_DWithin(location, 'SRID=4326;POINT(-110 29)':: ↩
    geography, 1000000);
```

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (LINESTRING(-122.33 47.606, 0.0 51.5)) ⊠⊠⊠⊠⊠⊠ (POINT(-21.96 64.15)) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠⊠.

Code

```
SELECT ST_Distance('LINESTRING(-122.33 47.606,0.0 51.5)'::geography, 'POINT(-21.96 64.15)' ↩
    ::geography);
```

Output

```
st_distance
-----------------
 122235.23815667
```

⊠⊠⊠⊠ (Great Circle mapper) ⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠” ⊠ (degree)” ⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠” ⊠” ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

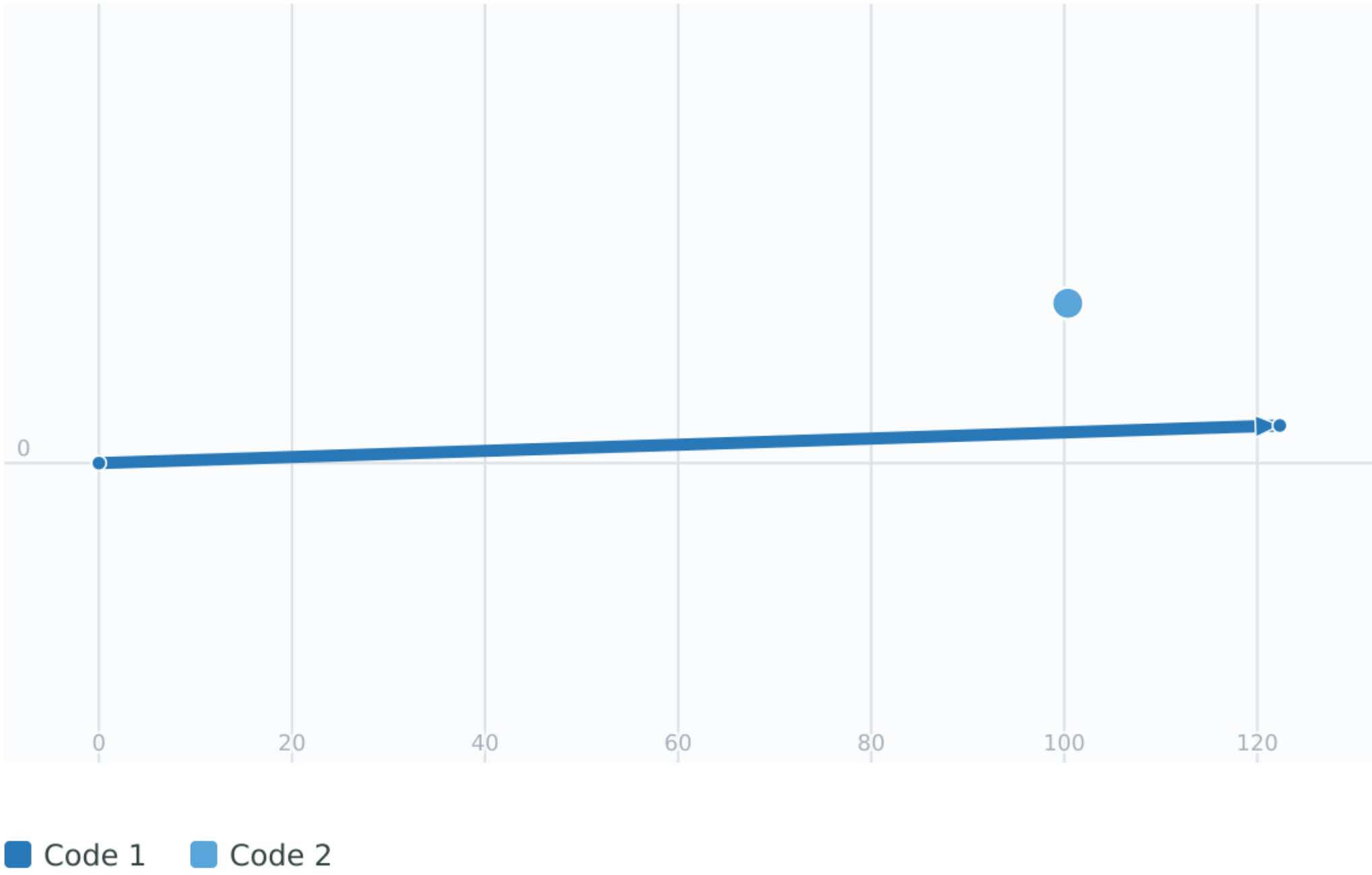
Code

```
SELECT ST_Distance('LINESTRING(-122.33 47.606,0.0 51.5)'::geometry, 'POINT(-21.96 64.15)':: ↩
    geometry);
```

Output

```
st_distance
--------------------
 13.342271221453624
```

Figure



4.3.3 [illegible]

[illegible]/[illegible], [illegible]. [illegible], [illegible] CPU [illegible].

[illegible]. [illegible], [illegible], [illegible]?

- [illegible], [illegible], [illegible].
- [illegible], [illegible]. [illegible]/[illegible], [illegible].
- [illegible], [illegible], [illegible], [illegible]. [illegible]/[illegible].

[illegible] 13.11[illegible]. [illegible] 13.4[illegible].

4.3.4 ⊠⊠⊠⊠⊠ FAQ

1. *⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠?*

   By default, all distance and area calculations are done on the spheroid. You should find that the results of calculations in local areas match up well with local planar results in good local projections. Over larger areas, the spheroidal calculations will be more accurate than any calculation done on a projected plane. The difference is that the spheroid takes into account the curvature and non-spherical shape of the Earth, whereas planar calculations assume a flat surface. ⊠⊠⊠⊠⊠⊠⊠⊠’FALSE’ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

2. *⊠⊠⊠⊠⊠⊠⊠⊠/⊠⊠⊠⊠⊠⊠⊠⊠?*

   ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠ (⊠⊠/⊠⊠) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

3. *⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠?*

   ⊠⊠⊠⊠⊠⊠⊠” ⊠⊠⊠⊠” ⊠⊠⊠⊠⊠ (great circle arc) ⊠⊠⊠⊠⊠⊠. ⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠’ ⊠⊠’ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠, 180 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

4. *⊠⊠⊠?*

   ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠! ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠ n ⊠⊠⊠ (⊠⊠ n ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠, ⊠⊠⊠” ⊠⊠⊠⊠” ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (subquery) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

4.4 Geometry Validation

PostGIS is compliant with the Open Geospatial Consortium’s (OGC) Simple Features specification. That standard defines the concepts of geometry being *simple* and *valid*. These definitions allow the Simple Features geometry model to represent spatial objects in a consistent and unambiguous way that supports efficient computation. (Note: the OGC SF and SQL/MM have the same definitions for simple and valid.)

4.4.1 Simple Geometry

A *simple* geometry is one that has no anomalous geometric points, such as self intersection or self tangency.

`POINT` ⊠ 0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠.

`MULTIPOINT` ⊠⊠⊠⊠⊠⊠ (`POINT`) ⊠⊠⊠⊠⊠⊠ (⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠) ⊠⊠⊠⊠⊠⊠.

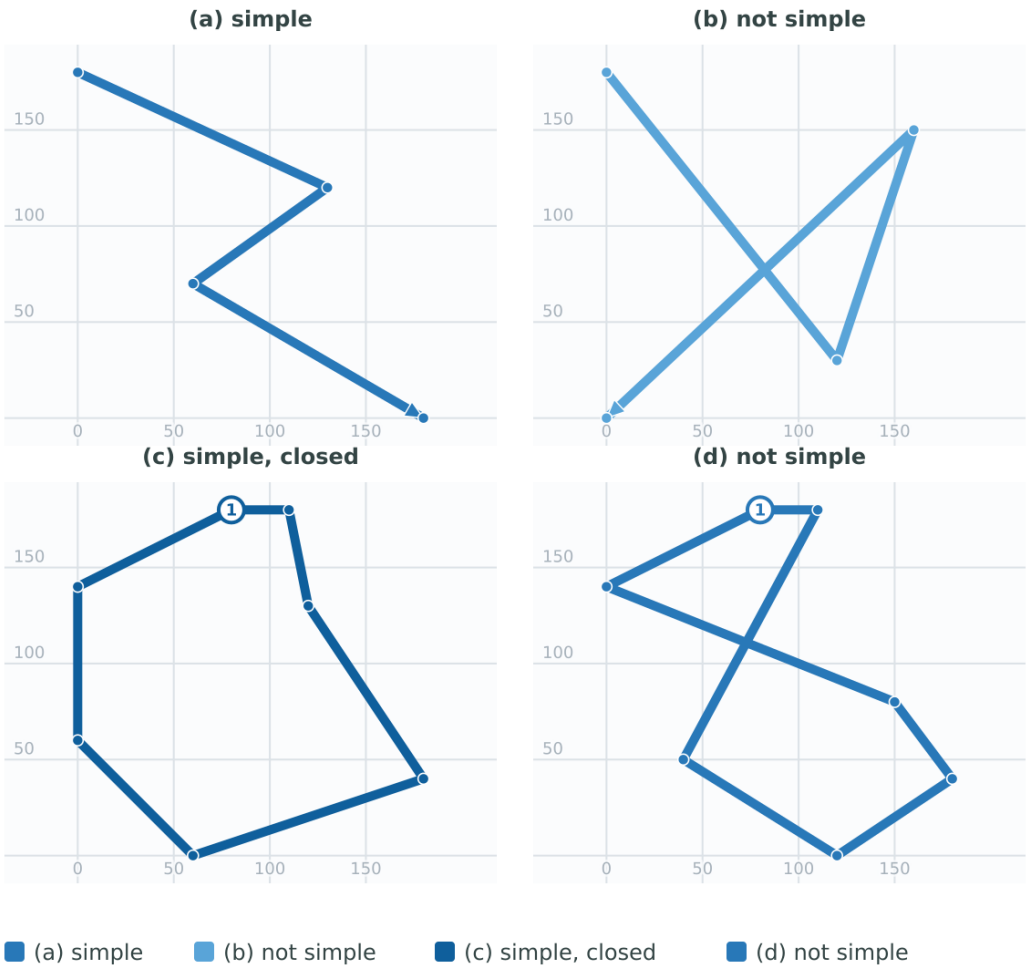
A `LINESTRING` is *simple* if it does not pass through the same point twice, except for the endpoints. If the endpoints of a simple LineString are identical it is called *closed* and referred to as a Linear Ring.

Use ST_IsSimple to classify linear geometries. Each labeled result below is returned only when the function agrees with the stated classification; example **(c)** also verifies that the simple LineString is closed.

Code

```
WITH cases(label, geom) AS (VALUES
  ('a', 'LINESTRING (10 190, 140 130, 70 80, 190 10)'::geometry),
  ('b', 'LINESTRING (10 190, 130 40, 170 160, 10 10)'::geometry),
  ('c', 'LINESTRING (90 190, 120 190, 130 140, 190 50, 70 10, 10 70, 10 150, 90 190)':: ↩
    geometry),
  ('d', 'LINESTRING (90 190, 120 190, 50 60, 130 10, 190 50, 160 90, 10 150, 90 190)':: ↩
    geometry)
)
SELECT
  (SELECT geom FROM cases WHERE label = 'a' AND ST_IsSimple(geom))
    AS "input_(a) simple",
  (SELECT geom FROM cases WHERE label = 'b' AND NOT ST_IsSimple(geom))
    AS "input_(b) not simple",
  (SELECT geom FROM cases
            WHERE label = 'c' AND ST_IsSimple(geom) AND ST_IsClosed(geom))
    AS "input_(c) simple, closed",
  (SELECT geom FROM cases WHERE label = 'd' AND NOT ST_IsSimple(geom))
    AS "input_(d) not simple";
```

Figure



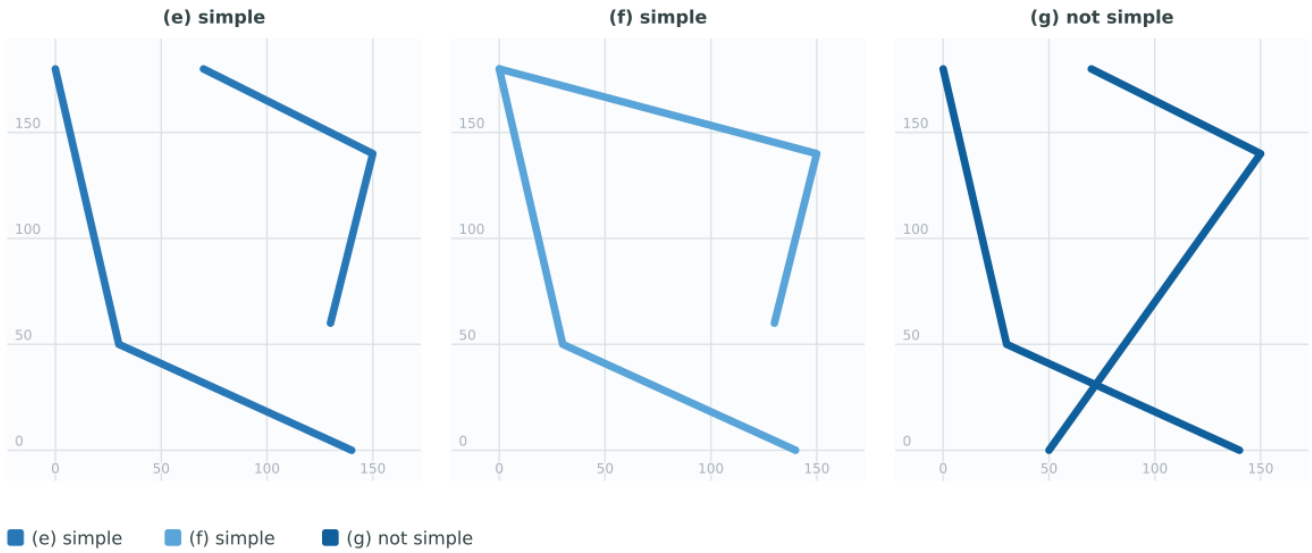
A `MULTILINESTRING` is *simple* only if all of its elements are simple and the only intersection between any two elements occurs at points that are on the boundaries of both elements.

The next example applies the same classification to the three MultiLineStrings. In **(f)** the component lines meet only at a shared endpoint, while in **(g)** they cross in their interiors.

Code

```
WITH cases(label, geom) AS (VALUES
  ('e', 'MULTILINESTRING ((30 190, 60 60, 170 10), (100 190, 180 150, 160 70))'::geometry),
  ('f', 'MULTILINESTRING ((30 190, 60 60, 170 10), (30 190, 180 150, 160 70))'::geometry),
  ('g', 'MULTILINESTRING ((30 190, 60 60, 170 10), (100 190, 180 150, 80 10))'::geometry)
)
SELECT
  (SELECT geom FROM cases WHERE label = 'e' AND ST_IsSimple(geom))
    AS "input_(e) simple",
  (SELECT geom FROM cases WHERE label = 'f' AND ST_IsSimple(geom))
    AS "input_(f) simple",
  (SELECT geom FROM cases WHERE label = 'g' AND NOT ST_IsSimple(geom))
    AS "input_(g) not simple";
```

Figure



POLYGONs are formed from linear rings, so valid polygonal geometry is always *simple*.

Generally, PostGIS functions do not require geometric arguments to be simple. Simplicity is primarily used as a basis for defining geometric validity. It is also a requirement for some kinds of spatial data models (for example, linear networks often disallow lines that cross). Multipoint and linear geometry can be made simple using ST_UnaryUnion.

4.4.2 Valid Geometry

Geometry validity primarily applies to 2-dimensional geometries (POLYGONs and MULTIPOLYGONs) . Validity is defined by rules that allow polygonal geometry to model planar areas unambiguously.

A POLYGON is *valid* if:

1. the polygon boundary rings (the exterior shell ring and interior hole rings) are *simple* (do not cross or self-touch). Because of this a polygon cannot have cut lines, spikes or loops. This implies that polygon holes must be represented as interior rings, rather than by the exterior ring self-touching (a so-called ”inverted hole”).
2. boundary rings do not cross
3. boundary rings may touch at points but only as a tangent (i.e. not in a line)

4. interior rings are contained in the exterior ring
5. the polygon interior is simply connected (i.e. the rings must not touch in a way that splits the polygon into more than one part)

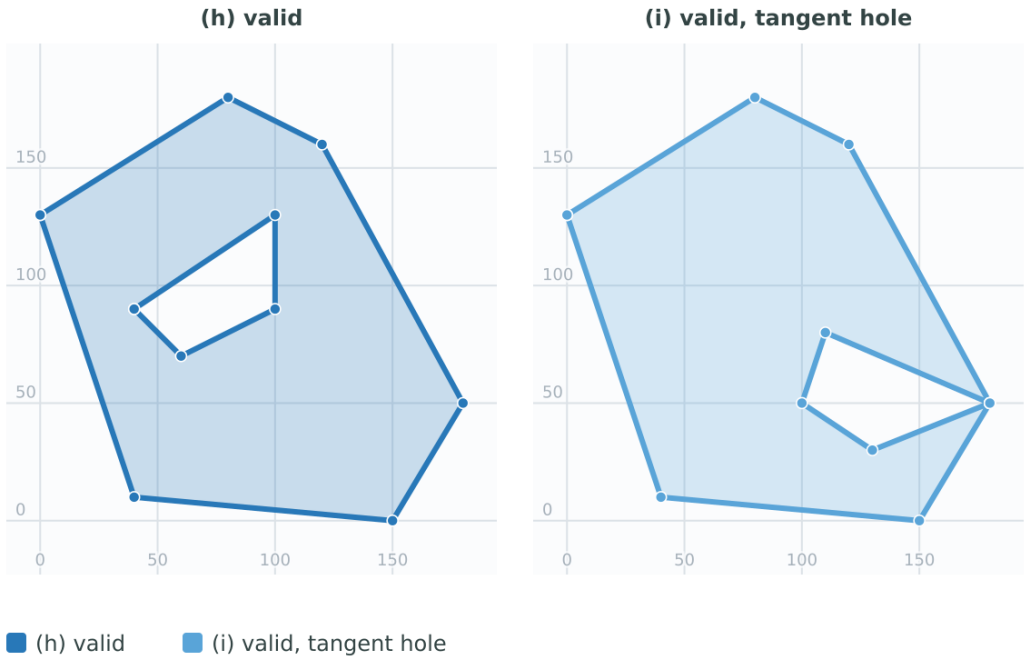
Consecutive repeated points in a polygon boundary ring do not make the ring non-simple. Under the SFS curve model, a curve is interpreted by a continuous parameterization; consecutive equal coordinates are topologically indistinguishable from a single vertex. PostGIS therefore treats them as redundant vertices, so they do not make an otherwise valid polygon invalid.

The following examples use ST_IsValid to check the validity rules. Each labeled geometry is returned only if its computed validity agrees with the label.

Code

```
WITH cases(label, geom) AS (VALUES
  ('h', 'POLYGON ((10 140, 90 190, 130 170, 190 60, 160 10, 50 20, 10 140),
                  (50 100, 70 80, 110 100, 110 140, 50 100))'::geometry),
  ('i', 'POLYGON ((10 140, 90 190, 130 170, 190 60, 160 10, 50 20, 10 140),
                  (190 60, 140 40, 110 60, 120 90, 190 60))'::geometry)
)
SELECT
  (SELECT geom FROM cases WHERE label = 'h' AND ST_IsValid(geom))
    AS "input_(h) valid",
  (SELECT geom FROM cases WHERE label = 'i' AND ST_IsValid(geom))
    AS "input_(i) valid, tangent hole";
```

Figure



Examples **(j-m)** are invalid. They show, respectively, a disconnected interior, crossing rings, a cut line, and a hole outside the shell. The invalidity reason is checked with ST_IsValidReason. The disconnected interior in **(j)** can instead be represented as a valid `MULTIPOLYGON`.

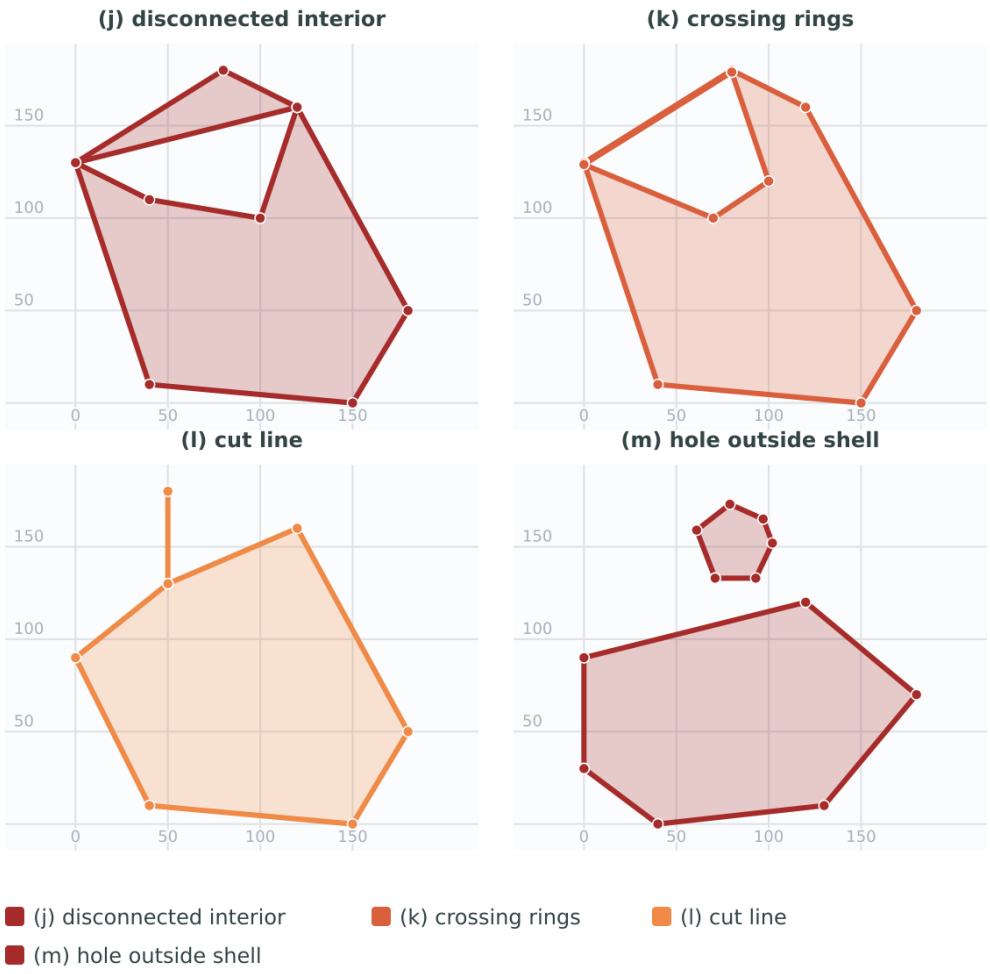
Code

```
WITH cases(label, geom, reason_pattern) AS (VALUES
  ('(j) disconnected interior',
   'POLYGON ((10 140, 90 190, 130 170, 190 60, 160 10, 50 20, 10 140),
             (130 170, 10 140, 50 120, 110 110, 130 170))'::geometry,
   'Interior is disconnected%'),
  ('(k) crossing rings',
   'POLYGON ((10 140, 90 190, 130 170, 190 60, 160 10, 50 20, 10 140),
             (90 189, 10 139, 80 110, 110 130, 90 189))'::geometry,
   'Self-intersection%'),
  ('(l) cut line',
   'POLYGON ((10 100, 60 140, 60 190, 60 140, 130 170,
              190 60, 160 10, 50 20, 10 100))'::geometry,
   'Ring Self-intersection%'),
  ('(m) hole outside shell',
   'POLYGON ((10 40, 10 100, 130 130, 190 80, 140 20, 50 10, 10 40),
             (81 143, 103 143, 112 162, 107 175, 89 183, 71 169, 81 143))'::geometry,
   'Hole lies outside shell%')
)
SELECT label, geom
FROM cases
WHERE NOT ST_IsValid(geom)
  AND ST_IsValidReason(geom) LIKE reason_pattern
ORDER BY label;
```

Output

```
label          | geom
-----------------------+-----------------------
```

Figure



A `MULTIPOLYGON` is *valid* if:

1. its element `POLYGON`s are valid
2. elements do not overlap (i.e. their interiors must not intersect)
3. elements touch only at points (i.e. not along a line)

The next example checks the MultiPolygon rules. In **(n)** the polygons touch at only one point. In **(o)** they share an edge, and in **(p)** their interiors overlap.

Code

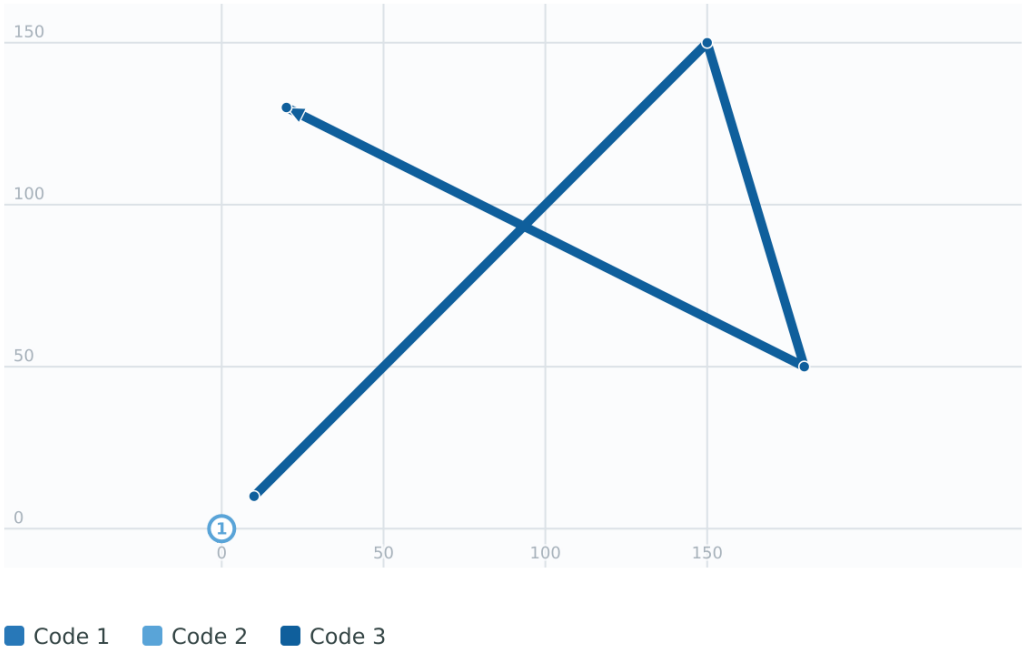
```
WITH cases(label, geom, valid, reason_pattern) AS (VALUES
  ('(n) point touch',
   'MULTIPOLYGON (((10 40, 29 118, 100 75, 180 88, 140 30, 70 10, 10 40)),
                  ((50 170, 100 190, 160 180, 180 140, 180 90, 120 120,
                    30 120, 50 170)))'::geometry,
   true, NULL),
  ('(o) shared edge',
   'MULTIPOLYGON (((10 40, 30 120, 120 120, 180 88, 140 30, 70 10, 10 40)),
                  ((50 170, 100 190, 160 180, 180 140, 180 90, 120 120,
                    30 120, 50 170)))'::geometry,
   false, 'Self-intersection%'),
  ('(p) overlap',
   'MULTIPOLYGON (((10 40, 42 93, 130 150, 180 86, 140 30, 70 10, 10 40)),
```

```
                  ((50 170, 100 190, 138 182, 149 156, 146 137, 100 100,
                    58 140, 50 170)))'::geometry,
   false, 'Self-intersection%')
)
SELECT label, geom
FROM cases
WHERE ST_IsValid(geom) = valid
  AND (valid OR ST_IsValidReason(geom) LIKE reason_pattern)
ORDER BY label;
```

Output

```
label       | geom
------------------+-----------------------
```

Figure



`POINT` and `MULTIPOINT` geometries have no validity rules.

4.4.3 Managing Validity

PostGIS allows creating and storing both valid and invalid Geometry. This allows invalid geometry to be detected and flagged or fixed. There are also situations where the OGC validity rules are stricter than desired (examples of this are zero-length linestrings and polygons with inverted holes.)

Many of the functions provided by PostGIS rely on the assumption that geometry arguments are valid. For example, it does not make sense to calculate the area of a polygon that has a hole defined outside of the polygon, or to construct a polygon from a non-simple boundary line. Assuming valid geometric inputs allows functions to operate more efficiently, since they do not need to check for topological correctness. (Notable exceptions are that zero-length lines and polygons with inversions are generally handled correctly.) Also, most PostGIS functions produce valid geometry output if the inputs are valid. This allows PostGIS functions to be chained together safely.

If you encounter unexpected error messages when calling PostGIS functions (such as "GEOS Intersection() threw an error!"), you should first confirm that the function arguments are valid. If they are not, then consider using one of the techniques below to ensure the data you are processing is valid.



□□
If a function reports an error with valid inputs, then you may have found an error in either PostGIS or one of the libraries it uses, and you should report this to the PostGIS project. The same is true if a PostGIS function returns an invalid geometry for valid input.

To test if a geometry is valid use the ST_IsValid function:

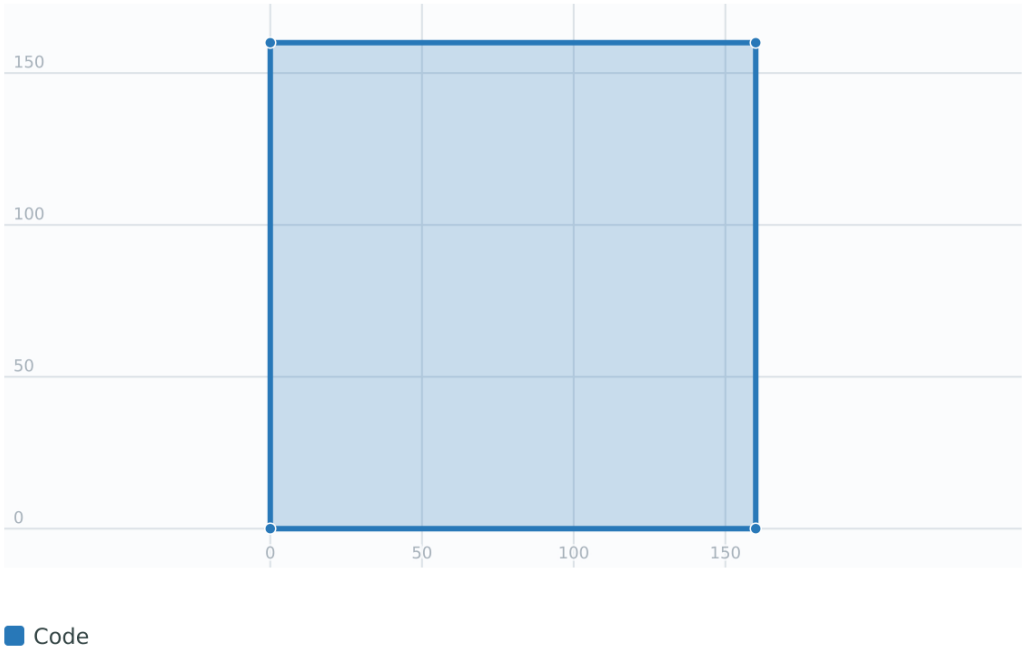
Code

```
SELECT ST_IsValid('POLYGON ((20 180,180 180,180 20,20 20,20 180))');
```

Output

```
t
```

Figure



Information about the nature and location of an geometry invalidity are provided by the ST_IsValidDetail function:

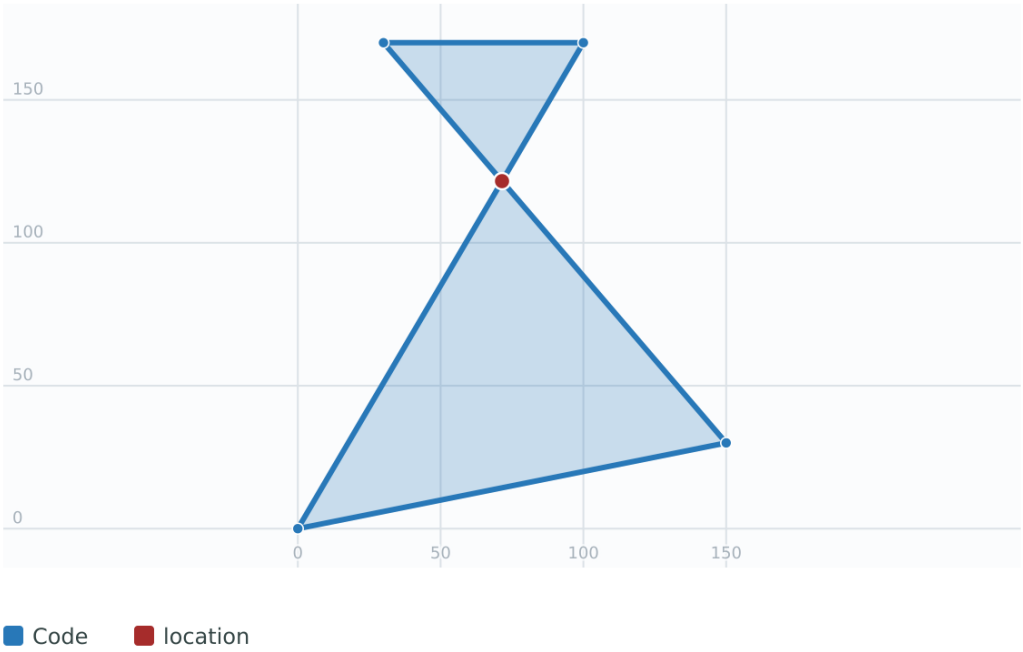
Code

```
SELECT valid, reason,location AS location
        FROM ST_IsValidDetail('POLYGON ((20 20,120 190,50 190,170 50,20 20))') AS t;
```

Output

```
valid |      reason       |                  location
-------+-------------------+--------------------------------------------
 f     | Self-intersection | POINT(91.51162790697674 141.56976744186048)
```

Figure



In some situations it is desirable to correct invalid geometry automatically. Use the ST_MakeValid function to do this. (`ST_MakeValid` is a case of a spatial function that *does* allow invalid input!)

By default, PostGIS does not check for validity when loading geometry, because validity testing can take a lot of CPU time for complex geometries. If you do not trust your data sources, you can enforce a validity check on your tables by adding a check constraint:

Code

```
ALTER TABLE mytable
ADD CONSTRAINT geometry_valid_check
  CHECK (ST_IsValid(geom));
```

4.5 SPATIAL_REF_SYS ☒☒☒☒☒☒☒☒☒☒

A Spatial Reference System (SRS) (also called a Coordinate Reference System (CRS)) defines how geometry is referenced to locations on the Earth's surface. There are three types of SRS:

- A **geodetic** SRS uses angular coordinates (longitude and latitude) which map directly to the surface of the earth.
- A **projected** SRS uses a mathematical projection transformation to "flatten" the surface of the spheroidal earth onto a plane. It assigns location coordinates in a way that allows direct measurement of quantities such as distance, area, and angle. The coordinate system is Cartesian, which means it has a defined origin point and two perpendicular axes (usually oriented North and East). Each projected SRS uses a stated length unit (usually metres or feet). A projected SRS may be limited in its area of applicability to avoid distortion and fit within the defined coordinate bounds.
- A **local** SRS is a Cartesian coordinate system which is not referenced to the earth's surface. In PostGIS this is specified by a SRID value of 0.

There are many different spatial reference systems in use. Common SRSes are standardized in the European Petroleum Survey Group EPSG database. For convenience PostGIS (and many other spatial systems) refers to SRS definitions using an integer identifier called a SRID.

A geometry is associated with a Spatial Reference System by its SRID value, which is accessed by ST_SRID. The SRID for a geometry can be assigned using ST_SetSRID. Some geometry constructor functions allow supplying a SRID (such as ST_Point and ST_MakeEnvelope). The EWKT format supports SRIDs with the `SRID=n;` prefix.

Spatial functions processing pairs of geometries (such as overlay and relationship functions) require that the input geometries are in the same spatial reference system (have the same SRID). Geometry data can be transformed into a different spatial reference system using ST_Transform and ST_TransformPipe Geometry returned from functions has the same SRS as the input geometries.

4.5.1 SPATIAL_REF_SYS Table

The `SPATIAL_REF_SYS` table used by PostGIS is an OGC-compliant database table that defines the available spatial reference systems. It holds the numeric SRIDs and textual descriptions of the coordinate systems.

`SPATIAL_REF_SYS` □□□□□□□□□□□□□□:

Code

```
CREATE TABLE spatial_ref_sys (
  srid       INTEGER NOT NULL PRIMARY KEY,
  auth_name  VARCHAR(256),
  auth_srid  INTEGER,
  srtext     VARCHAR(2048),
  proj4text  VARCHAR(2048)
)
```

□□□□□□□□□□□□□□□□:

srid □□□□□□□□□□□□□□□□□□ (SRS) □□□□□□□□□□□□□□□□.

auth_name □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□. □□□□"EPSG" □□□□ `AUTH_NAME` □□□□□□□□□.

auth_srid The ID of the Spatial Reference System as defined by the Authority cited in the `auth_name`. In the case of EPSG, this is the EPSG code.

srtext □□□□□□□□ WKT(Well-Known Text) □□□□□□. □□□ WKT SRS □□□□□□□□:

Code

```
PROJCS["NAD83 / UTM Zone 10N",
  GEOGCS["NAD83",
    DATUM["North_American_Datum_1983",
      SPHEROID["GRS 1980", 6378137, 298.257222101]
    ],
    PRIMEM["Greenwich", 0],
    UNIT["degree", 0.0174532925199433]
  ],
  PROJECTION["Transverse_Mercator"],
  PARAMETER["latitude_of_origin", 0],
  PARAMETER["central_meridian", -123],
  PARAMETER["scale_factor", 0.9996],
  PARAMETER["false_easting", 500000],
  PARAMETER["false_northing", 0],
  UNIT["metre", 1]
]
```

For a discussion of SRS WKT, see the OGC standard Well-known text representation of coordinate reference systems.

proj4text PostGIS ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ proj4 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. PROJ4TEXT ⊠⊠⊠⊠ SRID ⊠⊠⊠⊠⊠ proj4 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠:

Code

```
+proj=utm +zone=10 +ellps=clrk66 +datum=NAD27 +units=m
```

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ http://trac.osgeo.org/proj/ ⊠⊠⊠ proj4 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. spatial_ref_sys.sql ⊠⊠⊠⊠⊠ EPSG ⊠⊠⊠⊠⊠ SRTEXT ⊠ PROJ4TEXT ⊠⊠⊠⊠⊠⊠⊠⊠⊠.

When retrieving spatial reference system definitions for use in transformations, PostGIS uses fhe following strategy:

- If auth_name and auth_srid are present (non-NULL) use the PROJ SRS based on those entries (if one exists).
- If srtext is present create a SRS using it, if possible.
- If proj4text is present create a SRS using it, if possible.

4.5.2 SPATIAL_REF_SYS ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠

PostGIS ⊠ SPATIAL_REF_SYS ⊠⊠⊠⊠ proj ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ 3000 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠ proj4 ⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠ SPATIAL_REF_SYS ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ http://spatialreference.org/ ⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠.

Some commonly used spatial reference systems are: 4326 - WGS 84 Long Lat, 4269 - NAD 83 Long Lat, 3395 - WGS 84 World Mercator, 2163 - US National Atlas Equal Area, and the 60 WGS84 UTM zones. UTM zones are one of the most ideal for measurement, but only cover 6-degree regions. For WGS84 longitude/latitude data, the UTM zone is least(floor((longitude + 180) / 6) + 1, 60); the EPSG code is 32600 + zone in the northern hemisphere or 32700 + zone in the southern hemisphere.

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (⊠⊠⊠⊠⊠⊠⊠⊠⊠) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠ SPATIAL_REF_SYS ⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠ ESRI ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠ spatialreference.org ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

You can even define non-Earth-based coordinate systems, such as Mars 2000 This Mars coordinate system is non-planar (it’s in degrees spheroidal), but you can use it with the geography type to obtain length and proximity measurements in meters instead of degrees.

Here is an example of loading a custom coordinate system using an unassigned SRID and the PROJ definition for a US-centric Lambert Conformal projection:

Code

```
INSERT INTO spatial_ref_sys (srid, proj4text)
VALUES ( 990000,
  '+proj=lcc  +lon_0=-95 +lat_0=25 +lat_1=25 +lat_2=25 +x_0=0 +y_0=0 +datum=WGS84 +units=m ←
    +no_defs'
);
```

4.6 □□□□□□□

4.6.1 □□□□□□□

You can create a table to store geometry data using the CREATE TABLE SQL statement with a column of type `geometry`. The following example creates a table with a geometry column storing 2D (XY) LineStrings in the BC-Albers coordinate system (SRID 3005):

Code

```
CREATE TABLE roads (
  id SERIAL PRIMARY KEY,
  name VARCHAR(64),
  geom geometry(LINESTRING, 3005)
);
```

The `geometry` type supports two optional **type modifiers**:

- □□□□□□□□□□□□□□□□□□□□. POINT, LINESTRING, POLYGON, MULTIPOINT, MULTILINESTRING, MULTIPOLYGON. □□□□□□ Z, M □ ZM □□□□□□□□□□□□□□□□□□□□. □□□, □□□□□'LINESTRINGM' □□□□□ 3 □□□□□□□□□□□□□□□□□□□, □□□□□□□ □□□□□□□□□□□. □□□□□'POINTZM' □□□□□□□□□□□□□□□□□□□□.
- the **SRID modifier** restricts the spatial reference system SRID to a particular number. If omitted, the SRID defaults to 0.

Examples of creating tables with geometry columns:

Create a table holding any kind of geometry with the default SRID:

Code

```
CREATE TABLE geoms(gid serial PRIMARY KEY, geom geometry);
```

Create a table with 2D POINT geometry with the default SRID:

Code

```
CREATE TABLE pts(gid serial PRIMARY KEY, geom geometry(POINT));
```

Create a table with 3D (XYZ) POINTs and an explicit SRID of 3005:

Code

```
CREATE TABLE pts(gid serial PRIMARY KEY, geom geometry(POINTZ, 3005));
```

Create a table with 4D (XYZM) LINESTRING geometry with the default SRID:

Code

```
CREATE TABLE lines(gid serial PRIMARY KEY, geom geometry(LINESTRINGZM));
```

Create a table with 2D POLYGON geometry with the SRID 4267 (NAD 1927 long lat):

Code

```
CREATE TABLE polys(gid serial PRIMARY KEY, geom geometry(POLYGON, 4267));
```

It is possible to have more than one geometry column in a table. This can be specified when the table is created, or a column can be added using the ALTER TABLE SQL statement. This example adds a column that can hold 3D LineStrings:

Code

```
ALTER TABLE roads ADD COLUMN geom2 geometry(LINESTRINGZ, 4326);
```

4.6.2 The GEOMETRY_COLUMNS VIEW

OpenGIS ⊠"SQL ⊠⊠⊠⊠⊠⊠⊠⊠ (Simple Features Specification for SQL)" ⊠⊠⊠ GIS ⊠⊠⊠⊠, ⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ OpenGIS ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

Code

```
\d geometry_columns
```

Output

```
View "public.geometry_columns"
      Column       |          Type          | Modifiers
-------------------+------------------------+-----------
 f_table_catalog   | character varying(256) |
 f_table_schema    | character varying(256) |
 f_table_name      | character varying(256) |
 f_geometry_column | character varying(256) |
 coord_dimension   | integer                |
 srid              | integer                |
 type              | character varying(30)  |
```

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠:

f_table_catalog, f_table_schema, f_table_name ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. " ⊠⊠⊠⊠⊠" ⊠" ⊠⊠⊠⊠" ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. " ⊠⊠⊠⊠⊠" ⊠⊠⊠⊠⊠⊠ PostgreSQL ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. " ⊠⊠⊠⊠" ⊠⊠⊠ PostgreSQL ⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠ (⊠⊠⊠⊠⊠ `public` ⊠⊠⊠).

f_geometry_column ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

coord_dimension The coordinate dimension (2, 3 or 4) of the column. The coordinate dimension is at least 2, representing X and Y coordinates.

srid ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ID ⊠, `SPATIAL_REF_SYS` ⊠⊠⊠⊠⊠⊠⊠⊠ ⊠ (foreign key) ⊠⊠⊠.

type ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. POINT, LINESTRING, POLYGON, MULTIPOINT, MULTILINESTRING, MULTIPOLYGON, GEOMETRYCOLLECTION ⊠⊠⊠⊠⊠⊠⊠⊠ XYM ⊠⊠⊠ POINTM, LINESTRINGM, POLYGONM, MULTIPOINTM, MULTILINESTRINGM, MULTIPOLYGONM, GEOMETRYCOLLECTIONM. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠"GEOMETRY" ⊠⊠⊠⊠⊠⊠⊠⊠⊠.

4.6.3 geometry_columns ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠

AddGeometryColumn() ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, SQL ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (bulk insert) ⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ geometry_columns ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠. PostGIS 2.0 ⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠ typmod ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

For example, suppose you have a view created like this.

Code

```
CREATE VIEW public.vwmytablemercator AS
SELECT gid, ST_Transform(geom, 3395) As geom, f_name
FROM public.mytable;
```

To register the geometry column correctly, cast the transformed geometry.

Code

```
DROP VIEW public.vwmytablemercator;
CREATE VIEW  public.vwmytablemercator AS
    SELECT gid, ST_Transform(geom, 3395)::geometry(Geometry, 3395) As geom, f_name
    FROM public.mytable;
```

If the geometry type is known to be a 2D Polygon, specify it in the cast.

Code

```
DROP VIEW public.vwmytablemercator;
CREATE VIEW  public.vwmytablemercator AS
    SELECT gid, ST_Transform(geom, 3395)::geometry(Polygon, 3395) As geom, f_name
    FROM public.mytable;
```

For a derivative table created by a bulk insert, create the table first.

Code

```
SELECT poi.gid, poi.geom, citybounds.city_name
INTO myschema.my_special_pois
FROM poi INNER JOIN citybounds ON ST_Intersects(citybounds.geom, poi.geom);
```

Create a 2D spatial index on the new table.

Code

```
CREATE INDEX idx_myschema_myspecialpois_geom_gist
ON myschema.my_special_pois USING gist(geom);
```

If the points are 3D or measured, an n-D index may be more appropriate.

Code

```
CREATE INDEX my_special_pois_geom_gist_nd
ON my_special_pois USING gist(geom gist_geometry_ops_nd);
```

Manually register the new table's geometry column in `geometry_columns`. This also changes the underlying table structure to make the column typmod-based.

Code

```
SELECT Populate_Geometry_Columns('myschema.my_special_pois'::regclass);
```

To retain constraint-based definition behavior, such as for inherited tables where all children do not have the same type and SRID, set the optional `use_typmod` argument to false.

Code

```
SELECT Populate_Geometry_Columns('myschema.my_special_pois'::regclass, false);
```

Although the old-constraint based method is still supported, a constraint-based geometry column used directly in a view, will not register correctly in geometry_columns, as will a typmod one. In this example we define a column using typmod and another using constraints.

Code

```
CREATE TABLE pois_ny(gid SERIAL PRIMARY KEY, poi_name text, cat text, geom geometry(POINT,  ↩
    4326));
SELECT AddGeometryColumn('pois_ny', 'geom_2160', 2160, 'POINT', 2, false);
```

If we run **\d pois_ny** in psql, we observe they are defined differently -- one is typmod, one is constraint.

Code

```
\d pois_ny
```

Output

```
Table "public.pois_ny"
  Column   |         Type          |                       Modifiers

-----------+-----------------------+-------------------------------------------------------
 gid       | integer               | not null default nextval('pois_ny_gid_seq'::regclass)
 poi_name  | text                  |
 cat       | character varying(20) |
 geom      | geometry(Point,4326)  |
 geom_2160 | geometry              |
Indexes:
    "pois_ny_pkey" PRIMARY KEY, btree (gid)
Check constraints:
    "enforce_dims_geom_2160" CHECK (st_ndims(geom_2160) = 2)
    "enforce_geotype_geom_2160" CHECK (geometrytype(geom_2160) = 'POINT'::text
        OR geom_2160 IS NULL)
    "enforce_srid_geom_2160" CHECK (st_srid(geom_2160) = 2160)
```

☒☒ geometry_columns ☒☒☒☒☒☒☒☒☒☒☒☒.

Code

```
SELECT f_table_name, f_geometry_column, srid, type
FROM geometry_columns
WHERE f_table_name = 'pois_ny';
```

Output

```
f_table_name | f_geometry_column | srid | type
-------------+-------------------+------+-------
pois_ny      | geom              | 4326 | POINT
pois_ny      | geom_2160         | 2160 | POINT
```

☒☒☒ -- ☒☒☒☒☒☒☒☒☒☒☒☒☒☒

Code

```
CREATE VIEW vw_pois_ny_parks AS
SELECT *
  FROM pois_ny
  WHERE cat='park';

SELECT f_table_name, f_geometry_column, srid, type
    FROM geometry_columns
    WHERE f_table_name = 'vw_pois_ny_parks';
```

The typmod based geom view column registers correctly, but the constraint based one does not.

Output

```
f_table_name   | f_geometry_column | srid |   type
-----------------+-------------------+------+----------
 vw_pois_ny_parks | geom              | 4326 | POINT
 vw_pois_ny_parks | geom_2160         |    0 | GEOMETRY
```

This may change in future versions of PostGIS, but for now to force the constraint-based view column to register correctly, you need to do this:

Code

```
DROP VIEW vw_pois_ny_parks;
CREATE VIEW vw_pois_ny_parks AS
SELECT gid, poi_name, cat,
  geom,
  geom_2160::geometry(POINT,2160) As geom_2160
  FROM pois_ny
  WHERE cat = 'park';
SELECT f_table_name, f_geometry_column, srid, type
    FROM geometry_columns
    WHERE f_table_name = 'vw_pois_ny_parks';
```

Output

```
f_table_name   | f_geometry_column | srid | type
-----------------+-------------------+------+-------
 vw_pois_ny_parks | geom              | 4326 | POINT
 vw_pois_ny_parks | geom_2160         | 2160 | POINT
```

4.7 GIS (□□) □□□□□

□□□□□□□□□□□□□, □□□□□□□□□□□□ GIS □□□□□□□□□□□□□□□□□□. □□, □□□□ SQL □□□□□□□□□ shapefile □□/□□□□□□□□□□□□□□□□ PostGIS/PostgreSQL □□□□□□□□□ □□□□□□□□□□□□.

4.7.1 SQL □□□□□□□□□□□

□□□□□□□□□□□□□□□□□□□□□□□□□□, PostGIS □□□□□□□□□□□□□□□□□□□□□□□□□□ (formatted) SQL □□□□□□□□□□. Oracle □□□□ SQL □□□□□□□□□□□□□, SQL □□□□□□□□ SQL "INSERT" □□□□□□□□□□□□□□□□□□□□□□□□ (piping) □□□□□□□□□□□□□□□□□□□□ □□.

□□□□□□□□ (□□□□ `roads.sql`) □□□□□□□□□□□:

Code

```
BEGIN;
INSERT INTO roads (road_id, roads_geom, road_name)
  VALUES (1, 'LINESTRING(191232 243118,191108 243242)', 'Jeff Rd');
INSERT INTO roads (road_id, roads_geom, road_name)
  VALUES (2, 'LINESTRING(189141 244158,189265 244817)', 'Geordie Rd');
INSERT INTO roads (road_id, roads_geom, road_name)
  VALUES (3, 'LINESTRING(192783 228138,192612 229814)', 'Paul St');
INSERT INTO roads (road_id, roads_geom, road_name)
  VALUES (4, 'LINESTRING(189412 252431,189631 259122)', 'Graeme Ave');
INSERT INTO roads (road_id, roads_geom, road_name)
  VALUES (5, 'LINESTRING(190131 224148,190871 228134)', 'Phil Tce');
INSERT INTO roads (road_id, roads_geom, road_name)
  VALUES (6, 'LINESTRING(198231 263418,198213 268322)', 'Dave Cres');
COMMIT;
```

"psql" SQL □□□□□□□□□□□ PostgreSQL □□□□□□□□□□□□□□□□□□□□.

Code

```
psql -d [database] -f roads.sql
```

4.7.2 shp2pgsql: ESRI shapefile ☒☒☒☒☒☒☒

`shp2pgsql` ☒☒☒☒☒☒ ESRI shapefile ☒, ☒☒☒☒☒☒☒☒☒☒☒☒☒☒, PostGIS/PostgreSQL ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒ SQL ☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒ (command line) ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.

The input must be an unpacked shapefile path. The loader does not read `.zip` archives directly; unpack the `.shp`, `.shx`, and `.dbf` files before loading.

shp2pgsql ☒☒☒☒☒☒☒☒, ☒☒☒☒☒ PostGIS ☒☒☒ `shp2pgsql-gui` ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒. `shp2pgsql-gui` ☒ pgAdmin III ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.

c|a|d|p -- ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒:

- **-c** ☒☒☒☒☒☒☒☒☒☒☒ shapefile ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒☒.
- **-a** ☒☒☒☒☒☒☒☒☒☒☒☒☒☒ shapefile ☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒, ☒☒.
- **-d** ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒ (drop) ☒☒☒ shapefile ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.
- **-p** ☒☒☒☒☒☒☒☒☒ SQL ☒☒☒☒☒☒☒☒☒, ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒.

-? ☒☒☒☒☒☒☒☒☒☒☒☒☒☒.

-D ☒☒☒☒☒☒☒☒☒☒☒☒☒ PostgreSQL ” ☒☒ (dump)” ☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒ -a, -c ☒ -d ☒☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒” ☒☒” SQL ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.

-f <fid_column>, --feature-id-column <fid_column> Specify the name of the feature id column. The default is `gid`. The value must be a simple PostgreSQL identifier and cannot use a PostgreSQL system column name.

-s [<FROM_SRID>:]<SRID> ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒ SRID ☒☒☒☒☒☒. ☒☒ shapefile ☒☒☒☒ FROM_SRID ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒ SRID ☒☒☒☒☒☒☒☒☒☒. FROM_SRID ☒ -D ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.

-k ☒☒☒☒☒☒☒☒☒ (☒, ☒☒☒☒☒☒☒) ☒☒☒☒☒☒☒. shapefile ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.

-i DBF ☒☒☒☒☒☒ 64 ☒☒ bigint ☒☒☒☒☒☒☒☒☒☒☒☒, ☒☒☒☒☒☒☒☒ 32 ☒☒☒☒☒☒☒☒☒☒☒☒ 64 ☒☒ bigint ☒☒☒☒☒☒☒☒☒☒☒☒☒.

-I ☒☒☒☒☒ GiST ☒☒☒☒☒☒☒☒☒☒.

-u Create the target table as `UNLOGGED`. This can speed up loading transient staging data, but PostgreSQL does not preserve unlogged table contents after a crash or unclean shutdown.

-m Specify a file containing a set of mappings of (long) column names to 10 character DBF column names. The content of the file is one or more lines of two names separated by white space and no trailing or leading space. For example:

```
COLUMNNAME DBFFIELD1
AVERYLONGCOLUMNNAME DBFFIELD2
```

-S ☒☒ (multi) ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒ (☒: ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒) ☒☒☒☒☒☒☒☒☒☒☒.

-t <dimensionality> ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠: 2D, 3DZ, 3DM, 4D

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠ 0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

-w WKB ⊠⊠ WKT ⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

-e ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠. ” ⊠⊠” ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ -D ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

-W <encoding> ⊠⊠⊠⊠⊠ (DBF ⊠⊠) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠, DBF ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ UTF8 ⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠ SQL ⊠⊠⊠⊠ `SET CLIENT_ENCODING to UTF8` ⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠ UTF8 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

-N <policy> NULL ⊠⊠⊠⊠⊠⊠ -- insert*(⊠⊠⊠⊠⊠⊠), skip(⊠⊠⊠⊠), abort(⊠⊠)

-n DBF ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ shapefile ⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ DBF ⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠ shapefile ⊠⊠.

-G (⊠⊠/⊠⊠⊠⊠⊠⊠⊠) ⊠⊠⊠⊠ WGS84 ⊠⊠⊠ (SRID=4326) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

-T <tablespace> ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. -X ⊠⊠. PostgreSQL ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

-X <tablespace> ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠ (primary key) ⊠⊠⊠⊠⊠⊠⊠⊠, -I ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ GiST ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

-Z When used, this flag will prevent the generation of `ANALYZE` statements. Without the -Z flag (default behavior), the `ANALYZE` statements will be generated.

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠:

Code

```
shp2pgsql -c -D -s 4269 -i -I shaperoads.shp myschema.roadstable > roads.sql
psql -d roadsdb -f roads.sql
```

UNIX ⊠⊠⊠ (pipe) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠:

Code

```
shp2pgsql shaperoads.shp myschema.roadstable | psql -d roadsdb
```

To load a directory of Shapefiles, run the loader once per file and derive the table name from the file name. On Windows `cmd.exe`, use a single percent sign interactively and double percent signs in a batch file:

Code

```
REM In a .bat file, use %%F. At an interactive cmd.exe prompt, use %F.
for %%F in ("C:\data\shapefiles\*.shp") do (
  shp2pgsql -d -D -s 4269 -I -W UTF-8 "%%~fF" "myschema.%%~nF" | psql -d roadsdb
)
```

If every Shapefile has the same attributes and data types and should be appended into one table, create the table from the first file and use `-a` for the remaining files.

4.8 ⌧⌧⌧⌧⌧⌧⌧

SQL ⌧⌧ shapefile ⌧⌧/⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧. SQL ⌧⌧⌧⌧⌧⌧⌧⌧ ⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧.

4.8.1 SQL ⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧

⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧ SQL ⌧⌧ (select) ⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧ ⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧:

Code

```
SELECT road_id,road_geom AS geom, road_name FROM roads;
```

Output

```
road_id | geom                                    | road_name
--------+-----------------------------------------+-----------
      1 | LINESTRING(191232 243118,191108 243242) | Jeff Rd
      2 | LINESTRING(189141 244158,189265 244817) | Geordie Rd
      3 | LINESTRING(192783 228138,192612 229814) | Paul St
      4 | LINESTRING(189412 252431,189631 259122) | Graeme Ave
      5 | LINESTRING(190131 224148,190871 228134) | Phil Tce
      6 | LINESTRING(198231 263418,198213 268322) | Dave Cres
      7 | LINESTRING(218421 284121,224123 241231) | Chris Way
(6 rows)
```

⌧⌧⌧, ⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧. ⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧, ⌧⌧⌧ ⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧ SQL ⌧⌧⌧⌧⌧⌧⌧⌧⌧. ⌧⌧⌧⌧⌧⌧⌧, ⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧.

ST_Intersects This function tells whether two geometries share any space.

= ⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧. ⌧⌧⌧⌧, 'POLYGON((0 0,1 1,1 0,0 0))' ⌧'POLYGON((0 0,1 1,1 0,0 0))' ⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧ (⌧⌧⌧⌧⌧⌧).

⌧⌧⌧⌧, ⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧. SQL ⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧, "ST_GeomFromText()" ⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧. ⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧ ⌧ 312 ⌧⌧⌧. ⌧⌧⌧⌧⌧⌧⌧⌧⌧:

Code

```
SELECT road_id, road_name
FROM roads
WHERE roads_geom='SRID=312;LINESTRING(191232 243118,191108 243242)'::geometry;
```

⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧"ROADS_GEOM" ⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧.

To check whether some of the roads passes in the area defined by a polygon:

Code

```
SELECT road_id, road_name
FROM roads
WHERE ST_Intersects(roads_geom, 'SRID=312;POLYGON((...))');
```

⌧⌧⌧" ⌧⌧⌧⌧ (map frame)" ⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧" ⌧⌧⌧⌧⌧ (frame-based)" ⌧⌧⌧⌧⌧⌧⌧⌧.

"&&" ⌧⌧⌧⌧⌧⌧⌧, ⌧⌧⌧⌧⌧⌧ BOX3D ⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧. ⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧, ⌧⌧⌧⌧⌧ ⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧.

Using a "BOX3D" object for the frame, such a query looks like this:

Code

```
SELECT roads_geom AS geom
FROM roads
WHERE
  roads_geom && ST_MakeEnvelope(191232, 243117,191232, 243119,312);
```

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ SRID 312 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

4.8.2 ⊠⊠⊠⊠⊠⊠

pgsql2shp ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠) ⊠⊠⊠⊠ shapefile ⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠:

Code

```
pgsql2shp [<options
>] <database
> [<schema
>.]<table>
```

Code

```
pgsql2shp [<options
>] <database
> <query>
```

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠:

-f <filename> ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

-h <host> ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

-p <port> ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

-P <password> ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

-u <user> ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

-g <geometry column> ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, shapefile ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

-b ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠ (非) ⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (cast) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

-r ⊠ (raw) ⊠⊠⊠⊠⊠. gid ⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

-m filename ⊠⊠⊠⊠ 10 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (remap) ⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. VERYLONGSYMBOL SHORTONE ANOTHERVERYLONGSYMBOL SHORTER ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

4.9 ⊠⊠⊠⊠⊠⊠⊠

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠” ⊠⊠⊠⊠⊠” ⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. PostgreSQL ⊠⊠⊠⊠⊠⊠ B-Tree, R-Tree, GiST ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

The B-tree index method commonly used for attribute data is not very useful for spatial data, since it only supports storing and querying data in a single dimension. Data such as geometry (which has 2 or more dimensions) requires an index method that supports range query across all the data dimensions. One of the key advantages of PostgreSQL for spatial data handling is that it offers several kinds of index methods which work well for multi-dimensional data: GiST, BRIN and SP-GiST indexes.

- GiST(Generalized Search Tree) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠” ⊠⊠⊠⊠⊠⊠⊠”, ” ⊠⊠⊠⊠⊠”, ” ⊠⊠⊠⊠⊠⊠⊠” ⊠⊠⊠⊠⊠⊠ GIS ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. PostGIS ⊠ GiST ⊠⊠⊠ GIS ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ R-Tree ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- **BRIN (Block Range Index)** indexes operate by summarizing the spatial extent of ranges of table records. Search is done via a scan of the ranges. BRIN is only appropriate for use for some kinds of data (spatially sorted, with infrequent or no update). But it provides much faster index create time, and much smaller index size.
- **SP-GiST (Space-Partitioned Generalized Search Tree)** is a generic index method that supports partitioned search trees such as quad-trees, k-d trees, and radix trees (tries).

Spatial indexes store only the bounding box of geometries. Spatial queries use the index as a **primary filter** to quickly determine a set of geometries potentially matching the query condition. Most spatial queries require a **secondary filter** that uses a spatial predicate function to test a more specific spatial condition. For more information on queying with spatial predicates see 5.2⊠.

See also the PostGIS Workshop section on spatial indexes, and the PostgreSQL manual.

4.9.1 GiST ⊠⊠⊠

GiST ⊠” ⊠⊠⊠⊠⊠⊠⊠⊠⊠” ⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. GIS ⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠ B-Tree ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠ GiST ⊠⊠⊠⊠⊠⊠.

GIS ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠).

” ⊠⊠” ⊠⊠⊠⊠ GiST ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠:

Code

```
CREATE INDEX [indexname] ON [tablename] USING GIST ( [geometryfield]);
```

⊠⊠⊠⊠⊠⊠ 2D ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠ PostGIS 2.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠ n ⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠:

Code

```
CREATE INDEX [indexname] ON [tablename] USING GIST ([geometryfield] gist_geometry_ops_nd);
```

Building a spatial index is a computationally intensive exercise. It also blocks write access to your table for the time it creates, so on a production system you may want to do in in a slower CONCURRENTLY-aware way:

Code

```
CREATE INDEX CONCURRENTLY [indexname] ON [tablename] USING GIST ( [geometryfield]);
```

After building an index, it is sometimes helpful to force PostgreSQL to collect table statistics, which are used to optimize query plans:

Code

```
VACUUM ANALYZE [table_name] [(column_name)];
```

4.9.2 GiST ☒☒☒

BRIN stands for "Block Range Index". It is a general-purpose index method provided by PostgreSQL. BRIN is a *lossy* index method, meaning that a secondary check is required to confirm that a record matches a given search condition (which is the case for all provided spatial indexes). It provides much faster index creation and much smaller index size, with reasonable read performance. Its primary purpose is to support indexing very large tables on columns which have a correlation with their physical location within the table. In addition to spatial indexing, BRIN can speed up searches on various kinds of attribute data structures (integer, arrays etc). For more information see the PostgreSQL manual.

GIS ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒, ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒ (☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒, ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒).

A BRIN index stores the bounding box enclosing all the geometries contained in the rows in a contiguous set of table blocks, called a *block range*. When executing a query using the index the block ranges are scanned to find the ones that intersect the query extent. This is efficient only if the data is physically ordered so that the bounding boxes for block ranges have minimal overlap (and ideally are mutually exclusive). The resulting index is very small in size, but is typically less performant for read than a GiST index over the same data.

Building a BRIN index is much less CPU-intensive than building a GiST index. It's common to find that a BRIN index is ten times faster to build than a GiST index over the same data. And because a BRIN index stores only one bounding box for each range of table blocks, it's common to use up to a thousand times less disk space than a GiST index.

You can choose the number of blocks to summarize in a range. If you decrease this number, the index will be bigger but will probably provide better performance.

For BRIN to be effective, the table data should be stored in a physical order which minimizes the amount of block extent overlap. It may be that the data is already sorted appropriately (for instance, if it is loaded from another dataset that is already sorted in spatial order). Otherwise, this can be accomplished by sorting the data by a one-dimensional spatial key. One way to do this is to create a new table sorted by the geometry values (which in recent PostGIS versions uses an efficient Hilbert curve ordering):

Code

```
CREATE TABLE table_sorted AS
SELECT * FROM table  ORDER BY geom;
```

Alternatively, data can be sorted in-place by using a GeoHash as a (temporary) index, and clustering on that index:

Code

```
CREATE INDEX idx_temp_geohash ON table
    USING btree (ST_GeoHash(ST_Transform(geom, 4326 ), 20));
CLUSTER table USING idx_temp_geohash;
```

" ☒☒" ☒☒☒☒ GiST ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒:

Code

```
CREATE INDEX [indexname] ON [tablename] USING BRIN ( [geome_col]);
```

□□□□□□ 2D □□□□□□□□□□□□. □□□□□□□□ PostGIS 2.0 □□□□□□□□□□ n □□□□□□□□ □□□□, □□□□□□□□□□□□□□□□:

Code

```
CREATE INDEX [indexname] ON [tablename]
USING BRIN ([geome_col] brin_geometry_inclusion_ops_3d);
```

You can also get a 4D-dimensional index using the 4D operator class:

Code

```
CREATE INDEX [indexname] ON [tablename]
USING BRIN ([geome_col] brin_geometry_inclusion_ops_4d);
```

The above commands use the default number of blocks in a range, which is 128. To specify the number of blocks to summarise in a range, use this syntax

Code

```
CREATE INDEX [indexname] ON [tablename]
USING BRIN ( [geome_col] ) WITH (pages_per_range = [number]);
```

Keep in mind that a BRIN index only stores one index entry for a large number of rows. If your table stores geometries with a mixed number of dimensions, it's likely that the resulting index will have poor performance. You can avoid this performance penalty by choosing the operator class with the least number of dimensions of the stored geometries

" □□" □□□□ GiST □□□□□□□□□□□□□□□□□□□□□□□□□□:

Code

```
CREATE INDEX [indexname] ON [tablename] USING BRIN ( [geog_col]);
```

□□□□□□ 2D □□□□□□□□□□□□. □□□□□□□□ PostGIS 2.0 □□□□□□□□□□ n □□□□□□□□ □□□□, □□□□□□□□□□□□□□□□:

Currently, only "inclusion support" is provided, meaning that just the &&, ~ and @ operators can be used for the 2D cases (for both `geometry` and `geography`), and just the &&& operator for 3D geometries. There is currently no support for kNN searches.

An important difference between BRIN and other index types is that the database does not maintain the index dynamically. Changes to spatial data in the table are simply appended to the end of the index. This will cause index search performance to degrade over time. The index can be updated by performing a `VACUUM`, or by using a special function `brin_summarize_new_values(regclass)`. For this reason BRIN may be most appropriate for use with data that is read-only, or only rarely changing. For more information refer to the manual.

To summarize using BRIN for spatial data:

- Index build time is very fast, and index size is very small.
- Index query time is slower than GiST, but can still be very acceptable.
- Requires table data to be sorted in a spatial ordering.
- Requires manual index maintenance.
- Most appropriate for very large tables, with low or no overlap (e.g. points), which are static or change infrequently.
- More effective for queries which return relatively large numbers of data records.

4.9.3 GiST ⊠⊠⊠

SP-GiST stands for ”Space-Partitioned Generalized Search Tree” and is a generic form of indexing for multi-dimensional data types that supports partitioned search trees, such as quad-trees, k-d trees, and radix trees (tries). The common feature of these data structures is that they repeatedly divide the search space into partitions that need not be of equal size. In addition to spatial indexing, SP-GiST is used to speed up searches on many kinds of data, such as phone routing, ip routing, substring search, etc. For more information see the PostgreSQL manual.

As it is the case for GiST indexes, SP-GiST indexes are lossy, in the sense that they store the bounding box enclosing spatial objects. SP-GiST indexes can be considered as an alternative to GiST indexes.

GIS ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠).

Code

```
CREATE INDEX [indexname] ON [tablename] USING SPGIST ( [geometryfield]);
```

⊠⊠⊠⊠⊠⊠⊠ 2D ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠ PostGIS 2.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ n ⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠:

Code

```
CREATE INDEX [indexname] ON [tablename] USING SPGIST ([geometryfield] ↩
    spgist_geometry_ops_3d);
```

Building a spatial index is a computationally intensive operation. It also blocks write access to your table for the time it creates, so on a production system you may want to do in in a slower CONCURRENTLY-aware way:

Code

```
CREATE INDEX CONCURRENTLY [indexname] ON [tablename] USING SPGIST ( [geometryfield]);
```

After building an index, it is sometimes helpful to force PostgreSQL to collect table statistics, which are used to optimize query plans:

Code

```
VACUUM ANALYZE [table_name] [(column_name)];
```

An SP-GiST index can accelerate queries involving the following operators:

- <<, &<, &>, >>, <<|, &<|, |&>, |>>, &&, @>, <@, and ~=, for 2-dimensional indexes,
- &/&, ~==, @>>, and <<@, for 3-dimensional indexes.

There is no support for kNN searches at the moment.

4.9.4 ⊠⊠⊠⊠⊠⊠⊠⊠

⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠. ⊠⊠⊠⊠⊠⊠ PostgreSQL ⊠⊠⊠ ⊠⊠⊠ GiST ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠:

- Examine the query plan and check your query actually computes the thing you need. An erroneous JOIN, either forgotten or to the wrong table, can unexpectedly retrieve table records multiple times. To get the query plan, execute with `EXPLAIN` in front of the query.

- Make sure statistics are gathered about the number and distributions of values in a table, to provide the query planner with better information to make decisions around index usage. **VACUUM ANALYZE** will compute both.

  You should regularly vacuum your databases anyways. Many PostgreSQL DBAs run **VACUUM** as an off-peak cron job on a regular basis.

- ☒☒☒, ☒☒☒☒☒☒☒☒☒☒☒☒☒☒ **SET ENABLE_SEQSCAN=OFF** ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒ ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒ ☒☒, ☒☒☒☒☒☒☒☒☒☒☒ B-Tree ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒, ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒ `ENABLE_SEQSCAN` ☒☒☒☒☒☒☒☒☒☒☒☒☒☒.

- ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒ (cost) ☒☒☒☒☒☒☒☒☒☒☒, postgresql.conf ☒☒☒ random_page_cost ☒☒☒☒☒☒☒☒☒”SET random_page_cost=#” ☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒☒ 4 ☒☒☒, 1 ☒☒ 2 ☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.

- If **SET ENABLE_SEQSCAN TO OFF;** does not help your query, the query may be using a SQL construct that the Postgres planner is not yet able to optimize. It may be possible to rewrite the query in a way that the planner is able to handle. For example, a subquery with an inline SELECT may not produce an efficient plan, but could possibly be rewritten using a LATERAL JOIN.

For more information see the Postgres manual section on Query Planning.

Chapter 5

Spatial Queries

The *raison d'etre* of spatial databases is to perform queries inside the database which would ordinarily require desktop GIS functionality. Using PostGIS effectively requires knowing what spatial functions are available, how to use them in queries, and ensuring that appropriate indexes are in place to provide good performance.

5.1 Determining Spatial Relationships

Spatial relationships indicate how two geometries interact with one another. They are a fundamental capability for querying geometry.

5.1.1 Dimensionally Extended 9-Intersection Model

According to the OpenGIS Simple Features Implementation Specification for SQL, "the basic approach to comparing two geometries is to make pair-wise tests of the intersections between the Interiors, Boundaries and Exteriors of the two geometries and to classify the relationship between the two geometries based on the entries in the resulting 'intersection' matrix."

In the theory of point-set topology, the points in a geometry embedded in 2-dimensional space are categorized into three sets:

Boundary

The boundary of a geometry is the set of geometries of the next lower dimension. For `POINT`s, which have a dimension of 0, the boundary is the empty set. The boundary of a `LINESTRING` is the two endpoints. For `POLYGON`s, the boundary is the linework of the exterior and interior rings.

Interior

The interior of a geometry are those points of a geometry that are not in the boundary. For `POINT`s, the interior is the point itself. The interior of a `LINESTRING` is the set of points between the endpoints. For `POLYGON`s, the interior is the areal surface inside the polygon.

Exterior

The exterior of a geometry is the rest of the space in which the geometry is embedded; in other words, all points not in the interior or on the boundary of the geometry. It is a 2-dimensional non-closed surface.

The Dimensionally Extended 9-Intersection Model (DE-9IM) describes the spatial relationship between two geometries by specifying the dimensions of the 9 intersections between the above sets for each geometry. The intersection dimensions can be formally represented in a 3x3 **intersection matrix**.

For a geometry *g* the *Interior*, *Boundary*, and *Exterior* are denoted using the notation *I(g)*, *B(g)*, and *E(g)*. Also, *dim(s)* denotes the dimension of a set *s* with the domain of `{0,1,2,F}`:

- `0` => point
- `1` => line
- `2` => area
- `F` => empty set

Using this notation, the intersection matrix for two geometries *a* and *b* is:

	Interior	**Boundary**	**Exterior**
Interior	*dim(I(a) ∩ I(b))*	*dim(I(a) ∩ B(b))*	*dim(I(a) ∩ E(b))*
Boundary	*dim(B(a) ∩ I(b))*	*dim(B(a) ∩ B(b))*	*dim(B(a) ∩ E(b))*
Exterior	*dim(E(a) ∩ I(b))*	*dim(E(a) ∩ B(b))*	*dim(E(a) ∩ E(b))*

The following overlapping polygons provide a concrete example. ST_Relate computes the matrix `212101212`; it matches the overlap pattern `T*T***T**` used by ST_Overlaps.

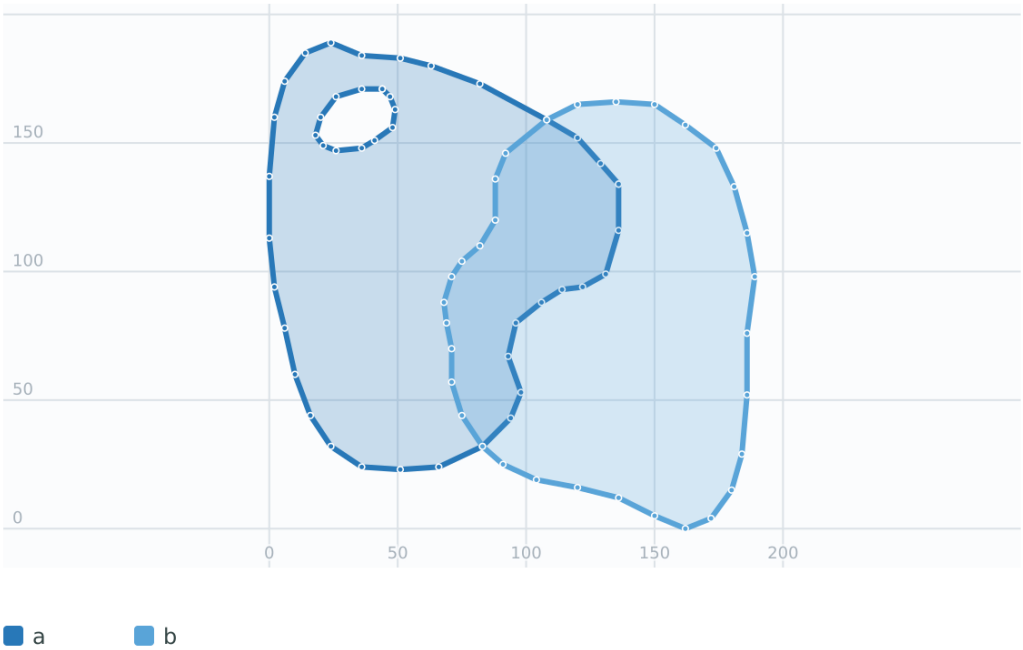
Code

```
WITH example AS (
  SELECT
    'POLYGON ((140 140,140 122,135 105,126 100,118 99,110 94,100 86,97 73,102 59,98 49,87 ↩
    38,70 30,55 29,40 30,28 38,20 50,14 66,10 84,6 100,4 119,4 143,6 166,10 180,18 191,28 ↩
    195,40 190,55 189,67 186,86 179,112 165,124 158,133 148,140 140),(48 177,40 177,30 ↩
    174,24 166,22 159,25 155,30 153,40 154,45 157,52 162,53 169,51 174,48 177))'::geometry ↩
    AS a,
    'POLYGON ((75 63,79 50,87 38,95 31,108 25,124 22,140 18,154 11,166 6,176 10,184 21,188 ↩
    35,190 58,190 82,193 104,190 121,185 139,178 154,166 163,154 171,139 172,124 171,112 ↩
    165,96 152,92 142,92 126,86 116,79 110,75 104,72 94,73 86,75 76,75 63))'::geometry AS b
), relation AS (
  SELECT a, b, ST_Relate(a, b) AS matrix
  FROM example
)
SELECT matrix,
       ST_RelateMatch(matrix, 'T*T***T**') AS matches_overlap_pattern,
       ST_Overlaps(a, b) AS overlaps
FROM relation;
```

Output

```
matrix   | matches_overlap_pattern | overlaps
-----------+-------------------------+----------
 212101212 | t                       | t
(1 row)
```

Figure



The executable query and generated overlay above are the visual source for the relation. The query below renders the nine cells of the same kind of matrix as a 3-by-3 set of examples. The final exterior/exterior cell is clipped to a finite viewport, since the true exterior/exterior intersection is unbounded.

Code

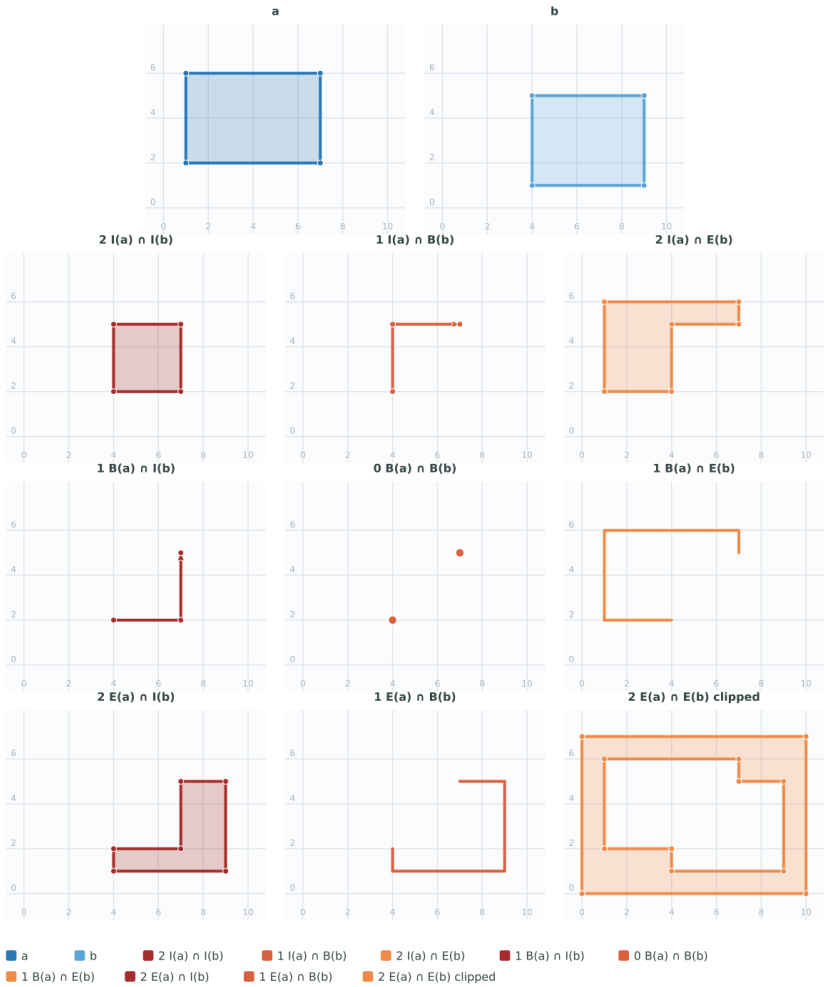
```
WITH example AS (
  SELECT
    ST_MakeEnvelope(0, 0, 6, 4) AS a,
    ST_MakeEnvelope(3, -1, 8, 3) AS b,
    ST_MakeEnvelope(-1, -2, 9, 5) AS viewport
), parts AS (
  SELECT
    a,
    b,
    ST_Boundary(a) AS ba,
    ST_Boundary(b) AS bb,
    viewport
  FROM example
)
SELECT
  ST_Relate(a, b) AS matrix,
  ST_AsText(ST_Normalize(a)) AS input_a,
  ST_AsText(ST_Normalize(b)) AS input_b,
  ST_AsText(ST_Normalize(ST_Intersection(a, b))) AS "2 I(a) b''ոb'' I(b)",
  ST_AsText(ST_Normalize(ST_Intersection(a, bb))) AS "1 I(a) b''ոb'' B(b)",
  ST_AsText(ST_Normalize(ST_Difference(a, b))) AS "2 I(a) b''ոb'' E(b)",
  ST_AsText(ST_Normalize(ST_Intersection(ba, b))) AS "1 B(a) b''ոb'' I(b)",
  ST_AsText(ST_Normalize(ST_Intersection(ba, bb))) AS "0 B(a) b''ոb'' B(b)",
  ST_AsText(ST_Normalize(ST_Collect(
      'LINESTRING(0 0,0 4,6 4,6 3)'::geometry,
      'LINESTRING(0 0,3 0)'::geometry
  ))) AS "1 B(a) b''ոb'' E(b)",
  ST_AsText(ST_Normalize(ST_Difference(b, a))) AS "2 E(a) b''ոb'' I(b)",
  ST_AsText(ST_Normalize(ST_Collect(
```

```
      'LINESTRING(3 -1,8 -1,8 3,6 3)'::geometry,
      'LINESTRING(3 -1,3 0)'::geometry
  ))) AS "1 E(a) b''nb'' B(b)",
  ST_AsText(ST_Normalize(ST_Difference(viewport, ST_Union(a, b))))
    AS "2 E(a) b''nb'' E(b) clipped"
FROM parts;
```

Output

```
-[ RECORD 1 ]----------+--------------------------------------------------------------------------------
matrix                 | 212101212
input_a                | POLYGON((0 0,0 4,6 4,6 0,0 0))
input_b                | POLYGON((3 -1,3 3,8 3,8 -1,3 -1))
2 I(a) b''nb'' I(b)          | POLYGON((3 0,3 3,6 3,6 0,3 0))
1 I(a) b''nb'' B(b)          | LINESTRING(3 0,3 3,6 3)
2 I(a) b''nb'' E(b)          | POLYGON((0 0,0 4,6 4,6 3,3 3,3 0,0 0))
1 B(a) b''nb'' I(b)          | LINESTRING(3 0,6 0,6 3)
0 B(a) b''nb'' B(b)          | MULTIPOINT((6 3),(3 0))
1 B(a) b''nb'' E(b)          | MULTILINESTRING((0 0,0 4,6 4,6 3),(0 0,3 0))
2 E(a) b''nb'' I(b)          | POLYGON((3 -1,3 0,6 0,6 3,8 3,8 -1,3 -1))
1 E(a) b''nb'' B(b)          | MULTILINESTRING((3 -1,8 -1,8 3,6 3),(3 -1,3 0))
2 E(a) b''nb'' E(b) clipped  | POLYGON((-1 -2,-1 5,9 5,9 -2,-1 -2),(0 0,3 0,3 -1,8 -1,8 3,6 3,6 4,0 4,0 0))
```

Figure



Reading from left to right and top to bottom, the intersection matrix is represented as the text string '**212101212**'.

For more information, refer to:

- OpenGIS Simple Features Implementation Specification for SQL (version 1.1, section 2.1.13.2)

- Wikipedia: Dimensionally Extended Nine-Intersection Model (DE-9IM)
- GeoTools: Point Set Theory and the DE-9IM Matrix

5.1.2 Named Spatial Relationships

To make it easy to determine common spatial relationships, the OGC SFS defines a set of *named spatial relationship predicates*. PostGIS provides these as the functions ST_Contains, ST_Crosses, ST_Disjoint, ST_Equals, ST_Intersects, ST_Overlaps, ST_Touches, ST_Within. It also defines the non-standard relationship predicates ST_Covers, ST_CoveredBy, and ST_ContainsProperly.

Spatial predicates are usually used as conditions in SQL `WHERE` or `JOIN` clauses. The named spatial predicates automatically use a spatial index if one is available, so there is no need to use the bounding box operator && as well. For example:

Code

```
SELECT city.name, state.name, city.geom
FROM city JOIN state ON ST_Intersects(city.geom, state.geom);
```

For more details and illustrations, see the PostGIS Workshop.

5.1.3 General Spatial Relationships

In some cases the named spatial relationships are insufficient to provide a desired spatial filter condition. These requirements can be expressed by computing the full DE-9IM intersection matrix with ST_Relate.

To test a particular spatial relationship, an **intersection matrix pattern** is used. This is the matrix representation augmented with the additional symbols `{T,*}`:

- `T` => intersection dimension is non-empty; i.e. is in `{0,1,2}`
- * => don't care

Using intersection matrix patterns, specific spatial relationships can be evaluated succinctly. The ST_Relate and ST_RelateMatch functions can both test these patterns.

These road segments share a line segment. ST_Crosses does not find this relationship for linear features, because it only returns `true` when their interiors intersect at a point. The pattern `1*1***1**` tests for a one-dimensional intersection instead.

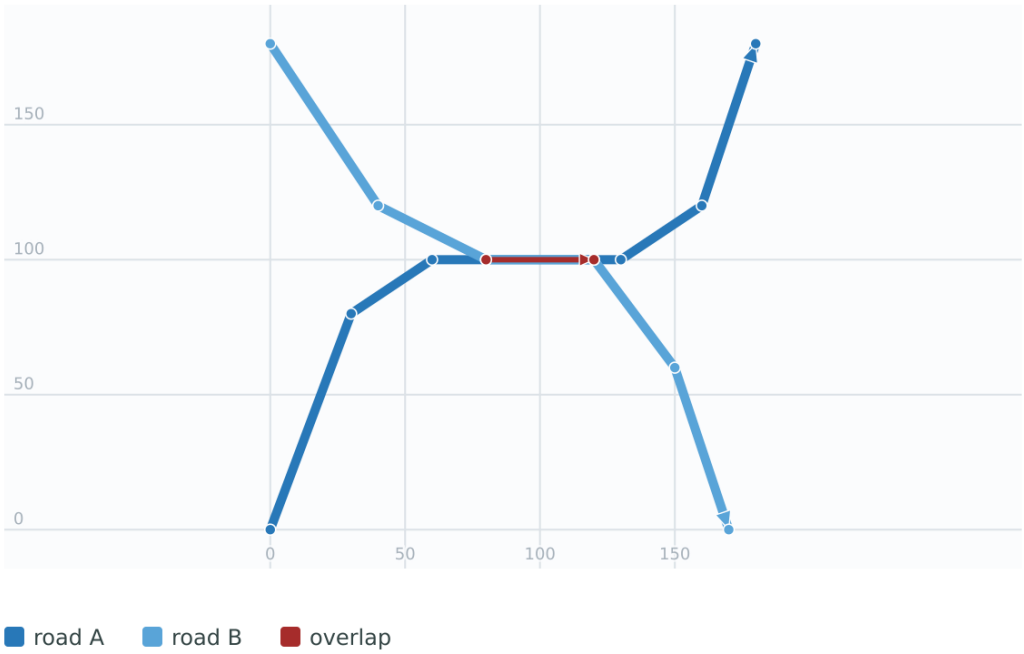
Code

```
WITH roads AS (
  SELECT
    'LINESTRING(10 10,40 90,70 110,140 110,170 130,190 190)'::geometry AS "road A",
    'LINESTRING(10 190,50 130,90 110,130 110,160 70,180 10)'::geometry AS "road B"
), relation AS (
  SELECT "road A", "road B", ST_Relate("road A", "road B") AS matrix
  FROM roads
)
SELECT
  ST_Intersection("road A", "road B") AS overlap,
  matrix,
  ST_Crosses("road A", "road B") AS "ST_Crosses",
  ST_RelateMatch(matrix, '1*1***1**') AS "matches 1*1***1**"
FROM relation;
```

Output

```
overlap           |  matrix   | ST_Crosses | matches 1*1***1**
--------------------------+-----------+------------+-------------------
 LINESTRING(90 110,130 110) | 1F1FF0102 | f          | t
(1 row)
```

Figure



The next example finds a wharf which runs from the lake interior onto its shoreline, with one endpoint on the shoreline. The other wharves provide nearby counterexamples in the figure. The pattern `102101FF2` expresses the complete required relationship in one test.

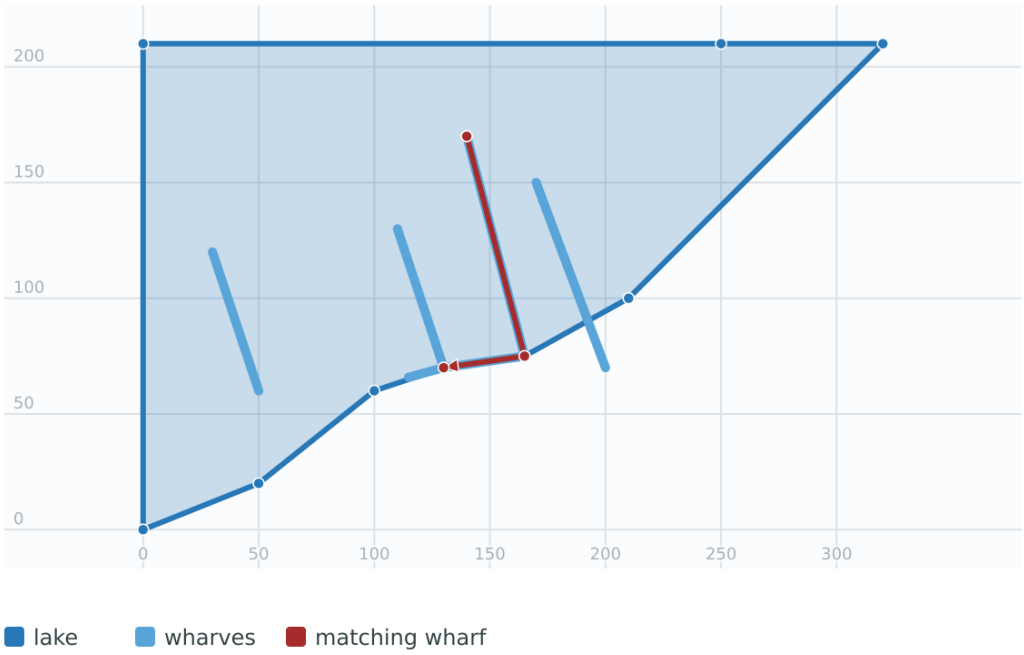
Code

```
WITH data AS (
  SELECT
    'POLYGON((-20 10,30 30,80 70,110 80,145 85,190 110,300 220,230 220,-20 220,-20 10))':: ↩
   geometry AS lake,
    'MULTILINESTRING((120 180,145 85,110 80),(10 130,30 70),(90 140,110  ↩
   80,94.81118881118876 75.83496503496498),(150 160,180 80))'::geometry AS wharves
), wharf AS (
  SELECT lake, (item).path[1] AS id, (item).geom
  FROM data CROSS JOIN LATERAL ST_Dump(wharves) AS item
), relation AS (
  SELECT id, geom, ST_Relate(lake, geom) AS matrix
  FROM wharf
)
SELECT
  id AS wharf,
  geom AS "matching wharf",
  matrix
FROM relation
WHERE ST_RelateMatch(matrix, '102101FF2')
ORDER BY id;
```

Output

```
wharf |           matching wharf           |  matrix
------+-------------------------------------+----------
     1 | LINESTRING(120 180,145 85,110 80) | 102101FF2
(1 row)
```

Figure



5.2 Using Spatial Indexes

When constructing queries using spatial conditions, for best performance it is important to ensure that a spatial index is used, if one exists (see 4.9☒). To do this, a spatial operator or index-aware function must be used in a `WHERE` or `ON` clause of the query.

Spatial operators include the bounding box operators (of which the most commonly used is &&; see 7.10.1☒ for the full list) and the distance operators used in nearest-neighbor queries (the most common being <->; see 7.10.2☒ for the full list.)

Index-aware functions automatically add a bounding box operator to the spatial condition. Index-aware functions include the named spatial relationship predicates ST_Contains, ST_ContainsProperly, ST_CoveredBy, ST_Covers, ST_Crosses, ST_Intersects, ST_Overlaps, ST_Touches, ST_Within, ST_Within, and ST_3DIntersects, and the distance predicates ST_DWithin, ST_DFullyWithin, ST_3DDFullyWithin, and ST_3DDWithin .)

Functions such as ST_Distance do *not* use indexes to optimize their operation. For example, the following query would be quite slow on a large table:

Code

```
SELECT geom
FROM geom_table
WHERE ST_Distance(geom, 'SRID=312;POINT(100000 200000)') < 100
```

This query selects all the geometries in `geom_table` which are within 100 units of the point (100000, 200000). It will be slow because it is calculating the distance between each point in the table and the specified point, ie. one `ST_Distance()` calculation is computed for **every** row in the table.

The number of rows processed can be reduced substantially by using the index-aware function ST_DWithin:

Code

```
SELECT geom
FROM geom_table
WHERE ST_DWithin(geom, 'SRID=312;POINT(100000 200000)', 100)
```

This query selects the same geometries, but it does it in a more efficient way. This is enabled by `ST_DWithin()` using the && operator internally on an expanded bounding box of the query geometry. If there is a spatial index on `geom`, the query planner will recognize that it can use the index to reduce the number of rows scanned before calculating the distance. The spatial index allows retrieving only records with geometries whose bounding boxes overlap the expanded extent and hence which *might* be within the required distance. The actual distance is then computed to confirm whether to include the record in the result set.

For more information and examples see the PostGIS Workshop.

5.3 Examples of Spatial SQL

The examples in this section make use of a table of linear roads, and a table of polygonal municipality boundaries. The definition of the `bc_roads` table is:

Output

```
Column    | Type              | Description
----------+-------------------+-------------------
gid       | integer           | Unique ID
name      | character varying | Road Name
geom      | geometry          | Location Geometry (Linestring)
```

The definition of the `bc_municipality` table is:

Output

```
Column   | Type              | Description
---------+-------------------+-------------------
gid      | integer           | Unique ID
code     | integer           | Unique ID
name     | character varying | City / Town Name
geom     | geometry          | Location Geometry (Polygon)
```

1. *What is the total length of all roads, expressed in kilometers?*

   You can answer this question with a very simple piece of SQL:

   Code

   ```
   SELECT sum(ST_Length(geom))/1000 AS km_roads FROM bc_roads;
   ```

   Output

   ```
   km_roads
   ------------------
   70842.1243039643
   ```

2. *How large is the city of Prince George, in hectares?*

   This query combines an attribute condition (on the municipality name) with a spatial calculation (of the polygon area):

Code

```
SELECT
  ST_Area(geom)/10000 AS hectares
FROM bc_municipality
WHERE name = 'PRINCE GEORGE';
```

Output

```
hectares
------------------
32657.9103824927
```

3. *What is the largest municipality in the province, by area?*

   This query uses a spatial measurement as an ordering value. There are several ways of approaching this problem, but the most efficient is below:

   Code

   ```
   SELECT
     name,
     ST_Area(geom)/10000 AS hectares
   FROM bc_municipality
   ORDER BY hectares DESC
   LIMIT 1;
   ```

   Output

   ```
   name           | hectares
   ---------------+-----------------
   TUMBLER RIDGE  | 155020.02556131
   ```

   Note that in order to answer this query we have to calculate the area of every polygon. If we were doing this a lot it would make sense to add an area column to the table that could be indexed for performance. By ordering the results in a descending direction, and them using the PostgreSQL "LIMIT" command we can easily select just the largest value without using an aggregate function like MAX().

4. *What is the length of roads fully contained within each municipality?*

   This is an example of a "spatial join", which brings together data from two tables (with a join) using a spatial interaction ("contained") as the join condition (rather than the usual relational approach of joining on a common key):

   Code

   ```
   SELECT
     m.name,
     sum(ST_Length(r.geom))/1000 as roads_km
   FROM bc_roads AS r
   JOIN bc_municipality AS m
     ON ST_Contains(m.geom, r.geom)
   GROUP BY m.name
   ORDER BY roads_km;
   ```

   Output

   ```
   name                         | roads_km
   -----------------------------+----------
   SURREY                       | 1539.476
   VANCOUVER                    | 1450.331
   LANGLEY DISTRICT             |  833.793
   ```

```
BURNABY                       |  773.769
PRINCE GEORGE                 |  694.376
...
```

This query takes a while, because every road in the table is summarized into the final result (about 250K roads for the example table). For smaller datasets (several thousand records on several hundred) the response can be very fast.

5. *Create a new table with all the roads within the city of Prince George.*

   This is an example of an "overlay", which takes in two tables and outputs a new table that consists of spatially clipped or cut resultants. Unlike the "spatial join" demonstrated above, this query creates new geometries. An overlay is like a turbo-charged spatial join, and is useful for more exact analysis work:

   Code

   ```
   CREATE TABLE pg_roads as
   SELECT
     ST_Intersection(r.geom, m.geom) AS intersection_geom,
     ST_Length(r.geom) AS rd_orig_length,
     r.*
   FROM bc_roads AS r
   JOIN bc_municipality AS m
     ON ST_Intersects(r.geom, m.geom)
   WHERE
     m.name = 'PRINCE GEORGE';
   ```

6. *What is the length in kilometers of "Douglas St" in Victoria?*

   Code

   ```
   SELECT
     sum(ST_Length(r.geom))/1000 AS kilometers
   FROM bc_roads r
   JOIN bc_municipality m
     ON ST_Intersects(m.geom, r.geom
   WHERE
     r.name = 'Douglas St'
     AND m.name = 'VICTORIA';
   ```

   Output

   ```
   kilometers
   ------------------
   4.89151904172838
   ```

7. *What is the largest municipality polygon that has a hole?*

   Code

   ```
   SELECT gid, name, ST_Area(geom) AS area
   FROM bc_municipality
   WHERE ST_NRings(geom)
   > 1
   ORDER BY area DESC LIMIT 1;
   ```

   Output

   ```
   gid  | name         | area
   -----+--------------+------------------
   12   | SPALLUMCHEEN | 257374619.430216
   ```

Chapter 6

⊠⊠⊠⊠⊠⊠

6.1 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠

6.1.1 ⊠⊠⊠⊠⊠

Current PostgreSQL versions suffer from a query optimizer weakness regarding TOAST tables. TOAST tables are a kind of ”extension room” used to store large (in the sense of data size) values that do not fit into normal data pages (like long texts, images or complex geometries with lots of vertices), see the PostgreSQL Documentation for TOAST for more information).

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, (⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠) ⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠ TOAST ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠ ⊠, ⊠⊠⊠⊠⊠⊠⊠⊠ 80 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ 3 ⊠⊠⊠⊠⊠⊠⊠⊠, TOAST ⊠⊠⊠⊠ 8,225 ⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠.

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ && ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ 3 ⊠⊠ 80 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠ GiST ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, && ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠, ⊠⊠⊠⊠ TOAST ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ”EXPLAIN ANALYZE” PostgreSQL ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, PostgreSQL ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠: http://archives.postgresql.org/pgsql-performance/2005-02/msg00030.php

and newer thread on PostGIS https://lists.osgeo.org/pipermail/postgis-devel/2017-June/026209.html

6.1.2 ⊠⊠⊠⊠

PostgreSQL ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ TOAST ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠:

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠”SET enable_seqscan TO off;” ⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠ GiST ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠”SET enable_seqscan TO on;” ⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠” ⊠⊠” ⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠:

Code

```
SELECT AddGeometryColumn('myschema', 'mytable', 'bbox', '4326', 'GEOMETRY', '2');
UPDATE mytable SET bbox = ST_Envelope(ST_Force2D(geom));
```

⊠⊠ geom_column ⊠⊠ bbox ⊠⊠⊠ && ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠:

Code

```
SELECT geom_column
FROM mytable
WHERE bbox && ST_SetSRID('BOX3D(0 0,1 1)'::box3d,4326);
```

⊠⊠, mytable ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, bbox ⊠" ⊠⊠⊠" ⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠ (trigger) ⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ bbox ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ UPDATE ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

6.2 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠

The PostgreSQL CLUSTER command physically reorders table rows according to an index. For a mostly read-only table whose queries frequently use the same spatial index, this can reduce I/O and improve cache locality.

PostgreSQL supports clustering on GiST indices, including indices on nullable geometry columns. To cluster a table using a geometry index:

Code

```
CLUSTER my_table USING my_geom_index;
```

Output

```
CLUSTER
```

6.3 ⊠⊠⊠⊠⊠⊠⊠

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ 3D ⊠⊠ 4D ⊠⊠⊠⊠⊠⊠⊠, ⊠⊠ 2D ⊠⊠⊠⊠⊠⊠⊠⊠ OpenGIS ⊠⊠ ST_AsText() ⊠ ⊠ ST_AsBinary() ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠ ST_Force2D() ⊠⊠⊠⊠ ⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

Code

```
UPDATE mytable SET geom = ST_Force2D(geom);
VACUUM FULL ANALYZE mytable;
```

AddGeometryColumn() ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. geometry_columns ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ WHERE ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠"VACUUM;" ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠, "WHERE dimension(the_geom)>2" ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ 2D ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠.

Chapter 7

PostGIS Reference

☒☒☒☒☒☒☒ PostGIS ☒☒☒☒☒☒☒☒☒☒☒☒☒☒, ☒☒☒☒☒☒☒☒☒☒ PostGIS ☒☒☒☒☒☒☒☒☒☒ ☒☒☒☒☒☒☒☒.

> ☒☒
> PostGIS ☒☒☒☒☒☒☒☒☒☒ SQL-MM-☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒, ☒☒☒☒☒☒ ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒ Spatial Type (ST) ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒. ☒ ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒, ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒ ☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒ ST_ ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒ ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.

7.1 PostgreSQL PostGIS Geometry/Geography/Box ☒☒

7.1.1 box2d

box2d — The type representing a 2-dimensional bounding box.

☒☒

box3d ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒ postgis ☒☒☒☒☒☒☒☒☒☒☒. ST_3DExtent ☒ box3d ☒☒☒☒☒☒☒☒.

The representation contains the values `xmin, ymin, xmax, ymax`. These are the minimum and maximum values of the X and Y extents.

`box2d` objects have a text representation which looks like `BOX(1 2,5 6)`.

☒☒☒☒☒

☒☒.

☒☒☒☒☒	☒☒
box3d	☒☒☒
geometry	☒☒☒

☒☒

13.7☒

7.1.2 box3d

box3d — The type representing a 3-dimensional bounding box.

☒☒

box3d ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒ postgis ☒☒☒☒☒☒☒☒☒☒. ST_3DExtent ☒ box3d ☒☒☒☒☒☒☒☒.

The representation contains the values `xmin, ymin, zmin, xmax, ymax, zmax`. These are the minimum and maximum values of the X, Y and Z extents.

`box3d` objects have a text representation which looks like `BOX3D(1 2 3,5 6 5)`.

☒☒☒☒☒

☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.

☒☒☒☒☒	☒☒
box	☒☒☒
box2d	☒☒☒
geometry	☒☒☒

☒☒

13.7☒

7.1.3 geometry

geometry — geography ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.

☒☒

geography ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.

All spatial operations on geometry use the units of the Spatial Reference System the geometry is in.

☒☒☒☒☒

☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.

☒☒☒☒☒	☒☒
box	☒☒☒
box2d	☒☒☒
box3d	☒☒☒

⊠⊠⊠⊠⊠	⊠⊠
bytea	⊠⊠⊠
geography	⊠⊠⊠
text	⊠⊠⊠

⊠⊠

4.1⊠, 4.3⊠

7.1.4 geometry_dump

geometry_dump — A composite type used to describe the parts of complex geometry.

⊠⊠

`geometry_dump` is a composite data type containing the fields:

- `geom` - a geometry representing a component of the dumped geometry. The geometry type depends on the originating function.
- `path[]` - an integer array that defines the navigation path within the dumped geometry to the `geom` component. The path array is 1-based (i.e. `path[1]` is the first element.)

It is used by the `ST_Dump*` family of functions as an output type to explode a complex geometry into its constituent parts.

⊠⊠

13.6⊠

7.1.5 geography

geography — The type representing spatial features with geodetic (ellipsoidal) coordinate systems.

⊠⊠

geography ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

Spatial operations on the geography type provide more accurate results by taking the ellipsoidal model into account.

⊠⊠⊠⊠⊠

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠⊠⊠⊠	⊠⊠
geometry	⊠⊠⊠

☒☒

4.3☒, 4.3☒

7.2 ☒☒☒☒

7.2.1 AddGeometryColumn

AddGeometryColumn — ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.

☒☒

text **AddGeometryColumn**(varchar table_name, varchar column_name, integer srid, varchar type, integer dimension, boolean use_typmod=true);
text **AddGeometryColumn**(varchar schema_name, varchar table_name, varchar column_name, integer srid, varchar type, integer dimension, boolean use_typmod=true);
text **AddGeometryColumn**(varchar catalog_name, varchar schema_name, varchar table_name, varchar column_name, integer srid, varchar type, integer dimension, boolean use_typmod=true);

☒☒

☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒. `schema_name` ☒☒☒☒☒☒☒☒☒☒☒☒☒. `srid` ☒☒☒☒☒☒☒☒☒☒ SPATIAL_REF_SYS ☒☒☒☒☒☒☒☒☒☒☒☒☒. `type` ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒, ☒☒☒☒'POLYGON' ☒☒'MULTILINESTRING' ☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒ (☒☒☒☒ search_path ☒☒☒☒☒☒) ☒☒☒ SRID, ☒☒☒☒☒☒☒☒, ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.

☒☒☒☒: 2.0.0 ☒☒☒☒ geometry_columns ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒ geometry_columns ☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒, ☒☒☒ PostgreSQL ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒ WGS84 POINT ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒: `ALTER TABLE some_table ADD COLUMN geom geometry(Point,4326);`

☒☒☒☒: 2.0.0 ☒☒. ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒, ☒☒☒☒ `use_typmod` ☒☒☒☒☒, ☒☒☒☒☒☒☒☒☒☒.

☒☒☒☒: 2.0.0 ☒☒. ☒☒☒☒☒☒☒☒ geometry_columns ☒☒☒☒☒☒☒☒☒☒, ☒☒ typmod ☒☒☒☒☒☒☒☒☒☒☒☒☒☒, ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒ typmod ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒ geometry_columns ☒☒☒☒☒☒☒☒☒☒☒, ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒ typmod ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒. 4.6.3☒☒☒☒☒☒☒☒☒.

This method implements the OGC Simple Features Implementation Specification for SQL 1.1.

This function supports 3d and will not drop the z-index.

This method supports Circular Strings and Curves.

☒☒☒☒: 2.0.0 ☒☒. use_typmod ☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒ typmod ☒☒☒☒☒☒☒☒☒.

☒☒

This example creates a schema to hold data.

Code

```
CREATE SCHEMA my_schema;
```

Create a simple PostgreSQL table in the schema.

Code

```
CREATE TABLE my_schema.my_spatial_table (id serial);
```

Describing the table shows a single `id` column.

Code

```
\d my_schema.my_spatial_table
```

Output

```
Table "my_schema.my_spatial_table"
 Column |  Type   |                                 Modifiers
--------+---------+------------------------------------------------------------------------ ↩

 id     | integer | not null default nextval('my_schema.my_spatial_table_id_seq'::regclass)
```

Add a spatial column to the table.

Code

```
SELECT AddGeometryColumn('my_schema', 'my_spatial_table', 'geom', 4326, 'POINT', 2);
```

Add a point column using the old constraint-based behavior.

Code

```
SELECT AddGeometryColumn('my_schema', 'my_spatial_table', 'geom_c', 4326, 'POINT', 2, false ↩
    );
```

Add a CurvePolygon column using the old constraint-based behavior.

Code

```
SELECT AddGeometryColumn('my_schema', 'my_spatial_table', 'geomcp_c', 4326, 'CURVEPOLYGON', ↩
     2, false);
```

Output

```
my_schema.my_spatial_table.geomcp_c SRID:4326 TYPE:CURVEPOLYGON DIMS:2
```

Describe the table again to see the new geometry columns.

Code

```
\d my_schema.my_spatial_table
```

Output

```
Table "my_schema.my_spatial_table"

Columns:
  id       integer               not null default nextval('my_schema.my_spatial_table_id_seq'::regclass)
  geom     geometry(Point,4326)
  geom_c   geometry
  geomcp_c geometry

Check constraints:
```

```
    "enforce_dims_geom_c" CHECK (st_ndims(geom_c) = 2)
    "enforce_dims_geomcp_c" CHECK (st_ndims(geomcp_c) = 2)
    "enforce_geotype_geom_c" CHECK (geometrytype(geom_c) = 'POINT'::text OR geom_c IS NULL)
    "enforce_geotype_geomcp_c" CHECK (geometrytype(geomcp_c) = 'CURVEPOLYGON'::text OR geomcp_c IS NULL)
    "enforce_srid_geom_c" CHECK (st_srid(geom_c) = 4326)
    "enforce_srid_geomcp_c" CHECK (st_srid(geomcp_c) = 4326)
```

The `geometry_columns` view also registers the new columns.

Code

```
SELECT f_geometry_column As col_name, type, srid, coord_dimension As ndims
FROM geometry_columns
WHERE f_table_name = 'my_spatial_table' AND f_table_schema = 'my_schema';
```

Output

```
col_name |     type      | srid | ndims
----------+--------------+------+-------
 geom     | Point        | 4326 |     2
 geom_c   | Point        | 4326 |     2
 geomcp_c | CurvePolygon | 4326 |     2
```

☒☒

DropGeometryColumn, DropGeometryTable, 4.6.2☒, 4.6.3☒

7.2.2 DropGeometryColumn

DropGeometryColumn — ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.

☒☒

text **DropGeometryColumn**(varchar table_name, varchar column_name);
text **DropGeometryColumn**(varchar schema_name, varchar table_name, varchar column_name);
text **DropGeometryColumn**(varchar catalog_name, varchar schema_name, varchar table_name, varchar column_name);

☒☒

Removes a geometry column from a spatial table. Note that schema_name will need to match the f_table_schema field shown by the `geometry_columns` view.

☒☒
`DropGeometryColumn` is retained for backward compatibility and is not recommended for new applications. Use the standard **ALTER TABLE ... DROP COLUMN** command instead. The `geometry_columns` view updates automatically from the system catalogs.

This method implements the OGC Simple Features Implementation Specification for SQL 1.1.

This function supports 3d and will not drop the z-index.

This method supports Circular Strings and Curves.

Changed: 2.0.0. Retained for backward compatibility after `geometry_columns` became a view of the system catalogs.

□□

Use standard SQL to drop the geometry column. The `geometry_columns` view updates automatically.

Code

```
ALTER TABLE my_schema.my_spatial_table DROP COLUMN geom;
```

The legacy compatibility function performs the equivalent operation.

Code

```
SELECT DropGeometryColumn ('my_schema', 'my_spatial_table', 'geom');
```

Output

```
my_schema.my_spatial_table.geom effectively removed.
```

□□

AddGeometryColumn, DropGeometryTable, 4.6.2□

7.2.3 DropGeometryTable

DropGeometryTable — □□□□ geometry_columns □□□□□□□□□□□.

□□

boolean **DropGeometryTable**(varchar table_name);
boolean **DropGeometryTable**(varchar schema_name, varchar table_name);
boolean **DropGeometryTable**(varchar catalog_name, varchar schema_name, varchar table_name);

□□

□□□□ geometry_columns □□□□□□□□□□□□□□□□□□□. □□: □□□□□□□□□□□□□□□□□□□□□□□□ (schema-aware) pgSQL □□□□□□ current_schema() □□□□□□□□□.



□□
`DropGeometryTable` is retained for backward compatibility and is not recommended for new applications. Use the standard **DROP TABLE** command instead. The `geometry_columns` view updates automatically from the system catalogs.

Changed: 2.0.0. Retained for backward compatibility after `geometry_columns` became a view of the system catalogs.

□□

Use standard SQL to drop a table containing geometry columns. The `geometry_columns` view updates automatically.

Code

```
DROP TABLE my_schema.my_spatial_table;
```

The legacy compatibility function performs the equivalent operation.

Code

```
SELECT DropGeometryTable ('my_schema', 'my_spatial_table');
```

Output

```
my_schema.my_spatial_table dropped.
```

□□

AddGeometryColumn, DropGeometryColumn, 4.6.2□

7.2.4 Find_SRID

Find_SRID — Returns the SRID defined for a geometry column.

□□

integer **Find_SRID**(varchar a_schema_name, varchar a_table_name, varchar a_geomfield_name);

□□

Returns the integer SRID of the specified geometry column by searching through the GEOMETRY_COLUMNS table. If the geometry column has not been properly added (e.g. with the AddGeometryColumn function), this function will not work.

□□

Code

```
SELECT Find_SRID('public', 'tiger_us_state_2007', 'geom_4269');
```

Output

```
4269
```

□□

ST_SRID

7.2.5 Populate_Geometry_Columns

Populate_Geometry_Columns — Ensures geometry columns are defined with type modifiers or have appropriate spatial constraints.

⊠⊠

text **Populate_Geometry_Columns**(boolean use_typmod=true);
int **Populate_Geometry_Columns**(oid relation_oid, boolean use_typmod=true);

⊠⊠

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠ `geometry_columns` ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠ `use_typmod=false` ⊠⊠⊠⊠

⊠⊠, ⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, `use_typmod=false` ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ 3 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠, ⊠⊠⊠⊠ ⊠⊠⊠⊠⊠ 3 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠:

- `enforce_dims_the_geom` - ensures every geometry has the same dimension (see ST_NDims)
- `enforce_geotype_the_geom` - ensures every geometry is of the same type (see GeometryType)
- `enforce_srid_the_geom` - ensures every geometry is in the same projection (see ST_SRID)

`oid` ⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ SRID, ⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠ geometry_columns ⊠⊠⊠⊠⊠⊠⊠ ⊠⊠. ⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

`oid` ⊠⊠⊠⊠⊠⊠ `oid` ⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠ `geometry_columns` ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠ SRID, ⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠ geometry_columns ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (truncate) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ `geometry_columns` ⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ `geometry_columns` ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠.

1.4.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠⊠⊠: 2.0.0 ⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠ `use_typmod` ⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠⊠⊠: 2.0.0 ⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ `use_typmod` ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

This example populates geometry column metadata using typmod behavior. For this to work, there must be existing data.

Code

```
CREATE TABLE public.myspatial_table(gid serial, geom geometry);
INSERT INTO myspatial_table(geom) VALUES(ST_GeomFromText('LINESTRING(1 2,3 4)', 4326) );
SELECT Populate_Geometry_Columns('public.myspatial_table'::regclass);
```

Output

```
1
```

Describe the table to see the typmod-based geometry column.

Code

```
\d myspatial_table
```

Output

```
Table "public.myspatial_table"

Columns:
  gid  integer                    not null default nextval('myspatial_table_gid_seq':: ↩
    regclass)
  geom geometry(LineString,4326)
```

This example changes the geometry columns to use constraints if they are not typmod or already constrained. For this to work, there must be existing data.

Code

```
CREATE TABLE public.myspatial_table_cs(gid serial, geom geometry);
INSERT INTO myspatial_table_cs(geom) VALUES(ST_GeomFromText('LINESTRING(1 2,3 4)', 4326) );
SELECT Populate_Geometry_Columns('public.myspatial_table_cs'::regclass, false);
```

Output

```
1
```

Describe the table to see the constraint-based geometry column.

Code

```
\d myspatial_table_cs
```

Output

```
Table "public.myspatial_table_cs"
 Column |   Type   |                            Modifiers
--------+----------+-------------------------------------------------------------------
 gid    | integer  | not null default nextval('myspatial_table_cs_gid_seq'::regclass)
 geom   | geometry |
Check constraints:
    "enforce_dims_geom" CHECK (st_ndims(geom) = 2)
    "enforce_geotype_geom" CHECK (geometrytype(geom) = 'LINESTRING'::text OR geom IS NULL)
    "enforce_srid_geom" CHECK (st_srid(geom) = 4326)
```

7.2.6 UpdateGeometrySRID

UpdateGeometrySRID — Updates the SRID of all features in a geometry column, and the table metadata.

⊠⊠

text **UpdateGeometrySRID**(varchar table_name, varchar column_name, integer srid);
text **UpdateGeometrySRID**(varchar schema_name, varchar table_name, varchar column_name, integer srid);
text **UpdateGeometrySRID**(varchar catalog_name, varchar schema_name, varchar table_name, varchar column_name, integer srid);

⊠⊠

⊠⊠⊠⊠⊠⊠⊠, geometry_columns ⊠⊠⊠⊠⊠⊠ srid ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ SRID ⊠⊠⊠⊠⊠⊠. ⊠⊠: ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ schema-aware pgsql installations ⊠⊠⊠⊠⊠ current_schema() ⊠⊠

This function supports 3d and will not drop the z-index.

This method supports Circular Strings and Curves.

⊠⊠

Insert geometries into roads table with a SRID set already using EWKT format:

Code

```
COPY roads (geom) FROM STDIN;
SRID=4326;LINESTRING(0 0, 10 10)
SRID=4326;LINESTRING(10 10, 15 0)
\.
```

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ SRID ⊠⊠⊠⊠⊠ SRID ⊠ 4326 ⊠⊠⊠⊠⊠⊠⊠⊠⊠:

Code

```
SELECT UpdateGeometrySRID('roads', 'geom', 4326);
```

⊠⊠⊠⊠⊠⊠⊠ DDL ⊠⊠⊠⊠⊠⊠⊠⊠⊠:

Code

```
ALTER TABLE roads
ALTER COLUMN geom TYPE geometry(MULTILINESTRING, 4326)
  USING ST_SetSRID(geom, 4326);
```

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (⊠⊠'unknown' ⊠⊠) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, DDL ⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠ PostGIS ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

Code

```
ALTER TABLE roads
ALTER COLUMN geom TYPE geometry(MULTILINESTRING, 3857) USING ST_Transform(ST_SetSRID(geom, ↩
    4326), 3857) ;
```

⊠⊠

UpdateRasterSRID, ST_SetSRID, ST_Transform

7.3 □□□□□ (constructor)

7.3.1 ST_Collect

ST_Collect — Creates a GeometryCollection or Multi* geometry from a set of geometries.

□□

geometry **ST_Collect**(geometry g1, geometry g2);
geometry **ST_Collect**(geometry[] g1_array);
geometry **ST_Collect**(geometry set g1field);

□□

Collects geometries into a geometry collection. The result is either a Multi* or a GeometryCollection, depending on whether the input geometries have the same or different types (homogeneous or heterogeneous). The input geometries are left unchanged within the collection.

Variant 1: accepts two input geometries

Variant 2: accepts an array of geometries

Variant 3: aggregate function accepting a rowset of geometries.



□□
`ST_Collect` does not dissolve boundaries or resolve overlaps. Collecting overlapping polygons can produce an invalid MultiPolygon. If the result must be a valid polygonal geometry with overlaps merged, use ST_Union instead. `ST_Union` also splits linestrings at intersections and may return a single geometry after dissolving boundaries.

If any input geometry is a collection (Multi* or GeometryCollection), `ST_Collect` returns a GeometryCollection, since that is the only type which can contain nested collections. To prevent this, use ST_Dump in a subquery to expand input collections to their atomic elements (see example below).

1.4.0 □□□□□□□□□□□□□. □□□□□□ ST_MakeLine □□□□□□□□□□□□□□□□□. □□□□□□□□ □□□□□□□□□□ ST_MakeLine □□□□□□□□□□□□.

 This function supports 3d and will not drop the z-index.

 This method supports Circular Strings and Curves.

□□

Two-input variant.

Collect 2D points.

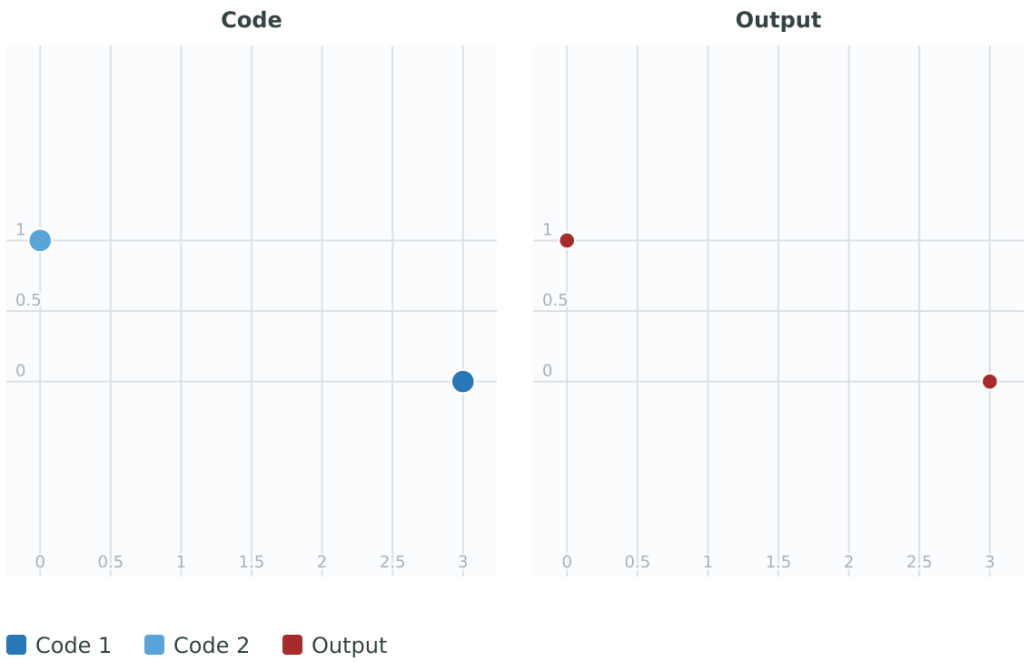
Code

```
SELECT ST_Collect(
    'POINT(1 2)',
    'POINT(-2 3)' );
```

Output

```
MULTIPOINT((1 2),(-2 3))
```

Figure



Collect 3D points.

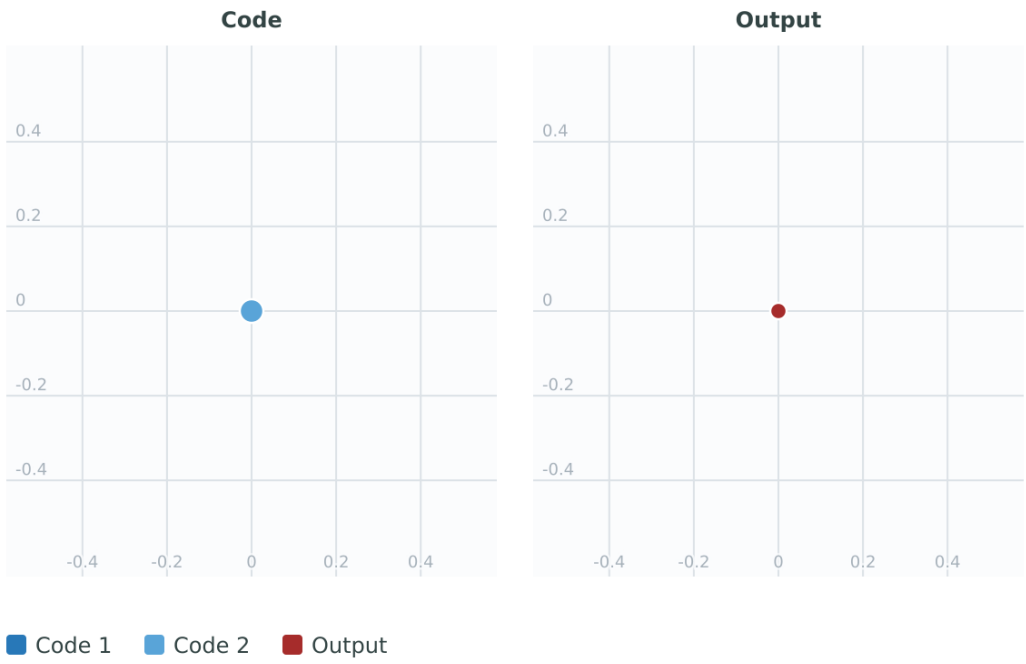
Code

```
SELECT ST_Collect(
        'POINT(1 2 3)',
        'POINT(1 2 4)' );
```

Output

```
MULTIPOINT(1 2 3,1 2 4)
```

Figure

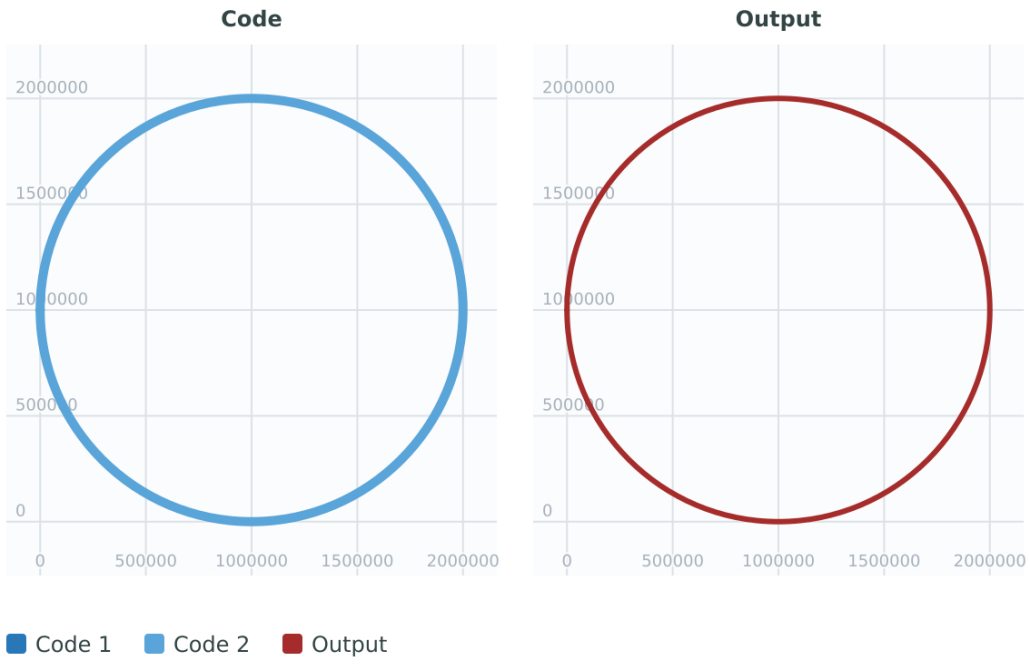


Collect curves.

Code

```
SELECT ST_Collect(
        'CIRCULARSTRING(220268 150415,220227 150505,220227 150406)',
        'CIRCULARSTRING(220227 150406,2220227 150407,220227 150406)');
```

Figure



Array variant.

Using an array constructor for a subquery.

Code

```
SELECT ST_Collect(ARRAY(SELECT geom FROM sometable));
```

Using an array constructor for values.

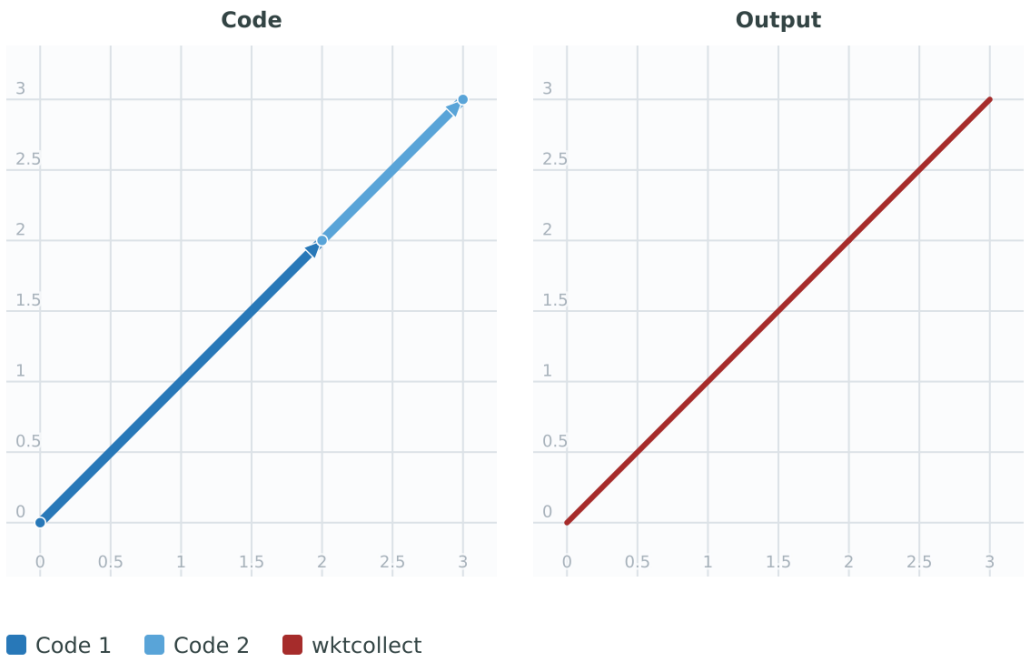
Code

```
SELECT ST_Collect(
        ARRAY[ 'LINESTRING(1 2,3 4)'::geometry,
            'LINESTRING(3 4,4 5)'::geometry ] ) As wktcollect;
```

Output

```
MULTILINESTRING((1 2,3 4),(3 4,4 5))
```

Figure



Aggregate variant.

Creating multiple collections by grouping geometries in a table.

Code

```
SELECT stusps, ST_Collect(geom) AS geom
FROM (
  SELECT stusps, (ST_Dump(geom)).geom AS geom
  FROM somestatetable
) AS f
GROUP BY stusps;
```

☒☒

ST_Dump, ST_AsBinary

7.3.2 ST_LineFromMultiPoint

ST_LineFromMultiPoint — ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.

☒☒

geometry **ST_LineFromMultiPoint**(geometry aMultiPoint);

☒☒

☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.

Use ST_MakeLine to create lines from Point or LineString inputs.

This function supports 3d and will not drop the z-index.

☒☒

☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.

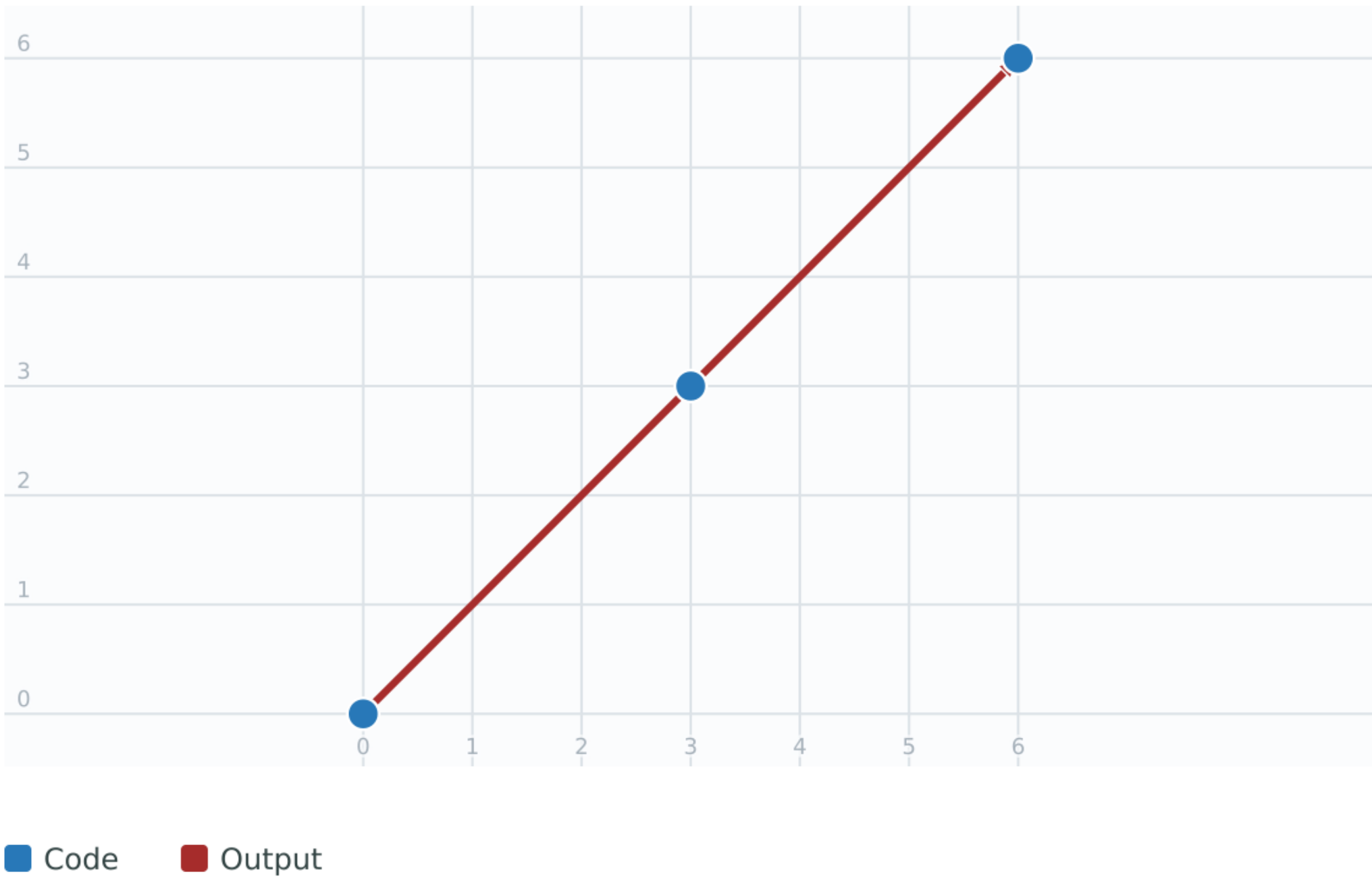
Code

```
SELECT ST_LineFromMultiPoint('MULTIPOINT(1 2 3,4 5 6,7 8 9)');
```

Output

```
LINESTRING(1 2 3,4 5 6,7 8 9)
```

Figure



☒☒

ST_AsEWKT, ST_AsKML

7.3.3 ST_MakeEnvelope

ST_MakeEnvelope — □□□□□□□□□□□□□□□□□□□□□□□□□□□□. □□□□□ SRID □□□□□ SRS □□□□□□□□□□□.

□□

geometry **ST_MakeEnvelope**(float xmin, float ymin, float xmax, float ymax, integer srid=unknown);

□□

□□□□□□□□□□□□□□□□□□□□□□□□□. □□□□ SRID □□□□ SRS □□□□□□□□□□. □□□ SRID □□□□□□□□□□□□□□□□□□□□□□□□□□□.

1.5 □□□□□□□□□□□□.

□□□□: 2.0 □□□□ SRID □□□□□□□□□□□□ (envelope) □□□□□□□□□□□□□□□□.

□□

Building a bounding box polygon.

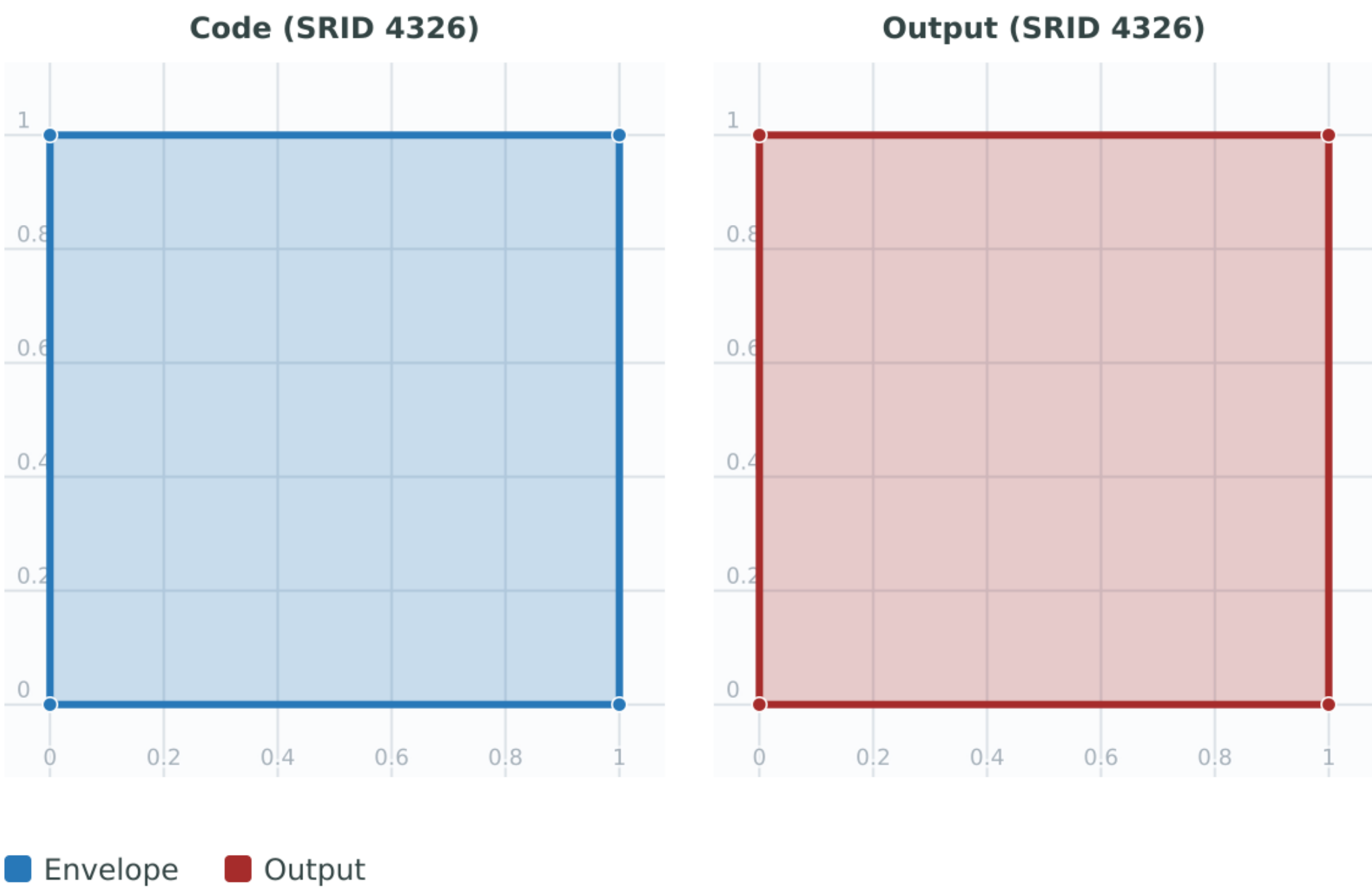
Code

```
SELECT ST_MakeEnvelope(10, 10, 11, 11, 4326);
```

Output

```
POLYGON((10 10, 10 11, 11 11, 11 10, 10 10))
```

Figure



⊠⊠

ST_MakePoint, ST_MakePoint, ST_Point, ST_SRID

7.3.4 ST_MakeLine

ST_MakeLine — ⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

geometry **ST_MakeLine**(geometry geom1, geometry geom2);
geometry **ST_MakeLine**(geometry[] geoms_array);
geometry **ST_MakeLine**(geometry set geoms);

⊠⊠

Creates a LineString containing the points of Point, MultiPoint, or LineString geometries. Other geometry types cause an error.

Variant 1: accepts two input geometries

Variant 2: accepts an array of geometries

Variant 3: aggregate function accepting a rowset of geometries. To ensure the order of the input geometries use `ORDER BY` in the function call, or a subquery with an `ORDER BY` clause.

Repeated nodes at the beginning of input LineStrings are collapsed to a single point. Repeated points in Point and MultiPoint inputs are not collapsed. Components of MultiLineString are handled in the order they appear in the collection. ST_RemoveRepeatedPoints can be used to collapse repeated points from the output LineString.

This function supports 3d and will not drop the z-index.

Availability: 3.7.0 - Support for MultiLineString input elements was introduced

2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

1.4.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠ ST_MakeLine ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠ ST_MakeLine ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

Two-input variant.

Create a line composed of two points.

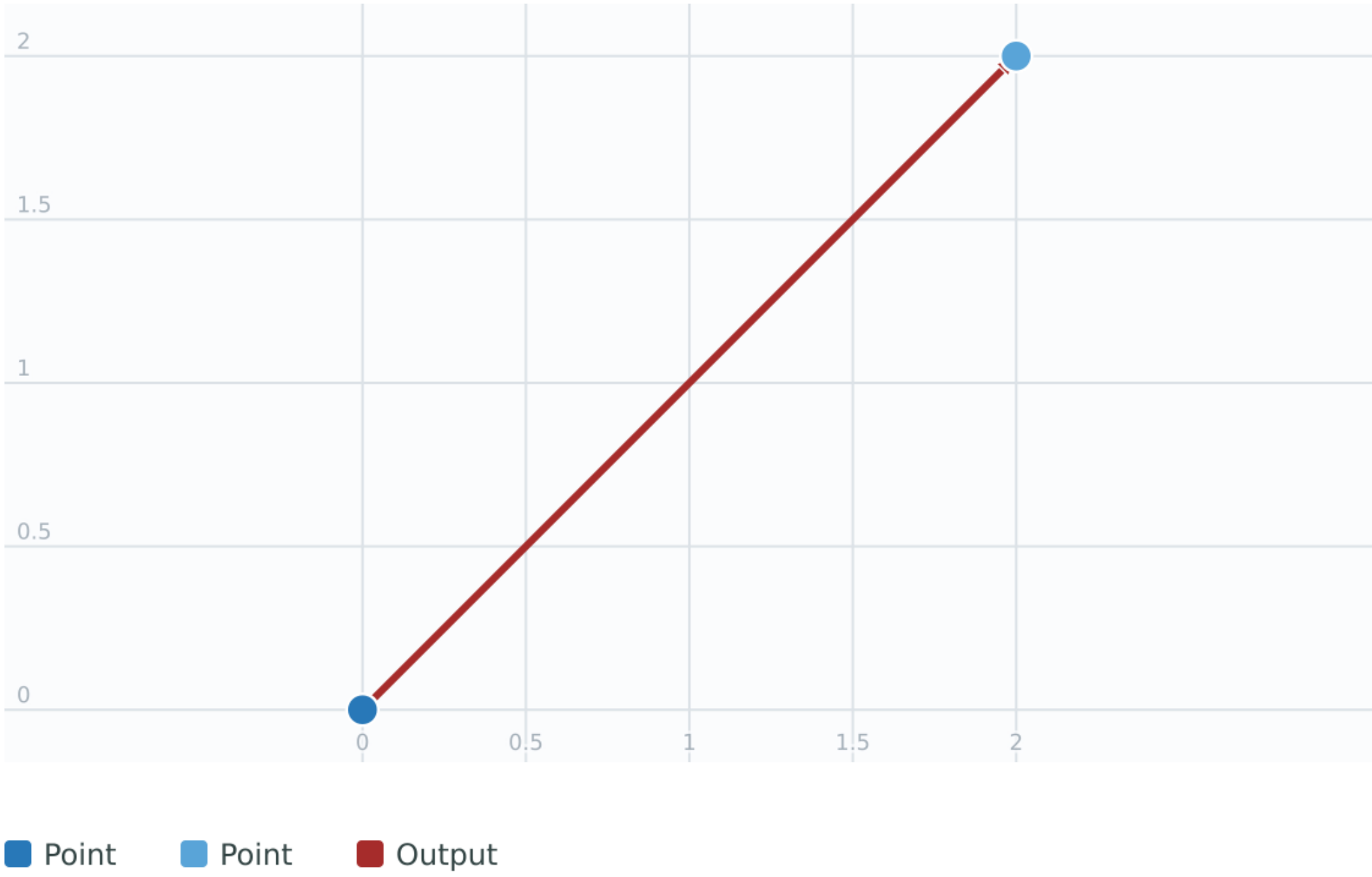
Code

```
SELECT ST_MakeLine(ST_Point(1, 2), ST_Point(3, 4));
```

Output

```
LINESTRING(1 2,3 4)
```

Figure



⌧⌧⌧ 3D ⌧⌧⌧⌧⌧ 2 ⌧⌧⌧⌧⌧⌧ BOX3D ⌧⌧⌧⌧⌧⌧.

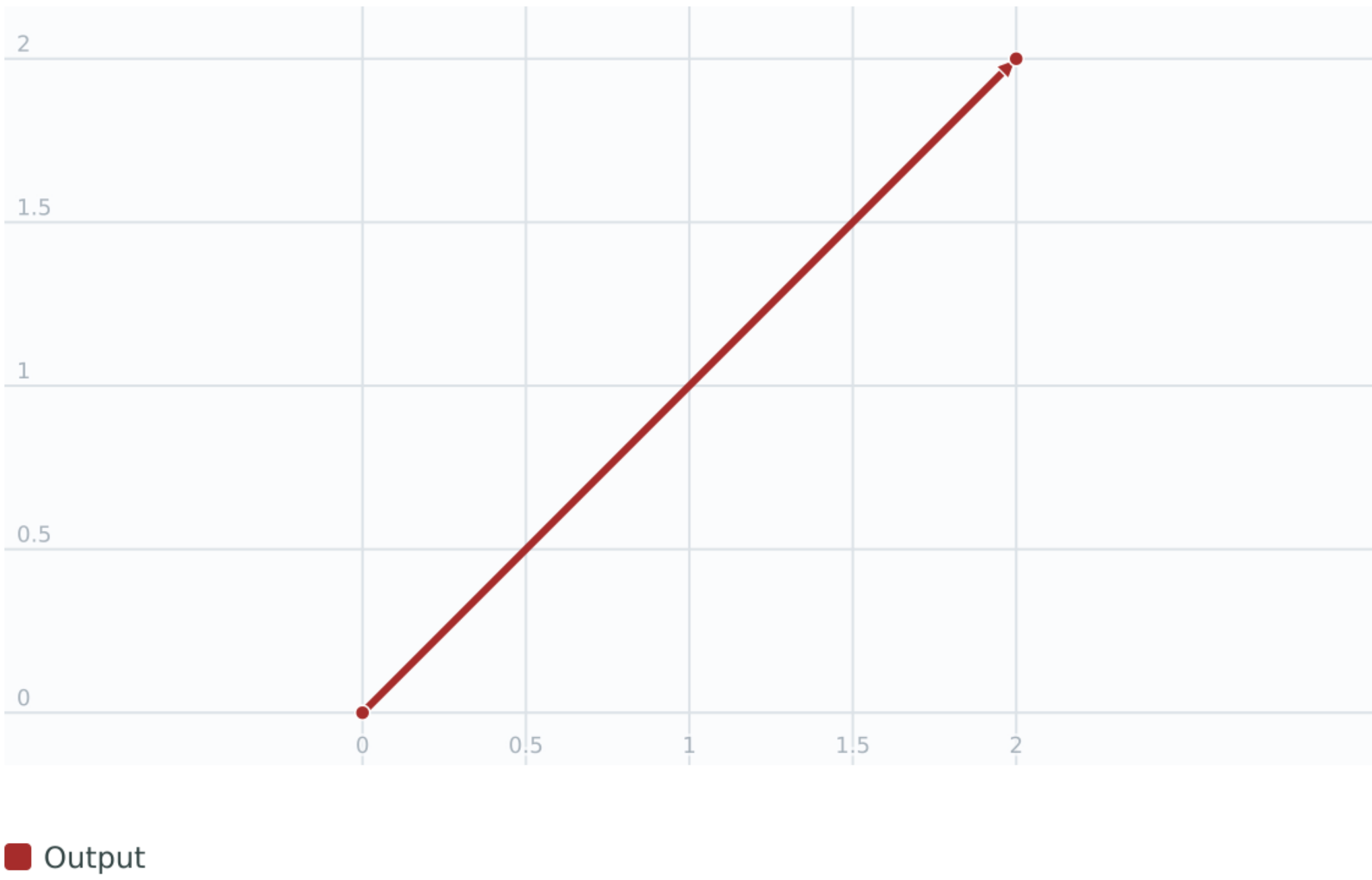
Code

```
SELECT ST_MakeLine(ST_MakePoint(1, 2, 3), ST_MakePoint(3, 4, 5));
```

Output

```
LINESTRING(1 2 3,3 4 5)
```

Figure



⌧⌧⌧, ⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧.

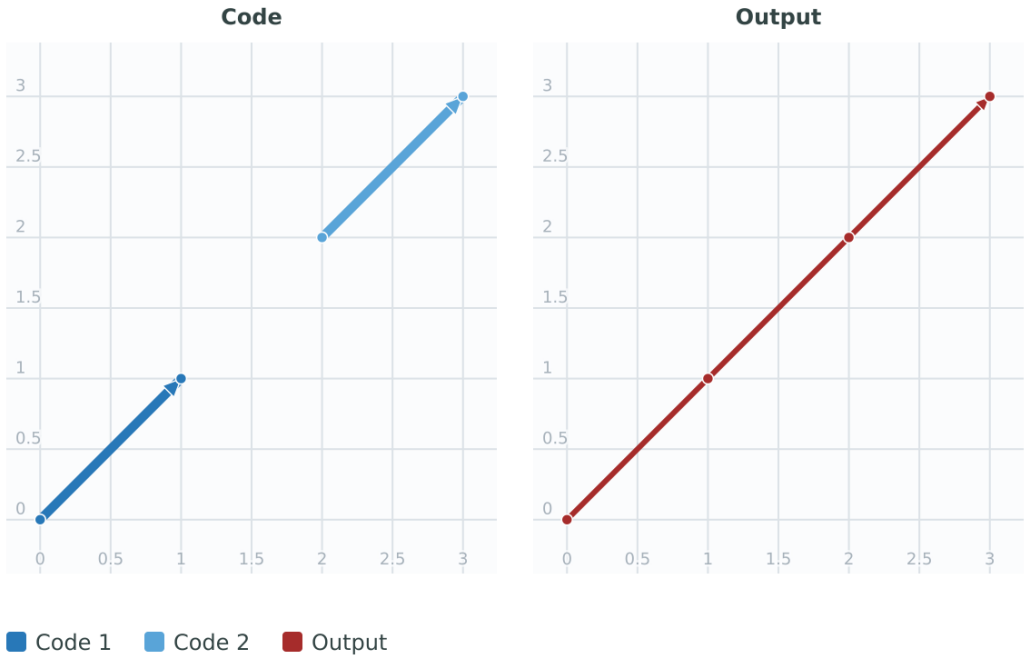
Code

```
SELECT ST_MakeLine('LINESTRING(0 0,1 1)', 'LINESTRING(2 2,3 3)');
```

Output

```
LINESTRING(0 0,1 1,2 2,3 3)
```

Figure



Array variant.

Create a line from an array formed by a subquery with ordering.

Code

```
SELECT ST_MakeLine(ARRAY(SELECT ST_Centroid(geom) FROM visit_locations ORDER BY visit_time) ↩
    );
```

Create a 3D line from an array of 3D points

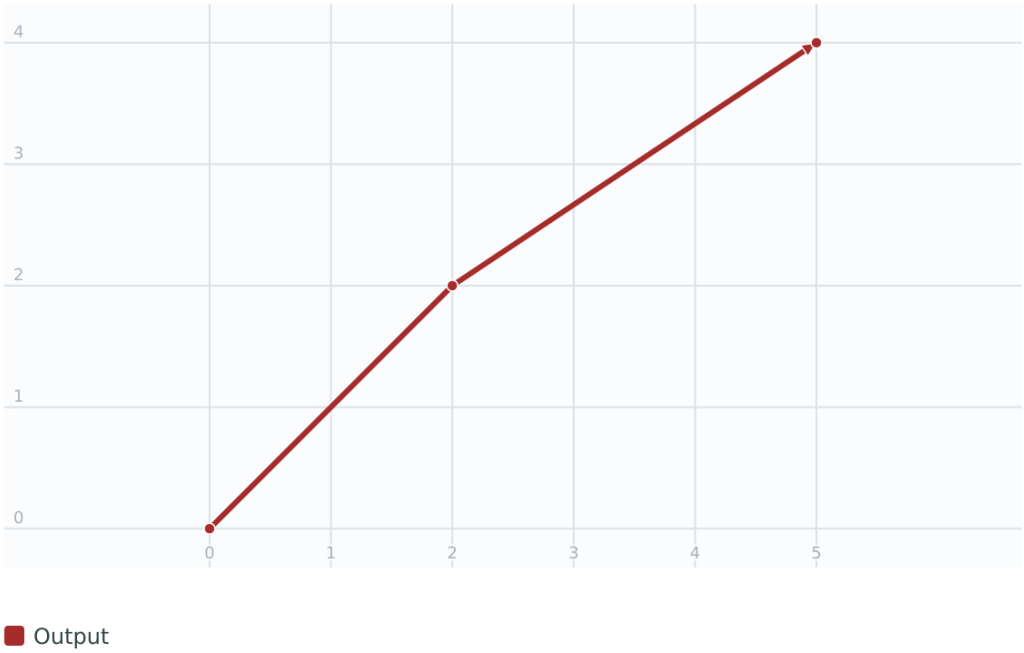
Code

```
SELECT ST_MakeLine(ARRAY[ST_MakePoint(1, 2, 3), ST_MakePoint(3, 4, 5), ST_MakePoint(6, 6, ↩
    6)]);
```

Output

```
LINESTRING(1 2 3,3 4 5,6 6 6)
```

Figure



Aggregate variant.

□□□□ GPS □□□□□□□□□□□, □□□□□□□□□□□□ GPS □□□□□□□□□□□□□□□ GPS □□□□□□□□□□□□□□□□□□.

Using aggregate `ORDER BY` provides a correctly-ordered LineString.

Code

```
SELECT gps.track_id, ST_MakeLine(gps.geom ORDER BY gps_time) AS geom
FROM gps_points AS gps
GROUP BY gps.track_id;
```

When ordering within the aggregate is not possible, ordering in a subquery can be used. However, sometimes the query plan may not respect the order of the subquery.

Code

```
SELECT gps.track_id, ST_MakeLine(gps.geom) AS geom
FROM (
  SELECT track_id, gps_time, geom
  FROM gps_points
  ORDER BY track_id, gps_time
) AS gps
GROUP BY gps.track_id;
```

□□

ST_RemoveRepeatedPoints, ST_AsText, ST_GeomFromText, ST_MakePoint

7.3.5 ST_MakePoint

ST_MakePoint — Creates a 2D, 3DZ or 4D Point.

□□

geometry **ST_MakePoint**(float x, float y);

geometry **ST_MakePoint**(float x, float y, float z);

geometry **ST_MakePoint**(float x, float y, float z, float m);

□□

Creates a 2D XY, 3D XYZ or 4D XYZM Point geometry. Use ST_MakePointM to make points with XYM coordinates.

Use ST_SetSRID to specify a SRID for the created point.

While not OGC-compliant, `ST_MakePoint` is faster than ST_GeomFromText and ST_PointFromText. It is also easier to use for numeric coordinate values.



□□
For geodetic coordinates, X is longitude and Y is latitude



□□
The functions ST_Point, ST_PointZ, ST_PointM, and ST_PointZM can be used to create points with a given SRID.



This function supports 3d and will not drop the z-index.

□□

This example creates a point with an unknown SRID.

Code

```
SELECT ST_MakePoint(-71.1043443253471, 42.3150676015829);
```

This example creates a point in the WGS 84 geodetic CRS.

Code

```
SELECT ST_SetSRID(ST_MakePoint(-71.1043443253471, 42.3150676015829), 4326);
```

This example creates a 3D point with altitude.

Code

```
SELECT ST_MakePoint(1, 2, 1.5);
```

This example returns the Z coordinate of a point.

Code

```
SELECT ST_Z(ST_MakePoint(1, 2, 1.5));
```

Output

```
1.5
```

□□

ST_GeomFromText, ST_PointFromText, ST_SetSRID, ST_MakePointM, ST_Point, ST_PointZ, ST_PointM, ST_PointZM

7.3.6 ST_MakePointM

ST_MakePointM — x, y □□□□□□□□□□□□□□□□□□□□□.

□□

geometry **ST_MakePointM**(float x, float y, float m);

□□

Creates a point with X, Y and M (measure) ordinates. Use ST_MakePoint to make points with XY, XYZ, or XYZM coordinates.

Use ST_SetSRID to specify a SRID for the created point.



□□
For geodetic coordinates, X is longitude and Y is latitude



□□
The functions ST_PointM, and ST_PointZM can be used to create points with an M value and a given SRID.

□□



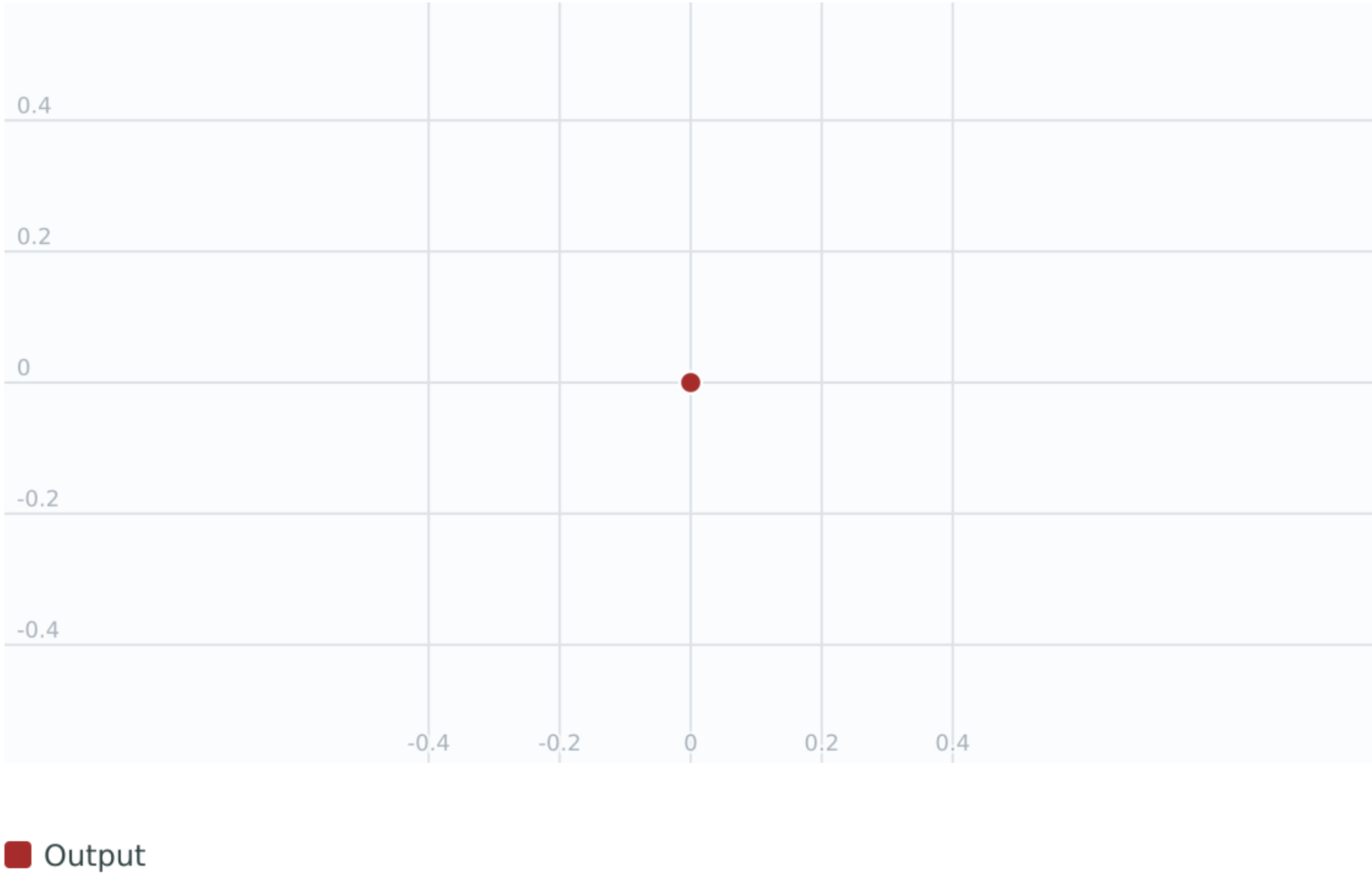
□□
ST_AsEWKT is used for text output because ST_AsText does not support M values.

Create point with unknown SRID.

Code

```
SELECT ST_MakePointM(-71.1043443253471, 42.3150676015829, 10);
```

Figure

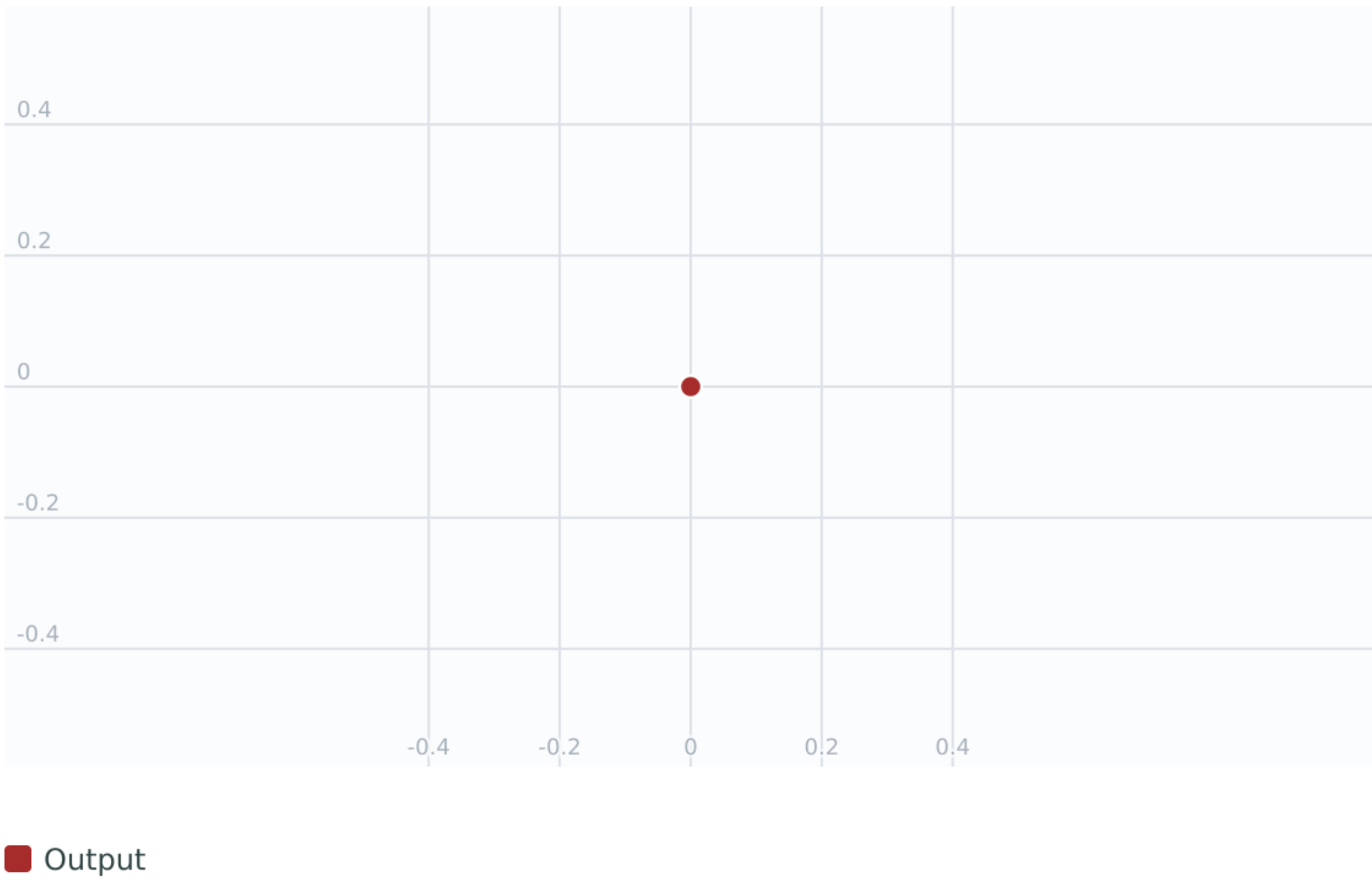


x, y ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

Code

```
SELECT ST_SetSRID(ST_MakePointM(-71.104, 42.315, 10), 4326);
```

Figure



Get measure of created point.

Code

```
SELECT ST_M(ST_MakePointM(-71.104, 42.315, 10));
```

Output

```
10
```

☒☒

ST_MakePoint, ST_SetSRID, ST_PointM, ST_PointZM

7.3.7 ST_MakePolygon

ST_MakePolygon — Creates a Polygon or CurvePolygon from a shell and optional list of holes.

☒☒

geometry **ST_MakePolygon**(geometry linestring);

geometry **ST_MakePolygon**(geometry outerlinestring, geometry[] interiorlinestrings);

☒☒

Creates a Polygon or CurvePolygon formed by the given shell and optional array of holes. Input geometries must be closed LineStrings, CircularStrings, CompoundCurves, or NURBSCurves (rings). If any input ring is curved, the result is a CurvePolygon.

Variant 1: Accepts one shell ring.

Variant 2: Accepts a shell ring and an array of inner (hole) rings. A geometry array can be constructed using the PostgreSQL array_agg(), ARRAY[] or ARRAY() constructs.

> **Note:** ☒☒
> ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒ ST_LineMerge ☒☒ ST_Dump ☒☒☒☒☒☒☒.

This function supports 3d and will not drop the z-index.

Enhanced: 3.7.0 - Support for curved input rings was introduced.

☒☒

Single input variant.

☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.

Code

```
SELECT ST_MakePolygon('LINESTRING(75 29,77 29,77 29,75 29)');
```

Create a Polygon from an open LineString, using ST_StartPoint and ST_AddPoint to close it.

Code

```
SELECT ST_MakePolygon(ST_AddPoint(foo.open_line, ST_StartPoint(foo.open_line)) )
FROM (
  SELECT 'LINESTRING(75 29,77 29,77 29,75 29)'::geometry As open_line) As foo;
```

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

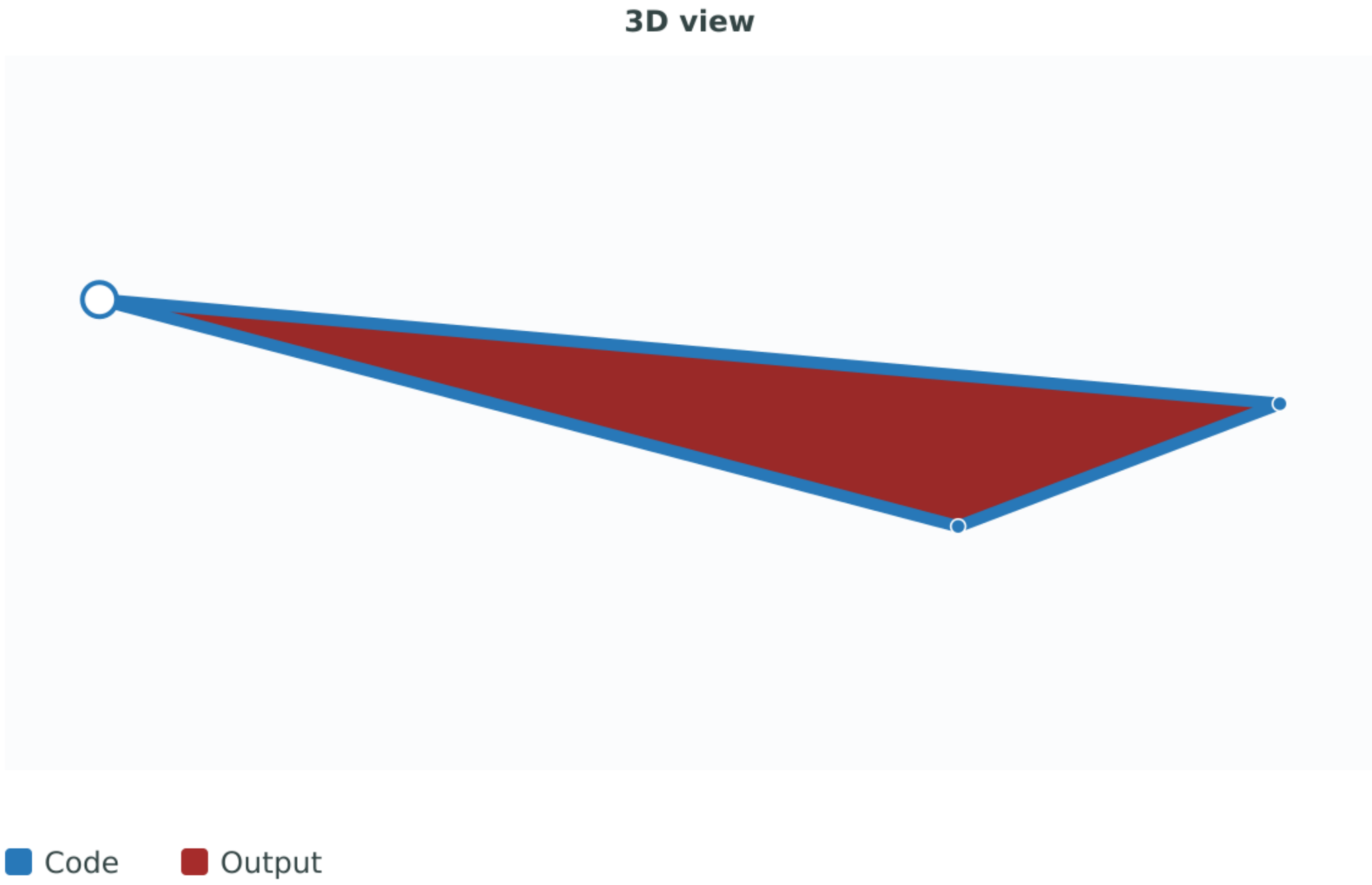
Code

```
SELECT ST_MakePolygon('LINESTRING(75.15 29.53 1,77 29 1,77.6 29.5 1,75.15 29.53 1)');
```

Output

```
POLYGON((75.15 29.53 1,77 29 1,77.6 29.5 1,75.15 29.53 1))
```

Figure

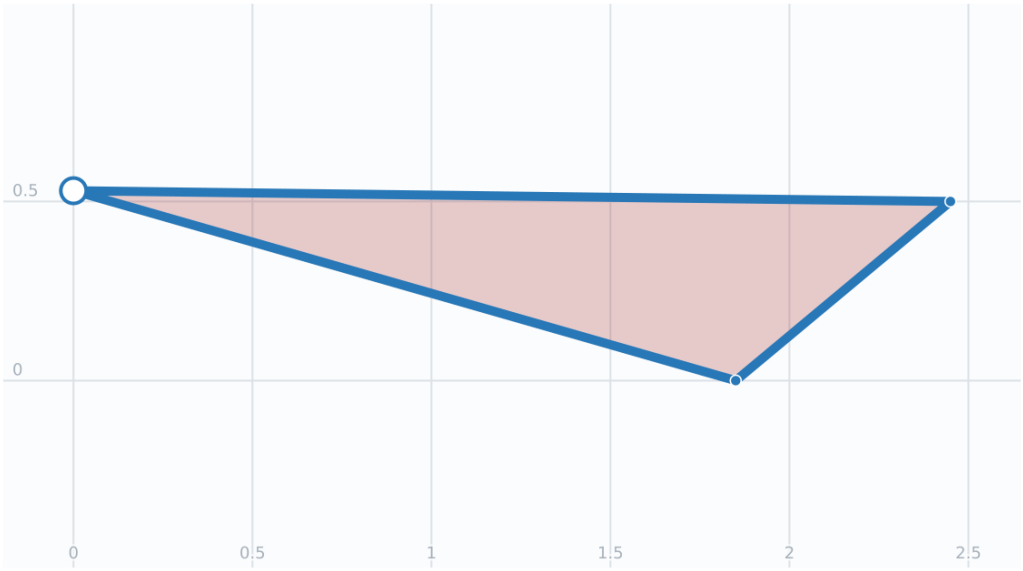


Create a Polygon from a LineString with measures

Code

```
SELECT ST_MakePolygon('LINESTRINGM(75.15 29.53 1,77 29 1,77.6 29.5 2, 75.15 29.53 2)');
```

Figure



Create a CurvePolygon from a CircularString shell.

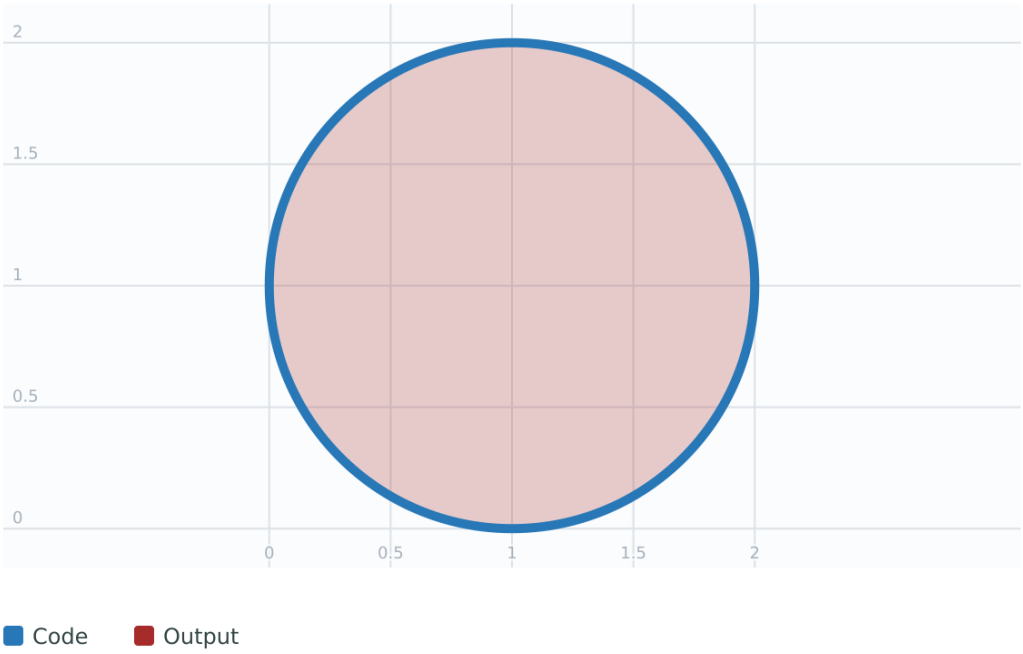
Code

```
SELECT ST_MakePolygon('CIRCULARSTRING(0 0,1 1,2 0,1 -1,0 0)');
```

Output

```
CURVEPOLYGON(CIRCULARSTRING(0 0,1 1,2 0,1 -1,0 0))
```

Figure



Outer shell with inner holes variant.

□□□□□□□□□□□□□□□□□□.

Code

```
SELECT ST_MakePolygon(
    ST_ExteriorRing(ST_Buffer(ring.line, 10)),
    ARRAY[  ST_Translate(ring.line, 1, 1),
        ST_ExteriorRing(ST_Buffer(ST_Point(20, 20), 1)) ]
    )
FROM (SELECT ST_ExteriorRing(ST_Buffer(ST_Point(10, 10), 10, 10)) AS line ) AS ring;
```

Create a set of province boundaries with holes representing lakes. The input is a table of province Polygons/MultiPolygons and a table of water linestrings. Lines forming lakes are determined by using ST_IsClosed. The province linework is extracted by using ST_Boundary. As required by ST_MakePolygon, the boundary is forced to be a single LineString by using ST_LineMerge. (However, note that if a province has more than one region or has islands this will produce an invalid polygon.) Using a LEFT JOIN ensures all provinces are included even if they have no lakes.



□□
NULL □□□ ST_MakePolygon □□□□□ NULL □□□□□□□□ CASE □□□□□□□□.

Code

```
SELECT p.gid, p.province_name,
  CASE
    WHEN array_agg(w.geom) IS NULL THEN p.geom
    ELSE ST_MakePolygon(
      ST_LineMerge(ST_Boundary(p.geom)),
      array_agg(w.geom)
    )
  END AS geom
FROM provinces p
LEFT JOIN waterlines w
  ON ST_Within(w.geom, p.geom)
  AND ST_IsClosed(w.geom)
GROUP BY p.gid, p.province_name, p.geom;
```

Another technique is to utilize a correlated subquery and the ARRAY() constructor that converts a row set to an array.

Code

```
SELECT p.gid, p.province_name,
  CASE
    WHEN EXISTS (
      SELECT w.geom
      FROM waterlines w
      WHERE ST_Within(w.geom, p.geom)
      AND ST_IsClosed(w.geom)
    )
    THEN ST_MakePolygon(
      ST_LineMerge(ST_Boundary(p.geom)),
      ARRAY(
        SELECT w.geom
        FROM waterlines w
        WHERE ST_Within(w.geom, p.geom)
```

```
        AND ST_IsClosed(w.geom)
      )
    )
    ELSE p.geom
  END AS geom
FROM provinces p;
```

□□

ST_BuildArea ST_Polygon

7.3.8 ST_Point

ST_Point — Creates a Point with X, Y and SRID values.

□□

geometry **ST_Point**(float x, float y);

geometry **ST_Point**(float x, float y, integer srid=unknown);

□□

Returns a Point with the given X and Y coordinate values. This is the SQL-MM equivalent for ST_MakePoint that takes just X and Y.



□□
For geodetic coordinates, X is longitude and Y is latitude

Enhanced: 3.2.0 srid as an extra optional argument was added. Older installs require combining with ST_SetSRID to mark the srid on the geometry.

 This method implements the SQL/MM specification. SQL-MM 3: 6.1.2

□□

Geometry.

Code

```
SELECT ST_Point(-71.104, 42.315);
```

Creating a point with SRID specified:

Code

```
SELECT ST_Point(-71.104, 42.315, 4326);
```

Alternative way of specifying SRID:

Code

```
SELECT ST_SetSRID(ST_Point(-71.104, 42.315), 4326);
```

Create geography points using the :: cast syntax:

Code

```
SELECT ST_Point(-71.104, 42.315, 4326)::geography;
```

Pre-PostGIS 3.2 code, using CAST:

Code

```
SELECT CAST(ST_SetSRID(ST_Point(-71.104, 42.315), 4326) AS geography);
```

If the point coordinates are not in a geodetic coordinate system (such as WGS84), then they must be reprojected before casting to a geography. In this example a point in Pennsylvania State Plane feet (SRID 2273) is projected to WGS84 (SRID 4326).

Code

```
SELECT ST_Transform(ST_Point(3637510, 3014852, 2273), 4326)::geography;
```

□□

ST_MakePoint, ST_PointZ, ST_PointM, ST_PointZM, ST_SetSRID, ST_Transform

7.3.9 ST_PointZ

ST_PointZ — Creates a Point with X, Y, Z and SRID values.

□□

geometry **ST_PointZ**(float x, float y, float z, integer srid=unknown);

□□

□□□□□□□□□□ ST_Point □□□□□□□. ST_MakePoint □□□□ OGC □□□□□□.

Enhanced: 3.2.0 srid as an extra optional argument was added. Older installs require combining with ST_SetSRID to mark the srid on the geometry.

□□

Code

```
SELECT ST_PointZ(-71.104, 42.315, 3.4, 4326)
```

Code

```
SELECT ST_PointZ(-71.104, 42.315, 3.4, srid => 4326)
```

Code

```
SELECT ST_PointZ(-71.104, 42.315, 3.4)
```

□□

ST_MakePoint, ST_PointFromText, ST_SetSRID, ST_MakePointM

7.3.10 ST_PointM

ST_PointM — Creates a Point with X, Y, M and SRID values.

□□

geometry **ST_PointM**(float x, float y, float m, integer srid=unknown);

□□

□□□□□□□□□□ ST_Point □□□□□□. ST_MakePoint □□□□ OGC □□□□□.

Enhanced: 3.2.0 srid as an extra optional argument was added. Older installs require combining with ST_SetSRID to mark the srid on the geometry.

□□

Code

```
SELECT ST_PointM(-71.104, 42.315, 3.4, 4326)
```

Code

```
SELECT ST_PointM(-71.104, 42.315, 3.4, srid => 4326)
```

Code

```
SELECT ST_PointM(-71.104, 42.315, 3.4)
```

□□

ST_MakePoint, ST_PointFromText, ST_SetSRID, ST_MakePointM

7.3.11 ST_PointZM

ST_PointZM — Creates a Point with X, Y, Z, M and SRID values.

□□

geometry **ST_PointZM**(float x, float y, float z, float m, integer srid=unknown);

□□

□□□□□□□□□□ ST_Point □□□□□□. ST_MakePoint □□□□ OGC □□□□□.
Enhanced: 3.2.0 srid as an extra optional argument was added. Older installs require combining with ST_SetSRID to mark the srid on the geometry.

□□

Code

```
SELECT ST_PointZM(-71.104, 42.315, 3.4, 4.5, 4326)
```

Code

```
SELECT ST_PointZM(-71.104, 42.315, 3.4, 4.5, srid => 4326)
```

Code

```
SELECT ST_PointZM(-71.104, 42.315, 3.4, 4.5)
```

□□

ST_MakePoint, ST_Point, ST_PointM, ST_PointZ, ST_SetSRID

7.3.12 ST_Polygon

ST_Polygon — Creates a Polygon from a LineString with a specified SRID.

□□

geometry **ST_Polygon**(geometry lineString, integer srid);

□□

Returns a polygon built from the given LineString and sets the spatial reference system from the `srid`.
ST_Polygon is similar to ST_MakePolygon Variant 1 with the addition of setting the SRID.
, ST_MakePoint, ST_SetSRID



□□
□□□□□□□□□□□□□□□□□□□□□□□□□. □□□□□□□□□□□ ST_LineMerge □□ ST_Dump □□□□□□□.

This method implements the OGC Simple Features Implementation Specification for SQL 1.1.

This method implements the SQL/MM specification. SQL-MM 3: 8.3.2

This function supports 3d and will not drop the z-index.

□□

Create a 2D polygon.

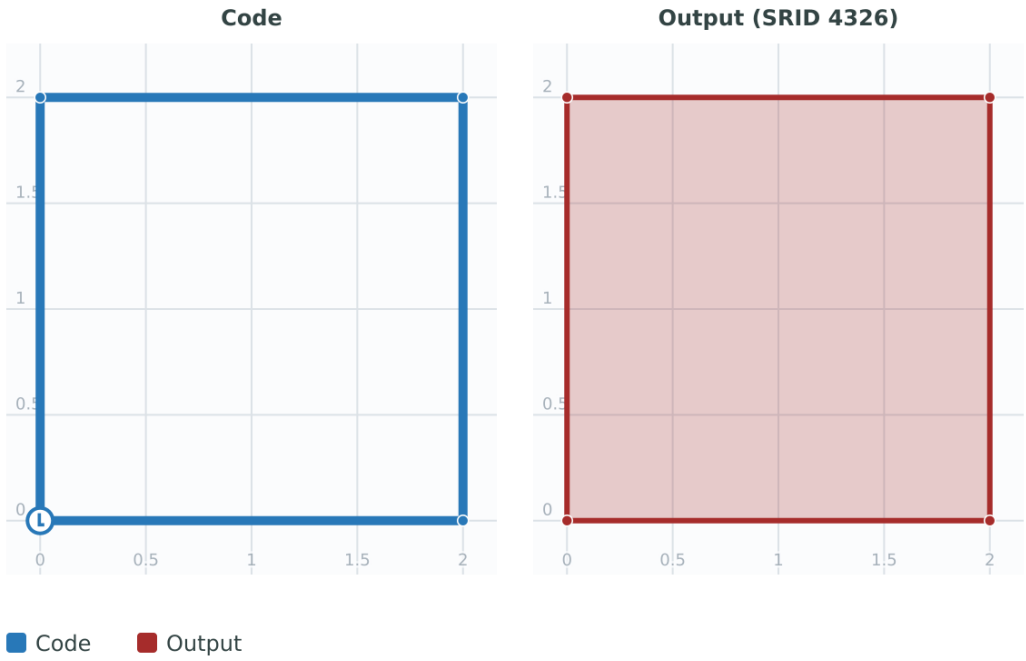
Code

```
SELECT ST_Polygon('LINESTRING(75 29,77 29,77 31,75 31,75 29)'::geometry, 4326);
```

Output

```
POLYGON((75 29,77 29,77 31,75 31,75 29))
```

Figure



Create a 3D polygon.

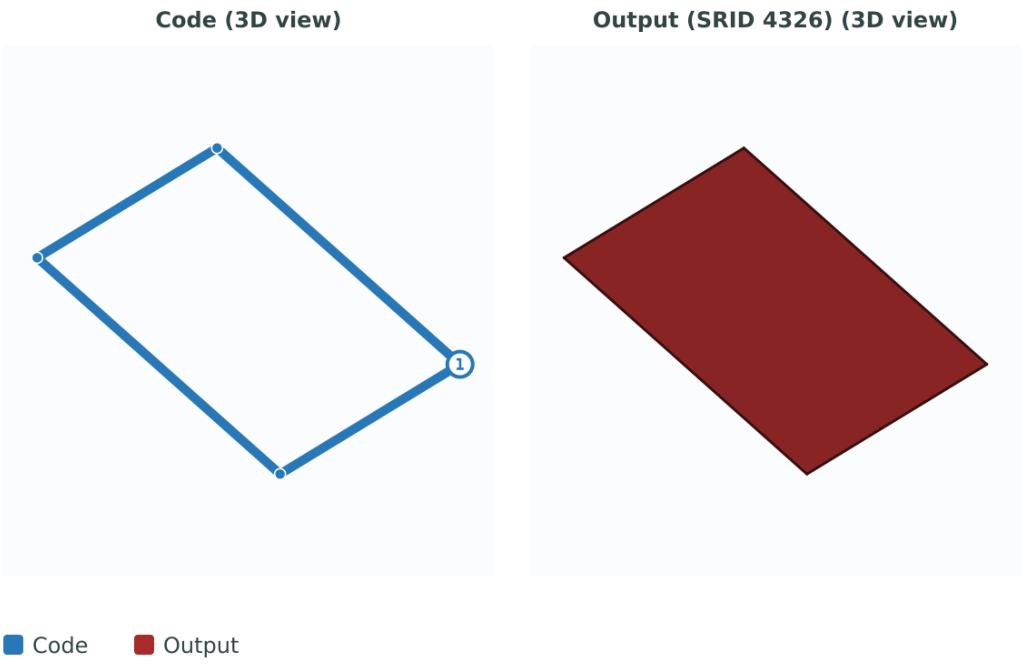
Code

```
SELECT ST_Polygon('LINESTRING(75 29 1,77 29 2,77 31 2,75 31 1,75 29 1)'::geometry, 4326);
```

Output

```
SRID=4326;POLYGON((75 29 1,77 29 2,77 31 2,75 31 1,75 29 1))
```

Figure



□□

ST_AsEWKT, ST_AsText, ST_GeomFromEWKT, ST_GeomFromText, ST_LineMerge, ST_MakePolygon

7.3.13 ST_TileEnvelope

ST_TileEnvelope — Creates a rectangular Polygon in Web Mercator (SRID:3857) using the XYZ tile system.

□□

geometry **ST_TileEnvelope**(integer tileZoom, integer tileX, integer tileY, geometry bounds=SRID=3857;LI
20037508.342789 -20037508.342789,20037508.342789 20037508.342789), float margin=0.0);

□□

Creates a rectangular Polygon giving the extent of a tile in the XYZ tile system. The tile is specified by the zoom level Z and the XY index of the tile in the grid at that level. Can be used to define the tile bounds required by ST_AsMVTGeom to convert geometry into the MVT tile coordinate space.

By default, the tile envelope is in the Web Mercator coordinate system (SRID:3857) using the standard range of the Web Mercator system (-20037508.342789, 20037508.342789). This is the most common coordinate system used for MVT tiles. The optional `bounds` parameter can be used to generate tiles in any coordinate system. It is a geometry that has the SRID and extent of the "Zoom Level zero" square within which the XYZ tile system is inscribed.

The optional `margin` parameter can be used to expand a tile by the given percentage. E.g. `margin=0.125` expands the tile by 12.5%, which is equivalent to buffer=512 when the tile extent size is 4096, as used

in ST_AsMVTGeom. This is useful to create a tile buffer to include data lying outside of the tile's visible area, but whose existence affects the tile rendering. For example, a city name (a point) could be near an edge of a tile, so its label should be rendered on two tiles, even though the point is located in the visible area of just one tile. Using expanded tiles in a query will include the city point in both tiles. Use a negative value to shrink the tile instead. Values less than -0.5 are prohibited because that would eliminate the tile completely. Do not specify a margin when using with `ST_AsMVTGeom`. See the example for ST_AsMVT.

□□□□: 2.0.0 □□□□□□□□□□ SRID □□□□□□□□□□□.

2.1.0 □□□□□□□□□□□.

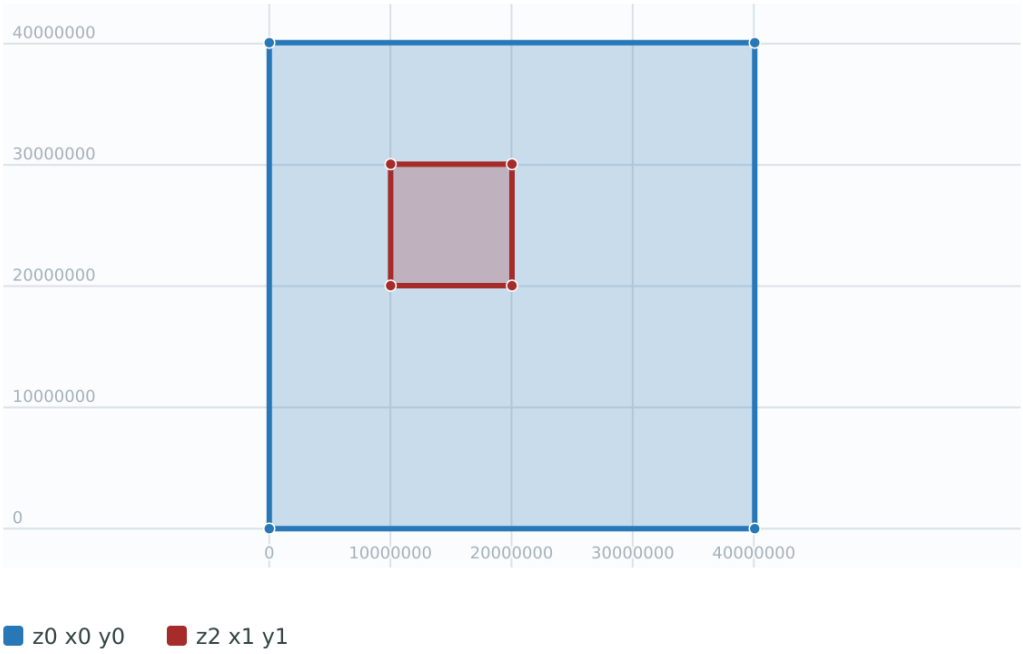
□□

Show tile (2, 1, 1) within the root tile (0, 0, 0).

Code

```
SELECT ST_TileEnvelope(0, 0, 0) AS input_z0_x0_y0, ST_TileEnvelope(2, 1, 1) AS z2_x1_y1;
```

Figure



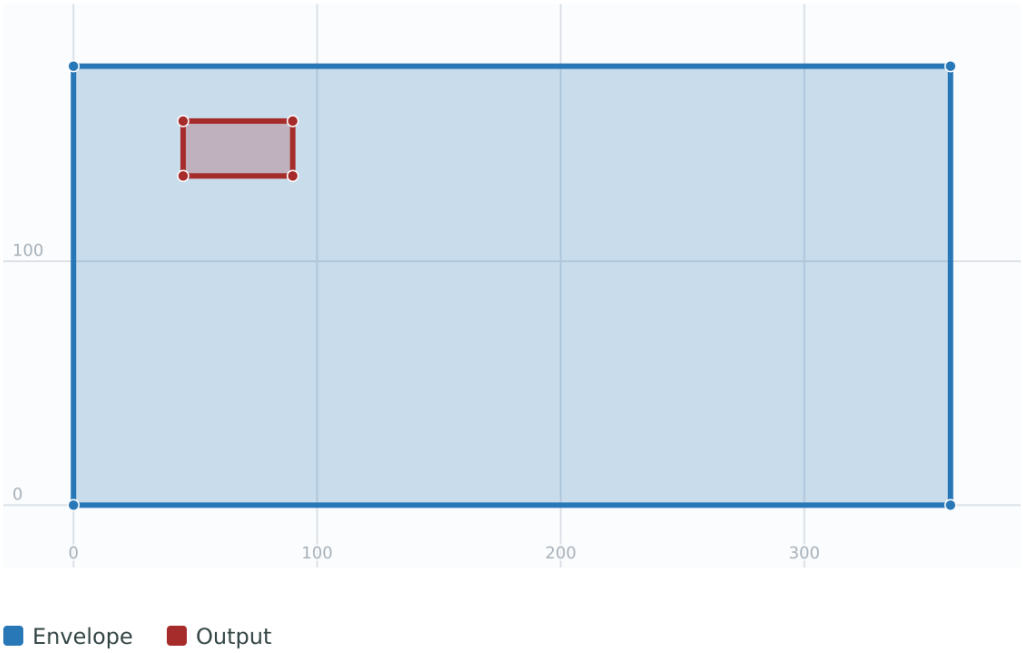
Code

```
SELECT ST_TileEnvelope(3, 1, 1, ST_MakeEnvelope(-180, -90, 180, 90, 4326));
```

Output

```
POLYGON((-135 45,-135 67.5,-90 67.5,-90 45,-135 45))
```

Figure



☒☒

ST_MakeEnvelope

7.3.14 ST_HexagonGrid

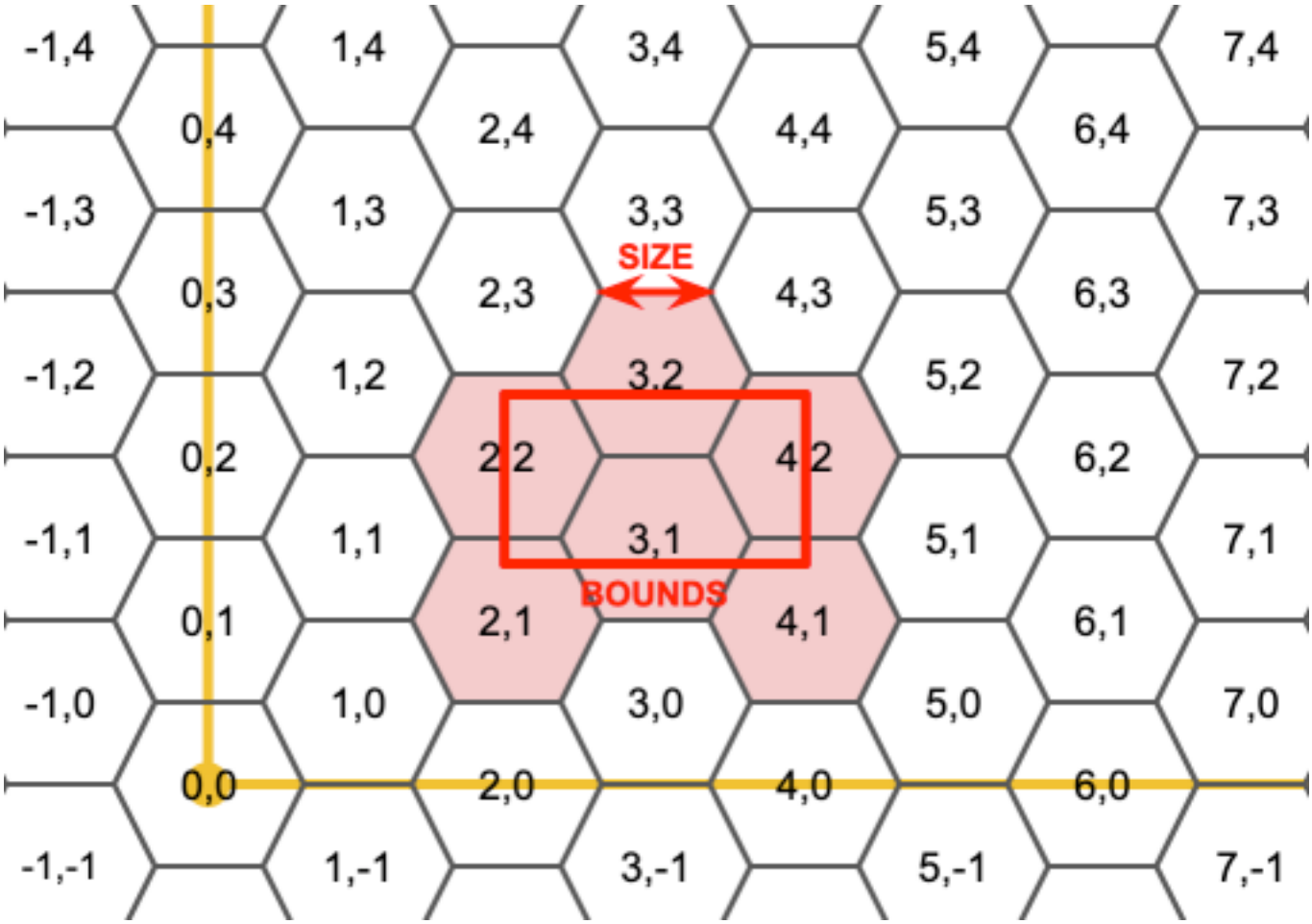
ST_HexagonGrid — Returns a set of hexagons and cell indices that completely cover the bounds of the geometry argument.

☒☒

setof record **ST_HexagonGrid**(float8 size, geometry bounds);

☒☒

Starts with the concept of a hexagon tiling of the plane. (Not a hexagon tiling of the globe; for H3 use the h3-pg extension.) For a given planar SRS, and a given edge size, starting at the origin of the SRS, there is one unique hexagonal tiling of the plane, Tiling(SRS, Size). This function answers the question: what hexagons in a given Tiling(SRS, Size) overlap with a given bounds.



The SRS for the output hexagons is the SRS provided by the bounds geometry.

Doubling or tripling the edge size of the hexagon generates a new parent tiling that fits with the origin tiling. Unfortunately, it is not possible to generate parent hexagon tilings that the child tiles perfectly fit inside.

2.1.0 □□□□□□□□□□□□.

□□

Unit hexagons covering rectangular bounds.

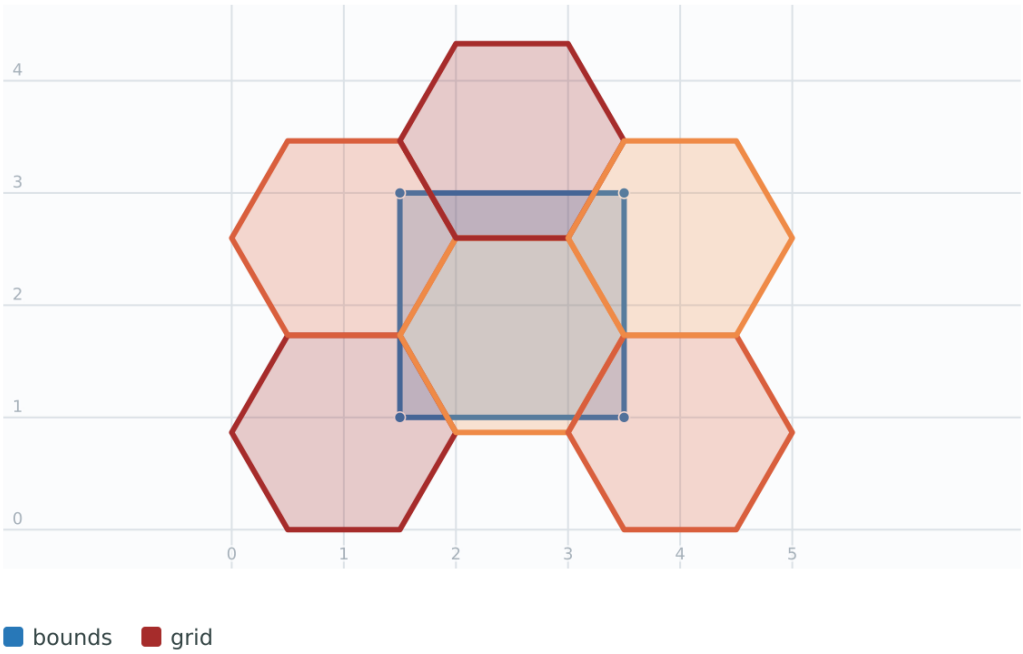
Code

```
WITH bounds AS (
    SELECT ST_MakeEnvelope(2, 1, 4, 3, 3857) AS geom
), grid AS (
    SELECT hex.*
    FROM bounds
    CROSS JOIN LATERAL ST_HexagonGrid(1, geom) AS hex
)
SELECT (SELECT geom FROM bounds) AS input_bounds,
    ST_Collect(geom ORDER BY i, j) AS grid
FROM grid;
```

Output

```
POLYGON((2 1,2 3,4 3,4 1,2 1)) | MULTIPOLYGON(((0.5 0.866,1 0,2 0,2.5 0.866,2 1.732,1 1.732,0.5 0.866)),((0.5 2.598,1 ↩
    1.732,2 1.732,2.5 2.598,2 3.464,1 3.464,0.5 2.598)),((2 1.732,2.5 0.866,3.5 0.866,4 1.732,3.5 2.598,2.5 2.598,2 ↩
    1.732)),((2 3.464,2.5 2.598,3.5 2.598,4 3.464,3.5 4.33,2.5 4.33,2 3.464)),((3.5 0.866,4 0,5 0,5.5 0.866,5 1.732,4 ↩
     1.732,3.5 0.866)),((3.5 2.598,4 1.732,5 1.732,5.5 2.598,5 3.464,4 3.464,3.5 2.598)))
```

Figure

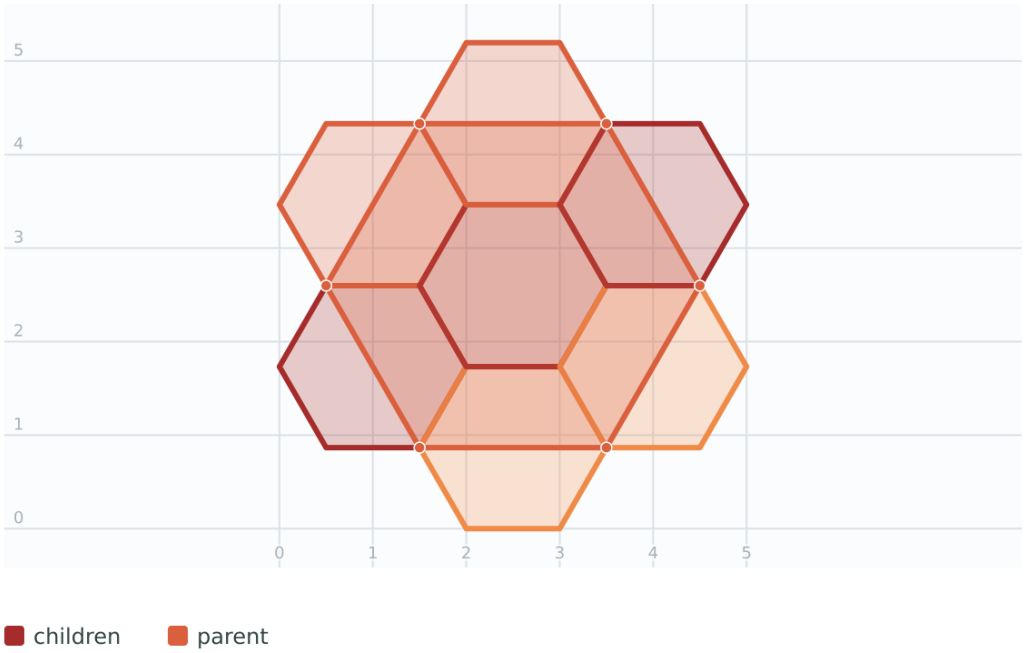


Child tiles that intersect a doubled-size parent hexagon share the tiling origin, but do not nest perfectly inside the parent.

Code

```
WITH parent AS (
    SELECT ST_Hexagon(2, 0, 0) AS geom
), children AS (
    SELECT child.*
    FROM parent
    CROSS JOIN LATERAL ST_HexagonGrid(1, geom) AS child
    WHERE ST_Area(ST_Intersection(child.geom, parent.geom)) > 0
)
SELECT ST_Collect(geom ORDER BY i, j) AS children,
    (SELECT geom FROM parent) AS parent
FROM children;
```

Figure



Counting points in hexagons.

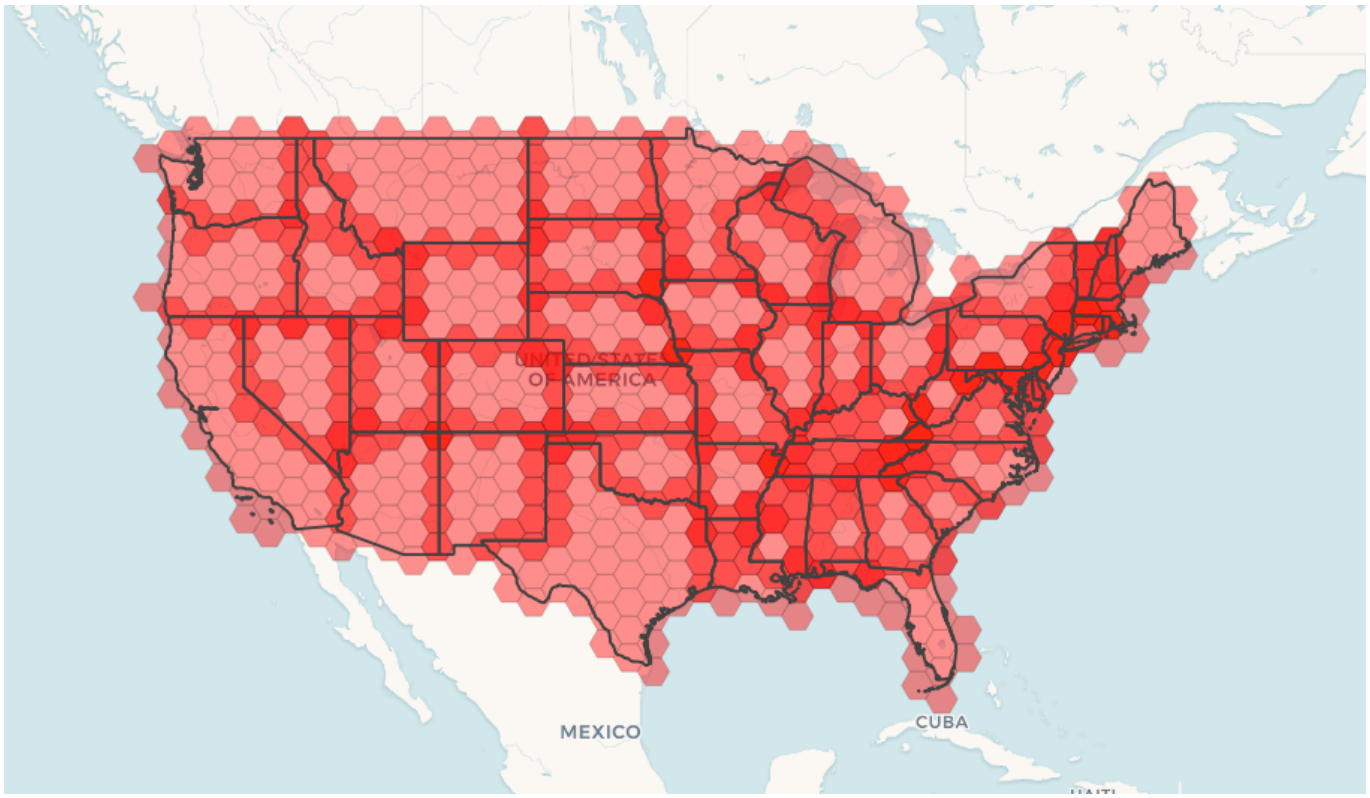
To do a point summary against a hexagonal tiling, generate a hexagon grid using the extent of the points as the bounds, then spatially join to that grid.

Code

```
SELECT COUNT(*), hexes.geom
FROM ST_HexagonGrid(
  10000,
  ST_SetSRID(ST_EstimatedExtent('pointtable', 'geom'), 3857)
) AS hexes
INNER JOIN pointtable AS pts
  ON ST_Intersects(pts.geom, hexes.geom)
GROUP BY hexes.geom;
```

Generating hex coverage of polygons.

If a grid is generated for each polygon boundary and filtered to intersecting cells, adjacent regions receive overlapping hexagons along their shared border. The build-time figure below uses two simple regions in place of an external administrative-boundary table, so the example is self-contained.



Code

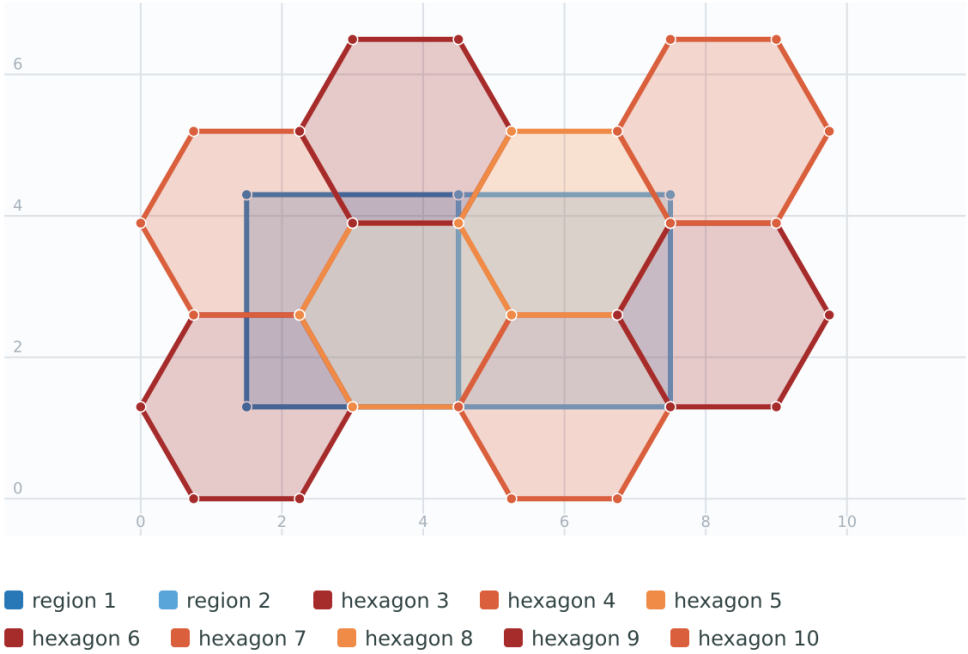
```
WITH admin1(gid, geom) AS (
    VALUES
        (1, ST_MakeEnvelope(0, 0, 3, 3, 3857)),
        (2, ST_MakeEnvelope(3, 0, 6, 3, 3857))
), coverage AS (
    SELECT hex.i, hex.j, hex.geom
    FROM admin1
    CROSS JOIN LATERAL ST_HexagonGrid(1.5, admin1.geom) AS hex
    WHERE ST_Intersects(admin1.geom, hex.geom)
    GROUP BY hex.i, hex.j, hex.geom
), layers AS (
    SELECT 'region' AS kind, gid AS position, geom
    FROM admin1
    UNION ALL
    SELECT
        'hexagon',
        1000 + row_number() OVER (ORDER BY i, j),
        geom
    FROM coverage
)
SELECT
    CASE kind WHEN 'region' THEN geom END AS input_region,
    CASE kind WHEN 'hexagon' THEN geom END AS hexagon
FROM layers
ORDER BY position;
```

Output

```
POLYGON((0 0,0 3,3 3,3 0,0 0)) | null
POLYGON((3 0,3 3,6 3,6 0,3 0)) | null
null | POLYGON((-1.5 0,-0.75 -1.3,0.75 -1.3,1.5 0,0.75 1.3,-0.75 1.3,-1.5 0))
null | POLYGON((-1.5 2.6,-0.75 1.3,0.75 1.3,1.5 2.6,0.75 3.9,-0.75 3.9,-1.5 2.6))
null | POLYGON((0.75 1.3,1.5 0,3 0,3.75 1.3,3 2.6,1.5 2.6,0.75 1.3))
null | POLYGON((0.75 3.9,1.5 2.6,3 2.6,3.75 3.9,3 5.2,1.5 5.2,0.75 3.9))
```

```
null | POLYGON((3 0,3.75 -1.3,5.25 -1.3,6 0,5.25 1.3,3.75 1.3,3 0))
null | POLYGON((3 2.6,3.75 1.3,5.25 1.3,6 2.6,5.25 3.9,3.75 3.9,3 2.6))
null | POLYGON((5.25 1.3,6 0,7.5 0,8.25 1.3,7.5 2.6,6 2.6,5.25 1.3))
null | POLYGON((5.25 3.9,6 2.6,7.5 2.6,8.25 3.9,7.5 5.2,6 5.2,5.25 3.9))
```

Figure



The same pattern applies to a real administrative-boundary table:



☒☒
The LATERAL keyword is implied for set-returning functions when referring to a prior table in the FROM list. So CROSS JOIN LATERAL, CROSS JOIN, or just plain , are equivalent constructs for this example.

Code

```
SELECT admin1.gid, hex.geom
FROM
    admin1
    CROSS JOIN
    ST_HexagonGrid(100000, admin1.geom) AS hex
WHERE
    adm0_a3 = 'USA'
    AND
    ST_Intersects(admin1.geom, hex.geom)
```

☒☒

ST_EstimatedExtent, ST_MakePoint, ST_Point, ST_SRID

7.3.15 ST_Hexagon

ST_Hexagon — Returns a single hexagon, using the provided edge size and cell coordinate within the hexagon grid space.

□□

geometry **ST_Hexagon**(float8 size, integer cell_i, integer cell_j, geometry origin);

□□

Uses the same hexagon tiling concept as ST_HexagonGrid, but generates just one hexagon at the desired cell coordinate. Optionally, can adjust origin coordinate of the tiling, the default origin is at 0,0.

Hexagons are generated with no SRID set, so use ST_SetSRID to set the SRID to the one you expect.

2.1.0 □□□□□□□□□□□□.

□□

Creating a hexagon at the origin.

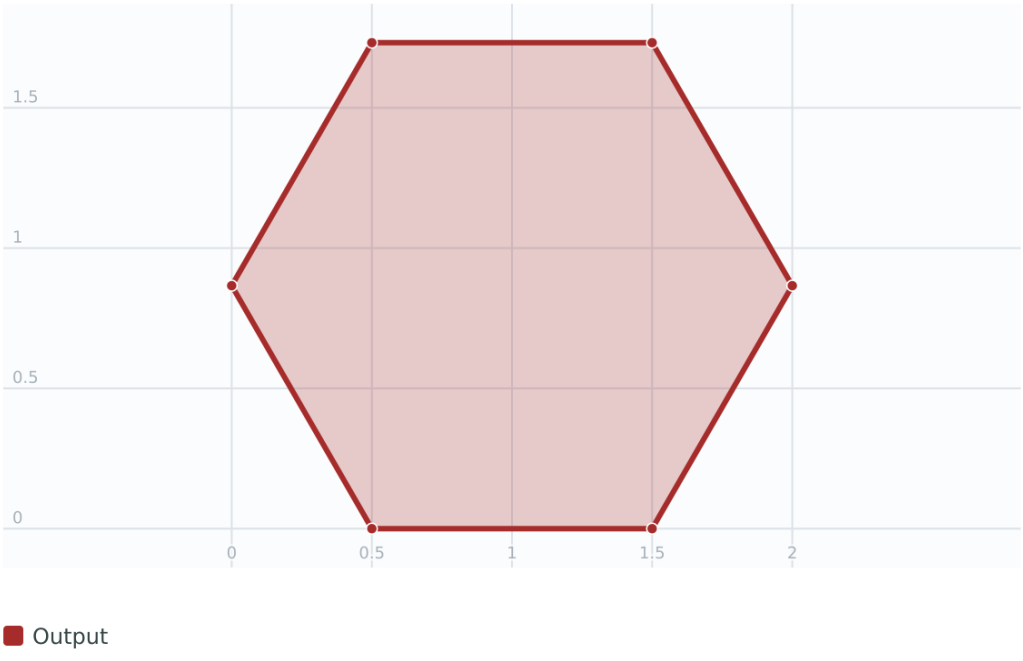
Code

```
SELECT ST_SetSRID(ST_Hexagon(1.0, 0, 0), 3857);
```

Output

```
POLYGON((-1 0,-0.5 -0.8660254037844387,0.5 -0.8660254037844387,1 0,0.5 0.8660254037844387,-0.5 0.8660254037844387,-1 ↩
    0))
```

Figure



□□

ST_TileEnvelope, ST_MakePoint, ST_SetSRID

7.3.16 ST_SquareGrid

ST_SquareGrid — Returns a set of grid squares and cell indices that completely cover the bounds of the geometry argument.

□□

setof record **ST_SquareGrid**(float8 size, geometry bounds);

□□

Starts with the concept of a square tiling of the plane. For a given planar SRS, and a given edge size, starting at the origin of the SRS, there is one unique square tiling of the plane, Tiling(SRS, Size). This function answers the question: what grids in a given Tiling(SRS, Size) overlap with a given bounds.

The SRS for the output squares is the SRS provided by the bounds geometry.

Doubling or edge size of the square generates a new parent tiling that perfectly fits with the original tiling. Standard web map tilings in mercator are just powers-of-two square grids in the mercator plane.

2.1.0 □□□□□□□□□□□□.

□□

Generating a 1 degree grid for a country.

The grid will fill the whole bounds of the country, so if you want just squares that touch the country you will have to filter afterwards with ST_Intersects.

Code

```
WITH grid AS (
SELECT (ST_SquareGrid(1, ST_Transform(geom, 4326))).*
FROM admin0 WHERE name = 'Canada'
)
  SELEcT geom
  FROM grid
```

Counting points in squares (using single chopped grid).

To do a point summary against a square tiling, generate a square grid using the extent of the points as the bounds, then spatially join to that grid. Note the estimated extent might be off from actual extent, so be cautious and at very least make sure you've analyzed your table.

Code

```
SELECT COUNT(*), squares.geom
FROM pointtable AS pts
INNER JOIN ST_SquareGrid(
  1000,
  ST_SetSRID(ST_EstimatedExtent('pointtable', 'geom'), 3857)
) AS squares
  ON ST_Intersects(pts.geom, squares.geom)
GROUP BY squares.geom;
```

Counting points in squares using set of grid per point.

This yields the same result as the first example but will be slower for a large number of points

Code

```
SELECT COUNT(*), squares.geom
FROM pointtable AS pts
INNER JOIN ST_SquareGrid(
  1000,
  pts.geom
) AS squares
  ON ST_Intersects(pts.geom, squares.geom)
GROUP BY squares.geom;
```

□□

ST_TileEnvelope, ST_Point, ST_SetSRID, ST_SRID

7.3.17 ST_Square

ST_Square — Returns a single square, using the provided edge size and cell coordinate within the square grid space.

□□

geometry **ST_Square**(float8 size, integer cell_i, integer cell_j, geometry origin='POINT(0 0)');

□□

Uses the same square tiling concept as ST_SquareGrid, but generates just one square at the desired cell coordinate. Optionally, can adjust origin coordinate of the tiling, the default origin is at 0,0.

Squares are generated with the SRID of the given origin. Use ST_SetSRID to set the SRID if the given origin has an unknown SRID (as is the case by default).

2.1.0 □□□□□□□□□□□□.

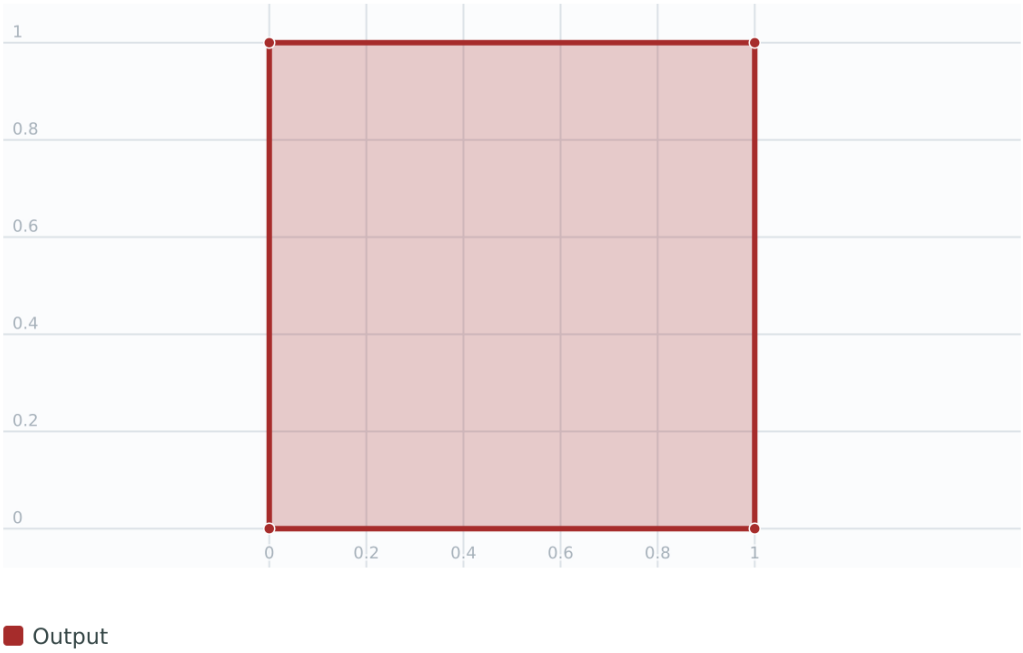
□□

Creating a square at the origin.

Code

```
SELECT ST_SetSRID(ST_Square(1.0, 0, 0), 3857);
```

Figure



☒☒

ST_TileEnvelope, ST_MakeLine, ST_MakePolygon

7.3.18 ST_Letters

ST_Letters — Returns the input letters rendered as geometry with a default start position at the origin and default text height of 100.

☒☒

geometry **ST_Letters**(text letters, json font);

☒☒

Uses a built-in font to render out a string as a multipolygon geometry. The default text height is 100.0, the distance from the bottom of a descender to the top of a capital. The default start position places the start of the baseline at the origin. Over-riding the font involves passing in a json map, with a character as the key, and base64 encoded TWKB for the font shape, with the fonts having a height of 1000 units from the bottom of the descenders to the tops of the capitals.

The text is generated at the origin by default, so to reposition and resize the text, first apply the `ST_Scale` function and then apply the `ST_Translate` function.

Availability: 3.3.0

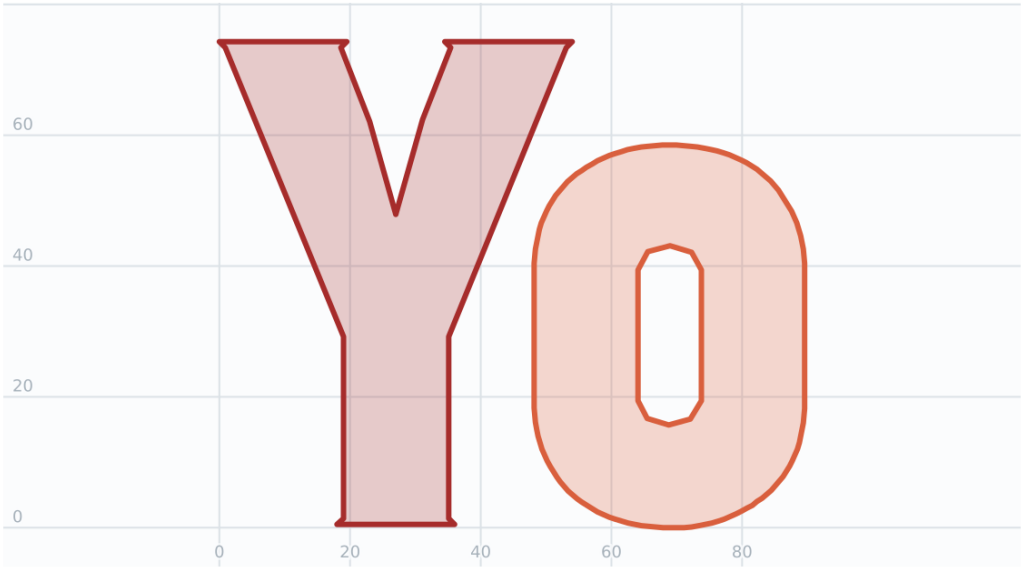
☒☒

Generating the word 'Yo'.

Code

```
SELECT ST_Letters('Yo');
```

Figure



Output

Generating the word 'PostGIS'.

Code

```
SELECT ST_Letters('PostGIS') AS letters;
```

Figure



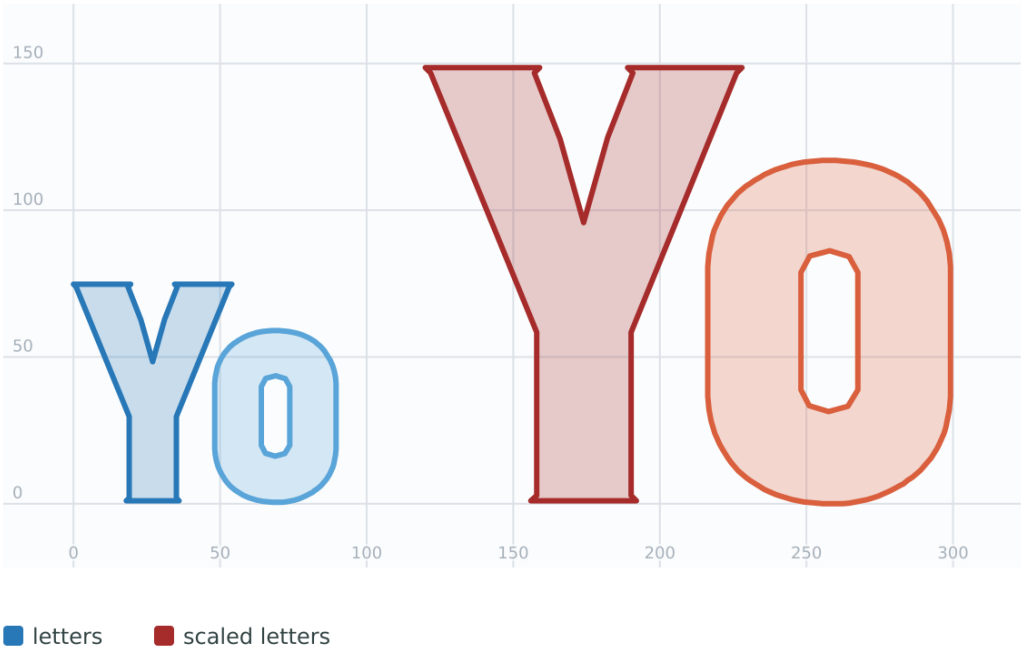
Output

Scaling and moving words.

Code

```
WITH letters AS (
    SELECT ST_Letters('Yo') AS geom
)
SELECT geom AS input_letters,
       ST_Translate(ST_Scale(geom, 2, 2), 120, 0)
           AS scaled_letters
FROM letters;
```

Figure



☒☒

ST_AsTWKB, ST_Scale, ST_Translate

7.3.19 ST_MakeNurbsCurve

ST_MakeNurbsCurve — Creates a NURBS (Non-Uniform Rational B-Spline) curve from control points, optional weights, and optional knot vector.

☒☒

geometry **ST_MakeNurbsCurve**(integer degree, geometry control_points, float8[] weights, float8[] knots);

☒☒

Creates a NURBS (Non-Uniform Rational B-Spline) curve geometry from the specified parameters. NURBS curves are smooth parametric curves that provide precise control over curve shape through control points, weights, and knot values.

The `degree` parameter specifies the polynomial degree of the curve (must be between 1 and 10). Higher degrees produce smoother curves.

The `control_points` must be a LINESTRING geometry defining the control polygon. The number of control points must be at least `degree + 1`.

The optional `weights` array assigns a weight to each control point. All weights must be positive. If NULL or omitted, uniform weights (1.0) are used, resulting in a non-rational B-spline.

The optional `knots` array defines the parameter space. If NULL or omitted, a uniform clamped knot vector is automatically generated. If provided, the knot vector must have exactly (`npoints + degree + 1`) elements and must be non-decreasing.

Availability: 3.7.0

□□

This example creates a degree-2 NURBS curve with uniform weights.

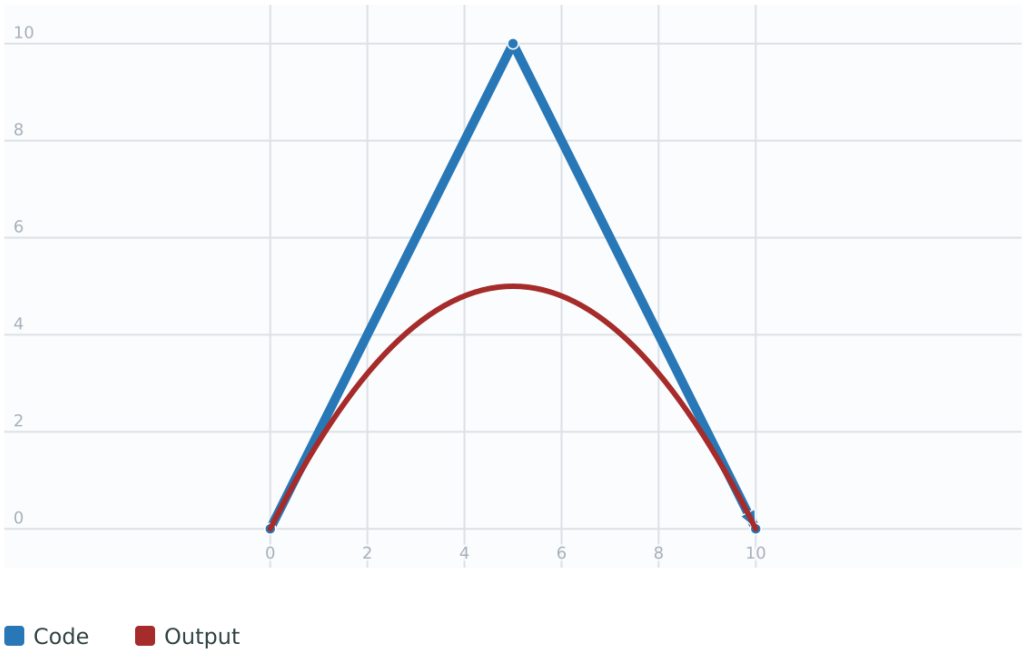
Code

```
SELECT ST_MakeNurbsCurve(
    2,
    'LINESTRING(0 0,5 10,10 0)'::geometry
);
```

Output

```
NURBSCURVE(DEGREE 2, CONTROLPOINTS(NURBSPOINT(WEIGHTEDPOINT(0 0),WEIGHT 1), NURBSPOINT(WEIGHTEDPOINT(5 10),WEIGHT 1),
    NURBSPOINT(WEIGHTEDPOINT(10 0),WEIGHT 1)),KNOTS (KNOT(0,3), KNOT(1,3)))
```

Figure



This example creates a rational NURBS curve with custom weights.

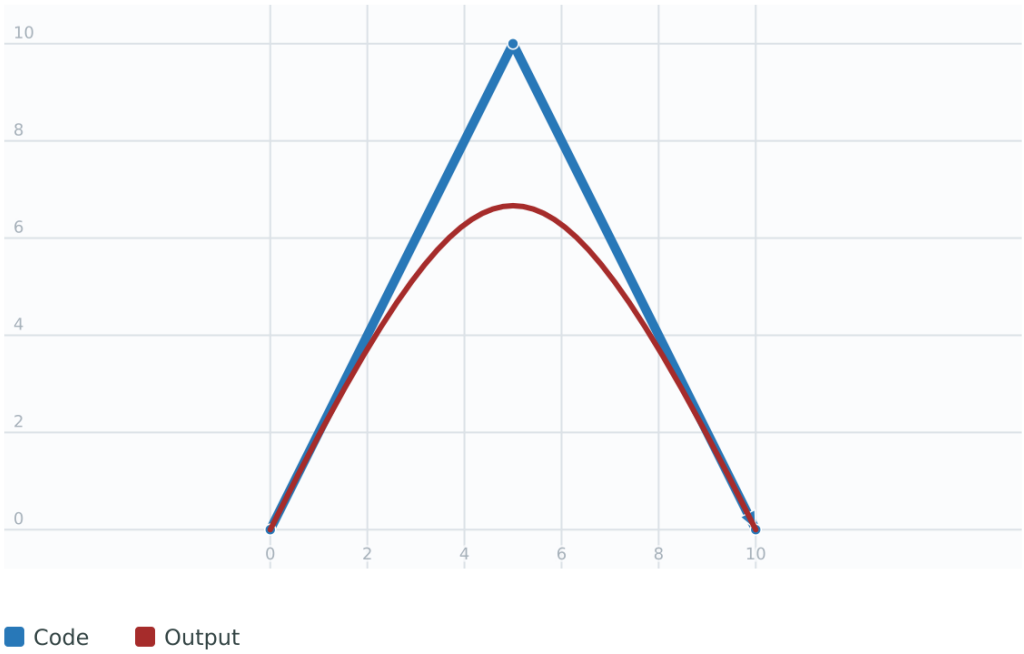
Code

```
SELECT ST_MakeNurbsCurve(
    2,
    'LINESTRING(0 0,5 10,10 0)'::geometry,
    ARRAY[1.0, 2.0, 1.0]
);
```

Output

```
NURBSCURVE(DEGREE 2, CONTROLPOINTS(NURBSPOINT(WEIGHTEDPOINT(0 0),WEIGHT 1), NURBSPOINT(WEIGHTEDPOINT(5 10),WEIGHT 2),  ↩
    NURBSPOINT(WEIGHTEDPOINT(10 0),WEIGHT 1)),KNOTS (KNOT(0,3), KNOT(1,3)))
```

Figure



This example creates a NURBS curve with a non-uniform knot vector.

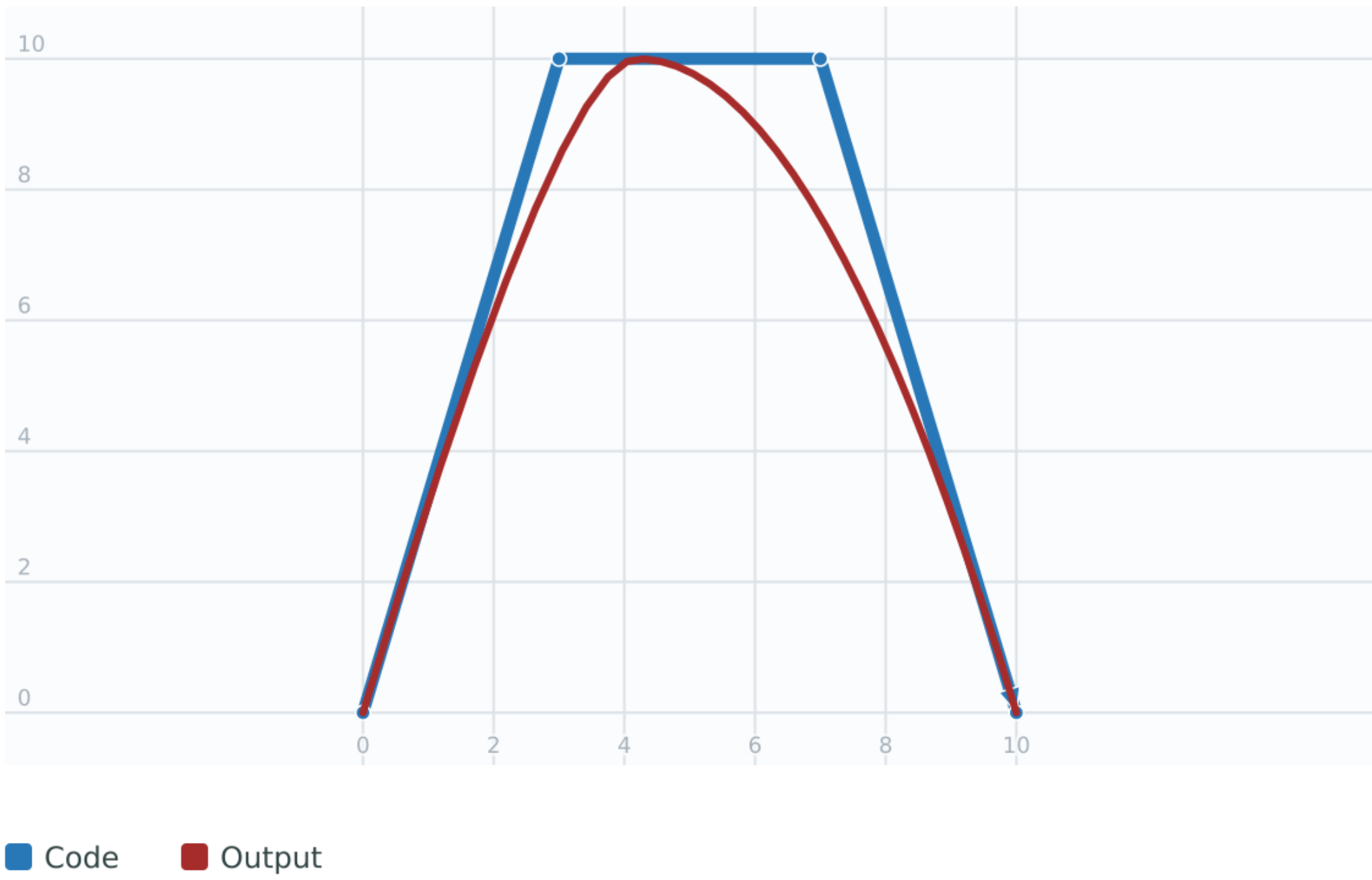
Code

```
SELECT ST_MakeNurbsCurve(
    2,
    'LINESTRING(0 0,3 10,7 10,10 0)'::geometry,
    ARRAY[1.0, 1.0, 1.0, 1.0],
    ARRAY[0.0, 0.0, 0.0, 0.3, 1.0, 1.0, 1.0]
);
```

Output

```
NURBSCURVE(DEGREE 2, CONTROLPOINTS(NURBSPOINT(WEIGHTEDPOINT(0 0),WEIGHT 1), NURBSPOINT(WEIGHTEDPOINT(3 10),WEIGHT 1),  ↩
    NURBSPOINT(WEIGHTEDPOINT(7 10),WEIGHT 1), NURBSPOINT(WEIGHTEDPOINT(10 0),WEIGHT 1)),KNOTS (KNOT(0,3), KNOT(0.3,1) ↩
    , KNOT(1,3)))
```

Figure



⊠⊠

ST_NurbsToLineString, ST_Evaluate, ST_ControlPoints

7.4 ⊠⊠⊠⊠⊠ (accessor)

7.4.1 GeometryType

GeometryType — ST_Geometry ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

text **GeometryType**(geometry geomA);

⊠⊠

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠: 'LINESTRING', 'POLYGON', 'MULTIPOINT' ⊠⊠.

OGC ⊠⊠ s2.1.1.1 - ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠.

> **Note**
> ⊠⊠
> ⊠⊠⊠⊠'POINTM' ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

□□□□: 2.0.0 □□□□□□□□□□, □□□□ TIN □□□□□□□□□□□□.

This method implements the OGC Simple Features Implementation Specification for SQL 1.1.

This method supports Circular Strings and Curves.

This function supports 3d and will not drop the z-index.

This function supports Polyhedral surfaces.

This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).

□□

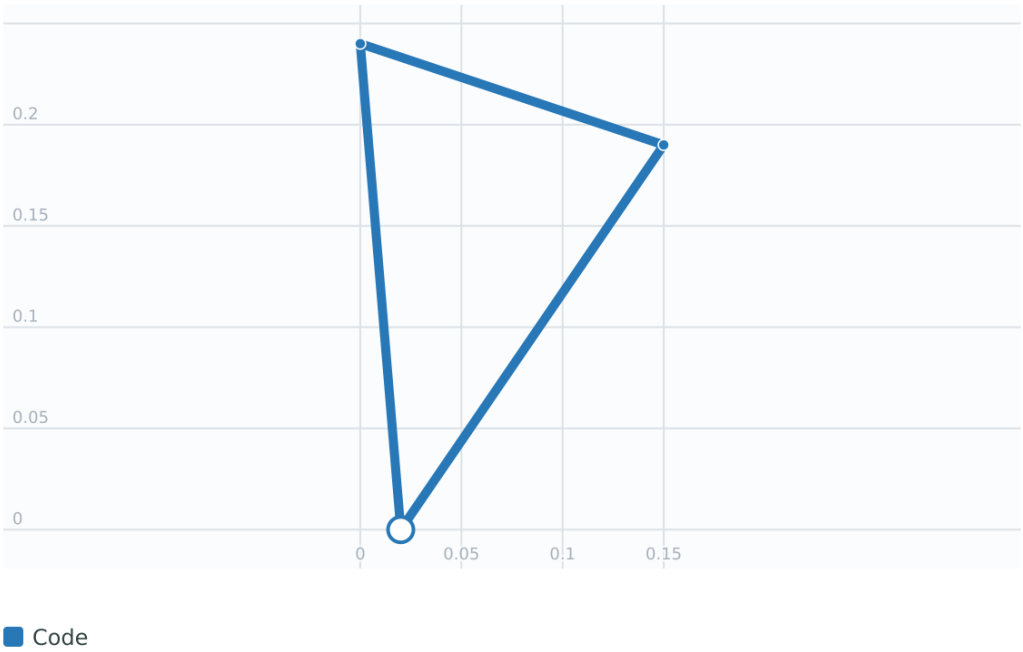
Code

```
SELECT GeometryType('LINESTRING(77.29 29.07,77.42 29.26,77.27 29.31,77.29 29.07)'::geometry ↩
    );
```

Output

```
LINESTRING
```

Figure



Code

```
SELECT ST_GeometryType('POLYHEDRALSURFACE( ((0 0 0,0 0 1,0 1 1,0 1 0,0 0 0)),
((0 0 0,0 1 0,1 1 0,1 0 0,0 0 0)),((0 0 0,1 0 0,1 0 1,0 0 1,0 0 0)),
((1 1 0,1 1 1,1 0 1,1 0 0,1 1 0)),
((0 1 0,0 1 1,1 1 1,1 1 0,0 1 0)),((0 0 1,1 0 1,1 1 1,0 1 1,0 0 1)) )');
    POLYHEDRALSURFACE
```

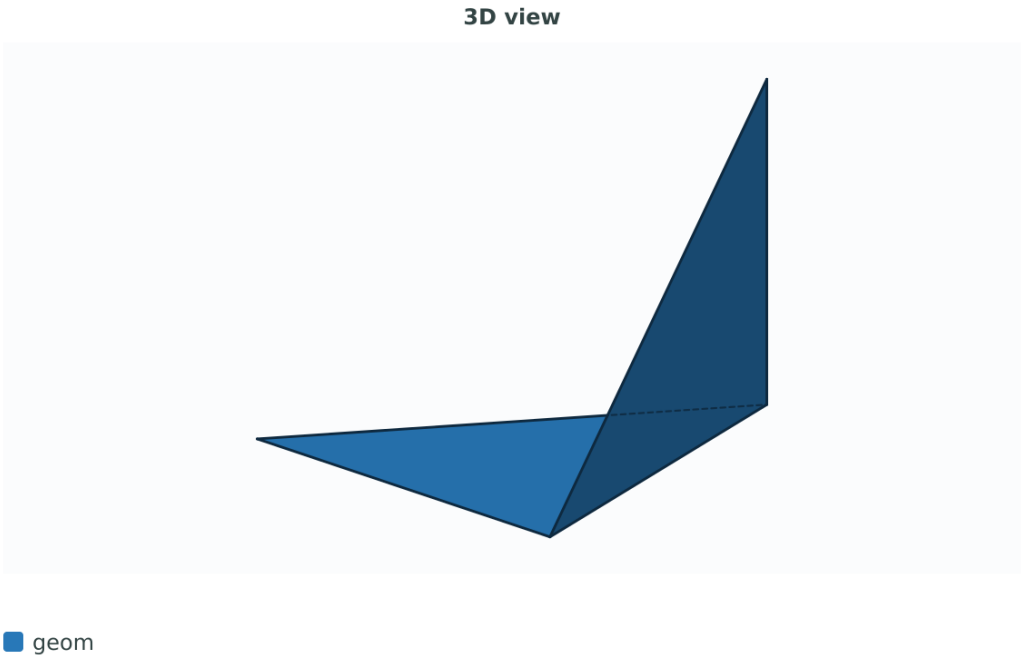
Code

```
SELECT GeometryType(geom) as result
FROM
  (SELECT
     'TIN (((
              0 0 0,
              0 0 1,
              0 1 0,
              0 0 0
          )),((
              0 0 0,
              0 1 0,
              1 1 0,
              0 0 0
          ))
          )'::geometry  AS geom
  ) AS g;
```

Output

```
TIN
```

Figure



□□

ST_GeometryType

7.4.2 ST_Boundary

ST_Boundary — □□□□□□□□□□□□□□□□□□□□□□□□□□□□□.

⊠⊠

geometry **ST_Boundary**(geometry geomA);

⊠⊠

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (closure) ⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠ (combinatorial boundary) ⊠ OGC ⊠⊠ ⊠⊠ 3.12.3.2 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠ (位相的) ⊠⊠⊠⊠⊠⊠ ⊠⊠⊠, OGC ⊠⊠⊠ 3.12.2 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (primitive) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠.

GEOS ⊠⊠⊠⊠⊠

> **⊠⊠**
> 2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ GEOMETRYCOLLECTION ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. 2.0.0 ⊠⊠⊠ ⊠⊠⊠⊠ (⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠) NULL ⊠⊠⊠⊠⊠⊠.

This method implements the OGC Simple Features Implementation Specification for SQL 1.1. OGC SPEC s2.1.1.1

This method implements the SQL/MM specification. SQL-MM IEC 13249-3: 5.1.17

This function supports 3d and will not drop the z-index.

⊠⊠⊠⊠: 2.1.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

Changed: 3.2.0 support for TIN, does not use geos, does not linearize curves

⊠⊠

LineString with boundary points overlaid.

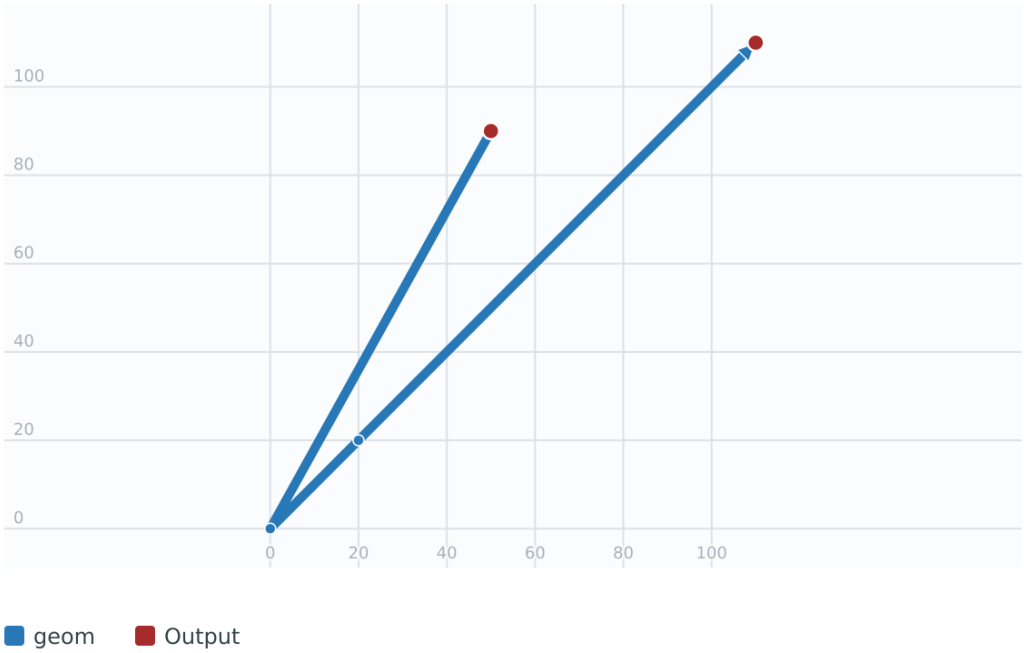
Code

```
SELECT ST_Boundary(geom)
FROM (SELECT 'LINESTRING(100 150,50 60,70 80,160 170)'::geometry As geom) As f;
```

Output

```
MULTIPOINT((100 150),(160 170))
```

Figure



Polygon holes with the boundary MultiLineString overlaid.

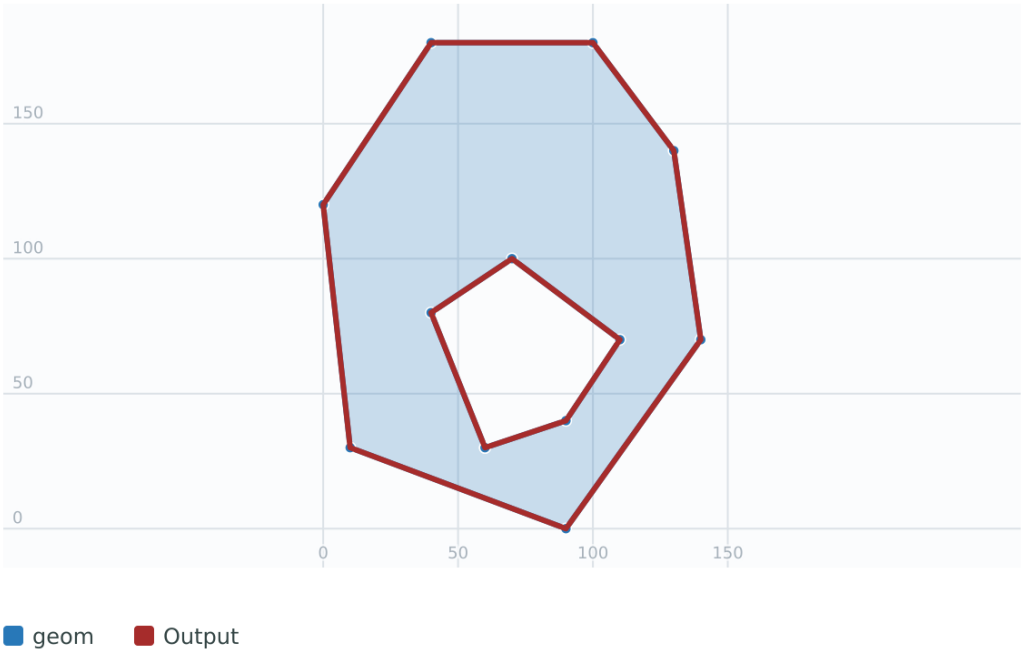
Code

```
SELECT ST_Boundary(geom)
FROM (SELECT
'POLYGON (( 10 130,50 190,110 190,140 150,150 80,100 10,20 40,10 130 ),
    ( 70 40,100 50,120 80,80 110,50 90,70 40 ))'::geometry As geom) As f;
```

Output

```
MULTILINESTRING((10 130,50 190,110 190,140 150,150 80,100 10,20 40,10 130),
       (70 40,100 50,120 80,80 110,50 90,70 40))
```

Figure



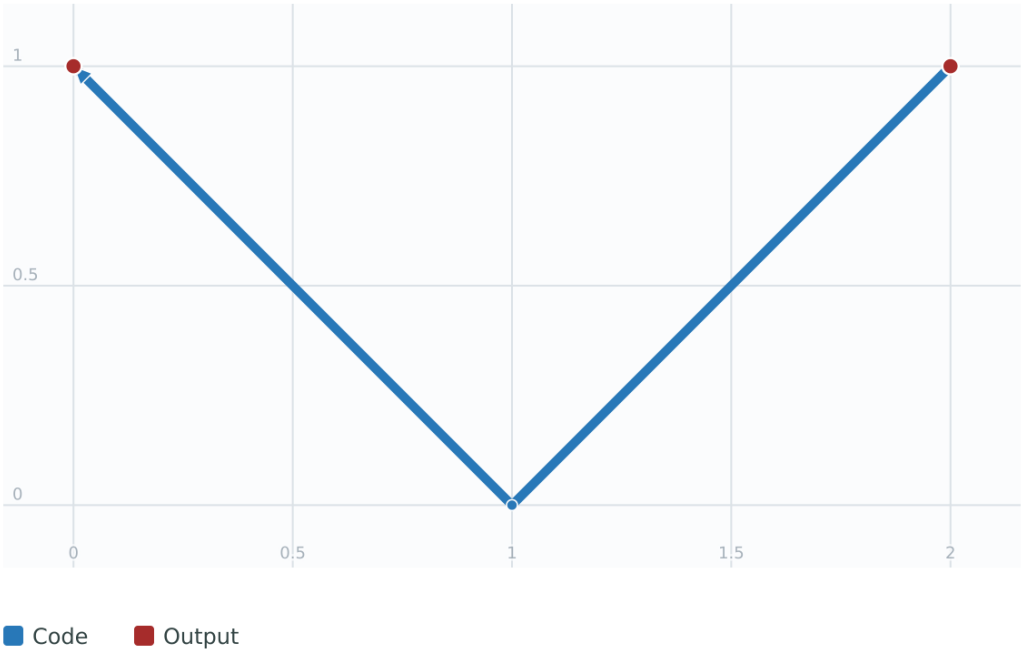
Code

```
SELECT ST_Boundary('LINESTRING(1 1,0 0,-1 1)');
```

Output

```
MULTIPOINT((1 1),(-1 1))
```

Figure



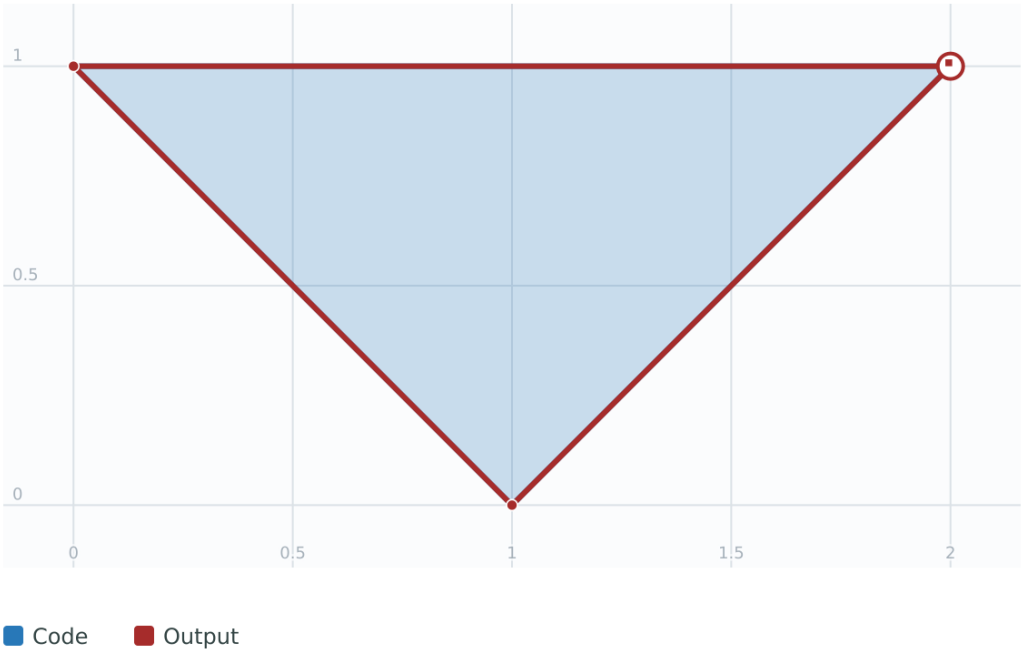
Code

```
SELECT ST_Boundary('POLYGON((1 1,0 0,-1 1,1 1))');
```

Output

```
LINESTRING(1 1,0 0,-1 1,1 1)
```

Figure



This example uses a 3D polygon.

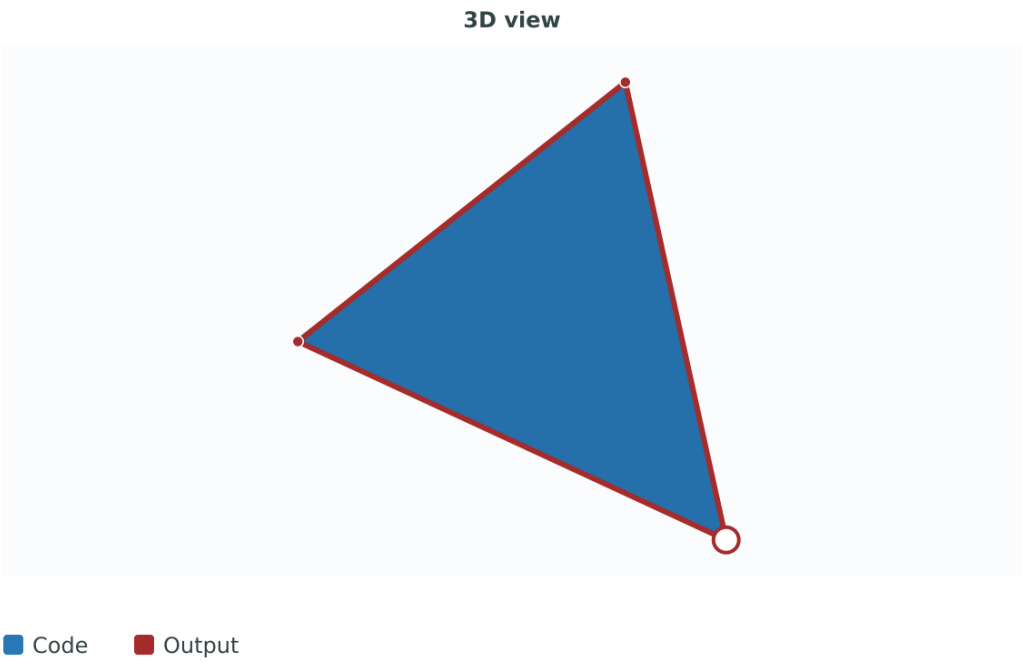
Code

```
SELECT ST_Boundary('POLYGON((1 1 1,0 0 1,-1 1 1,1 1 1))');
```

Output

```
LINESTRING(1 1 1,0 0 1,-1 1 1,1 1 1)
```

Figure



This example uses a 3D MultiLineString.

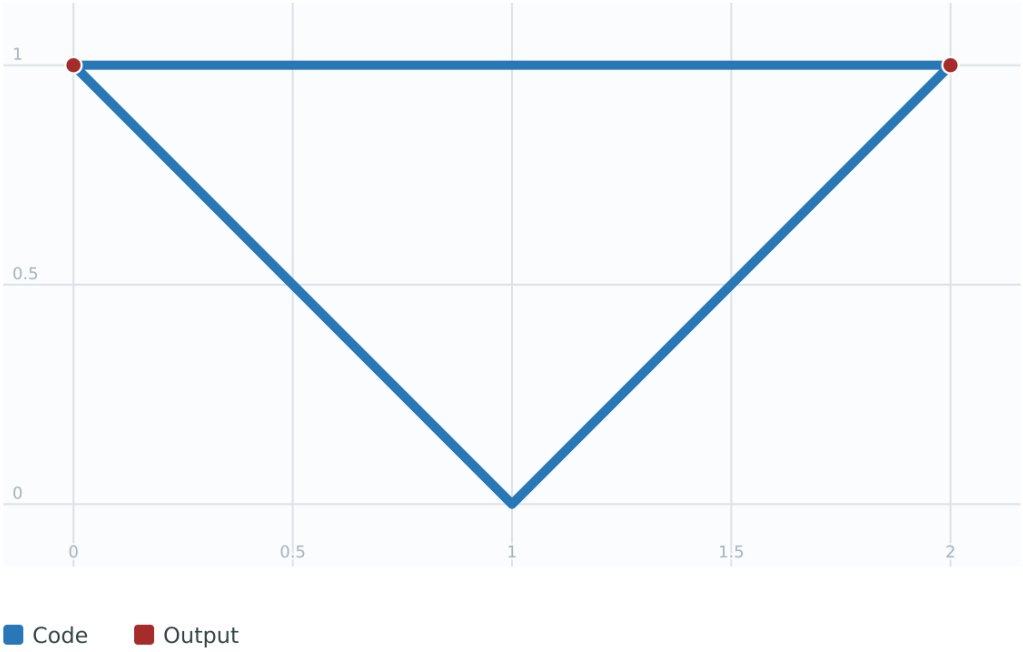
Code

```
SELECT ST_Boundary('MULTILINESTRING((1 1 1,0 0 0.5,-1 1 1),(1 1 0.5,0 0 0.5,-1 1 0.5,1 1 0.5) )');
```

Output

```
MULTIPOINT(1 1 1,-1 1 1)
```

Figure



☒☒

ST_AsText, ST_ExteriorRing, ST_MakePolygon

7.4.3 ST_BoundingDiagonal

ST_BoundingDiagonal — ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.

☒☒

geometry **ST_BoundingDiagonal**(geometry geom, boolean fits=false);

☒☒

☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒ 2 ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒, ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒ ☒☒☒.

fits ☒☒☒☒☒☒☒☒ (best fit) ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒ ☒☒☒☒☒☒☒☒☒☒ (☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒). ☒☒☒☒☒☒☒☒☒☒☒☒☒☒ ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.

☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒ SRID ☒☒☒☒☒☒☒☒☒☒.

> **☒☒**
>
> ☒☒☒☒☒☒ (☒☒☒☒☒☒☒☒☒☒) ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒ (☒☒☒☒☒) ☒ ☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.

2.2.0 ☒☒☒☒☒☒☒☒☒☒☒☒.

This function supports 3d and will not drop the z-index.

This function supports M coordinates.

☒☒

This example shows a polygon together with the diagonal of its bounding box.

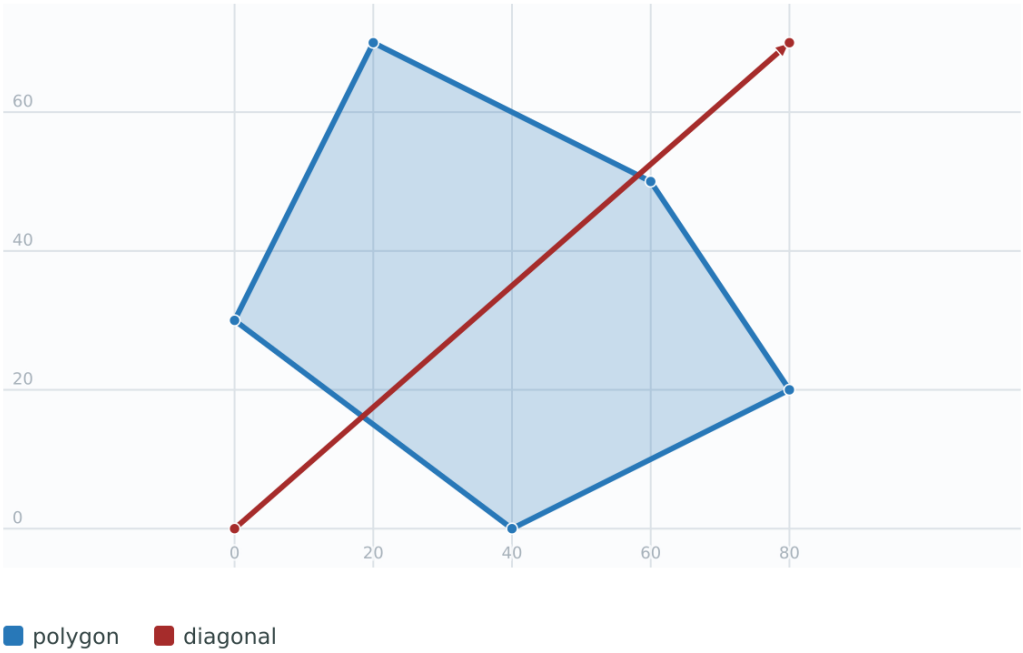
Code

```
WITH input AS (
  SELECT 'POLYGON((10 20,30 60,70 40,90 10,50 -10,10 20))'::geometry AS geom
)
SELECT geom AS input_polygon,
       ST_BoundingDiagonal(geom) AS diagonal
FROM input;
```

Output

```
POLYGON((10 20,30 60,70 40,90 10,50 -10,10 20)) | LINESTRING(10 -10,90 60)
```

Figure



This example returns the minimum X coordinate of a buffer around a point.

Code

```
SELECT ST_X(ST_StartPoint(ST_BoundingDiagonal(
  ST_Buffer(ST_Point(0, 0), 10)
)));
```

Output

```
-10
```

☒☒

ST_StartPoint, ST_EndPoint, ST_X, ST_Y, ST_Z, ST_M, &&&

7.4.4 ST_CoordDim

ST_CoordDim — ST_Geometry ☒☒☒☒☒☒☒☒☒☒☒☒.

☒☒

integer **ST_CoordDim**(geometry geomA);

☒☒

ST_Geometry ☒☒☒☒☒☒☒☒☒☒☒☒.
☒☒☒☒ MM ☒☒☒☒☒, ST_NDims ☒☒☒☒☒☒☒☒☒.

This method implements the OGC Simple Features Implementation Specification for SQL 1.1.

This method implements the SQL/MM specification. SQL-MM 3: 5.1.3

This method supports Circular Strings and Curves.

This function supports 3d and will not drop the z-index.

This function supports Polyhedral surfaces.

This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).

□□

Code

```
SELECT ST_CoordDim('CIRCULARSTRING(1 2 3,1 3 4,5 6 7,8 9 10,11 12 13)');
```

Output

```
3
```

Code

```
SELECT ST_CoordDim(ST_Point(1, 2));
```

Output

```
2
```

□□

ST_NDims

7.4.5 ST_Dimension

ST_Dimension — ST_Geometry □□□□□□□□□□□□.

□□

integer **ST_Dimension**(geometry g);

□□

□□□□□□□□□□□□□□□□, □□□□□□□□□□□□□□□□□□□□. OGC □□□ s2.1.1.1 □□□□□ `POINT` □ 0, `LINESTRING` □ 1, `POLYGON` □ 2, □□□ `GEOMETRYCOLLECTION` □□□□□□□□□□□□□□□□ □□□□. □□□□□□□ (□□) □□□□□□ null □□□□□□□.

This method implements the SQL/MM specification. SQL-MM 3: 5.1.2



□□
A TIN, including a closed TIN surface, is a surface geometry and has dimension 2. Use CG_MakeSolid to make the solid representation explicit; the resulting 3D solid has dimension 3. Solid geometry semantics, such as volume, are handled separately by the SQL/MM 3D solid functions (SQL-MM IEC 13249-3: 9.1).

□□□□: 2.0.0 □□□□□□□□□□ (polyhedral surface) □ TIN □□□□□□. □□□□□□□□□□□□□□□□□□□□□□□□□□□.



□□
2.0.0 □□□□□□□□□□□□□□□□□□□□□□□□□□□.

This function supports Polyhedral surfaces.

This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).

□□

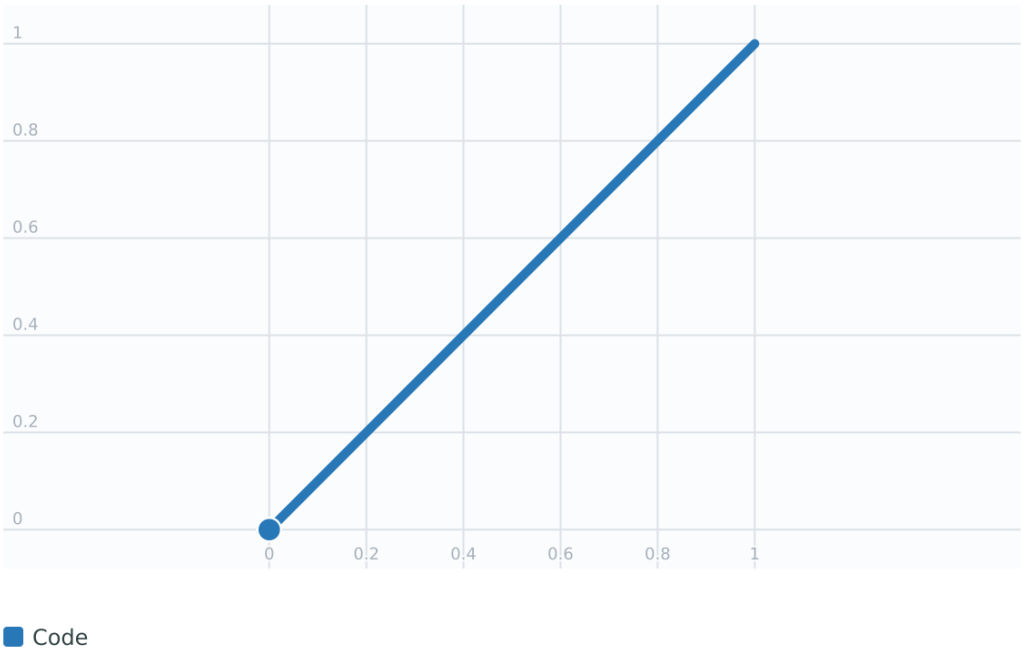
Code

```
SELECT ST_Dimension('GEOMETRYCOLLECTION(LINESTRING(1 1,0 0), POINT(0 0))');
```

Output

```
1
```

Figure



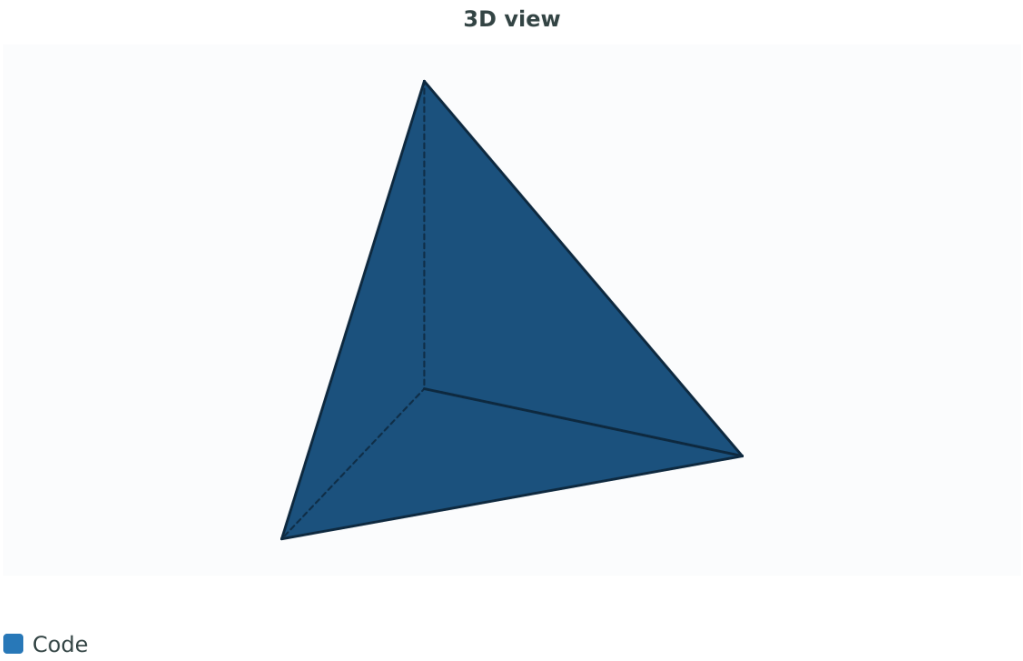
Code

```
SELECT ST_Dimension('TIN Z (((0 0 0,0 0 1,0 1 0,0 0 0)),((0 0 0,0 1 0,1 0 0,0 0 0)),((0 0 ↩
    0,1 0 0,0 0 1,0 0 0)),((1 0 0,0 1 0,0 0 1,1 0 0)))');
```

Output

```
2
```

Figure



□□

ST_NDims, CG_MakeSolid, CG_Volume

7.4.6 ST_Dump

ST_Dump — Returns a set of `geometry_dump` rows for the components of a geometry.

□□

geometry_dump[] **ST_Dump**(geometry g1);

□□

A set-returning function (SRF) that extracts the components of a geometry. It returns a set of geometry_dump rows, each containing a geometry (*geom* field) and an array of integers (*path* field).

For an atomic geometry type (POINT,LINESTRING,POLYGON) a single record is returned with an empty *path* array and the input geometry as *geom*. For a collection or multi-geometry a record is

returned for each of the collection components, and the *path* denotes the position of the component inside the collection.

ST_Dump is useful for expanding geometries. It is the inverse of a ST_Collect / GROUP BY, in that it creates new rows. For example it can be use to expand MULTIPOLYGONS into POLYGONS.

□□□□: 2.0.0 □□□□□□□□□□, □□□□ TIN □□□□□□□□□□□□.

Availability: PostGIS 1.0.0RC1. Requires PostgreSQL 7.3 or higher.



□□
1.3.4 □□□□□□□□□□□□□□□□ (curve) □□□□□□□□□□□□□□□□□□□□□□□□. 1.3.4 □□□□□□□□□□□□□□□□.

This method supports Circular Strings and Curves.

This function supports Polyhedral surfaces.

This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).

This function supports 3d and will not drop the z-index.

□□

Dump the components of geometries stored in a table.

Code

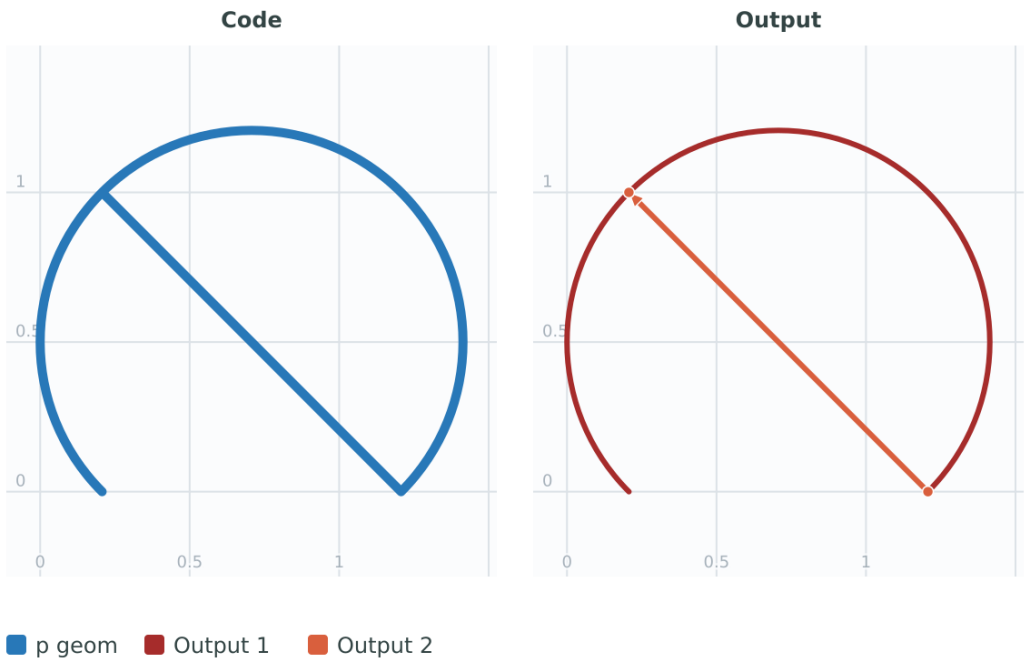
```
SELECT sometable.field1,
       (ST_Dump(sometable.geom)).geom AS geom
FROM sometable;
```

Break a CompoundCurve into its constituent LineStrings and CircularStrings.

Code

```
SELECT a.geom, ST_HasArc(a.geom)
  FROM ( SELECT (ST_Dump(p_geom)).geom AS geom
         FROM (SELECT 'COMPOUNDCURVE(CIRCULARSTRING(0 0,1 1,1 0),(1 0,0 1))'::geometry AS ↩
    p_geom) AS b
       ) AS a;
```

Figure

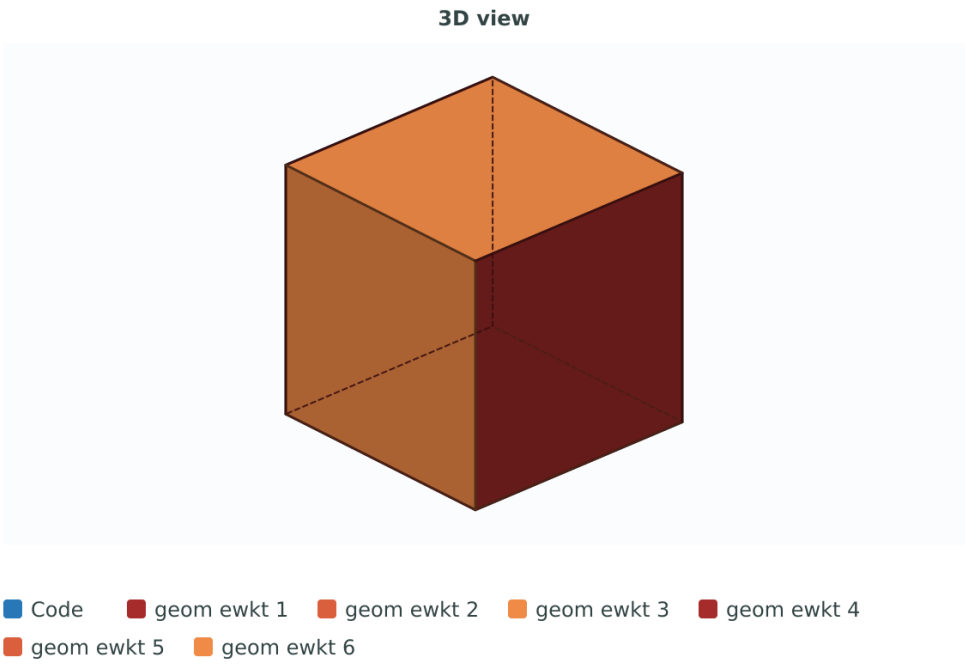


This example breaks a PolyhedralSurface into its faces.

Code

```
SELECT (a.p_geom).path[1] As path,(a.p_geom).geom As geom_ewkt
  FROM (SELECT ST_Dump('POLYHEDRALSURFACE(
((0 0 0,0 0 1,0 1 1,0 1 0,0 0 0)),
((0 0 0,0 1 0,1 1 0,1 0 0,0 0 0)),((0 0 0,1 0 0,1 0 1,0 0 1,0 0 0)),((1 1 0,1 1 1,1 0 1,1 0 ↩
    0,1 1 0)),
((0 1 0,0 1 1,1 1 1,1 1 0,0 1 0)),((0 0 1,1 0 1,1 1 1,0 1 1,0 0 1))
)' ) AS p_geom )  AS a;
```

Figure

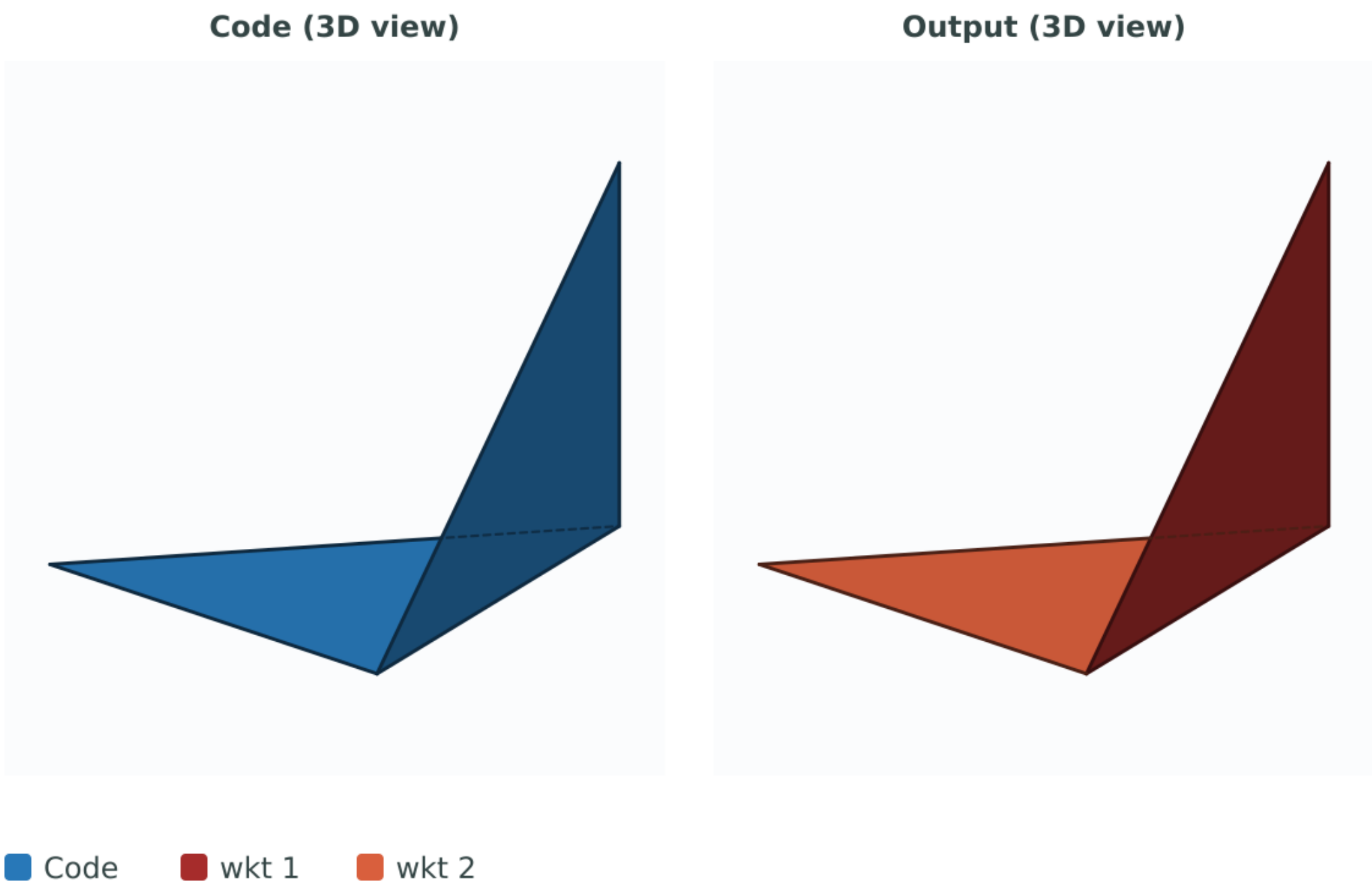


This example uses a TIN geometry.

Code

```
SELECT (g.gdump).path,(g.gdump).geom as wkt
  FROM
    (SELECT
       ST_Dump('TIN (((
                0 0 0,
                0 0 1,
                0 1 0,
                0 0 0
            )),((
                0 0 0,
                0 1 0,
                1 1 0,
                0 0 0
            ))
            )' ) AS gdump
    ) AS g;
```

Figure



⊠⊠

[geometry_dump](#), [ST_GeomFromEWKT](#), [ST_Dump](#), [ST_GeometryN](#), [ST_NumGeometries](#)

7.4.7 ST_DumpPoints

ST_DumpPoints — ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

geometry_dump[] **ST_DumpPoints**(geometry geom);

⊠⊠

A set-returning function (SRF) that extracts the coordinates (vertices) of a geometry. It returns a set of geometry_dump rows, each containing a geometry (*geom* field) and an array of integers (*path* field).

- the *geom* field `POINT`s represent the coordinates of the supplied geometry.
- the *path* field (an `integer[]`) is an index enumerating the coordinate positions in the elements of the supplied geometry. The indices are 1-based. For example, for a `LINESTRING` the paths are `{i}` where `i` is the `nth` coordinate in the `LINESTRING`. For a `POLYGON` the paths are `{i,j}` where `i` is the ring number (1 is outer; inner rings follow) and `j` is the coordinate position in the ring.

To obtain a single geometry containing the coordinates use ST_Points.

Enhanced: 2.1.0 Faster speed. Reimplemented as native-C.

⊠⊠⊠⊠: 2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠ TIN ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

1.5.0 □□□□□□□□□□□□.

This method supports Circular Strings and Curves.

This function supports Polyhedral surfaces.

This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).

This function supports 3d and will not drop the z-index.

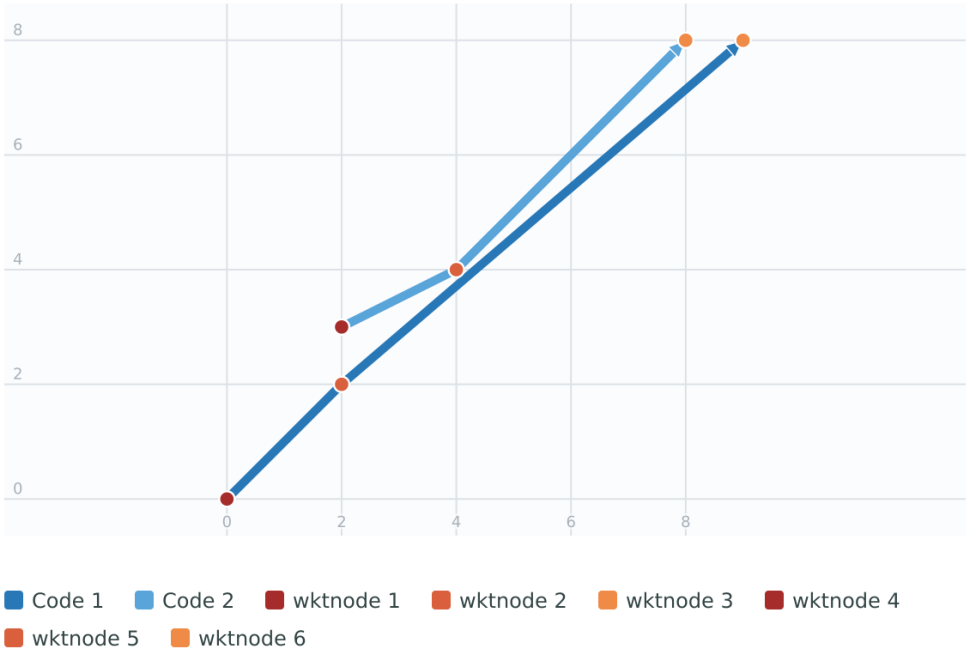
□□

Explode a table of LineStrings into their vertices.

Code

```
SELECT edge_id, (dp).path[1] As index,(dp).geom As wktnode
FROM (SELECT 1 As edge_id,
    ST_DumpPoints(ST_GeomFromText('LINESTRING(1 2,3 4,10 10)')) AS dp
     UNION ALL
     SELECT 2 As edge_id,
    ST_DumpPoints(ST_GeomFromText('LINESTRING(3 5,5 6,9 10)')) AS dp
   ) As foo;
```

Figure



Dump every vertex from a mixed GeometryCollection.

Code

```
SELECT path,geom
FROM (
  SELECT (ST_DumpPoints(g.geom)).*
  FROM
    (SELECT
       'GEOMETRYCOLLECTION(
```

```
          POINT ( 0 1 ),
          LINESTRING ( 0 3,3 4 ),
          POLYGON (( 2 0,2 3,0 2,2 0 )),
          POLYGON (( 3 0,3 3,6 3,6 0,3 0 ),
                   ( 5 1,4 2,5 2,5 1 )),
          MULTIPOLYGON (
                  (( 0 5,0 8,4 8,4 5,0 5 ),
                   ( 1 6,3 6,2 7,1 6 )),
                  (( 5 4,5 8,6 7,5 4 ))
          )
        )'::geometry AS geom
    ) AS g
  ) j;
```

Output

```
path    | st_astext
----------+-----------
 {1,1}    | POINT(0 1)
 {2,1}    | POINT(0 3)
 {2,2}    | POINT(3 4)
 {3,1,1}  | POINT(2 0)
 {3,1,2}  | POINT(2 3)
 {3,1,3}  | POINT(0 2)
 {3,1,4}  | POINT(2 0)
 {4,1,1}  | POINT(3 0)
 {4,1,2}  | POINT(3 3)
 {4,1,3}  | POINT(6 3)
 {4,1,4}  | POINT(6 0)
 {4,1,5}  | POINT(3 0)
 {4,2,1}  | POINT(5 1)
 {4,2,2}  | POINT(4 2)
 {4,2,3}  | POINT(5 2)
 {4,2,4}  | POINT(5 1)
 {5,1,1,1} | POINT(0 5)
 {5,1,1,2} | POINT(0 8)
 {5,1,1,3} | POINT(4 8)
 {5,1,1,4} | POINT(4 5)
 {5,1,1,5} | POINT(0 5)
 {5,1,2,1} | POINT(1 6)
 {5,1,2,2} | POINT(3 6)
 {5,1,2,3} | POINT(2 7)
 {5,1,2,4} | POINT(1 6)
 {5,2,1,1} | POINT(5 4)
 {5,2,1,2} | POINT(5 8)
 {5,2,1,3} | POINT(6 7)
 {5,2,1,4} | POINT(5 4)
(29 rows)
```

Collect the dumped vertices to visualize them over the source geometry.

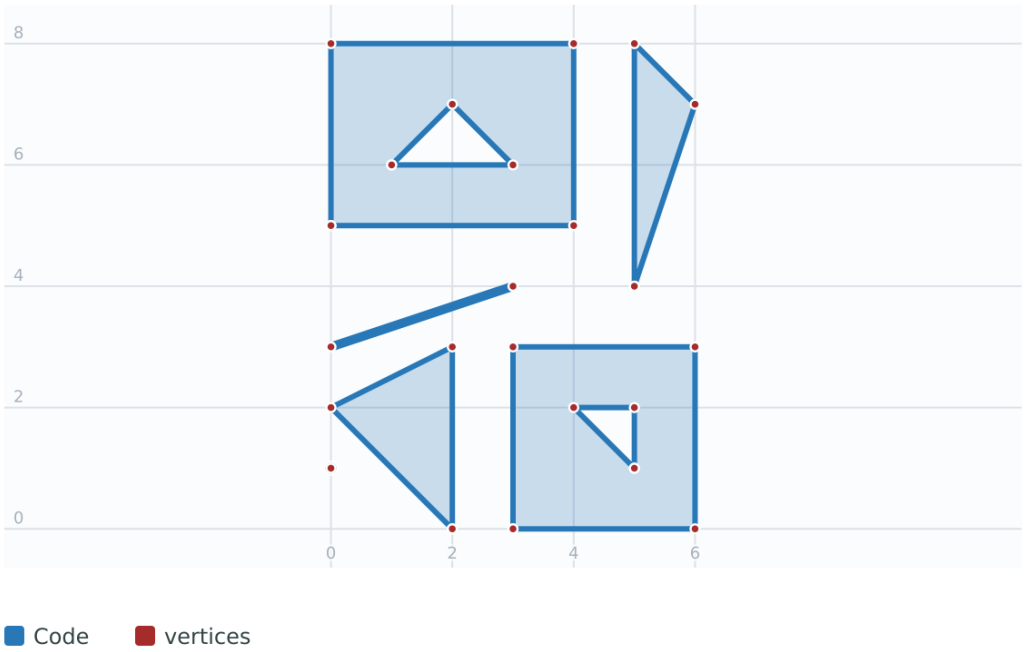
Code

```
WITH source(geom) AS (VALUES (
  'GEOMETRYCOLLECTION(
     POINT (0 1),
     LINESTRING (0 3,3 4),
     POLYGON ((2 0,2 3,0 2,2 0)),
     POLYGON ((3 0,3 3,6 3,6 0,3 0),(5 1,4 2,5 2,5 1)),
     MULTIPOLYGON (((0 5,0 8,4 8,4 5,0 5),(1 6,3 6,2 7,1 6)),
                   ((5 4,5 8,6 7,5 4))))'::geometry
))
SELECT ST_Collect((dp).geom ORDER BY (dp).path) AS vertices
FROM source
CROSS JOIN LATERAL ST_DumpPoints(source.geom) AS dp;
```

Output

```
MULTIPOINT((0 1),(0 3),(3 4),(2 0),(2 3),(0 2),(2 0),(3 0),(3 3),(6 3),(6 0),(3 0),(5 1),(4 2),(5 2),(5 1),(0 5),(0 8) ↩
    ,(4 8),(4 5),(0 5),(1 6),(3 6),(2 7),(1 6),(5 4),(5 8),(6 7),(5 4))
```

Figure



This example uses a cube represented as a PolyhedralSurface.

Code

```
SELECT (g.gdump).path,(g.gdump).geom as wkt
  FROM
    (SELECT
       ST_DumpPoints('POLYHEDRALSURFACE( ((0 0 0,0 0 1,0 1 1,0 1 0,0 0 0)),
((0 0 0,0 1 0,1 1 0,1 0 0,0 0 0)),((0 0 0,1 0 0,1 0 1,0 0 1,0 0 0)),
((1 1 0,1 1 1,1 0 1,1 0 0,1 1 0)),
((0 1 0,0 1 1,1 1 1,1 1 0,0 1 0)),((0 0 1,1 0 1,1 1 1,0 1 1,0 0 1)) )' ) AS gdump
    ) AS g;
```

Output

```
path   |     wkt
---------+-------------
 {1,1,1} | POINT(0 0 0)
 {1,1,2} | POINT(0 0 1)
 {1,1,3} | POINT(0 1 1)
 {1,1,4} | POINT(0 1 0)
 {1,1,5} | POINT(0 0 0)
 {2,1,1} | POINT(0 0 0)
 {2,1,2} | POINT(0 1 0)
 {2,1,3} | POINT(1 1 0)
 {2,1,4} | POINT(1 0 0)
 {2,1,5} | POINT(0 0 0)
 {3,1,1} | POINT(0 0 0)
 {3,1,2} | POINT(1 0 0)
 {3,1,3} | POINT(1 0 1)
 {3,1,4} | POINT(0 0 1)
 {3,1,5} | POINT(0 0 0)
 {4,1,1} | POINT(1 1 0)
 {4,1,2} | POINT(1 1 1)
 {4,1,3} | POINT(1 0 1)
 {4,1,4} | POINT(1 0 0)
 {4,1,5} | POINT(1 1 0)
 {5,1,1} | POINT(0 1 0)
```

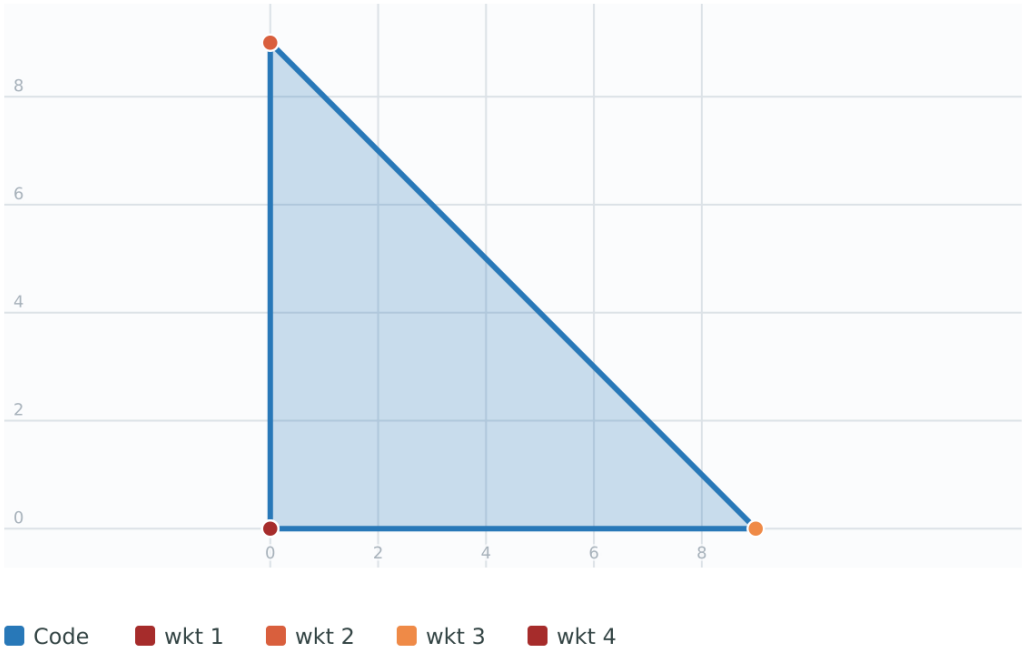
```
 {5,1,2} | POINT(0 1 1)
 {5,1,3} | POINT(1 1 1)
 {5,1,4} | POINT(1 1 0)
 {5,1,5} | POINT(0 1 0)
 {6,1,1} | POINT(0 0 1)
 {6,1,2} | POINT(1 0 1)
 {6,1,3} | POINT(1 1 1)
 {6,1,4} | POINT(0 1 1)
 {6,1,5} | POINT(0 0 1)
(30 rows)
```

This example uses a Triangle geometry.

Code

```
SELECT (g.gdump).path,(g.gdump).geom as wkt
  FROM
    (SELECT
       ST_DumpPoints('TRIANGLE ((
                0 0,
                0 9,
                9 0,
                0 0
            ))' ) AS gdump
    ) AS g;
```

Figure



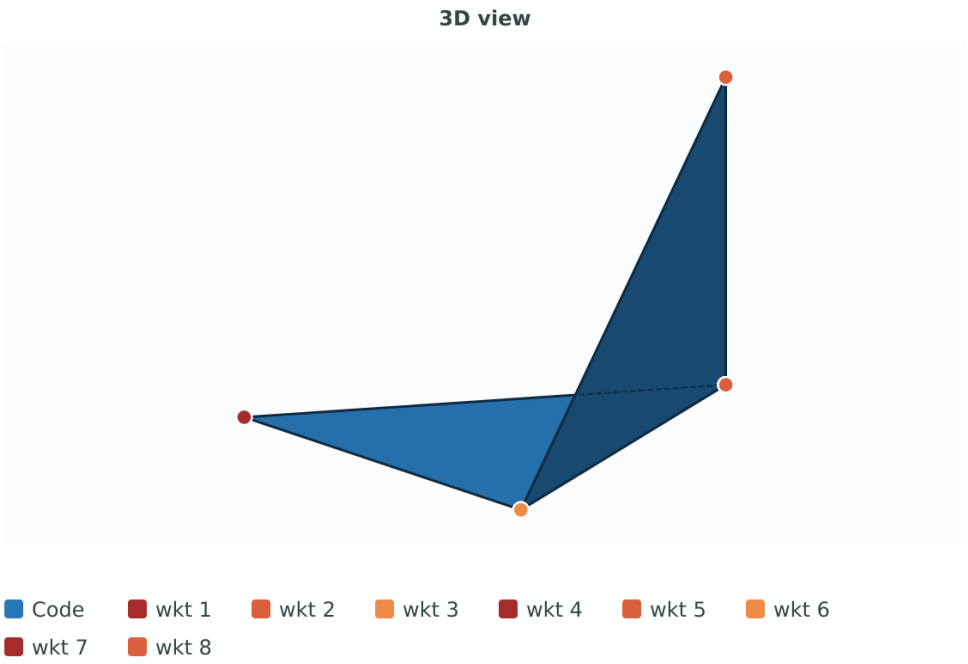
This example uses a TIN geometry.

Code

```
SELECT (g.gdump).path,(g.gdump).geom as wkt
  FROM
    (SELECT
       ST_DumpPoints('TIN (((
                0 0 0,
                0 0 1,
                0 1 0,
                0 0 0
```

```
            )),((
                0 0 0,
                0 1 0,
                1 1 0,
                0 0 0
            ))
            )' ) AS gdump
    ) AS g;
```

Figure



☒☒

geometry_dump, ST_GeomFromEWKT, ST_Dump, ST_GeometryN, ST_NumGeometries

7.4.8 ST_DumpSegments

ST_DumpSegments — ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.

☒☒

geometry_dump[] **ST_DumpSegments**(geometry geom);

☒☒

A set-returning function (SRF) that extracts the segments of a geometry. It returns a set of geometry_dump rows, each containing a geometry (*geom* field) and an array of integers (*path* field).

- the *geom* field `LINESTRING`s represent the linear segments of the supplied geometry, while the `CIRCULARSTRING`s represent the arc segments.

- the *path* field (an `integer[]`) is an index enumerating the segment start point positions in the elements of the supplied geometry. The indices are 1-based. For example, for a `LINESTRING` the paths are `{i}` where `i` is the `nth` segment start point in the `LINESTRING`. For a `POLYGON` the paths are `{i,j}` where `i` is the ring number (1 is outer; inner rings follow) and `j` is the segment start point position in the ring.

Availability: 3.2.0

This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).

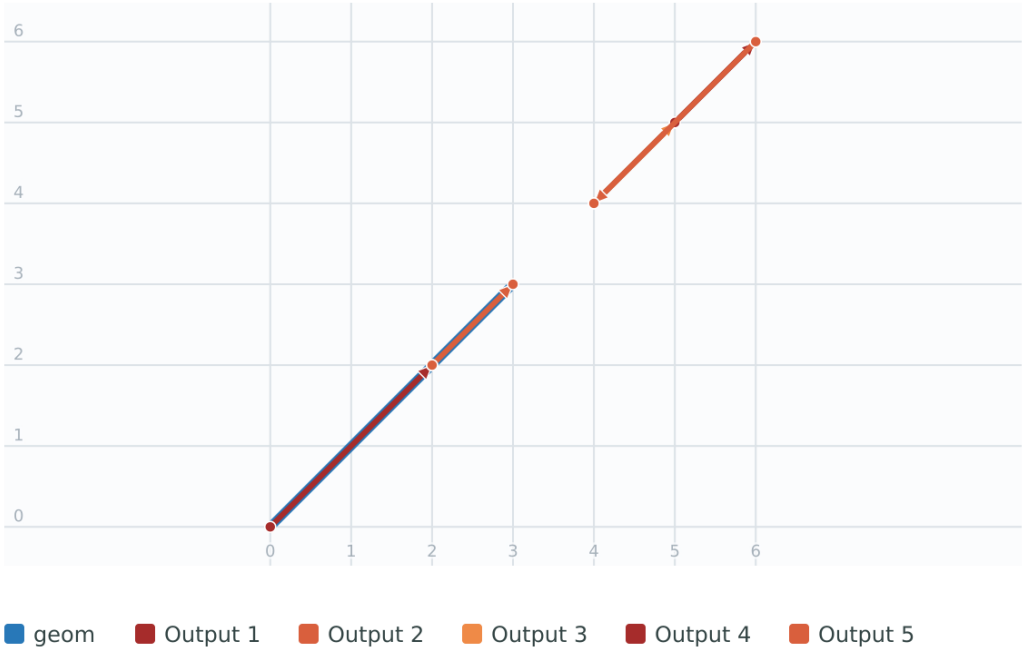
This function supports 3d and will not drop the z-index.

□□□□

Code

```
SELECT path,geom
FROM (
    SELECT (ST_DumpSegments(g.geom)).*
    FROM (SELECT 'GEOMETRYCOLLECTION(
    LINESTRING(1 1,3 3,4 4),
    POLYGON((5 5,6 6,7 7,5 5))
)'::geometry AS geom
        ) AS g
) j;
```

Figure



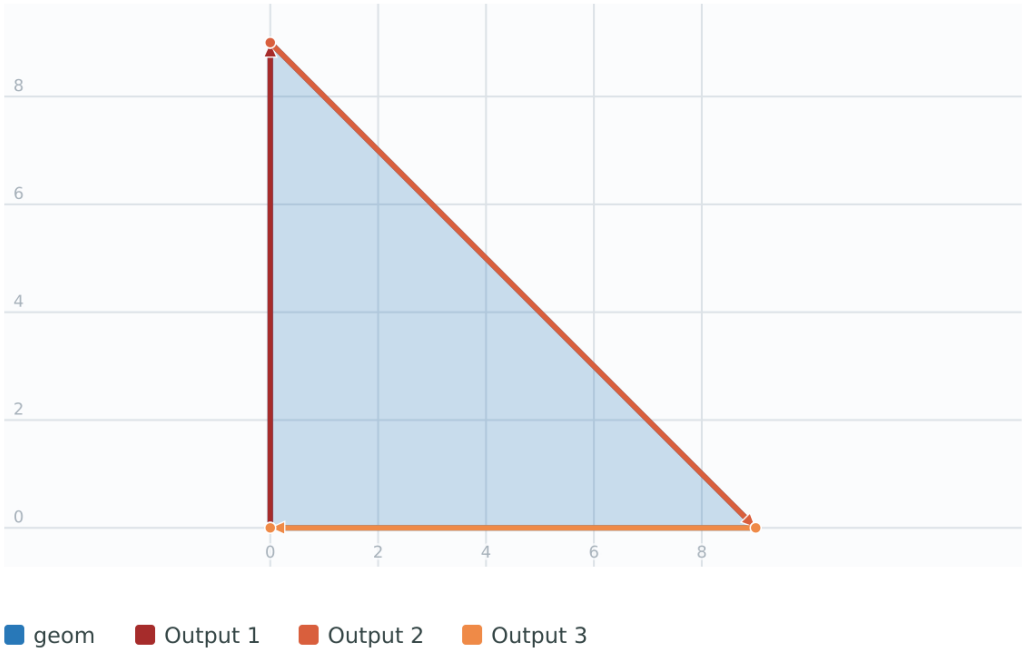
□□□□□, TIN □□□□□□

This example uses a Triangle geometry.

Code

```
SELECT path,geom
FROM (
    SELECT (ST_DumpSegments(g.geom)).*
    FROM (SELECT 'TRIANGLE((
        0 0,
        0 9,
        9 0,
        0 0
    ))'::geometry AS geom
        ) AS g
) j;
```

Figure

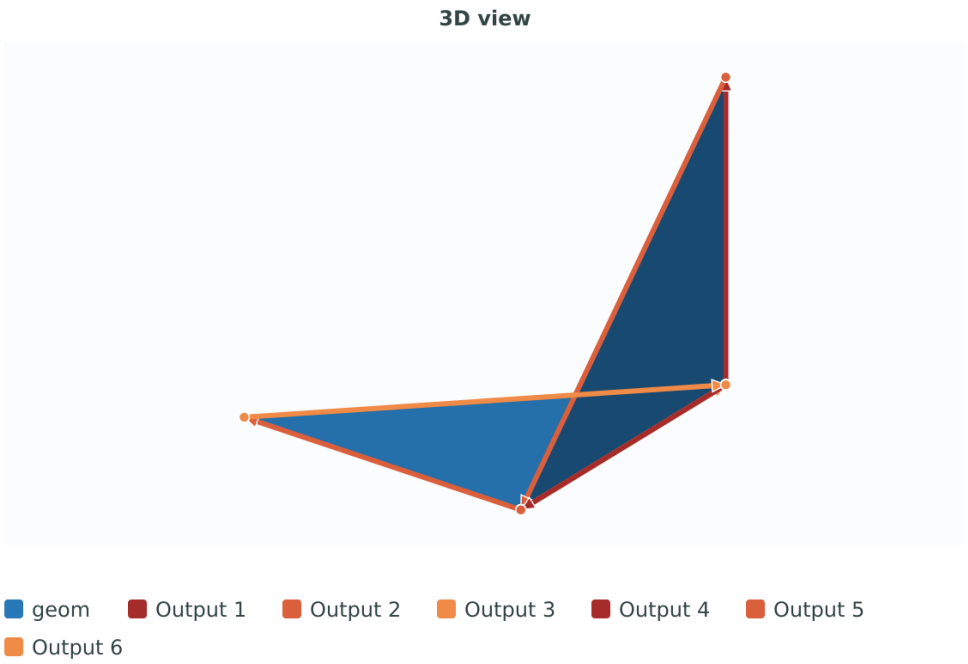


This example uses a TIN geometry.

Code

```
SELECT path,geom
FROM (
    SELECT (ST_DumpSegments(g.geom)).*
    FROM (SELECT 'TIN(((
        0 0 0,
        0 0 1,
        0 1 0,
        0 0 0
    )),((
        0 0 0,
        0 1 0,
        1 1 0,
        0 0 0
    ))
    )'::geometry AS geom
        ) AS g
) j;
```

Figure



□□

geometry_dump, ST_Collect, ST_Dump, ST_NumInteriorRing,

7.4.9 ST_DumpRings

ST_DumpRings — Returns a set of `geometry_dump` rows for the exterior and interior rings of a Polygon.

□□

geometry_dump[] **ST_DumpRings**(geometry a_polygon);

□□

A set-returning function (SRF) that extracts the rings of a polygon. It returns a set of geometry_dump rows, each containing a geometry (*geom* field) and an array of integers (*path* field).

The *geom* field contains each ring as a POLYGON. The *path* field is an integer array of length 1 containing the polygon ring index. The exterior ring (shell) has index 0. The interior rings (holes) have indices of 1 and higher.



□□
This only works for POLYGON geometries, not MULTIPOLYGONs. Use ST_Dump to extract polygon elements from polygonal geometries: `ST_DumpRings((ST_Dump(geom)).geom )`

Availability: PostGIS 1.1.3. Requires PostgreSQL 7.3 or higher.

 This function supports 3d and will not drop the z-index.

□□

Extracting all rings as polygons.

Code

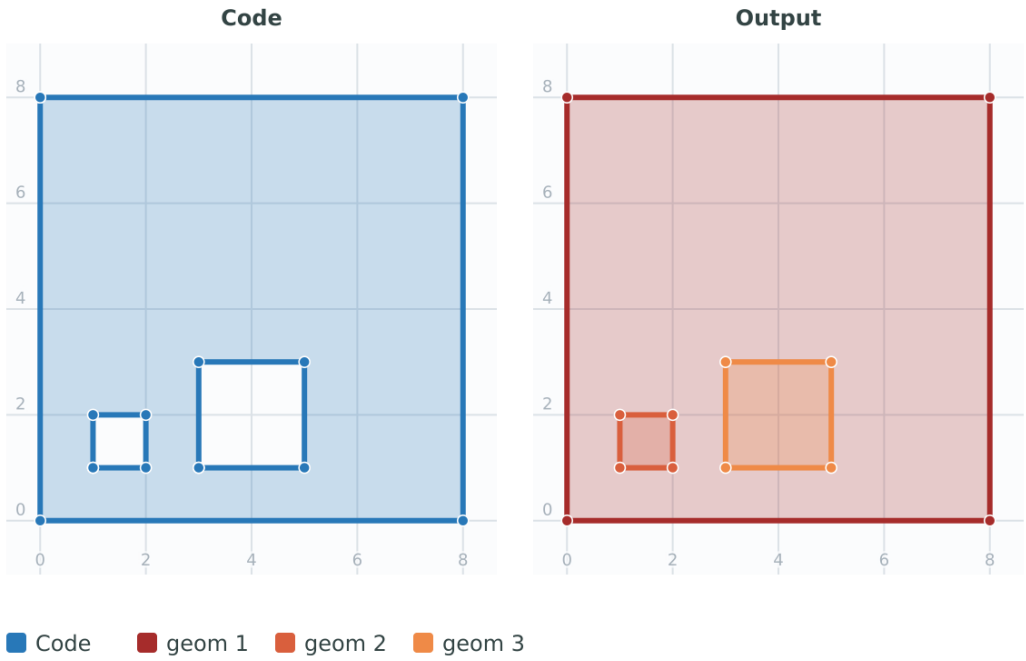
```
SELECT polyTable.field1, polyTable.field1,
      (ST_DumpRings(polyTable.geom)).geom As geom
FROM polyTable
```

Extracting shell and holes from a polygon.

Code

```
SELECT path,geom As geom
FROM ST_DumpRings('POLYGON ((1 9,9 9,9 1,1 1,1 9),(2 2,2 3,3 3,3 2,2 2),(4 2,4 4,6 4,6 2,4 ↩
    2))');
```

Figure



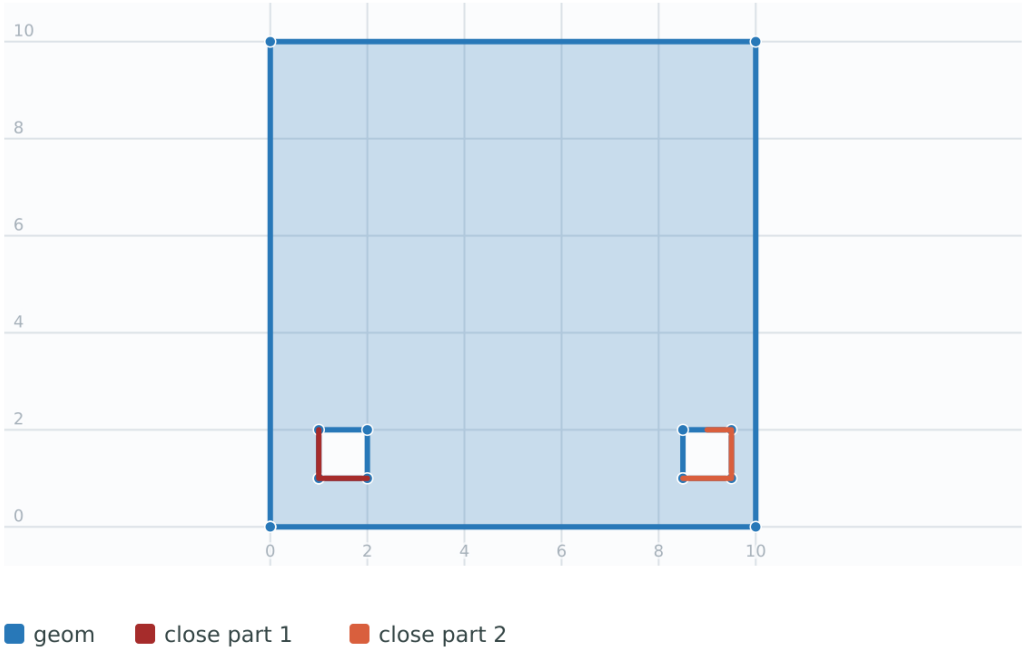
Finding interior rings which lie close to the exterior ring.

Code

```
WITH poly AS (
  SELECT 'POLYGON ((
      0 0, 10 0, 10 10, 0 10, 0 0
    ), (
      1 1, 1 2, 2 2, 2 1, 1 1
    ), (
      8.5 1, 8.5 2, 9.5 2, 9.5 1, 8.5 1
    ))'::geometry AS geom
),
rings AS (
  SELECT d.path[1] AS ring_no,
         ST_Boundary(d.geom) AS ring_geom
  FROM poly
  CROSS JOIN LATERAL ST_DumpRings(poly.geom) AS d
```

```
),
shell AS (
  SELECT ring_geom
  FROM rings
  WHERE ring_no = 0
),
holes AS (
  SELECT ring_no, ring_geom
  FROM rings
  WHERE ring_no > 0
)
SELECT h.ring_no,
       ST_Normalize(ST_Multi(ST_LineMerge(
         ST_Intersection(ST_Buffer(s.ring_geom, 1.0), h.ring_geom)
       ))) AS close_part
FROM shell AS s
JOIN holes AS h
  ON ST_DWithin(s.ring_geom, h.ring_geom, 1.0);
```

Figure



☒☒

geometry_dump, 13.6☒, ST_Boundary, ST_Buffer, ST_Dump, ST_DWithin, ST_ExteriorRing, ST_InteriorRing
ST_Intersection

7.4.10 ST_EndPoint

ST_EndPoint — Returns the last point of a LineString, CircularLineString, or NURBSCurve.

☒☒

geometry **ST_EndPoint**(geometry g);

⊠⊠

Returns the last point of a `LINESTRING`, `CIRCULARLINESTRING`, or `NURBSCURVE` geometry as a `POINT`. Returns `NULL` if the input is not a `LINESTRING`, `CIRCULARLINESTRING`, or `NURBSCURVE`.

This method implements the SQL/MM specification. SQL-MM 3: 7.1.4

This function supports 3d and will not drop the z-index.

This method supports Circular Strings and Curves.

⊠⊠⊠⊠: 2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. PostGIS ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. 2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ NULL ⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ 2.0 ⊠⊠⊠⊠ NULL ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

End point of a LineString

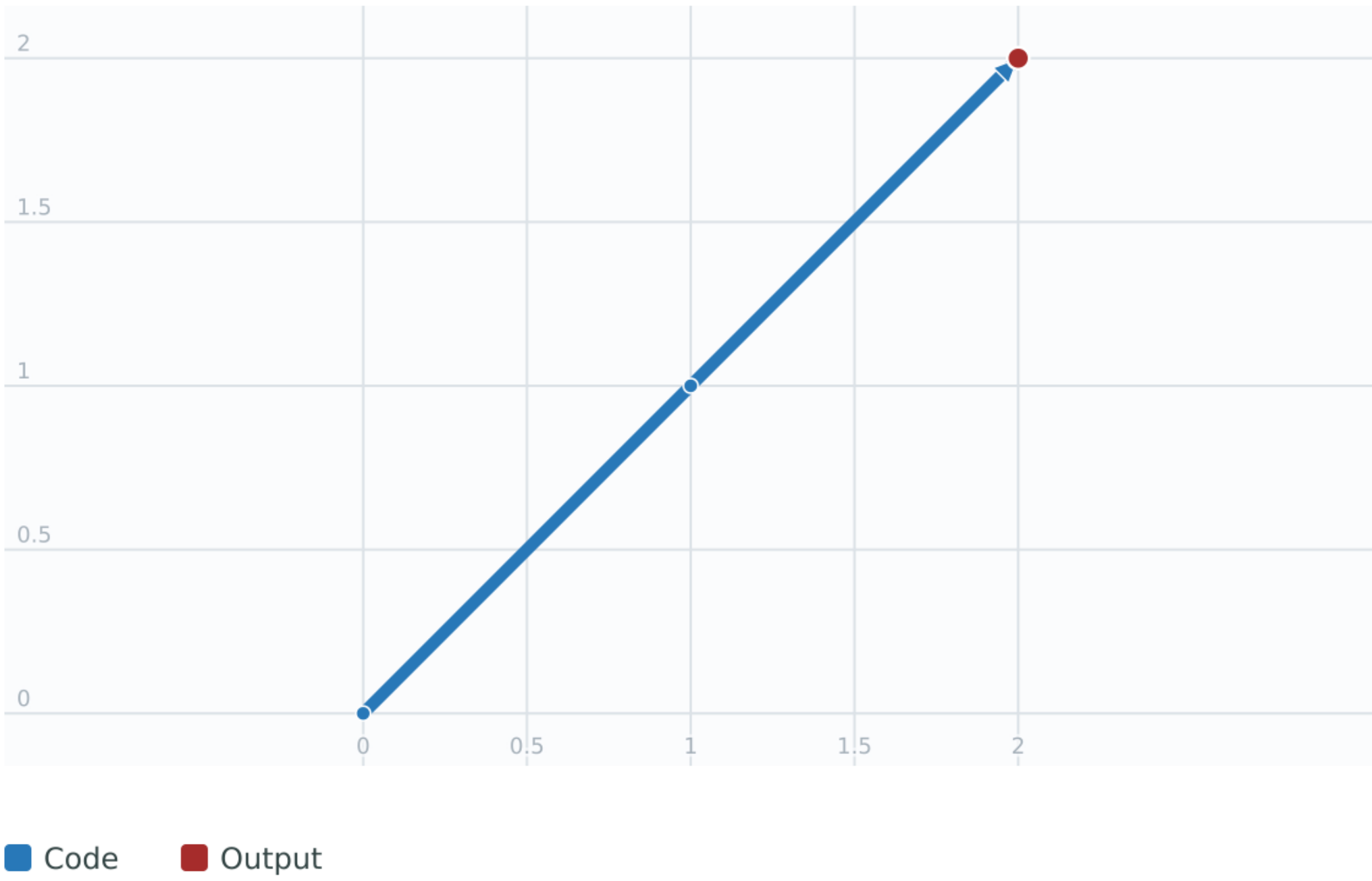
Code

```
SELECT ST_EndPoint('LINESTRING(1 1,2 2,3 3)'::geometry);
```

Output

```
POINT(3 3)
```

Figure



End point of a non-LineString is NULL

Code

```
SELECT ST_EndPoint('POINT(1 1)'::geometry) IS NULL AS is_null;
```

Output

```
t
```

End point of a 3D LineString

3D endpoint.

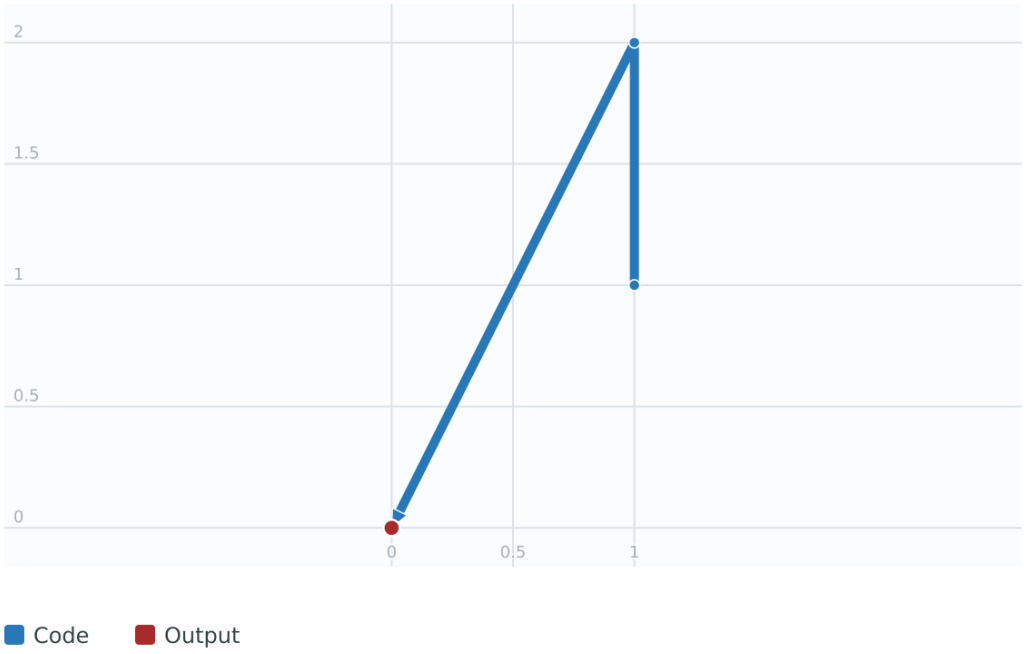
Code

```
SELECT ST_EndPoint('LINESTRING(1 1 2,1 2 3,0 0 5)');
```

Output

```
POINT(0 0 5)
```

Figure



ST_LineString ☒☒ ST_CircularString ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.

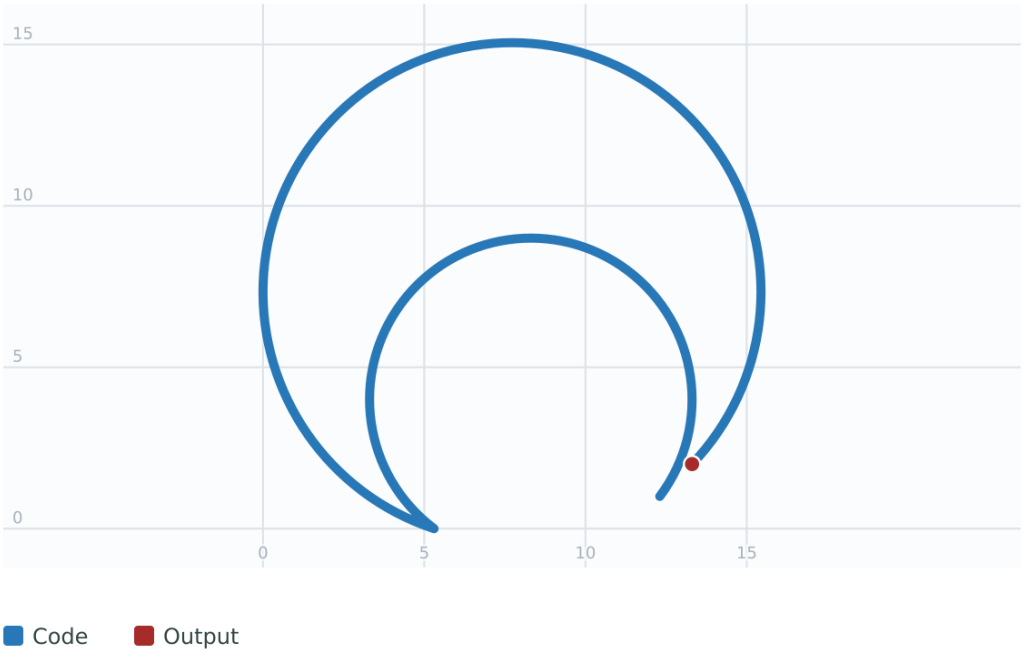
Code

```
SELECT ST_EndPoint('CIRCULARSTRING(5 2,-3 2,-2 1,-4 2,6 3)'::geometry);
```

Output

```
POINT(6 3)
```

Figure



End point of a NURBSCurve

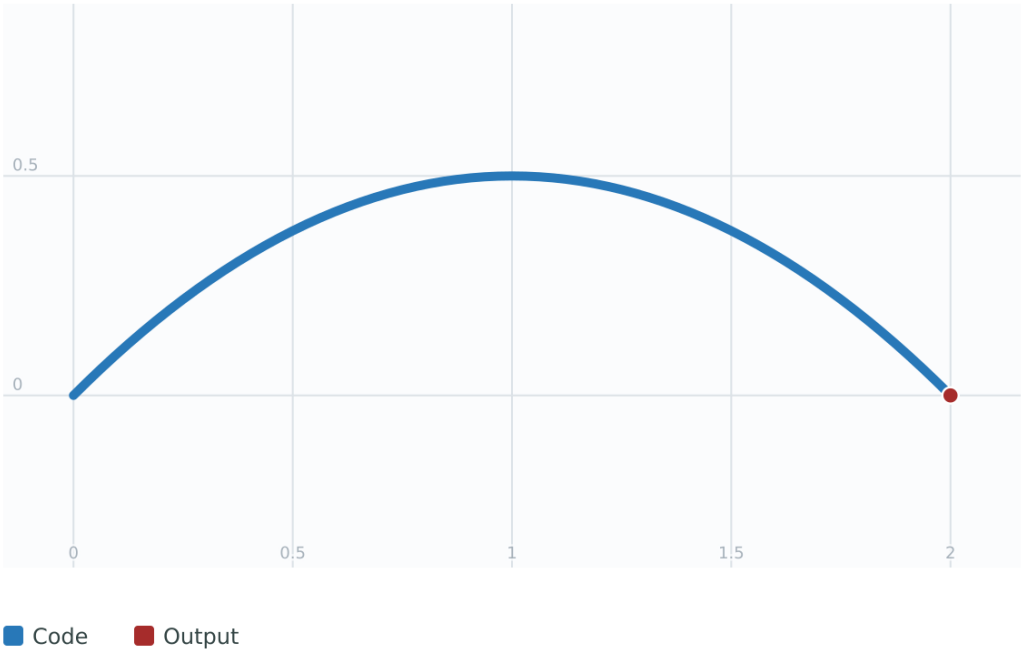
Code

```
SELECT ST_EndPoint('NURBSCURVE(2, (0 0, 1 1, 2 0))'::geometry);
```

Output

```
POINT(2 0)
```

Figure



⊠⊠

ST_PointN, ST_StartPoint

7.4.11 ST_Envelope

ST_Envelope — ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (double precision; float8) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

geometry **ST_Envelope**(geometry g1);

⊠⊠

⊠⊠⊠⊠⊠⊠⊠⊠⊠ float8 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ((`MINX`, `MINY`), (`MINX`, `MAXY`), (`MAXX`, `MAXY`), (`MAXX`, `MINY`), (`MINX`, `MINY`)). (PostGIS ⊠ `ZMIN`/`ZMAX` ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.)

⊠⊠⊠⊠⊠ (⊠⊠⊠⊠, ⊠⊠⊠) ⊠⊠ `POLYGON` ⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠ `POINT` ⊠⊠ `LINESTRING` ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

1.5.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, float4 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

This method implements the OGC Simple Features Implementation Specification for SQL 1.1. s2.1.1.1

This method implements the SQL/MM specification. SQL-MM 3: 5.1.19

⊠⊠

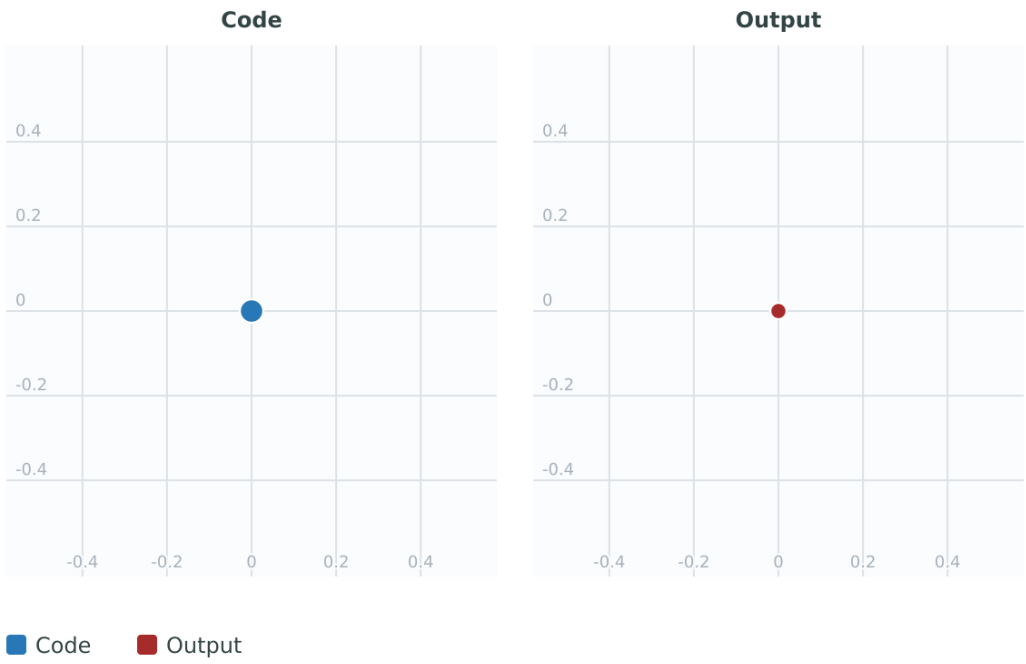
Code

```
SELECT ST_Envelope('POINT(1 3)'::geometry);
```

Output

```
POINT(1 3)
```

Figure



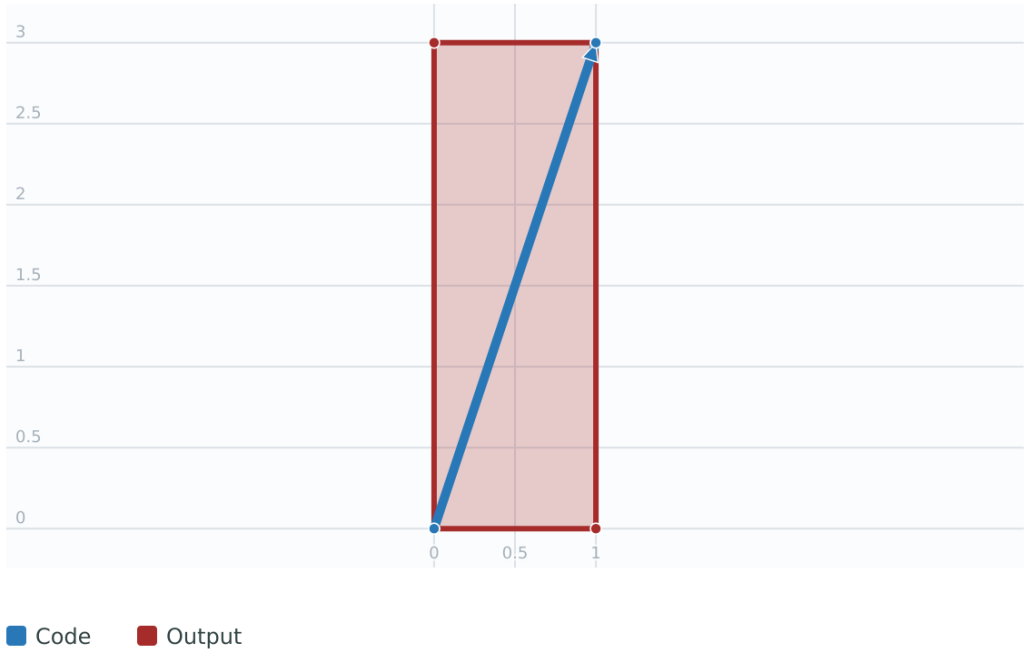
Code

```
SELECT ST_Envelope('LINESTRING(0 0,1 3)'::geometry);
```

Output

```
POLYGON((0 0,0 3,1 3,1 0,0 0))
```

Figure



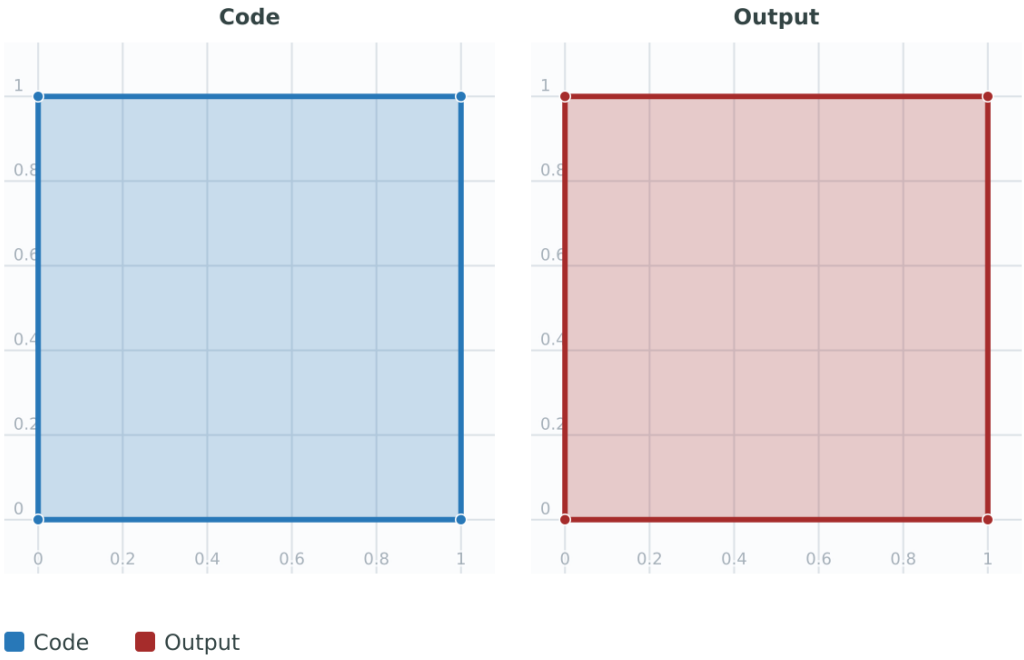
Code

```
SELECT ST_Envelope('POLYGON((0 0,0 1,1.0000001 1,1.0000001 0,0 0))'::geometry);
```

Output

```
POLYGON((0 0,0 1,1.0000001 1,1.0000001 0,0 0))
```

Figure



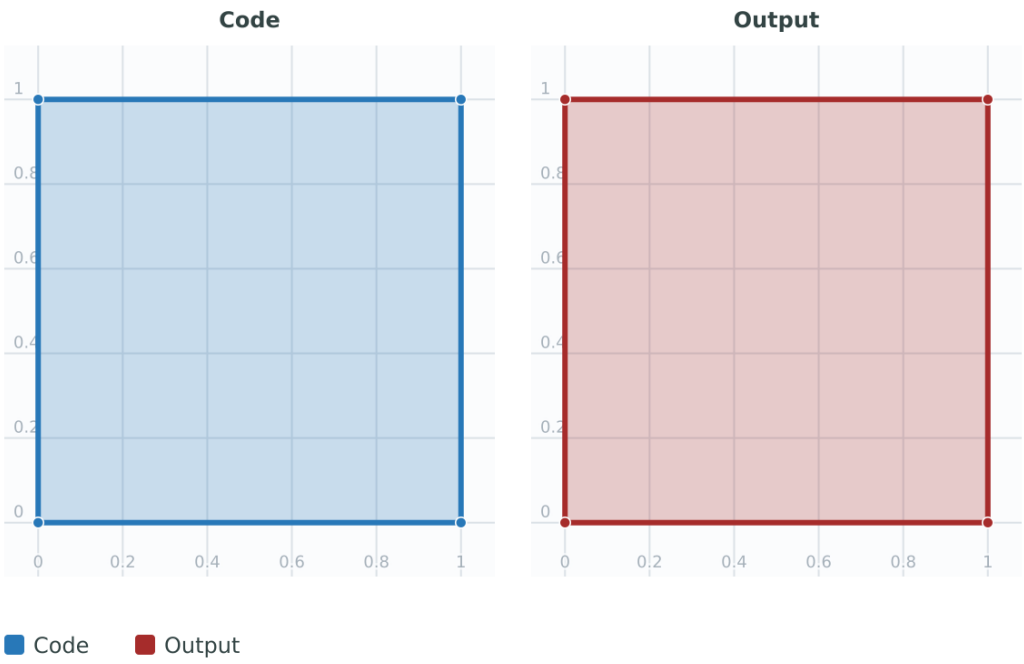
Code

```
SELECT ST_Envelope('POLYGON((0 0,0 1,1.0000000001 1,1.0000000001 0,0 0))'::geometry);
```

Output

```
POLYGON((0 0,0 1,1.0000000001 1,1.0000000001 0,0 0))
```

Figure



Code

```
SELECT Box3D(geom), Box2D(geom), ST_Envelope(geom) As envelopewkt
    FROM (SELECT 'POLYGON((0 0,0 1000012333334.34545678,1.0000001 1,1.0000001 0,0 0))':: ↩
    geometry As geom) As foo;
```

Envelope of a point and LineString.

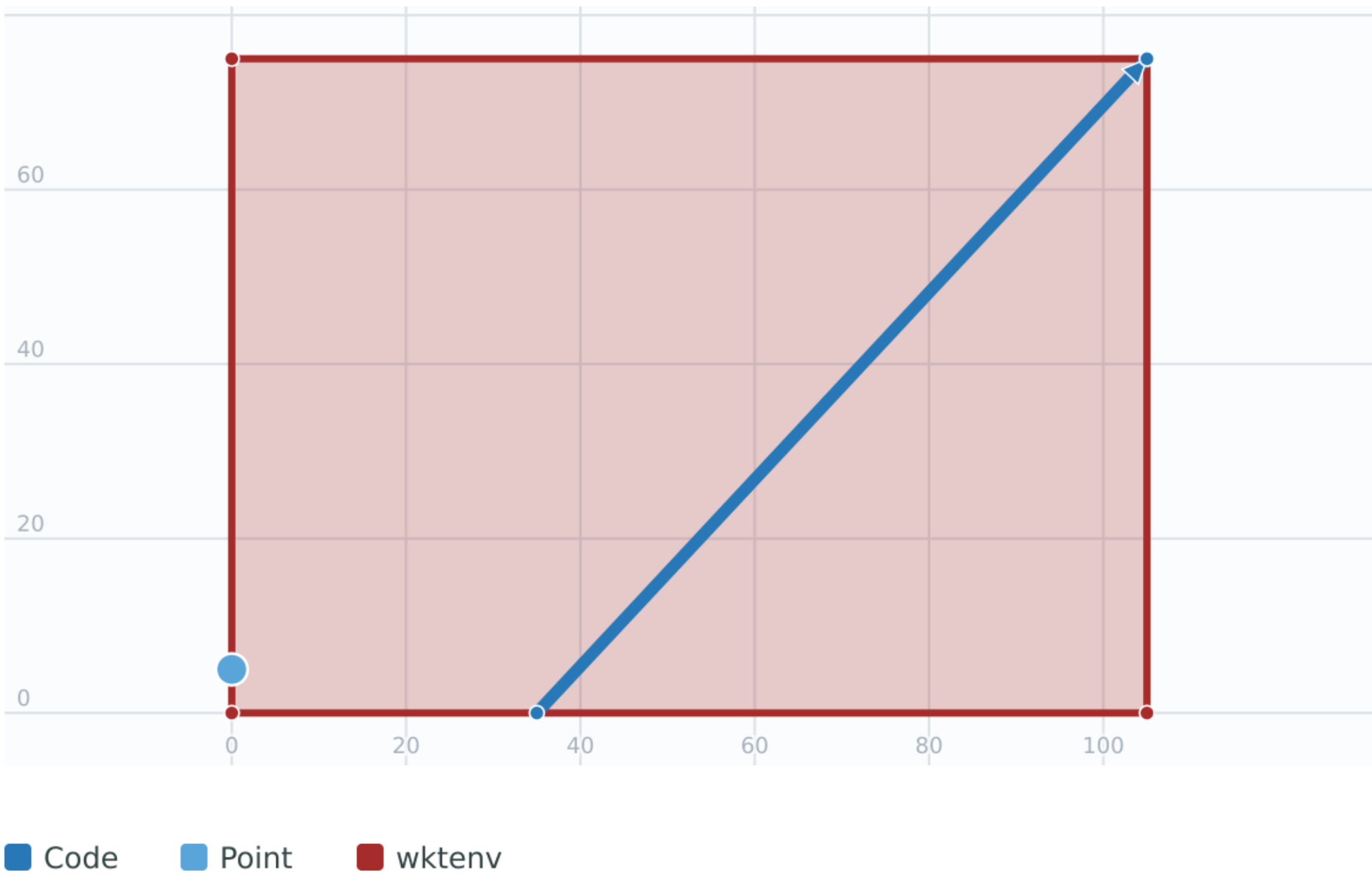
Code

```
SELECT ST_Envelope(ST_Collect(
            'LINESTRING(55 75,125 150)',
            ST_Point(20, 80)
            )) As wktenv;
```

Output

```
POLYGON((20 75,20 150,125 150,125 75,20 75))
```

Figure



⊠⊠

Box2D, Box3D, ST_OrientedEnvelope

7.4.12 ST_ExteriorRing

ST_ExteriorRing — ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

geometry **ST_ExteriorRing**(geometry a_polygon);

⊠⊠

`POLYGON` ⊠⊠⊠⊠⊠⊠⊠ (exterior ring) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ NULL ⊠⊠⊠⊠⊠⊠.

> **Note!**
> ⊠⊠
> ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠ ST_Dump ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

This method implements the OGC Simple Features Implementation Specification for SQL 1.1. 2.1.5.1

This method implements the SQL/MM specification. SQL-MM 3: 8.2.3, 8.3.3

This function supports 3d and will not drop the z-index.

□□

Extract the exterior ring from a Polygon with a hole.

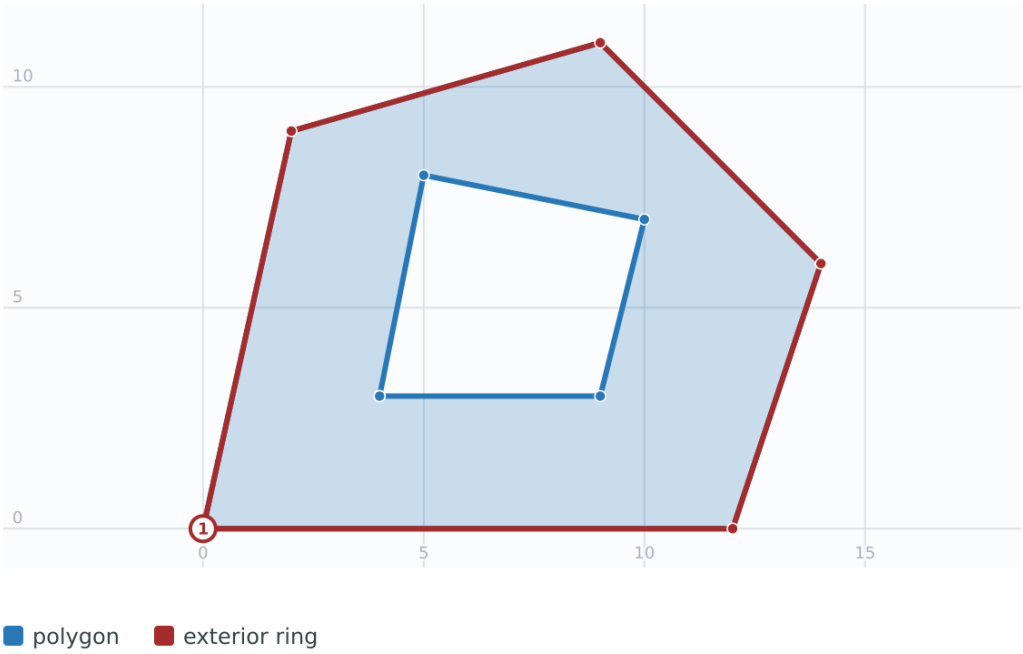
Code

```
WITH input(geom) AS (VALUES (
  'POLYGON((0 0,2 9,9 11,14 6,12 0,0 0),(4 3,9 3,10 7,5 8,4 3))'::geometry
))
SELECT geom AS input_polygon,
       ST_ExteriorRing(geom) AS exterior_ring
FROM input;
```

Output

```
POLYGON((0 0,2 9,9 11,14 6,12 0,0 0),(4 3,9 3,10 7,5 8,4 3)) | LINESTRING(0 0,2 9,9 11,14 6,12 0,0 0)
```

Figure



For a MultiPolygon, dump the component Polygons before extracting their exterior rings. Both component Polygons in this example have holes.

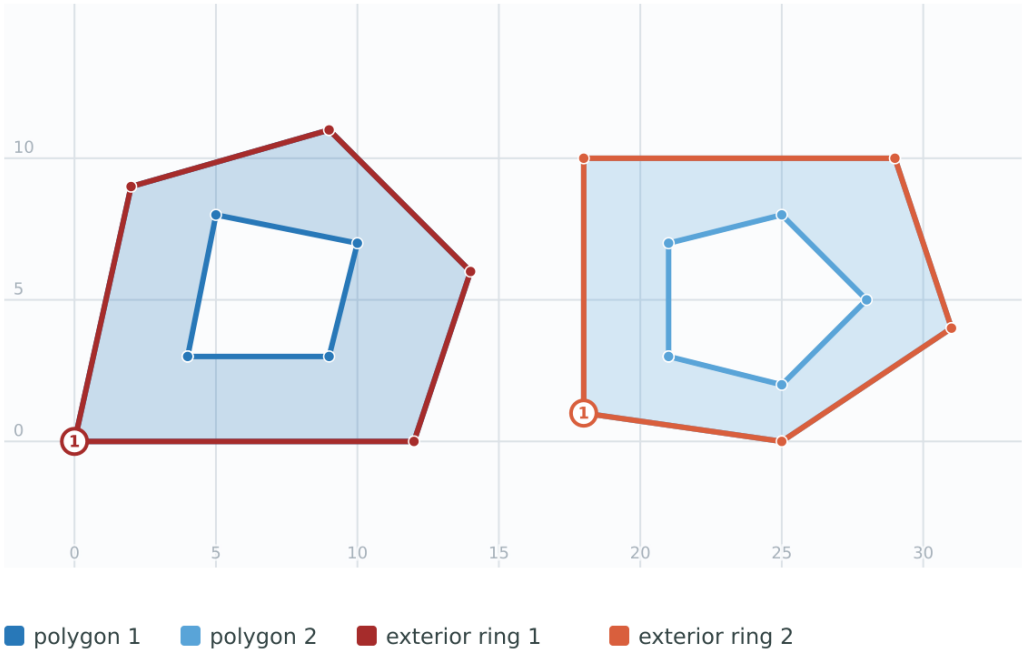
Code

```
WITH input(geom) AS (VALUES (
  'MULTIPOLYGON(
    ((0 0,2 9,9 11,14 6,12 0,0 0),(4 3,9 3,10 7,5 8,4 3)),
    ((18 1,18 10,29 10,31 4,25 0,18 1),(21 3,25 2,28 5,25 8,21 7,21 3))
  )'::geometry
))
SELECT part.geom AS input_polygon,
       ST_ExteriorRing(part.geom) AS exterior_ring
FROM input
CROSS JOIN LATERAL ST_Dump(geom) AS part
ORDER BY part.path;
```

Output

```
POLYGON((0 0,2 9,9 11,14 6,12 0,0 0),(4 3,9 3,10 7,5 8,4 3)) | LINESTRING(0 0,2 9,9 11,14 6,12 0,0 0)
POLYGON((18 1,18 10,29 10,31 4,25 0,18 1),(21 3,25 2,28 5,25 8,21 7,21 3)) | LINESTRING(18 1,18 10,29 10,31 4,25 0,18 ↩
    1)
```

Figure



Extract exterior rings from a table of Polygons.

Code

```
SELECT gid, ST_ExteriorRing(geom) AS ering
FROM sometable;
```

For a table of MultiPolygons, dump the component Polygons and collect their exterior rings into a MultiLineString.

Code

```
SELECT gid, ST_Collect(ST_ExteriorRing(geom)) AS erings
FROM (
  SELECT gid, (ST_Dump(geom)).geom AS geom
  FROM sometable
) AS foo
GROUP BY gid;
```

This example returns the 3D exterior ring of a Polygon.

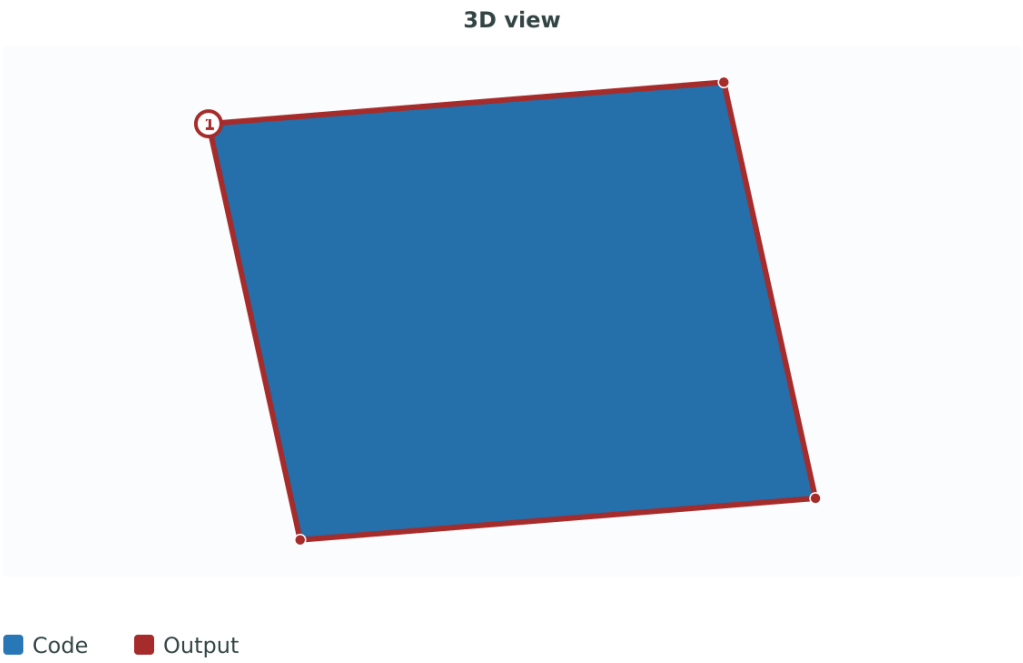
Code

```
SELECT ST_ExteriorRing(
    'POLYGON Z ((0 0 5,4 0 5,4 3 5,0 3 5,0 0 5))'
    );
```

Output

```
LINESTRING(0 0 5,4 0 5,4 3 5,0 3 5,0 0 5)
```

Figure



□□

ST_InteriorRingN, ST_Boundary, ST_NumInteriorRings

7.4.13 ST_GeometryN

ST_GeometryN — ST_Geometry □□□□□□□□□□□□□.

□□

geometry **ST_GeometryN**(geometry geomA, integer n);

□□

Returns the 1-based Nth element of a GEOMETRYCOLLECTION, MULTIPOINT, MULTILINESTRING, MULTICURVE, or MULTIPOLYGON. For a unitary geometry, including a TIN or PolyhedralSurface, an index of 1 returns the input geometry and any other index returns NULL. Use ST_PatchN and ST_NumPatches to access and count the faces of a TIN or PolyhedralSurface.



□□
0.8.0 □□□□□□□□ OGC □□□□□ 1-□□□□□. □□□□□□□ 0-□□□□□□□□□□□□□.



□□
□□□□□□□□□□□□□□□□ ST_Dump □□□□□□□□□□, □□□□□□□□□□□□□□□□□□□.

□□□□: 2.0.0 □□□□□□□□□□, □□□□□ TIN □□□□□□□□□□□□.

□□□□: 2.0.0 □□□□□□□□□□□□□□ NULL □□□□□□□□□. 2.0.0 □□□□□ ST_GeometryN(..,1) □ □□□□□□□□□□□□□□□□□□□□□□□□.

Changed: 3.6.0 TIN and PolyhedralSurface are treated as unitary geometries and are not decomposed into patches; use ST_PatchN to access faces.

This method implements the OGC Simple Features Implementation Specification for SQL 1.1.

This method implements the SQL/MM specification. SQL-MM 3: 9.1.5

This function supports 3d and will not drop the z-index.

This method supports Circular Strings and Curves.

This function supports Polyhedral surfaces.

This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).

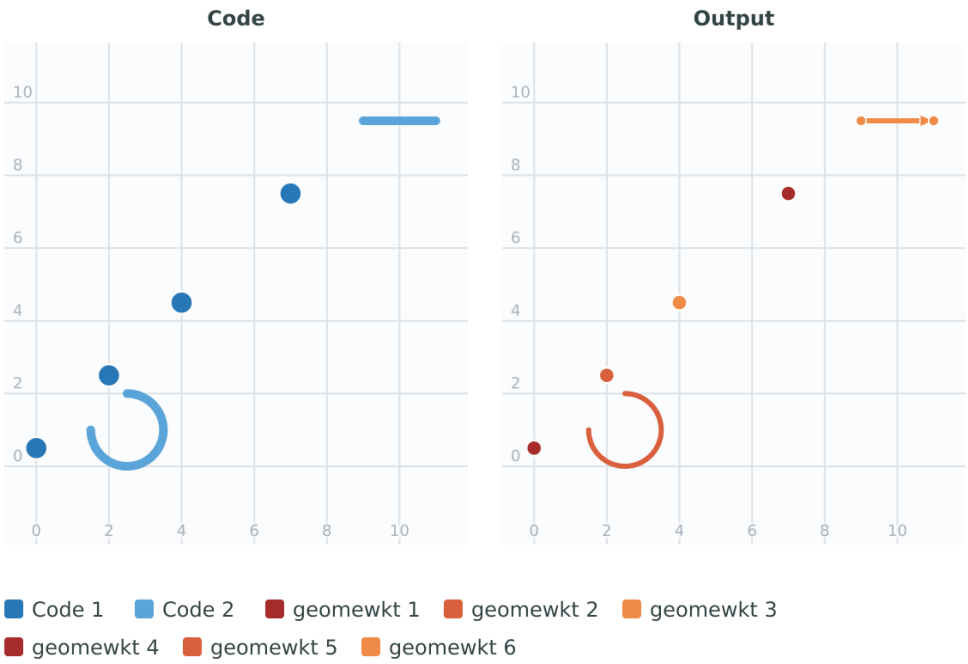
□□□□

Extracting a subset of points from a 3D multipoint.

Code

```
SELECT n, ST_GeometryN(geom, n) As geomewkt
FROM (
VALUES
    (1, 'MULTIPOINT((1 2 7), (3 4 7), (5 6 7), (8 9 10))'::geometry ),
    (2, 'MULTICURVE(CIRCULARSTRING(2.5 2.5,4.5 2.5, 3.5 3.5), (10 11, 12 11))'::geometry )
    ) As foo(id, geom)
    CROSS JOIN generate_series(1,100) n
WHERE n <= ST_NumGeometries(geom)
ORDER BY id, n;
```

Figure



Extracting all geometries (useful when you want to assign an id).

Code

```
SELECT gid, n, ST_GeometryN(geom, n)
FROM sometable CROSS JOIN generate_series(1,100) n
WHERE n <= ST_NumGeometries(geom);
```

☒☒☒☒☒, TIN ☒☒☒☒☒☒

A PolyhedralSurface or TIN is treated as a single geometry, not as a collection of faces. `ST_GeometryN` does not decompose the surface into patches, so an index greater than 1 returns `NULL`. Use ST_PatchN and ST_NumPatches to access individual faces.

This example requests the third element of a PolyhedralSurface.

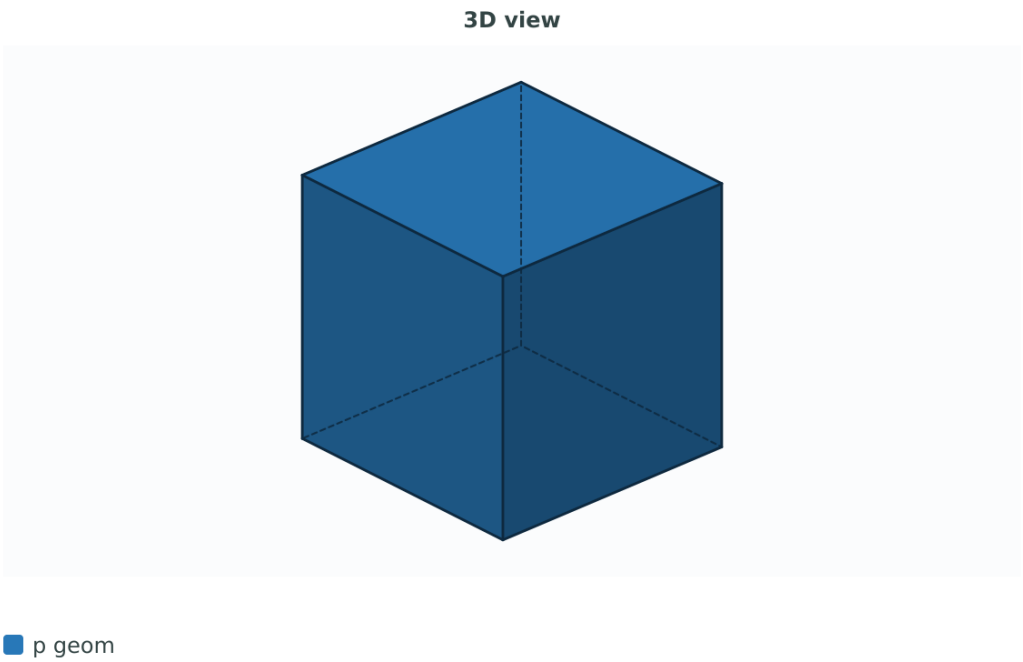
Code

```
SELECT ST_GeometryN(p_geom, 3) As geom_ewkt
  FROM (SELECT 'POLYHEDRALSURFACE(
((0 0 0,0 0 1,0 1 1,0 1 0,0 0 0)),
((0 0 0,0 1 0,1 1 0,1 0 0,0 0 0)),
((0 0 0,1 0 0,1 0 1,0 0 1,0 0 0)),
((1 1 0,1 1 1,1 0 1,1 0 0,1 1 0)),
((0 1 0,0 1 1,1 1 1,1 1 0,0 1 0)),
((0 0 1,1 0 1,1 1 1,0 1 1,0 0 1))
)'::geometry  AS p_geom )  AS a;
```

Output

```
NULL
```

Figure



This example requests the second element of a TIN.

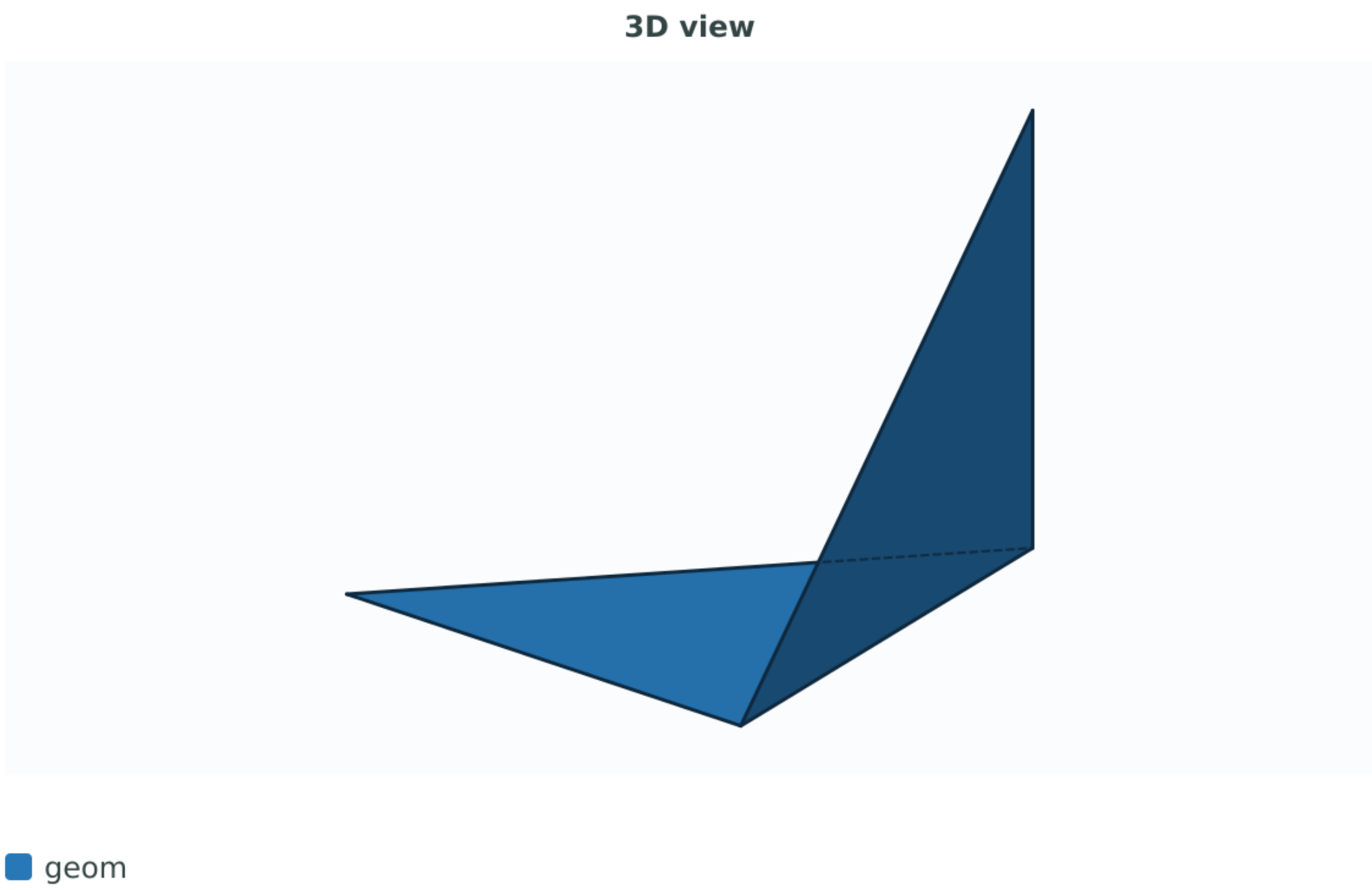
Code

```
SELECT ST_GeometryN(geom, 2) as wkt
  FROM
    (SELECT
       'TIN (((
                0 0 0,
                0 0 1,
                0 1 0,
                0 0 0
            )),((
                0 0 0,
                0 1 0,
                1 1 0,
                0 0 0
            ))
            )'::geometry  AS geom
    ) AS g;
```

Output

```
NULL
```

Figure



⊠⊠

ST_Dump, ST_NumGeometries, ST_PatchN, ST_NumPatches

7.4.14 ST_GeometryType

ST_GeometryType — ST_Geometry ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

text **ST_GeometryType**(geometry g1);

⊠⊠

⊠⊠⊠⊠⊠⊠’ST_LineString’, ’ST_Polygon’, ’ST_MultiPolygon’ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ST ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ GeometryType(geometry) ⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠⊠⊠: 2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠ (polyhedral surface) ⊠⊠⊠⊠⊠⊠.

This method implements the SQL/MM specification. SQL-MM 3: 5.1.4

This function supports 3d and will not drop the z-index.

This function supports Polyhedral surfaces.

⊠⊠

Code

```
SELECT ST_GeometryType('LINESTRING(77.29 29.07,77.42 29.26,77.27 29.31,77.29 29.07)':: ↩
    geometry);
```

Output

```
ST_LineString
```

Code

```
SELECT ST_GeometryType('POLYHEDRALSURFACE( ((0 0 0,0 0 1,0 1 1,0 1 0,0 0 0)),
((0 0 0,0 1 0,1 1 0,1 0 0,0 0 0)),((0 0 0,1 0 0,1 0 1,0 0 1,0 0 0)),
((1 1 0,1 1 1,1 0 1,1 0 0,1 1 0)),
((0 1 0,0 1 1,1 1 1,1 1 0,0 1 0)),((0 0 1,1 0 1,1 1 1,0 1 1,0 0 1)) )'::geometry);
```

Output

```
ST_PolyhedralSurface
```

Code

```
SELECT ST_GeometryType(geom) AS result
FROM
  (SELECT
     'TIN (((
               0 0 0,
               0 0 1,
               0 1 0,
               0 0 0
           )),((
               0 0 0,
               0 1 0,
               1 1 0,
               0 0 0
           ))
           )'::geometry AS geom
  ) AS g;
```

Output

```
ST_Tin
```

□□

GeometryType

7.4.15 ST_HasArc

ST_HasArc — Tests if a geometry contains a circular arc

□□

boolean **ST_HasArc**(geometry geomA);

⊠⊠

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠ TRUE ⊠⊠⊠⊠⊠⊠.

1.2.2 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

✅ This function supports 3d and will not drop the z-index.

✅ This method supports Circular Strings and Curves.

⊠⊠

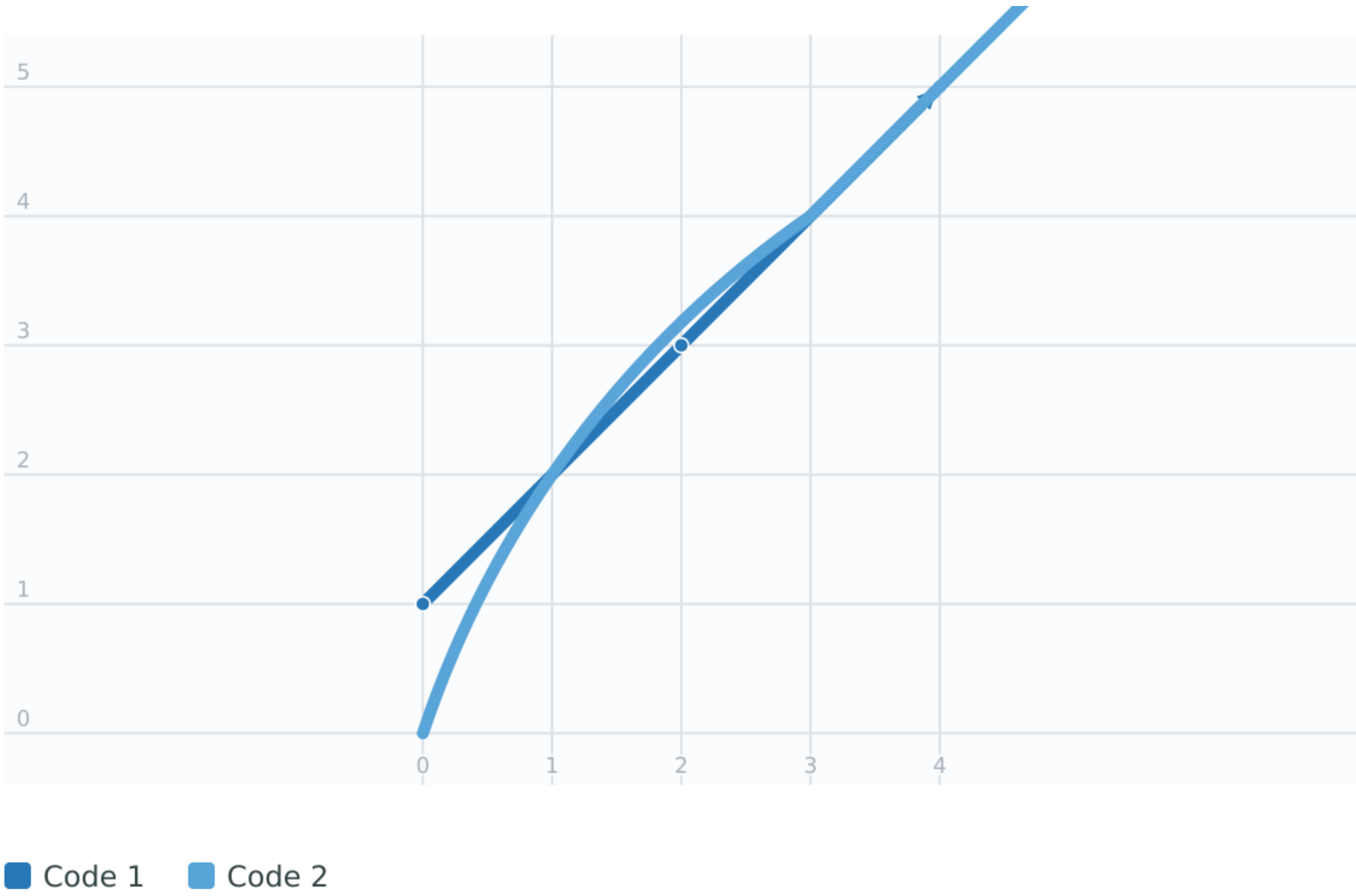
Code

```
SELECT ST_HasArc(ST_Collect('LINESTRING(1 2,3 4,5 6)', 'CIRCULARSTRING(1 1,2 3,4 5,6 7,5 6) ↩
    '));
```

Output

```
t
```

Figure



⊠⊠

ST_CurveToLine, ST_PointN

7.4.16 ST_InteriorRingN

ST_InteriorRingN — ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

□□

geometry **ST_InteriorRingN**(geometry a_polygon, integer n);

□□

□□□□□□ N □□□□□□□□□□□□□□□□□□. □□□□□□□□□□□□□□□ N □□□ (range) □□□ □□□ NULL □□□□□□.



□□
□□□□□□□□□□□□□□□□□□□□. □□□□□□□□ ST_Dump □□□□□□□□□□□□□.

This method implements the OGC Simple Features Implementation Specification for SQL 1.1.

This method implements the SQL/MM specification. SQL-MM 3: 8.2.6, 8.3.5

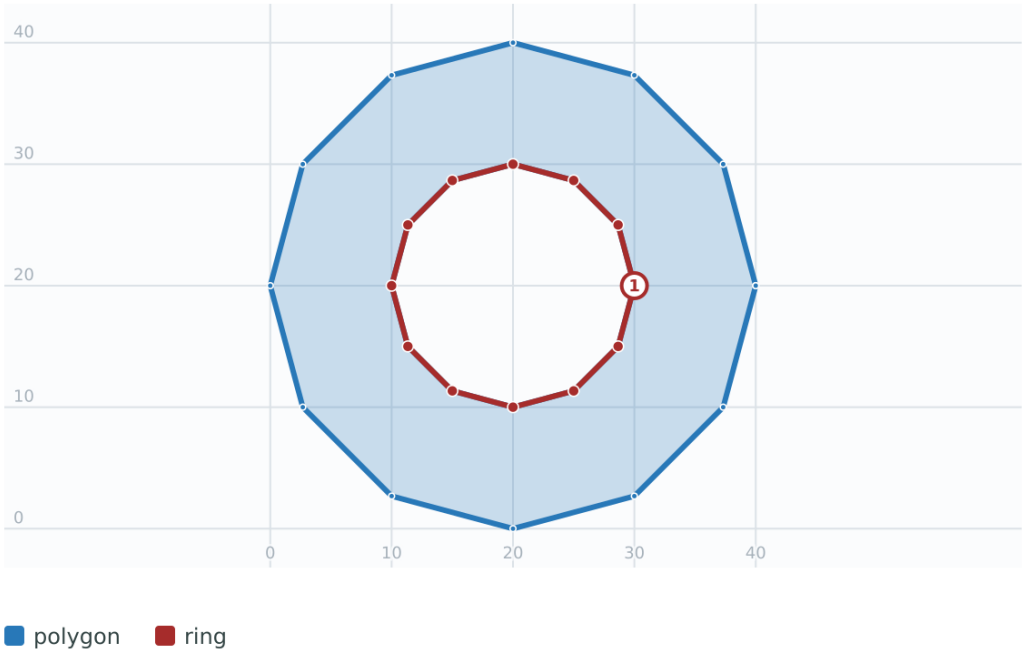
This function supports 3d and will not drop the z-index.

□□

Code

```
WITH data AS (
  SELECT ST_BuildArea(ST_Collect(
    ST_Buffer(ST_Point(1, 2), 20, 3),
    ST_Buffer(ST_Point(1, 2), 10, 3)
  )) AS geom
)
SELECT geom AS input_polygon,
       ST_InteriorRingN(geom, 1) AS ring
FROM data;
```

Figure



☒☒

ST_ExteriorRing, ST_M, ST_X, ST_Y, ST_ZMax, ST_ZMin

7.4.17 ST_NumCurves

ST_NumCurves — Return the number of component curves in a CompoundCurve.

☒☒

integer **ST_NumCurves**(geometry a_compoundcurve);

☒☒

Return the number of component curves in a CompoundCurve, zero for an empty CompoundCurve, or NULL for a non-CompoundCurve input.

This method implements the SQL/MM specification. SQL-MM 3: 8.2.6, 8.3.5

This function supports 3d and will not drop the z-index.

☒☒

This CompoundCurve contains three component curves.

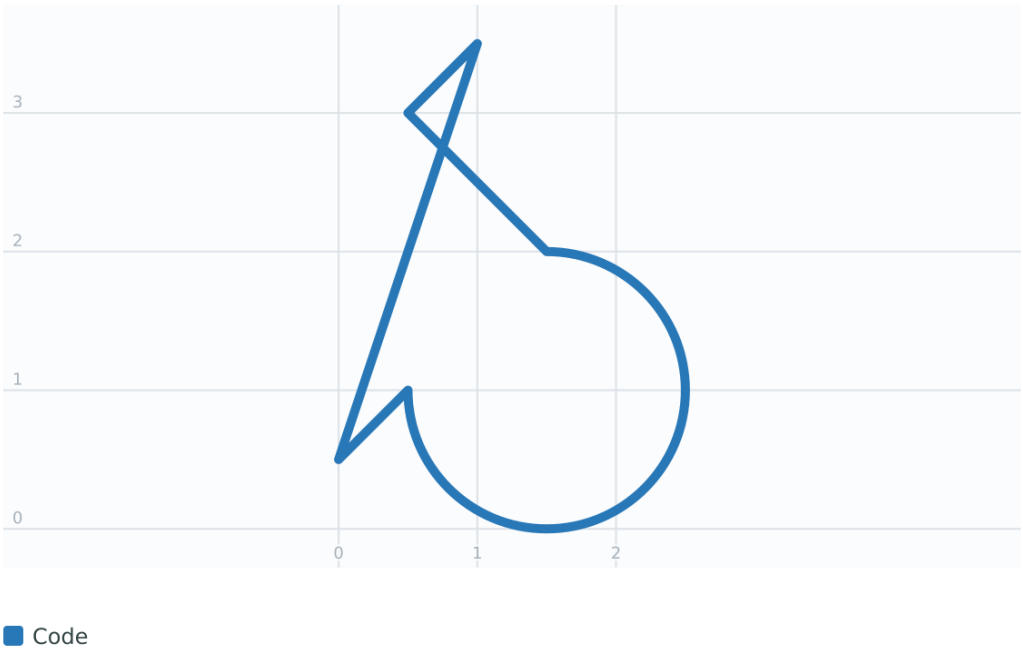
Code

```
SELECT ST_NumCurves('COMPOUNDCURVE(
    (2 2,2.5 2.5),
    CIRCULARSTRING(2.5 2.5,4.5 2.5,3.5 3.5),
    (3.5 3.5,2.5 4.5,3 5,2 2)
  )');
```

Output

```
3
```

Figure



An empty CompoundCurve has no component curves.

Code

```
SELECT ST_NumCurves('COMPOUNDCURVE EMPTY');
```

Output

```
0
```

☒☒

ST_CurveN, ST_Dump, ST_ExteriorRing, ST_NumInteriorRings, ST_NumGeometries

7.4.18 ST_CurveN

ST_CurveN — Returns the Nth component curve geometry of a CompoundCurve.

☒☒

geometry **ST_CurveN**(geometry a_compoundcurve, integer index);

□□

Returns the Nth component curve geometry of a CompoundCurve. The index starts at 1. Returns NULL if the geometry is not a CompoundCurve or the index is out of range.

This method implements the SQL/MM specification. SQL-MM 3: 8.2.6, 8.3.5

This function supports 3d and will not drop the z-index.

□□

This example extracts the second component curve, a CircularString, from a CompoundCurve.

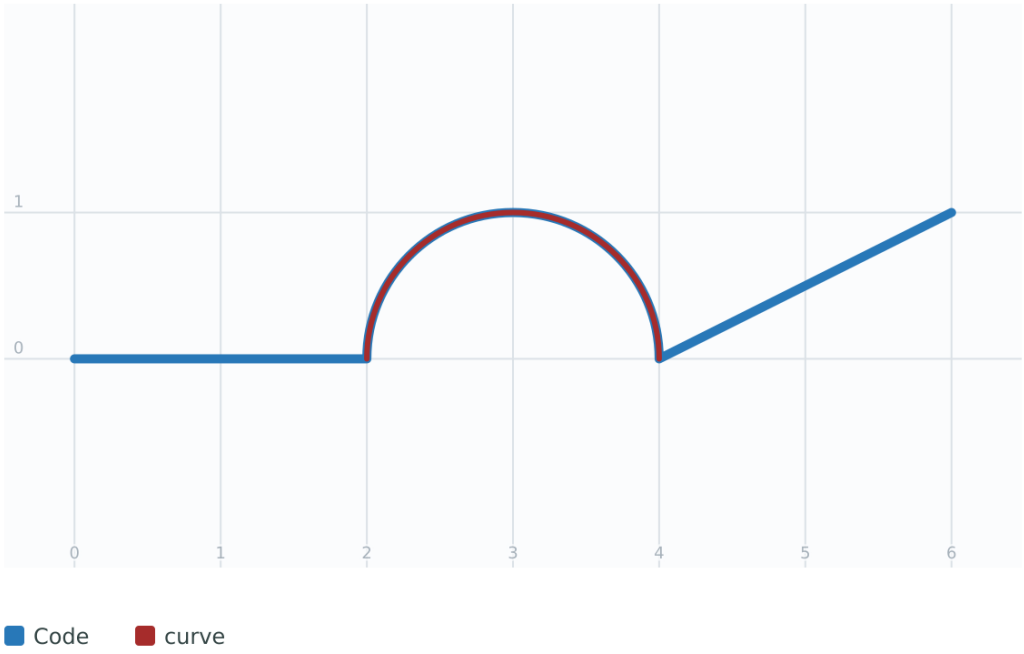
Code

```
SELECT ST_CurveN(
  'COMPOUNDCURVE(
    (0 0,2 0),
    CIRCULARSTRING(2 0,3 1,4 0),
    (4 0,6 1)
  )',
  2) AS curve;
```

Output

```
CIRCULARSTRING(2 0,3 1,4 0)
```

Figure



□□

ST_NumCurves, ST_Dump, ST_ExteriorRing, ST_NumInteriorRings, ST_NumGeometries

7.4.19 ST_IsClosed

ST_IsClosed — LINESTRING ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ TRUE ⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠ (⊠⊠⊠⊠⊠⊠) ⊠⊠⊠⊠ TRUE ⊠⊠⊠⊠⊠⊠.

⊠⊠

boolean **ST_IsClosed**(geometry g);

⊠⊠

LINESTRING ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ TRUE ⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠ (⊠⊠) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (⊠⊠) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

This method implements the OGC Simple Features Implementation Specification for SQL 1.1.

This method implements the SQL/MM specification. SQL-MM 3: 7.1.5, 9.3.3

> **⊠⊠**
> SQL-MM defines the result of `ST_IsClosed(NULL)` to be 0, while PostGIS returns `NULL`.

This function supports 3d and will not drop the z-index.

This method supports Circular Strings and Curves.

⊠⊠⊠⊠: 2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠ (polyhedral surface) ⊠⊠⊠⊠⊠⊠.

This function supports Polyhedral surfaces.

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠

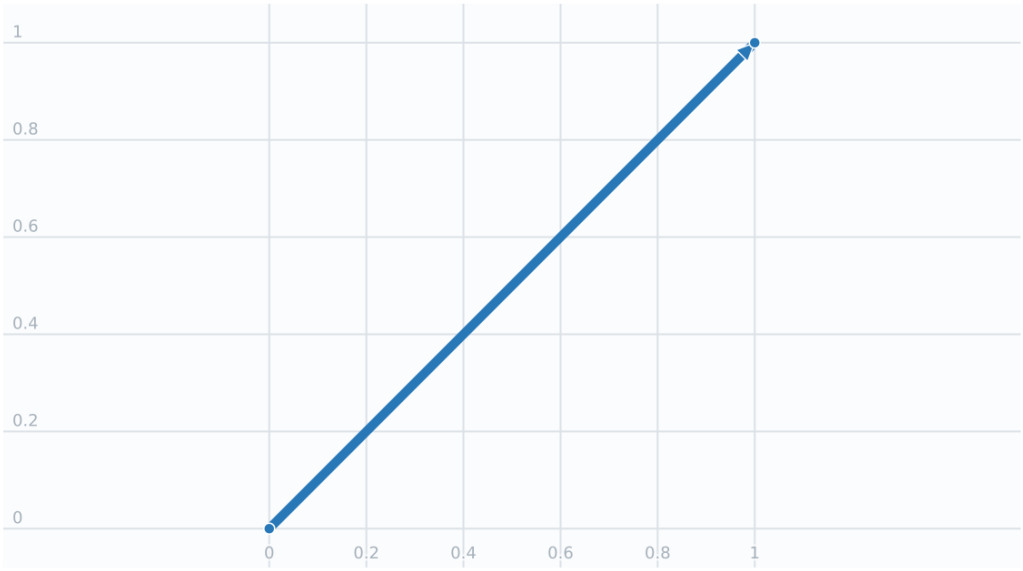
Code

```
SELECT ST_IsClosed('LINESTRING(0 0,1 1)'::geometry);
```

Output

```
f
```

Figure



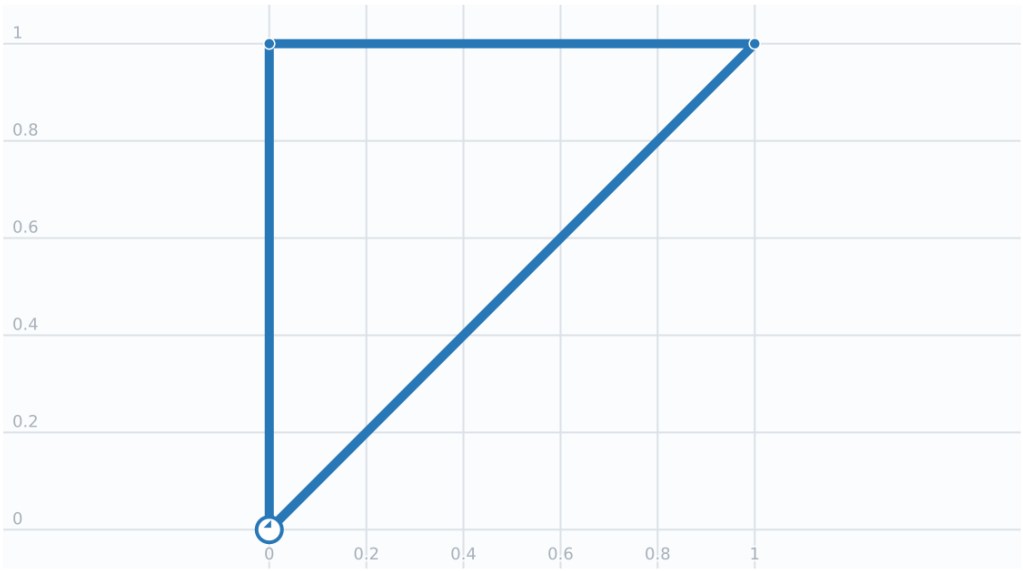
Code

```
SELECT ST_IsClosed('LINESTRING(0 0,0 1,1 1,0 0)'::geometry);
```

Output

```
t
```

Figure



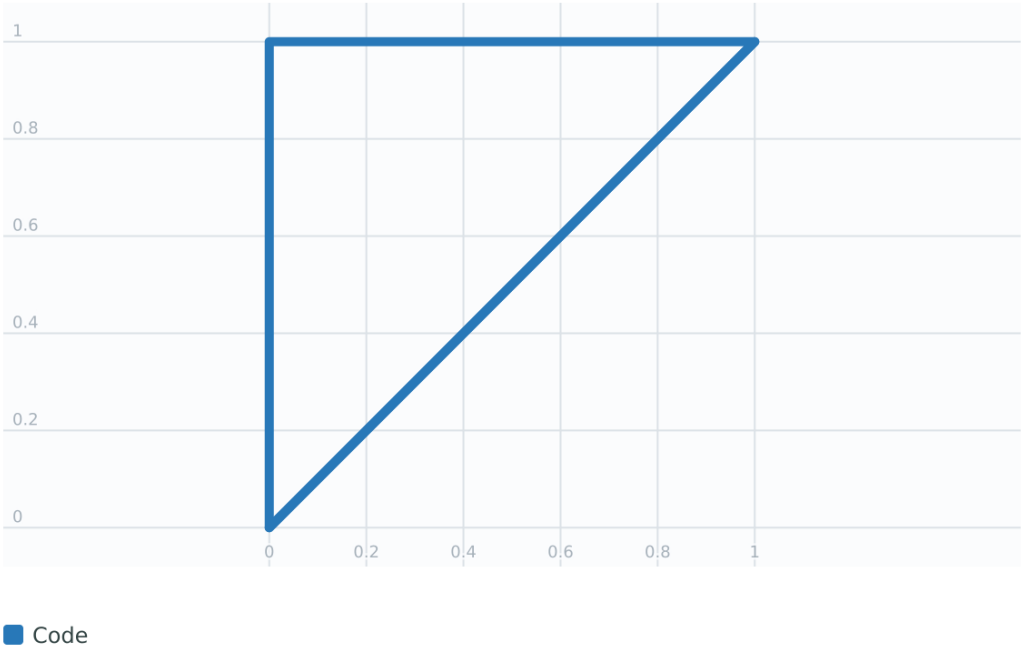
Code

```
SELECT ST_IsClosed('MULTILINESTRING((0 0,0 1,1 1,0 0),(0 0,1 1))'::geometry);
```

Output

```
f
```

Figure



Code

```
SELECT ST_IsClosed('POINT(0 0)'::geometry);
```

Output

```
t
```

Code

```
SELECT ST_IsClosed('MULTIPOINT((0 0),(1 1))'::geometry);
```

Output

```
t
```

□□□□□□□

This example uses a cube.

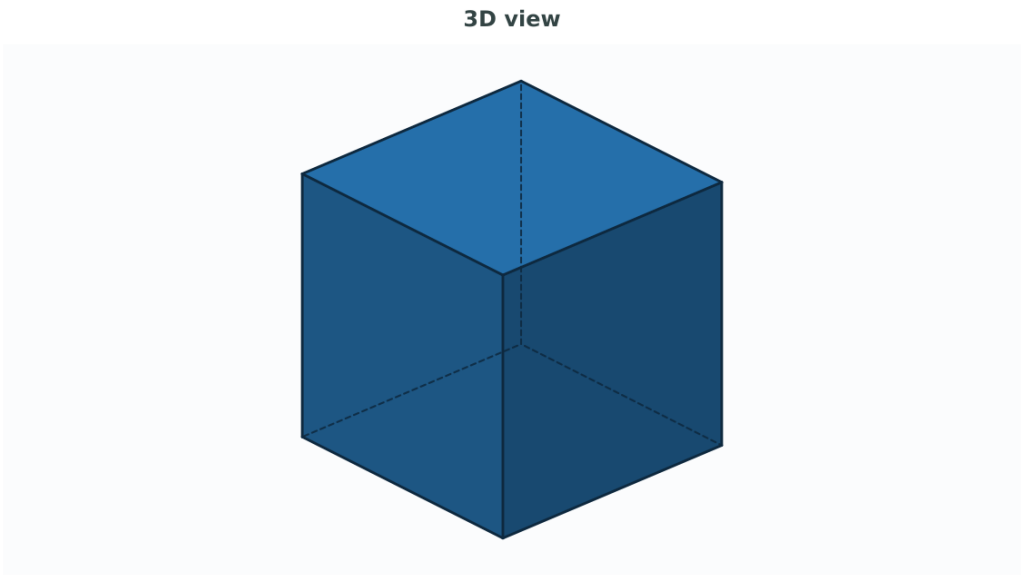
Code

```
SELECT ST_IsClosed('POLYHEDRALSURFACE( ((0 0 0,0 0 1,0 1 1,0 1 0,0 0 0)),
((0 0 0,0 1 0,1 1 0,1 0 0,0 0 0)),((0 0 0,1 0 0,1 0 1,0 0 1,0 0 0)),
((1 1 0,1 1 1,1 0 1,1 0 0,1 1 0)),
((0 1 0,0 1 1,1 1 1,1 1 0,0 1 0)),((0 0 1,1 0 1,1 1 1,0 1 1,0 0 1)) )');
```

Output

```
t
```

Figure



This example uses the same cube with one side missing.

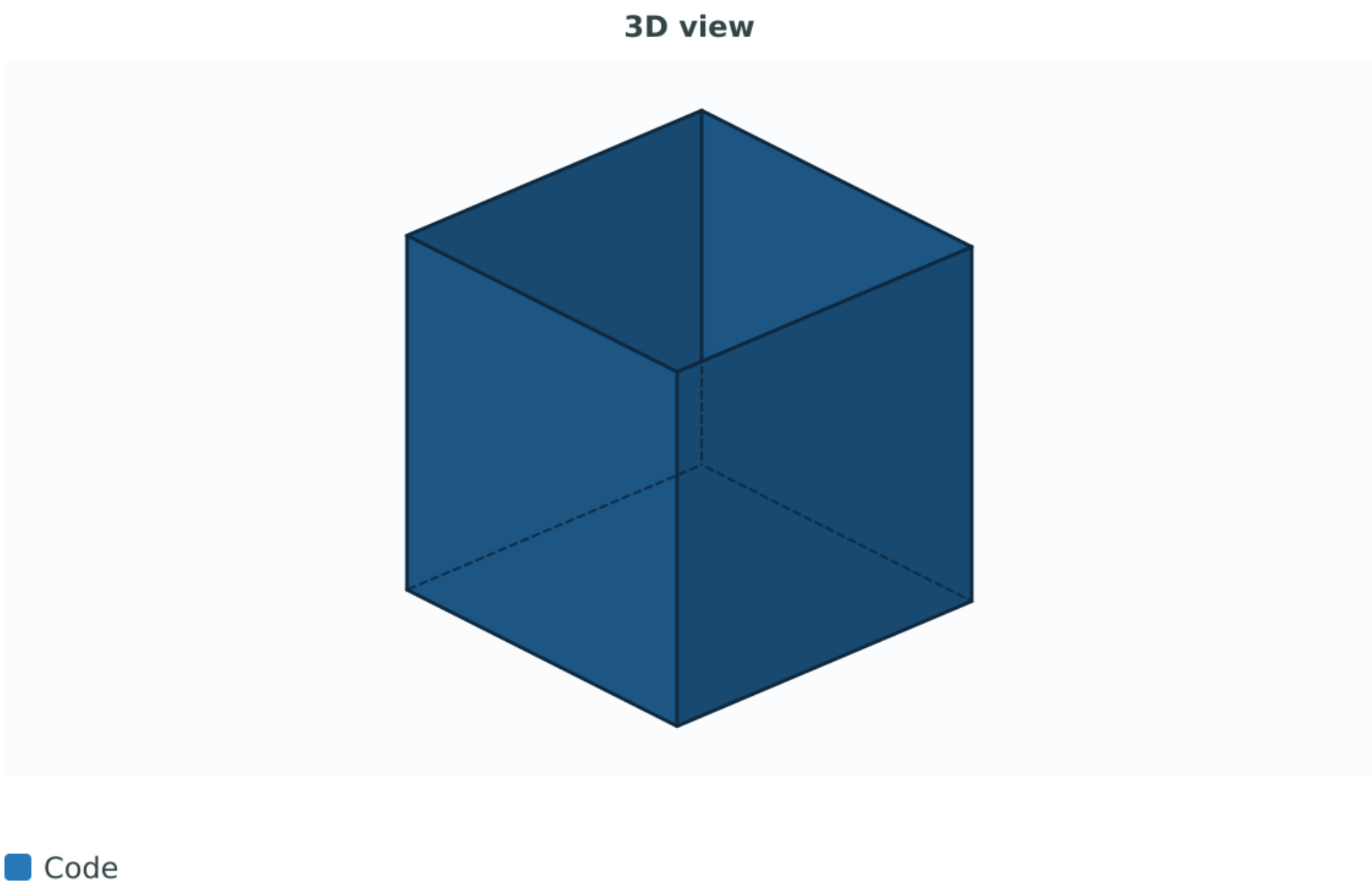
Code

```
SELECT ST_IsClosed('POLYHEDRALSURFACE( ((0 0 0,0 0 1,0 1 1,0 1 0,0 0 0)),
       ((0 0 0,0 1 0,1 1 0,1 0 0,0 0 0)),((0 0 0,1 0 0,1 0 1,0 0 1,0 0 0)),
       ((1 1 0,1 1 1,1 0 1,1 0 0,1 1 0)),
       ((0 1 0,0 1 1,1 1 1,1 1 0,0 1 0)) )');
```

Output

```
f
```

Figure



⊠⊠

ST_IsRing

7.4.20 ST_IsCollection

ST_IsCollection — ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠ TRUE ⊠⊠⊠⊠⊠⊠.

⊠⊠

boolean **ST_IsCollection**(geometry g);

⊠⊠

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ TRUE ⊠⊠⊠⊠⊠⊠:

- GEOMETRYCOLLECTION
- MULTI{POINT,POLYGON,LINESTRING,CURVE,SURFACE}
- COMPOUNDCURVE

> **Note!**
> ⊠⊠
> ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ TRUE ⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠.

This function supports 3d and will not drop the z-index.

This method supports Circular Strings and Curves.

□□

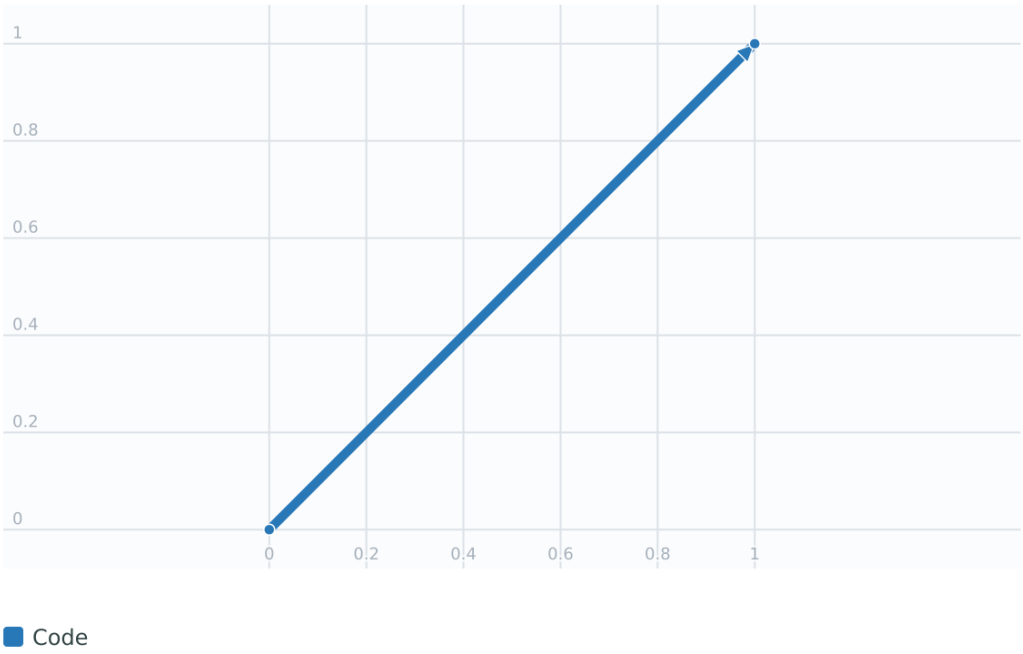
Code

```
SELECT ST_IsCollection('LINESTRING(0 0,1 1)'::geometry);
```

Output

```
f
```

Figure



Code

```
SELECT ST_IsCollection('MULTIPOINT EMPTY'::geometry);
```

Output

```
t
```

Code

```
SELECT ST_IsCollection('MULTIPOINT((0 0))'::geometry);
```

Output

```
t
```

Code

```
SELECT ST_IsCollection('MULTIPOINT((0 0),(42 42))'::geometry);
```

Output

```
t
```

Code

```
SELECT ST_IsCollection('GEOMETRYCOLLECTION(POINT(0 0))'::geometry);
```

Output

```
t
```

⊠⊠

ST_NumGeometries

7.4.21 ST_IsEmpty

ST_IsEmpty — Tests if a geometry is empty.

⊠⊠

boolean **ST_IsEmpty**(geometry geomA);

⊠⊠

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ TRUE ⊠⊠⊠⊠⊠⊠. TRUE ⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠.

> **⊠⊠**
> SQL-MM ⊠ ST_IsEmpty(NULL) ⊠⊠⊠⊠ 0 ⊠⊠⊠⊠⊠⊠⊠, PostGIS ⊠ NULL ⊠⊠⊠⊠⊠⊠.

This method implements the OGC Simple Features Implementation Specification for SQL 1.1. s2.1.1.1

This method implements the SQL/MM specification. SQL-MM 3: 5.1.7

This method supports Circular Strings and Curves.

⊠⊠⊠⊠: PostGIS 2.0.0 ⊠⊠⊠⊠⊠⊠⊠ ST_GeomFromText('GEOMETRYCOLLECTION(EMPTY)') ⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠. PostGIS 2.0.0 ⊠⊠⊠⊠, SQL/MM ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

Code

```
SELECT ST_IsEmpty('GEOMETRYCOLLECTION EMPTY'::geometry);
```

Output

```
t
```

Code

```
SELECT ST_IsEmpty('POLYGON EMPTY'::geometry);
```

Output

```
t
```

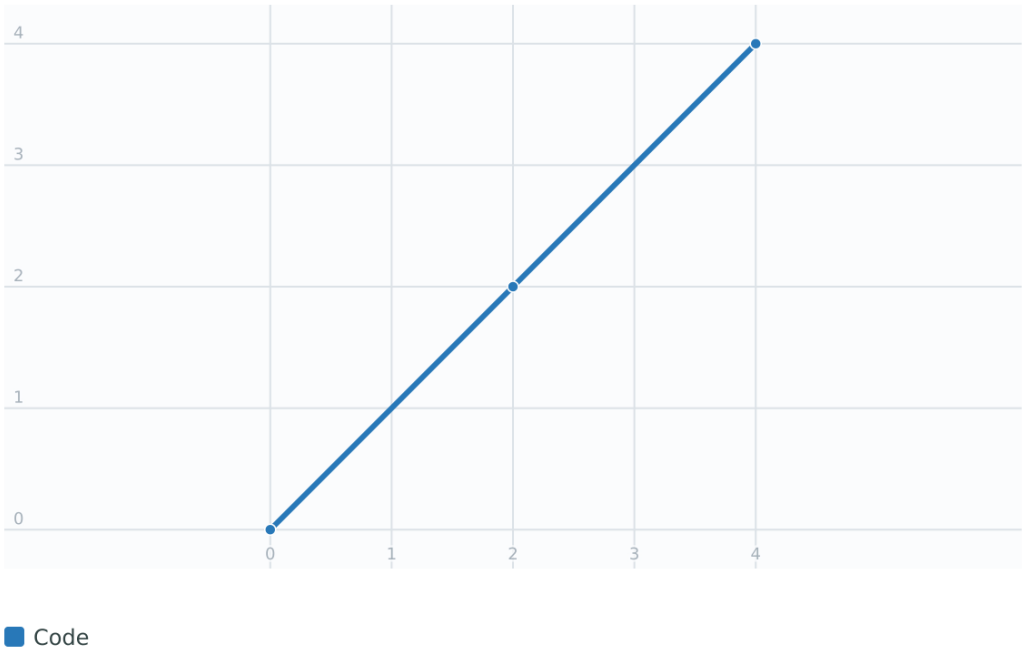
Code

```
SELECT ST_IsEmpty('POLYGON((1 2,3 4,5 6,1 2))'::geometry);
```

Output

```
f
```

Figure



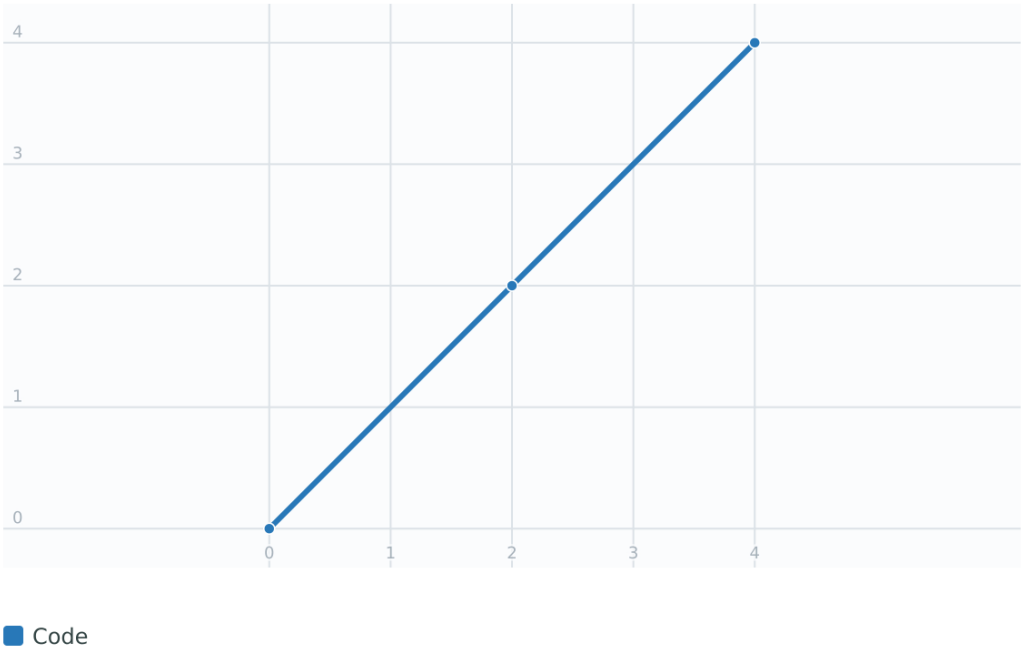
Code

```
SELECT ST_IsEmpty('POLYGON((1 2,3 4,5 6,1 2))'::geometry) = false;
```

Output

```
t
(1 row)
```

Figure



Code

```
SELECT ST_IsEmpty('CIRCULARSTRING EMPTY'::geometry);
```

Output

```
t
```

7.4.22 ST_IsPolygonCCW

ST_IsPolygonCCW — Tests if Polygons have exterior rings oriented counter-clockwise and interior rings oriented clockwise.

☒☒

boolean **ST_IsPolygonCCW**(geometry geom);

☒☒

Returns true if all polygonal components of the input geometry use a counter-clockwise orientation for their exterior ring, and a clockwise direction for all interior rings.

Returns true if the geometry has no polygonal components.



☒☒
Closed linestrings are not considered polygonal components, so you would still get a true return by passing a single closed linestring no matter its orientation.



□□
If a polygonal geometry does not use reversed orientation for interior rings (i.e., if one or more interior rings are oriented in the same direction as an exterior ring) then both ST_IsPolygonCW and ST_IsPolygonCCW will return false.

2.2.0 □□□□□□□□□□□□.



This function supports 3d and will not drop the z-index.



This function supports M coordinates.

□□

ST_ForcePolygonCW, ST_ForcePolygonCCW, ST_IsPolygonCW

7.4.23 ST_IsPolygonCW

ST_IsPolygonCW — Tests if Polygons have exterior rings oriented clockwise and interior rings oriented counter-clockwise.

□□

boolean **ST_IsPolygonCW**(geometry geom);

□□

Returns true if all polygonal components of the input geometry use a clockwise orientation for their exterior ring, and a counter-clockwise direction for all interior rings.

Returns true if the geometry has no polygonal components.



□□
Closed linestrings are not considered polygonal components, so you would still get a true return by passing a single closed linestring no matter its orientation.



□□
If a polygonal geometry does not use reversed orientation for interior rings (i.e., if one or more interior rings are oriented in the same direction as an exterior ring) then both ST_IsPolygonCW and ST_IsPolygonCCW will return false.

2.2.0 □□□□□□□□□□□□.



This function supports 3d and will not drop the z-index.



This function supports M coordinates.

□□

ST_ForcePolygonCW, ST_ForcePolygonCCW, ST_IsPolygonCW

7.4.24 ST_IsRing

ST_IsRing — Tests if a LineString is closed and simple.

□□

boolean **ST_IsRing**(geometry g);

□□

Returns TRUE if this LINESTRING is both ST_IsClosed (ST_StartPoint(g) ~= ST_EndPoint(g)) and ST_IsSimple (does not self intersect).

This method implements the OGC Simple Features Implementation Specification for SQL 1.1. 2.1.5.1

This method implements the SQL/MM specification. SQL-MM 3: 7.1.6



□□
SQL-MM defines the result of ST_IsRing(NULL) to be 0, while PostGIS returns NULL.

□□

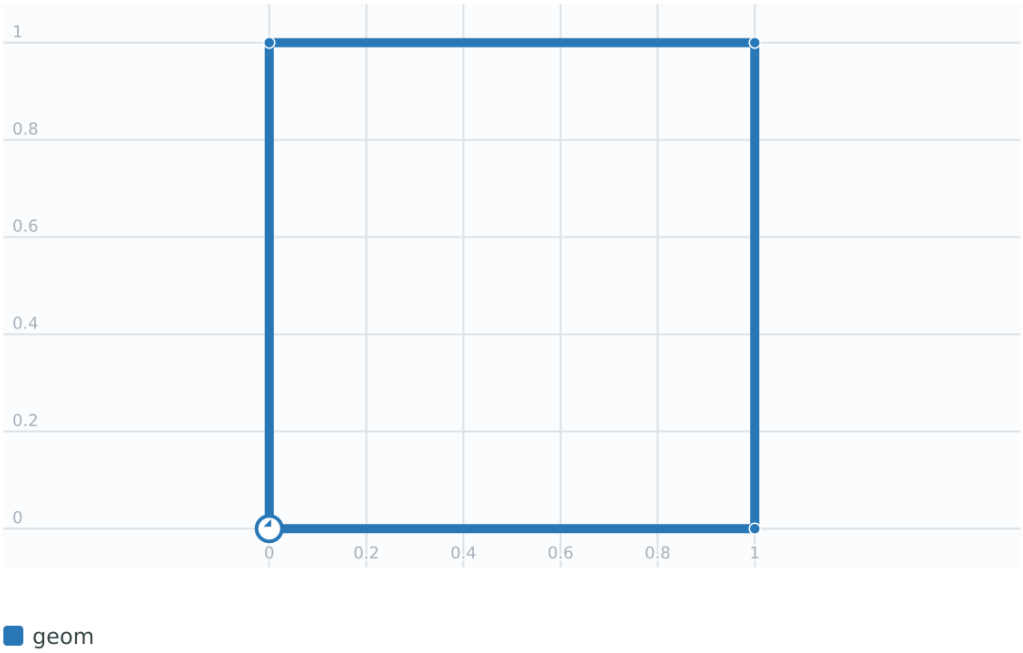
Code

```
SELECT ST_IsRing(geom), ST_IsClosed(geom), ST_IsSimple(geom)
FROM (SELECT 'LINESTRING(0 0,0 1,1 1,1 0,0 0)'::geometry AS geom) AS foo;
```

Output

```
st_isring | st_isclosed | st_issimple
-----------+-------------+-------------
 t         | t           | t
(1 row)
```

Figure



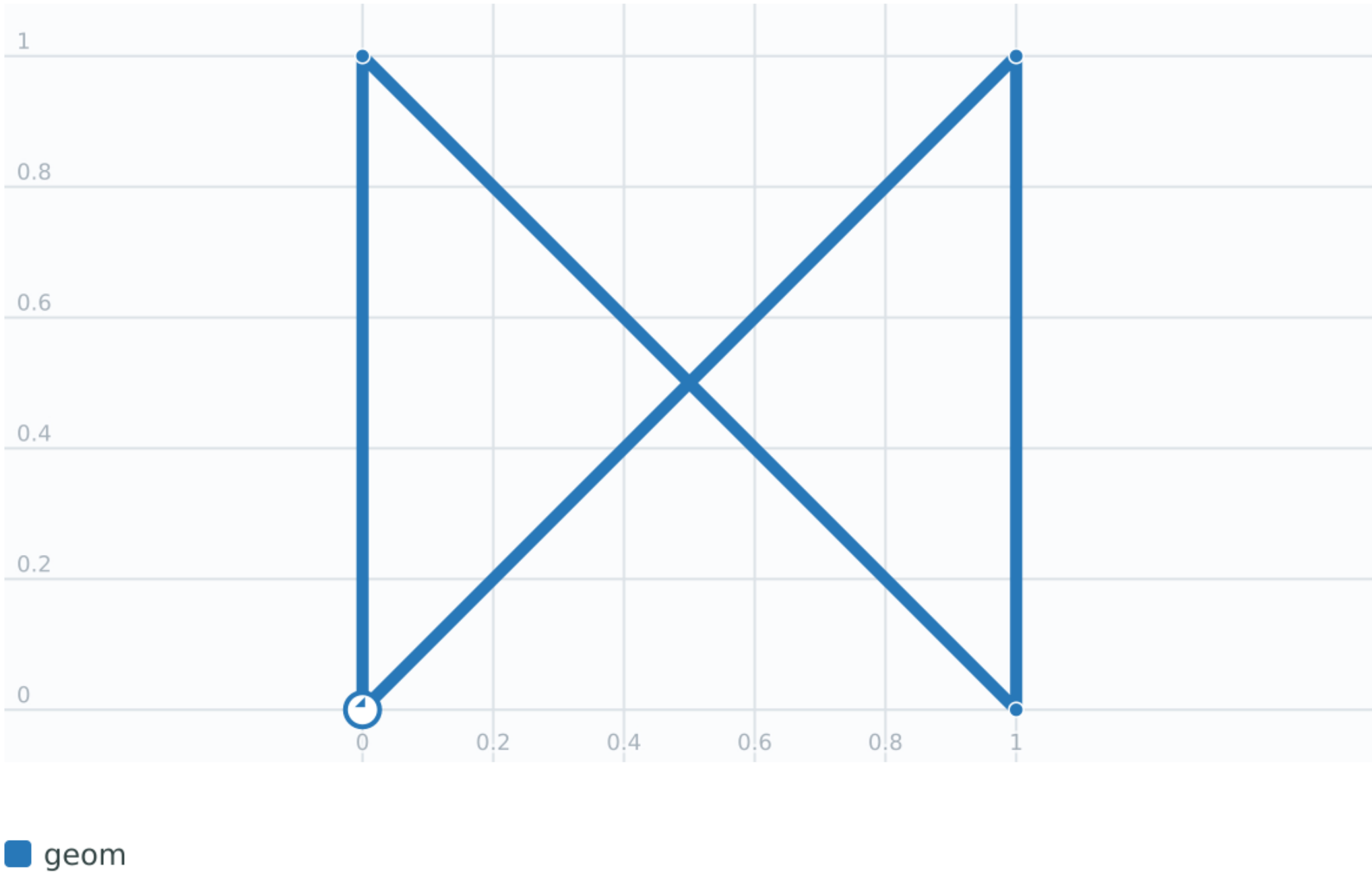
Code

```
SELECT ST_IsRing(geom), ST_IsClosed(geom), ST_IsSimple(geom)
FROM (SELECT 'LINESTRING(0 0,0 1,1 0,1 1,0 0)'::geometry AS geom) AS foo;
```

Output

```
st_isring | st_isclosed | st_issimple
-----------+-------------+-------------
 f         | t           | f
(1 row)
```

Figure



⊠⊠

ST_IsClosed, ST_IsSimple, ST_StartPoint, ST_EndPoint

7.4.25 ST_IsSimple

ST_IsSimple — ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ TRUE ⊠⊠⊠⊠⊠⊠.

⊠⊠

boolean **ST_IsSimple**(geometry geomA);

⊠⊠

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ TRUE ⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ OGC ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, "OpenGIS ⊠⊠⊠⊠⊠⊠⊠⊠ (Ensuring OpenGIS compliance of geometries)" ⊠⊠⊠⊠⊠⊠⊠.

> **Note!**
> ⊠⊠
> SQL-MM ⊠ ST_IsSimple(NULL) ⊠⊠⊠⊠ 0 ⊠⊠⊠⊠⊠⊠⊠, PostGIS ⊠ NULL ⊠⊠⊠⊠⊠⊠.

This method implements the OGC Simple Features Implementation Specification for SQL 1.1. s2.1.1.1

This method implements the SQL/MM specification. SQL-MM 3: 5.1.8

This function supports 3d and will not drop the z-index.

□□

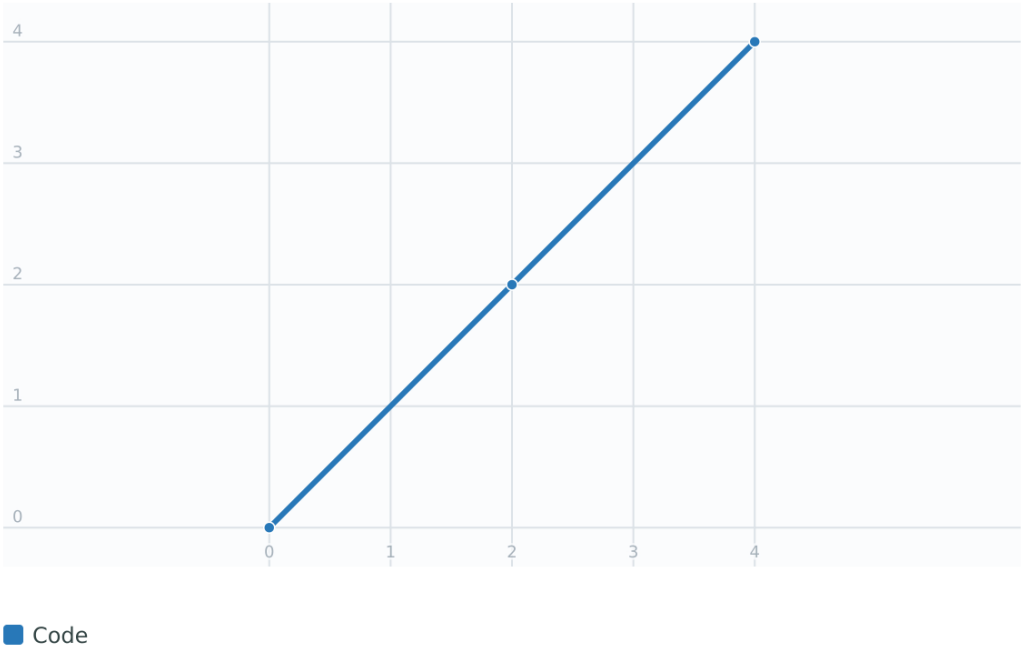
Code

```
SELECT ST_IsSimple('POLYGON((1 2,3 4,5 6,1 2))');
```

Output

```
f
```

Figure



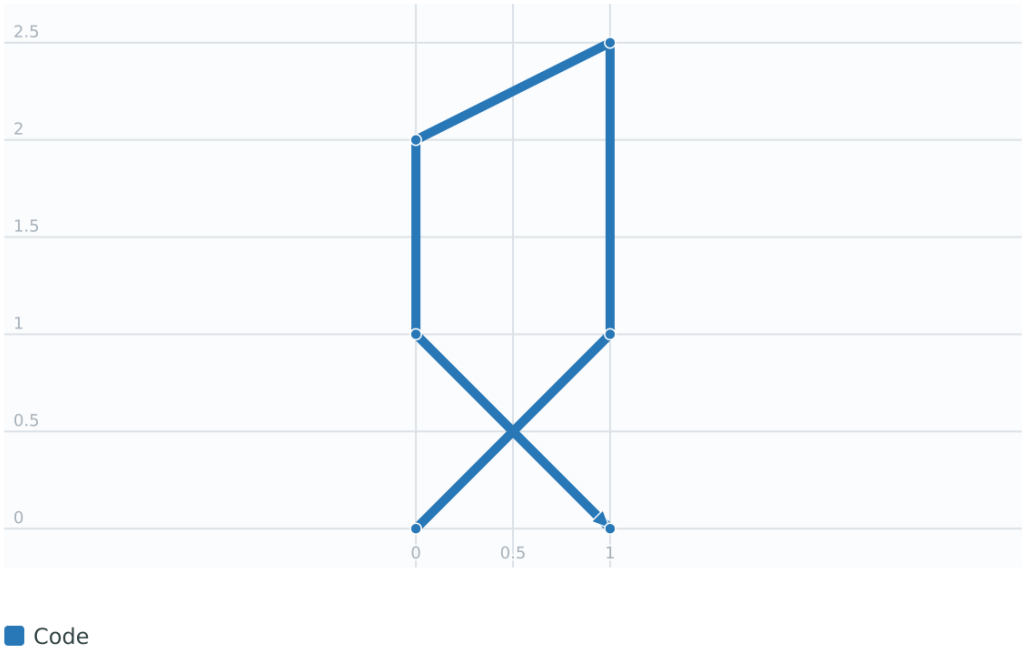
Code

```
SELECT ST_IsSimple('LINESTRING(1 1,2 2,2 3.5,1 3,1 2,2 1)');
```

Output

```
f
```

Figure



☒☒

ST_IsValid

7.4.26 ST_IsSimpleDetail

ST_IsSimpleDetail — Report locations of self-intersection or self-tangency, if any.

☒☒

simple_detail **ST_IsSimpleDetail**(geometry geom, integer findAllLocations = 0, integer boundaryNodeRule = 1);

☒☒

Returns a `simple_detail` row, containing a boolean (`simple`) stating if a geometry is simple and a geometry (`location`) pointing out where it is non-simple.

The optional `findAllLocations` parameter can be set to 1 to report all non-simple locations, rather than just one.

The optional `boundaryNodeRule` values are:

- 1: **OGC-Mod2** - line endpoints are in the Boundary if they occur an odd number of times. This is the rule defined by the OGC SFS standard, and is the default for `ST_Relate`.
- 2: **Endpoint** - all endpoints are in the Boundary.
- 3: **MultivalentEndpoint** - endpoints are in the Boundary if they occur more than once. In other words, the boundary is all the ”attached” or ”inner” endpoints (but not the ”unattached/outer” ones).

- 4: **MonovalentEndpoint** - endpoints are in the Boundary if they occur only once. In other words, the boundary is all the ”unattached” or ”outer” endpoints.

Availability: 3.7.0

☒☒

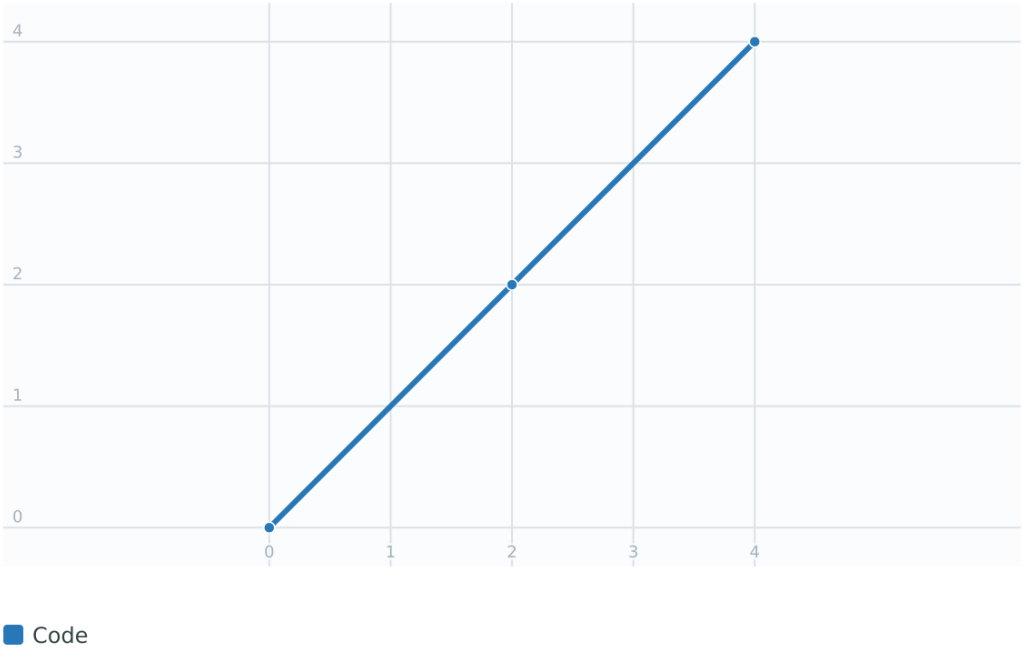
Code

```
SELECT ST_IsSimple('POLYGON((1 2,3 4,5 6,1 2))');
```

Output

```
f
```

Figure



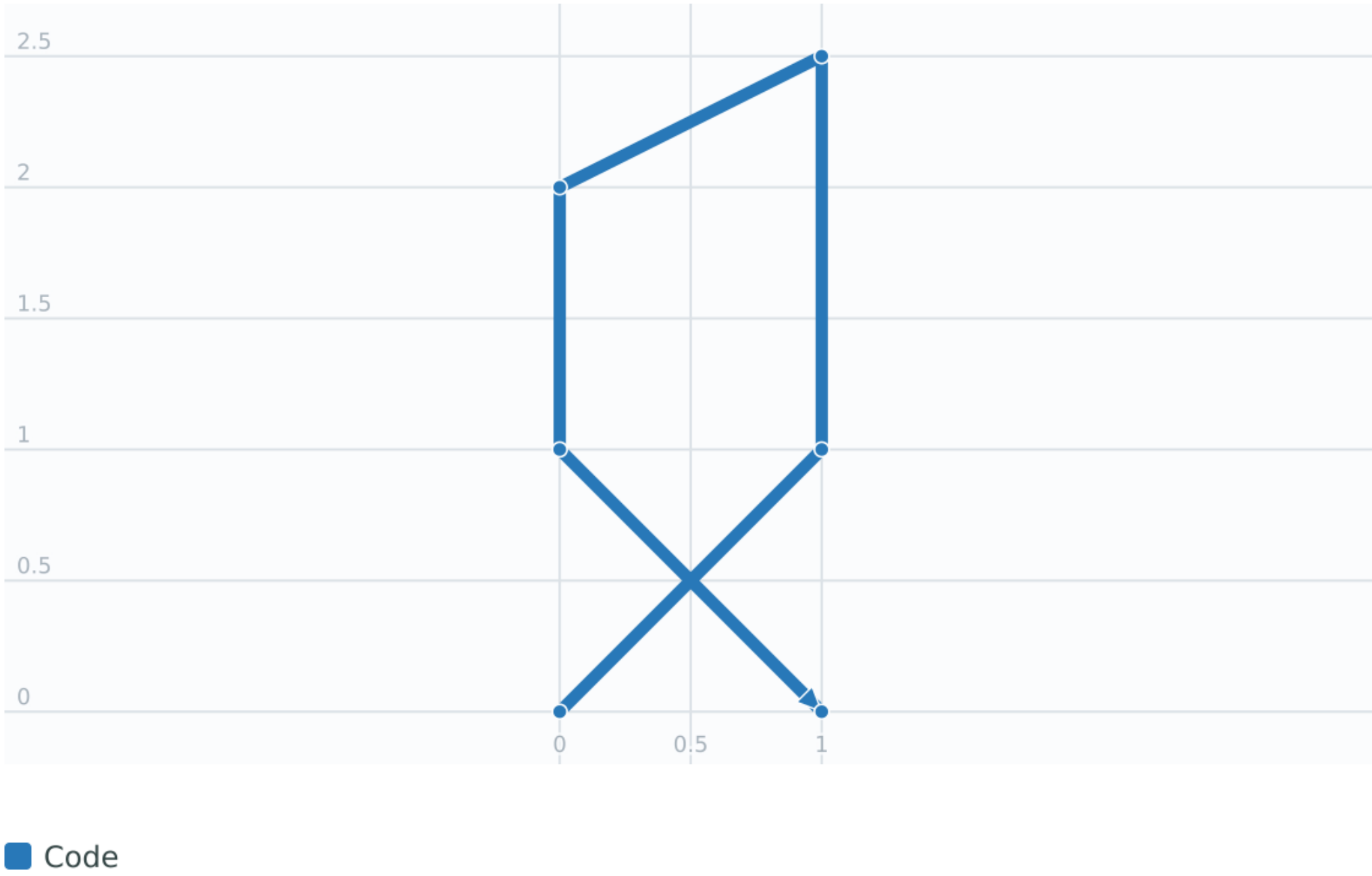
Code

```
SELECT ST_IsSimple('LINESTRING(1 1,2 2,2 3.5,1 3,1 2,2 1)');
```

Output

```
f
```

Figure



☒☒

ST_IsSimple

7.4.27 ST_M

ST_M — Returns the M coordinate of a Point.

☒☒

float **ST_M**(geometry a_point);

☒☒

☒☒☒☒ M ☒☒☒☒☒☒☒☒. M ☒☒☒☒☒☒☒ NULL ☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒☒☒.

> **Note!** ☒☒
> ☒☒☒☒ (☒☒) OGC ☒☒☒☒☒☒☒☒☒☒, ☒☒☒☒☒☒☒☒ (extractor) ☒☒☒☒☒☒☒☒☒☒☒☒ ☒☒☒☒☒☒☒☒☒.

This method implements the OGC Simple Features Implementation Specification for SQL 1.1.

This method implements the SQL/MM specification.

This function supports 3d and will not drop the z-index.

□□

Code

```
SELECT ST_M('POINT(1 2 3 4)');
```

Output

```
4
```

□□

ST_GeomFromEWKT, ST_X, ST_Y, ST_Z

7.4.28 ST_MemSize

ST_MemSize — ST_Geometry □□□□□□□□□□□□□.

□□

integer **ST_MemSize**(geometry geomA);

□□

ST_Geometry □□□□□□□□□□□□□.

This complements the PostgreSQL built-in database object functions pg_column_size, pg_size_pretty, pg_relation_size, pg_total_relation_size.



□□
pg_relation_size which gives the byte size of a table may return byte size lower than ST_MemSize. This is because pg_relation_size does not add toasted table contribution and large geometries are stored in TOAST tables.
pg_total_relation_size □□□□□□□, TOAST □□□□□□□□□□□□□□□□□□□.
pg_column_size returns how much space a geometry would take in a column considering compression, so may be lower than ST_MemSize


This function supports 3d and will not drop the z-index.


This method supports Circular Strings and Curves.


This function supports Polyhedral surfaces.


This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).

Changed: 2.2.0 name changed to ST_MemSize to follow naming convention.

□□

Compare the memory used by all geometries in a Massachusetts town dataset with the subset for Boston.

Code

```
SELECT pg_size_pretty(sum(ST_MemSize(geom))) AS totgeomsum,
       pg_size_pretty(sum(CASE WHEN town = 'BOSTON' THEN ST_MemSize(geom) ELSE 0 END)) AS ↩
    bossum,
       cast(
           sum(CASE WHEN town = 'BOSTON' THEN ST_MemSize(geom) ELSE 0 END) * 100.0
           / sum(ST_MemSize(geom)) AS numeric(10, 2)
       ) AS perbos
FROM towns;
```

Output

```
totgeomsum | bossum | perbos
------------+--------+--------
 1522 kB    | 30 kB  |   1.99
(1 row)
```

Return the memory size of a CircularString in bytes.

Code

```
SELECT ST_MemSize('CIRCULARSTRING(220268 150415,220227 150505,220227 150406)'::geometry);
```

Output

```
st_memsize
------------
         80
(1 row)
```

Calculate the percentage of a table's total size used by geometry data.

Code

```
SELECT pg_total_relation_size('public.neighborhoods') AS fulltable_size,
       sum(ST_MemSize(geom)) AS geomsize,
       sum(ST_MemSize(geom)) * 100.0
           / pg_total_relation_size('public.neighborhoods') AS pergeom
FROM neighborhoods;
```

Output

```
fulltable_size | geomsize |         pergeom
---------------+----------+-------------------------
         262144 |    96238 | 36.71188354492187500000
(1 row)
```

7.4.29 ST_NDims

ST_NDims — ST_Geometry □□□□□□□□□□□□□.

□□

integer **ST_NDims**(geometry g1);

⊠⊠

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. PostGIS ⊠ 2 - 2 ⊠⊠ (x,y), 3 - 3 ⊠⊠ (x,y,z), 3 - ⊠⊠⊠⊠⊠ 2 ⊠⊠ (x,y,m), ⊠ ⊠⊠ 4 - ⊠⊠⊠⊠⊠ 3 ⊠⊠⊠⊠ (x,y,z,m) ⊠⊠⊠⊠⊠⊠.

This function supports 3d and will not drop the z-index.

⊠⊠

Code

```
SELECT ST_NDims('POINT(1 1)') As d2point,
ST_NDims('POINT(1 1 2)') As d3point,
ST_NDims('POINTM(1 1 0.5)') As d2pointm;
```

Output

```
d2point | d3point | d2pointm
---------+---------+----------
       2 |       3 |        3
```

⊠⊠

ST_CoordDim, ST_Dimension, ST_GeomFromEWKT

7.4.30 ST_NPoints

ST_NPoints — ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (⊠⊠⊠) ⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

integer **ST_NPoints**(geometry g1);

⊠⊠

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠⊠⊠: 2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (polyhedral surface) ⊠⊠⊠⊠⊠⊠.

> **⊠⊠**
> 1.3.4 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (curve) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. 1.3.4 ⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

This function supports 3d and will not drop the z-index.

This method supports Circular Strings and Curves.

This function supports Polyhedral surfaces.

⊠⊠

Count the points in a 2D LineString.

Code

```
SELECT ST_NPoints('LINESTRING(77.29 29.07,77.42 29.26,77.27 29.31,77.29 29.07)');
```

Output

```
st_npoints
------------
          4
(1 row)
```

Count the points in a 3D LineString.

Code

```
SELECT ST_NPoints('LINESTRING(77.29 29.07 1,77.42 29.26 0,77.27 29.31 -1,77.29 29.07 3)');
```

Output

```
st_npoints
------------
          4
(1 row)
```

⊠⊠

ST_NumPoints

7.4.31 ST_NRings

ST_NRings — ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

integer **ST_NRings**(geometry geomA);

⊠⊠

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. NumInteriorRings ⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠.

This function supports 3d and will not drop the z-index.

This method supports Circular Strings and Curves.

⊠⊠

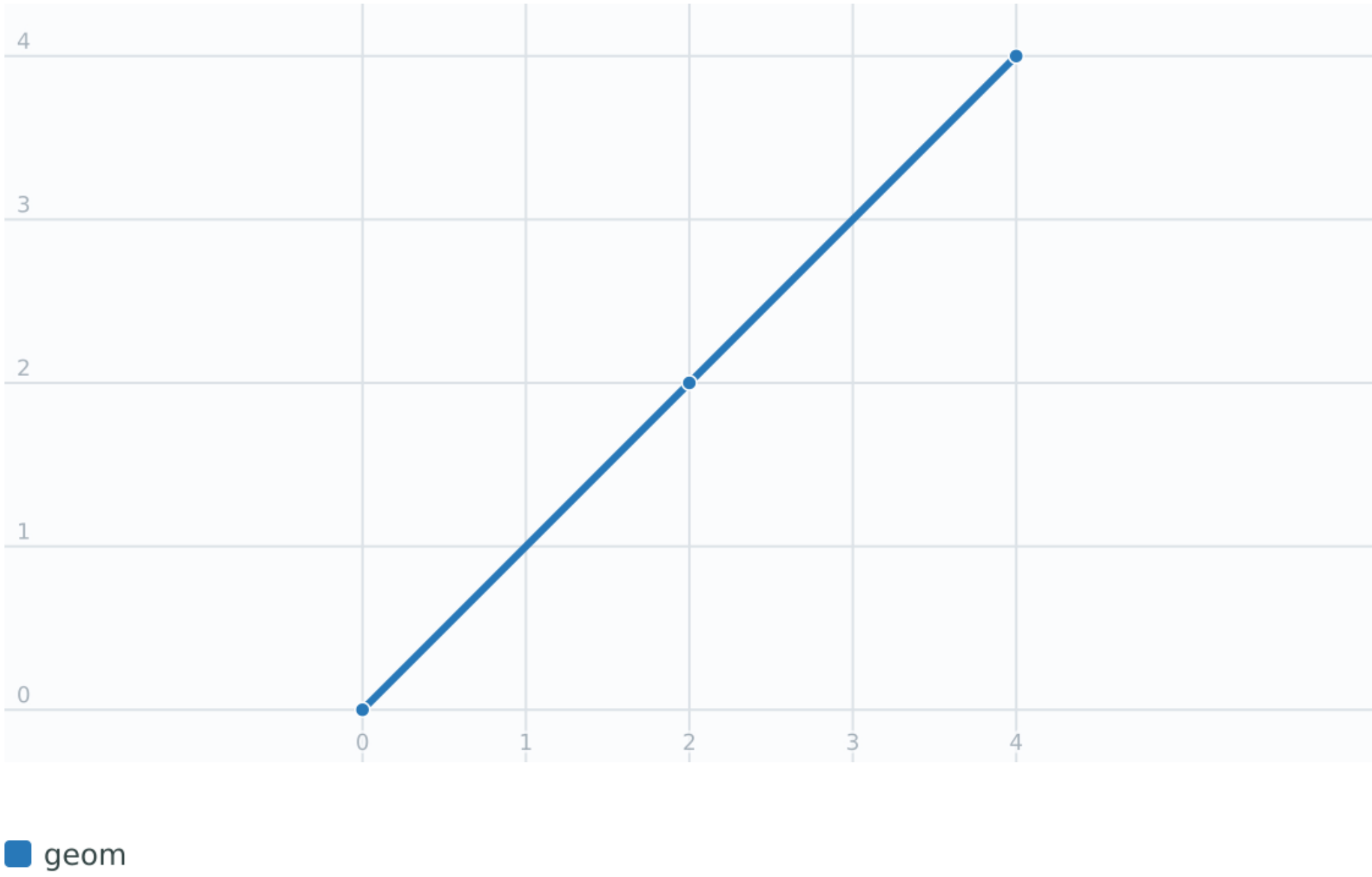
Code

```
SELECT ST_NRings(geom) As Nrings, ST_NumInteriorRings(geom) As ninterrings
FROM (SELECT 'POLYGON((1 2,3 4,5 6,1 2))'::geometry As geom) As foo;
```

Output

```
nrings | ninterrings
--------+-------------
      1 |           0
(1 row)
```

Figure



⊠⊠

ST_NumInteriorRings

7.4.32 ST_NumGeometries

ST_NumGeometries — ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

integer **ST_NumGeometries**(geometry geom);

⊠⊠

Returns the number of elements in a geometry collection (GEOMETRYCOLLECTION or MULTI*). For non-empty atomic geometries returns 1. For empty geometries returns 0.

⊠⊠⊠⊠: 2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠ TIN ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠⊠⊠: 2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ NULL ⊠⊠⊠⊠⊠⊠⊠. 2.0.0 ⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ 1 ⊠⊠⊠⊠⊠⊠.

Changed: 3.6.0 TIN and PolyhedralSurface are treated as unitary geometries and return 1; use ST_NumPatches to count faces.

This method implements the SQL/MM specification. SQL-MM 3: 9.1.4

This function supports 3d and will not drop the z-index.

This function supports Polyhedral surfaces.

This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).

□□

This example returns 1; prior versions would have returned NULL.

Code

```
SELECT ST_NumGeometries('LINESTRING(77.29 29.07,77.42 29.26,77.27 29.31,77.29 29.07)');
```

Output

```
st_numgeometries
-----------------
                1
(1 row)
```

A MultiPoint counts as one geometry within a GeometryCollection.

Code

```
SELECT ST_NumGeometries('GEOMETRYCOLLECTION(MULTIPOINT((-2 3),(-2 2)),
LINESTRING(5 5,10 10),
POLYGON((-7 4.2,-7.1 5,-7.1 4.3,-7 4.2)))');
```

Output

```
st_numgeometries
-----------------
                3
(1 row)
```

□□

ST_GeometryN, ST_Multi

7.4.33 ST_NumInteriorRings

ST_NumInteriorRings — □□□□□□□□□□□□□□□□□□□□□□□□□□□□□.

□□

integer **ST_NumInteriorRings**(geometry a_polygon);

☒☒

☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒ NULL ☒☒☒☒☒☒.

This method implements the SQL/MM specification. SQL-MM 3: 8.2.5

☒☒☒☒: 2.0.0 ☒☒.

☒☒

Count the interior rings of a Polygon.

Code

```
SELECT gid, field1, field2, ST_NumInteriorRings(geom) AS numholes
FROM sometable;
```

For a MultiPolygon, dump its component Polygons and sum their interior ring counts.

Code

```
SELECT gid, field1, field2, SUM(ST_NumInteriorRings(geom)) AS numholes
FROM (
    SELECT gid, field1, field2, (ST_Dump(geom)).geom AS geom
    FROM sometable
) AS foo
GROUP BY gid, field1, field2;
```

☒☒

ST_NumInteriorRing, ST_PointN

7.4.34 ST_NumInteriorRing

ST_NumInteriorRing — Returns the number of interior rings (holes) of a Polygon. Alias for ST_NumInteriorRi

☒☒

integer **ST_NumInteriorRing**(geometry a_polygon);

☒☒

This function is an alias for ST_NumInteriorRings.

1.2.2 ☒☒☒☒☒☒☒☒☒☒☒☒.

☒☒

ST_NumInteriorRings, ST_PointN

7.4.35 ST_NumPatches

ST_NumPatches — Returns the number of faces in a PolyhedralSurface or TIN.

□□

integer **ST_NumPatches**(geometry g1);

□□

Returns the number of faces in a PolyhedralSurface or TIN. Returns NULL for other geometries. Unlike ST_NumGeometries, this function counts individual faces.

2.0.0 □□□□□□□□□□□□.

This function supports 3d and will not drop the z-index.

This method implements the OGC Simple Features Implementation Specification for SQL 1.1.

This method implements the SQL/MM specification. SQL-MM ISO/IEC 13249-3: 8.5

This function supports Polyhedral surfaces.

□□

Code

```
SELECT ST_NumPatches('POLYHEDRALSURFACE( ((0 0 0,0 0 1,0 1 1,0 1 0,0 0 0)),
((0 0 0,0 1 0,1 1 0,1 0 0,0 0 0)),((0 0 0,1 0 0,1 0 1,0 0 1,0 0 0)),
((1 1 0,1 1 1,1 0 1,1 0 0,1 1 0)),
((0 1 0,0 1 1,1 1 1,1 1 0,0 1 0)),((0 0 1,1 0 1,1 1 1,0 1 1,0 0 1)) )');
```

Output

```
st_numpatches
---------------
             6
(1 row)
```

□□

ST_GeomFromEWKT, ST_NumGeometries, ST_PatchN

7.4.36 ST_NumPoints

ST_NumPoints — ST_LineString □□ ST_CircularString □□□□□□□□□□□□□□□□.

□□

integer **ST_NumPoints**(geometry g1);

⊠⊠

ST_LineString ⊠⊠ ST_CircularString ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. 1.4 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. 1.4 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ST_NPoints ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ST_NPoints ⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

This method implements the OGC Simple Features Implementation Specification for SQL 1.1.

This method implements the SQL/MM specification. SQL-MM 3: 7.2.4

⊠⊠

Code

```
SELECT ST_NumPoints('LINESTRING(77.29 29.07,77.42 29.26,77.27 29.31,77.29 29.07)');
```

Output

```
st_numpoints
--------------
            4
(1 row)
```

⊠⊠

ST_NPoints

7.4.37 ST_PatchN

ST_PatchN — Returns the Nth face of a PolyhedralSurface or TIN.

⊠⊠

geometry **ST_PatchN**(geometry geomA, integer n);

⊠⊠

Returns the 1-based Nth face of a PolyhedralSurface or TIN. Returns NULL for other geometries or an out-of-range index. Use this function to access faces, because ST_GeometryN treats PolyhedralSurface and TIN as unitary geometries.

> **⊠⊠**
> ⊠⊠⊠⊠ 1-⊠⊠⊠⊠⊠.

> **⊠⊠**
> ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ST_Dump ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

2.0.0 □□□□□□□□□□□□.

This method implements the SQL/MM specification. SQL-MM ISO/IEC 13249-3: 8.5

This function supports 3d and will not drop the z-index.

This function supports Polyhedral surfaces.

This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).

□□

Extract the 2nd face of the polyhedral surface.

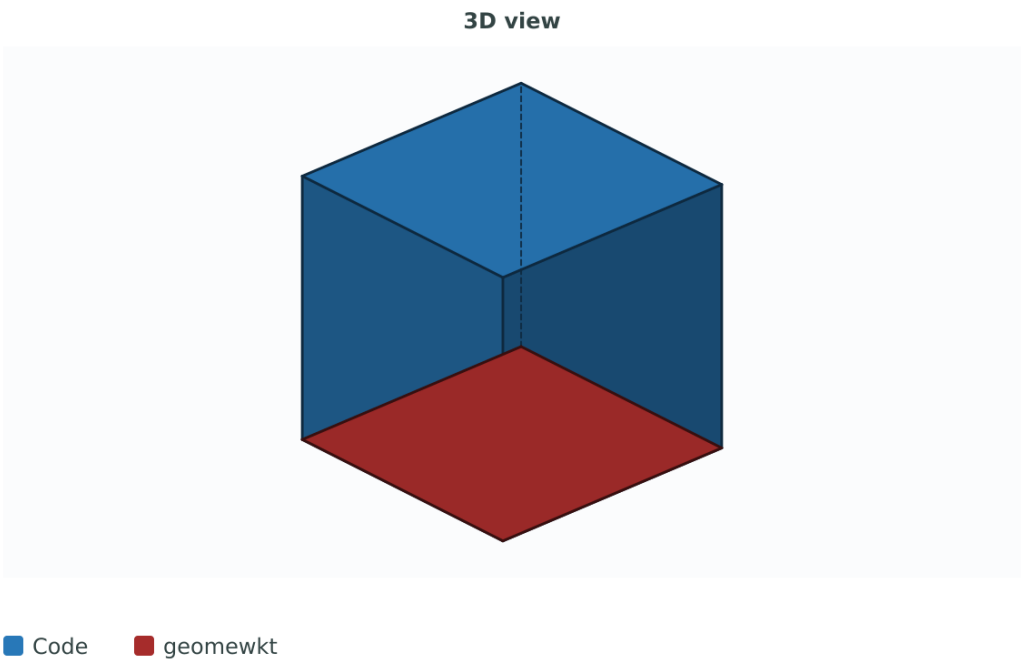
Code

```
SELECT ST_PatchN(geom, 2) As geomewkt
FROM (
VALUES ('POLYHEDRALSURFACE( ((0 0 0,0 0 1,0 1 1,0 1 0,0 0 0)),
    ((0 0 0,0 1 0,1 1 0,1 0 0,0 0 0)),((0 0 0,1 0 0,1 0 1,0 0 1,0 0 0)),
    ((1 1 0,1 1 1,1 0 1,1 0 0,1 1 0)),
    ((0 1 0,0 1 1,1 1 1,1 1 0,0 1 0)),((0 0 1,1 0 1,1 1 1,0 1 1,0 0 1)) )'::geometry) ) As ↩
    foo(geom);
```

Output

```
POLYGON((0 0 0,0 1 0,1 1 0,1 0 0,0 0 0))
```

Figure



□□

ST_AsEWKT, ST_GeomFromEWKT, ST_Dump, ST_GeometryN, ST_NumGeometries, ST_NumPatches

7.4.38 ST_PointN

ST_PointN — ST_LineString ⊠⊠ ST_CircularString ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

geometry **ST_PointN**(geometry a_linestring, integer n);

⊠⊠

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ N ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, -1 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ NULL ⊠⊠⊠⊠⊠⊠.

> **Note:** ⊠⊠
> Index is 1-based as for OGC specs since version 0.8.0. Backward indexing (negative index) is not in OGC. Previous versions implemented this as 0-based instead.

> **Note:** ⊠⊠
> ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ N ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠ ST_Dump ⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠.

This method implements the OGC Simple Features Implementation Specification for SQL 1.1.

This method implements the SQL/MM specification. SQL-MM 3: 7.2.5, 7.3.5

This function supports 3d and will not drop the z-index.

This method supports Circular Strings and Curves.

⊠⊠⊠⊠: 2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. PostGIS ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. 2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ NULL ⊠⊠⊠⊠⊠⊠ ⊠⊠.

⊠⊠⊠⊠: 2.3.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠ (-1 ⊠⊠⊠⊠⊠⊠⊠) ⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

This example extracts all POINT geometries from a LineString.

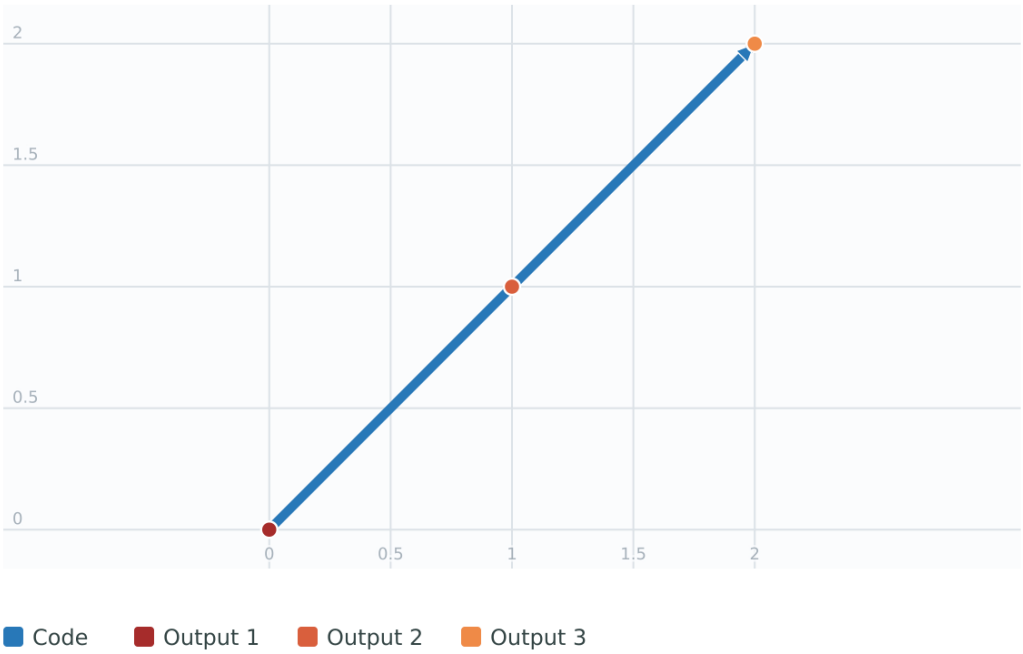
Code

```
SELECT ST_PointN(
      column1,
      generate_series(1, ST_NPoints(column1))
   )
FROM ( VALUES ('LINESTRING(0 0,1 1,2 2)'::geometry) ) AS foo;
```

Output

```
POINT(0 0)
 POINT(1 1)
 POINT(2 2)
(3 rows)
```

Figure



This example uses a CircularString.

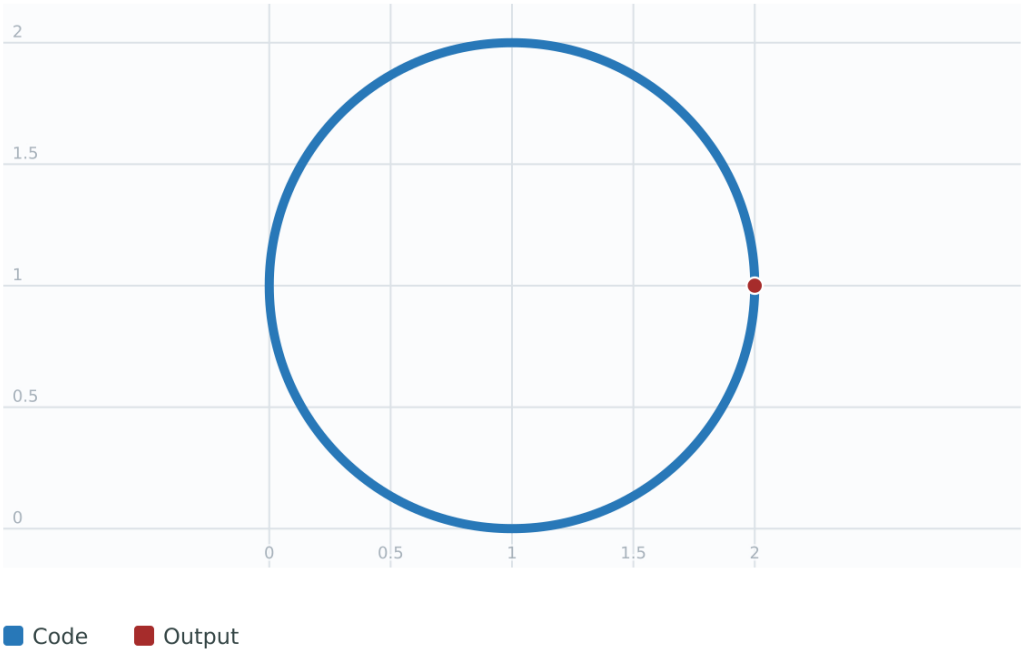
Code

```
SELECT ST_PointN('CIRCULARSTRING(1 2,3 2,1 2)', 2);
```

Output

```
POINT(3 2)
```

Figure



The index is 1-based. A negative index counts backward from the end of the LineString.

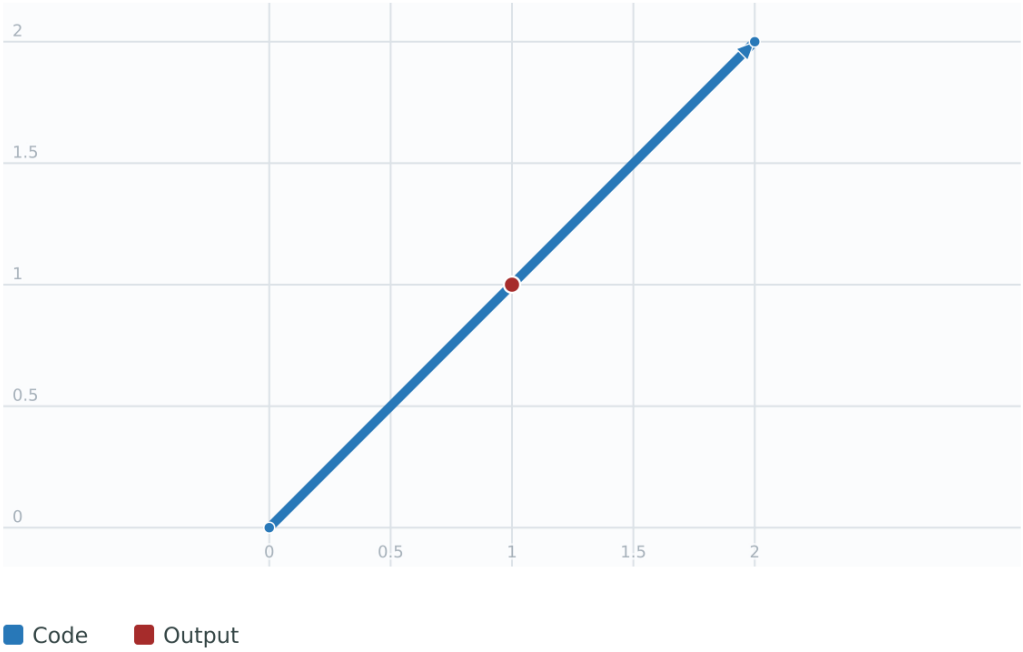
Code

```
SELECT f
FROM ST_GeomFromText('LINESTRING(0 0 0,1 1 1,2 2 2)') AS g
  CROSS JOIN ST_PointN(g, -2) AS f;
```

Output

```
POINT Z (1 1 1)
```

Figure



⊠⊠

ST_NPoints

7.4.39 ST_Points

ST_Points — ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

geometry **ST_Points**(geometry geom);

⊠⊠

Returns a MultiPoint containing all the coordinates of a geometry. Duplicate points are preserved, including the start and end points of ring geometries. (If desired, duplicate points can be removed by calling ST_RemoveRepeatedPoints on the result).

To obtain information about the position of each coordinate in the parent geometry use ST_DumpPoints.

M and Z coordinates are preserved if present.

This method supports Circular Strings and Curves.

This function supports 3d and will not drop the z-index.

2.3.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

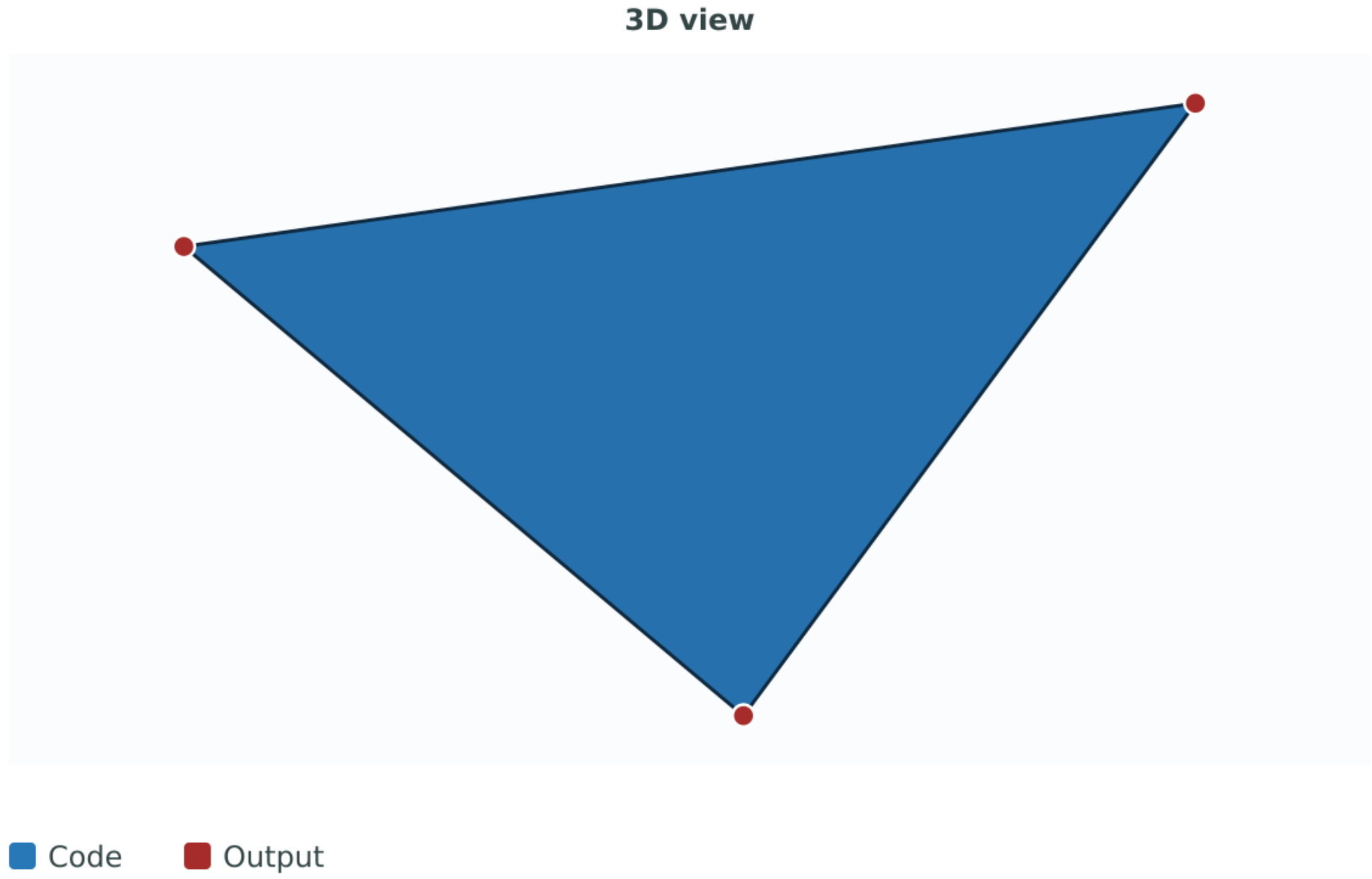
Code

```
SELECT ST_Points('POLYGON Z ((30 10 4,10 30 5,40 40 6,30 10 4))');
```

Output

```
MULTIPOINT Z ((30 10 4),(10 30 5),(40 40 6),(30 10 4))
```

Figure



⊠⊠

ST_RemoveRepeatedPoints, ST_PointN

7.4.40 ST_StartPoint

ST_StartPoint — Returns the first point of a LineString, CircularLineString, or NURBSCurve.

⊠⊠

geometry **ST_StartPoint**(geometry geomA);

⊠⊠

Returns the first point of a `LINESTRING`, `CIRCULARLINESTRING`, or `NURBSCURVE` geometry as a `POINT`. For other geometries, returns the first point in coordinate order.

This method implements the SQL/MM specification. SQL-MM 3: 7.1.3

This function supports 3d and will not drop the z-index.

This method supports Circular Strings and Curves.

Enhanced: 3.2.0 returns a point for all geometries. Prior behavior returns NULLs if input was not a LineString.

⊠⊠⊠⊠: 2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. PostGIS ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. 2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ NULL ⊠⊠⊠⊠⊠⊠ ⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ 2.0 ⊠⊠⊠⊠ NULL ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

□□

Start point of a LineString

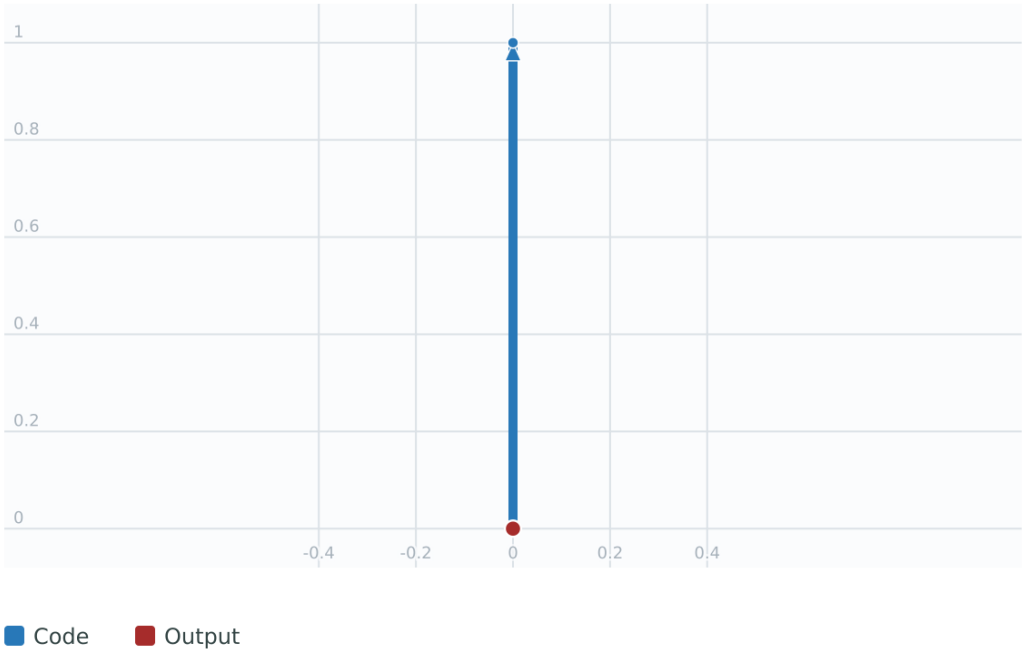
Code

```
SELECT ST_StartPoint('LINESTRING(0 1,0 2)'::geometry);
```

Output

```
POINT(0 1)
```

Figure



Start point of a Point is the Point itself

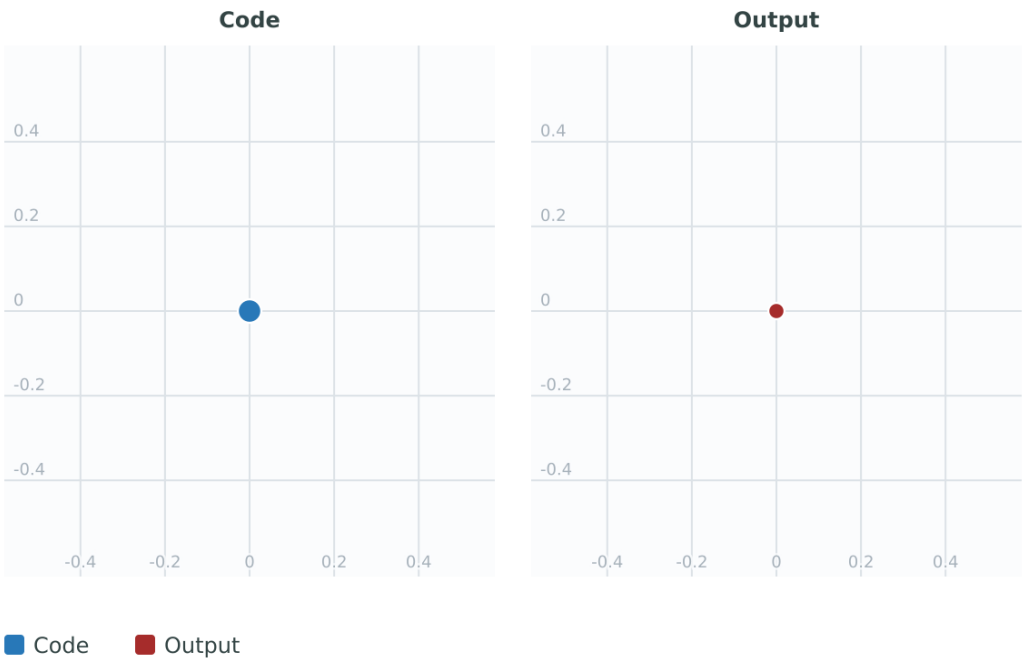
Code

```
SELECT ST_StartPoint('POINT(0 1)'::geometry);
```

Output

```
POINT(0 1)
```

Figure



Start point of a 3D LineString

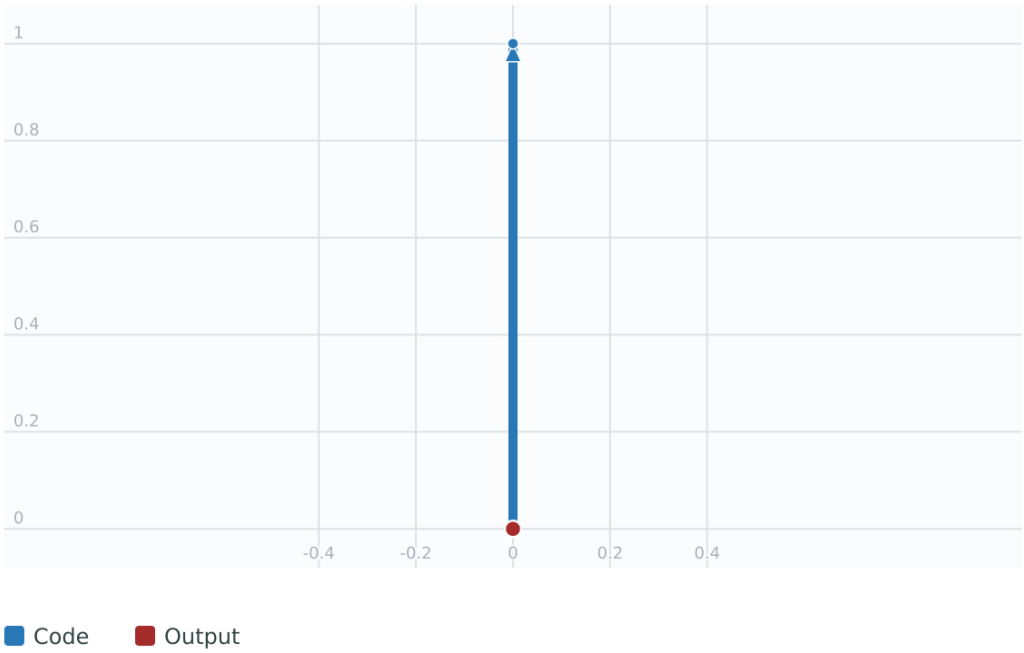
Code

```
SELECT ST_StartPoint('LINESTRING(0 1 1,0 2 2)'::geometry);
```

Output

```
POINT(0 1 1)
```

Figure



ST_LineString □□ ST_CircularString □□□□□□□□□□□□□□□□□□.

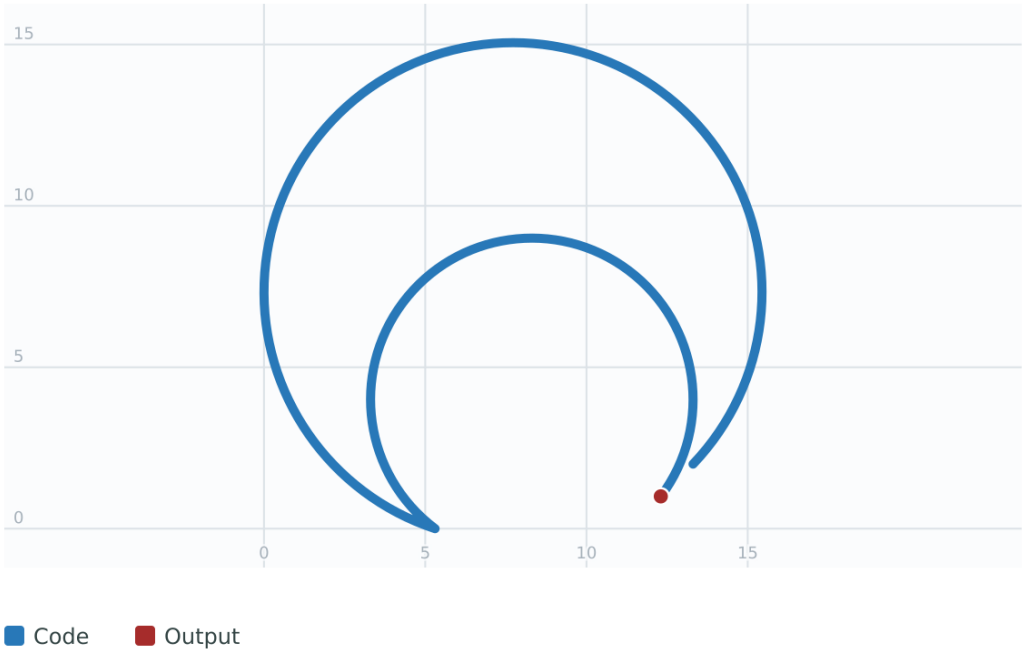
Code

```
SELECT ST_StartPoint('CIRCULARSTRING(5 2,-3 2,-2 1,-4 2,6 3)'::geometry);
```

Output

```
POINT(5 2)
```

Figure



Start point of a NURBSCurve

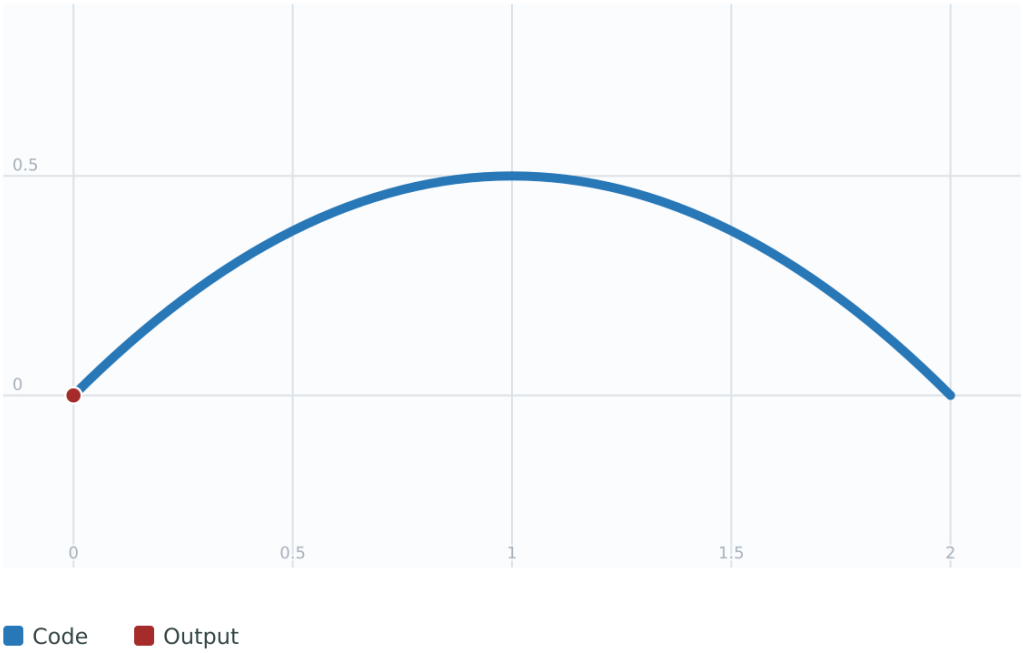
Code

```
SELECT ST_StartPoint('NURBSCURVE(2, (0 0, 1 1, 2 0))'::geometry);
```

Output

```
POINT(0 0)
```

Figure



Build edge node references from the start and end points of a LineString table. The example uses exact endpoint equality, so linework with nearly coincident endpoints should be snapped or otherwise cleaned first.

Code

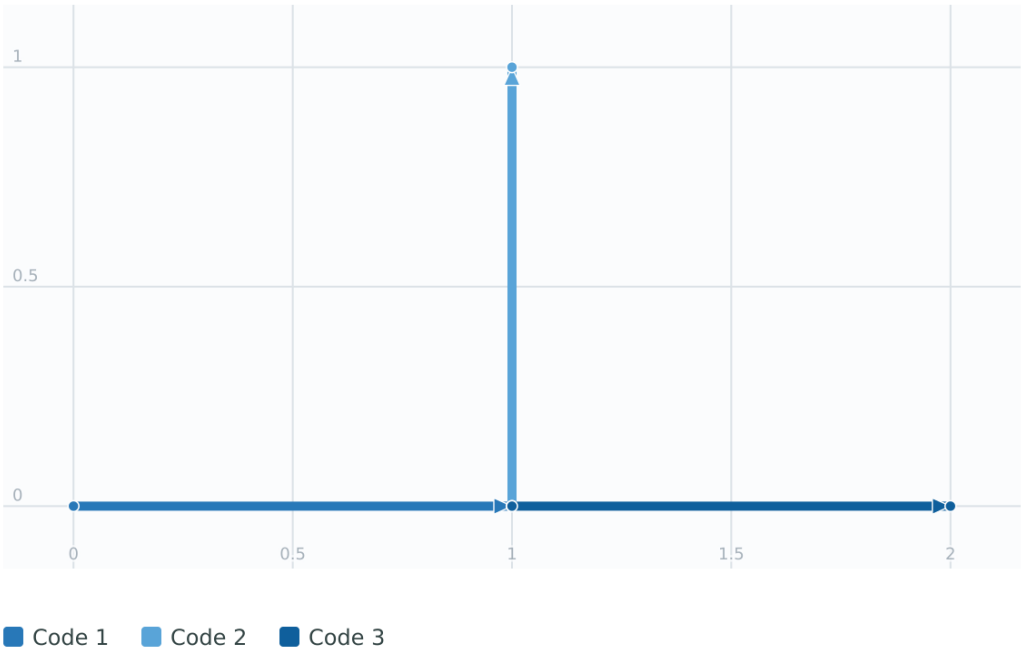
```
WITH roads(road_id, geom) AS (
  VALUES
    (1, 'LINESTRING(0 0, 1 0)'::geometry),
    (2, 'LINESTRING(1 0, 1 1)'::geometry),
    (3, 'LINESTRING(1 0, 2 0)'::geometry)
),
endpoints AS (
  SELECT road_id, 'from' AS end_name, ST_StartPoint(geom) AS geom,
         ST_AsEWKB(ST_StartPoint(geom)) AS endpoint_key
  FROM roads
  UNION ALL
  SELECT road_id, 'to' AS end_name, ST_EndPoint(geom) AS geom,
         ST_AsEWKB(ST_EndPoint(geom)) AS endpoint_key
  FROM roads
),
nodes AS (
  SELECT dense_rank() OVER (ORDER BY ST_X(geom), ST_Y(geom), endpoint_key) AS node_id,
         endpoint_key
  FROM (
    SELECT DISTINCT ON (endpoint_key) endpoint_key, geom
    FROM endpoints
    ORDER BY endpoint_key
  ) AS distinct_endpoints
)
SELECT e.road_id,
       max(n.node_id) FILTER (WHERE e.end_name = 'from') AS from_node,
       max(n.node_id) FILTER (WHERE e.end_name = 'to') AS to_node
FROM endpoints AS e
JOIN nodes AS n USING (endpoint_key)
GROUP BY e.road_id
```

```
ORDER BY e.road_id;
```

Output

```
road_id | from_node | to_node
--------+-----------+---------
     1 |         1 |       2
     2 |         2 |       3
     3 |         2 |       4
```

Figure



☒☒

ST_EndPoint, ST_PointN

7.4.41 ST_StartM

ST_StartM — Returns the M coordinate of the first point of a LineString, CircularLineString, or NURBSCurve.

☒☒

float8 **ST_StartM**(geometry geom);

☒☒

Returns the M coordinate of the first point of a `LINESTRING`, `CIRCULARLINESTRING`, or `NURBSCURVE` geometry. Returns `NULL` if the input geometry does not have an M dimension or is not a `LINESTRING`, `CIRCULARLINESTRING`, or `NURBSCURVE`.

This method supports Circular Strings and Curves.

□□

Start M value of a LineString

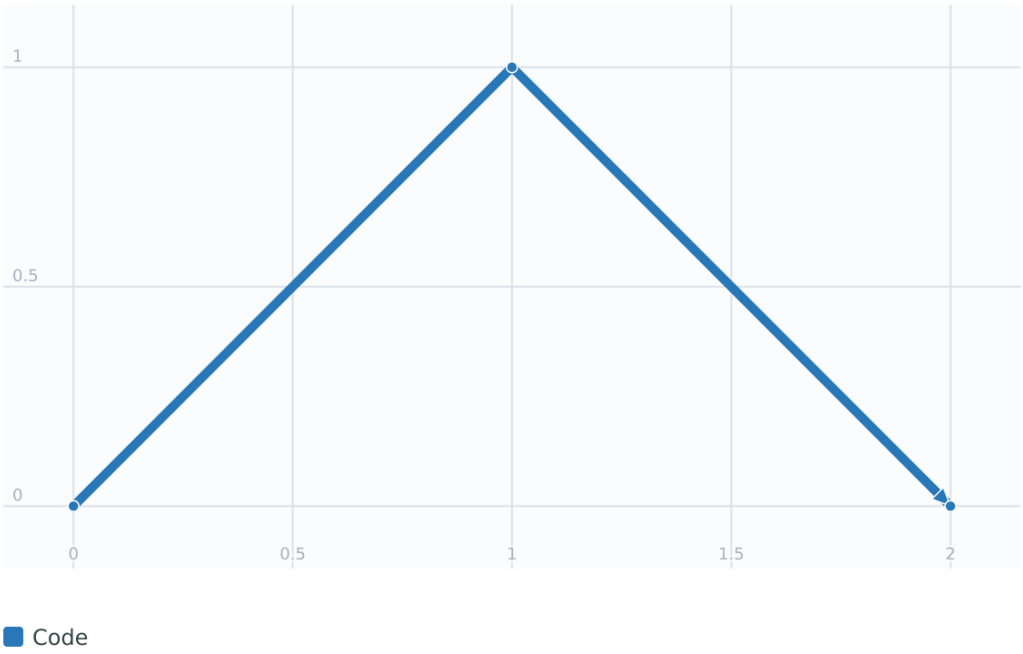
Code

```
SELECT ST_StartM('LINESTRING M(0 0 10,1 1 20,2 0 30)'::geometry);
```

Output

```
10
```

Figure



Start M value of a CircularString

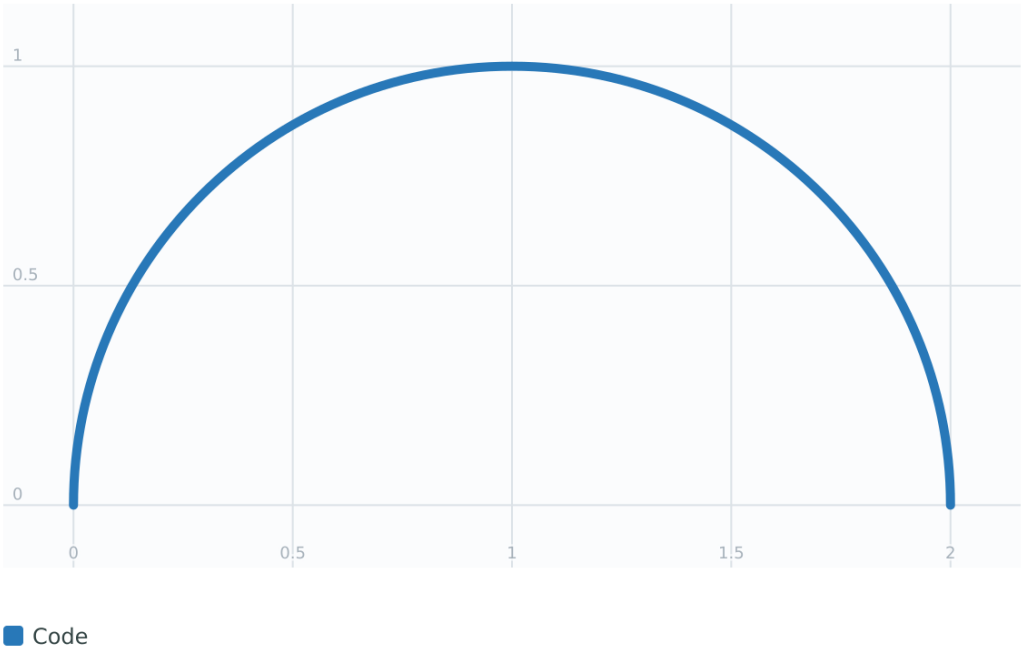
Code

```
SELECT ST_StartM('CIRCULARSTRING M(0 0 5,1 1 10,2 0 15)'::geometry);
```

Output

```
5
```

Figure



Start M value of a NURBSCurve

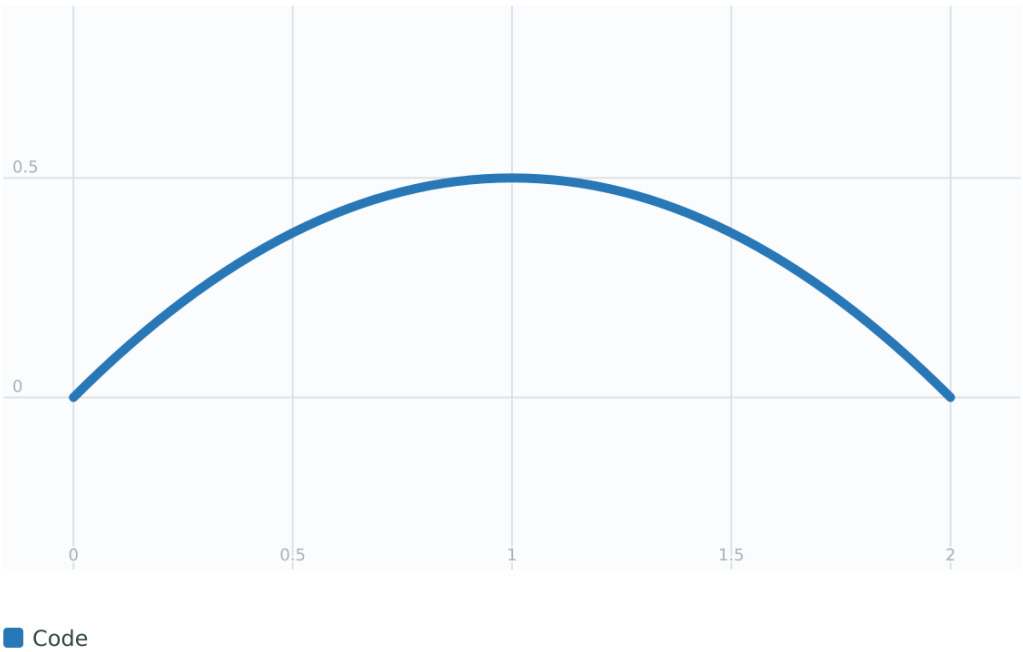
Code

```
SELECT ST_StartM('NURBSCURVE M(2, (0 0 100, 1 1 200, 2 0 300))'::geometry);
```

Output

```
100
```

Figure



Geometry without M dimension returns NULL

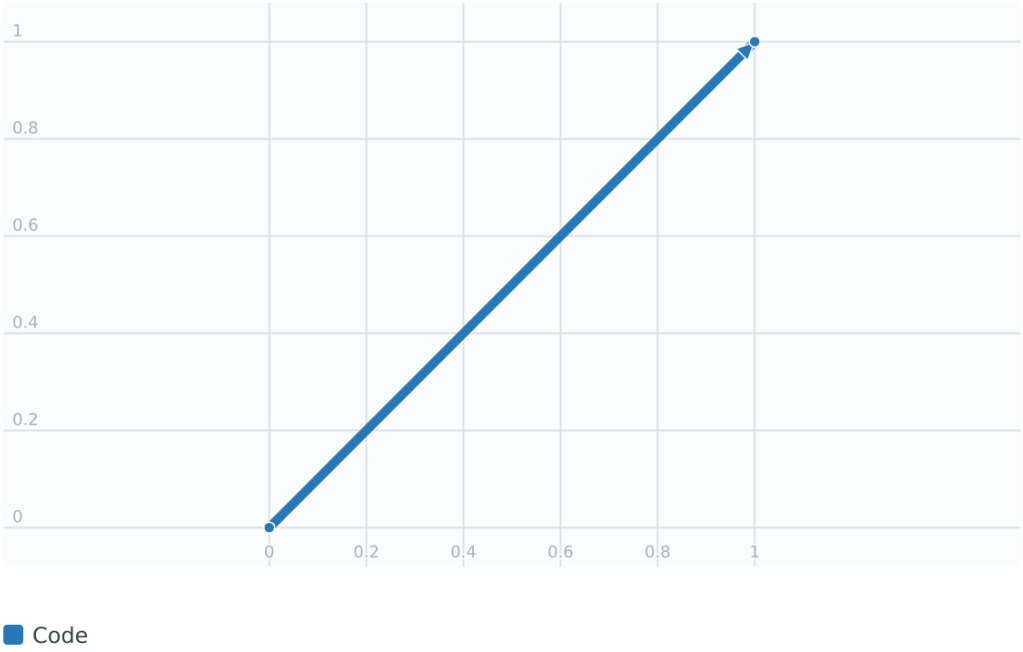
Code

```
SELECT ST_StartM('LINESTRING(0 0,1 1)'::geometry) IS NULL AS is_null;
```

Output

```
t
```

Figure



☒☒

ST_EndM, ST_SetStartM, ST_StartPoint

7.4.42 ST_EndM

ST_EndM — Returns the M coordinate of the last point of a LineString, CircularLineString, or NURBSCurve.

☒☒

float8 **ST_EndM**(geometry geom);

☒☒

Returns the M coordinate of the last point of a `LINESTRING`, `CIRCULARLINESTRING`, or `NURBSCURVE` geometry. Returns `NULL` if the input geometry does not have an M dimension or is not a `LINESTRING`, `CIRCULARLINESTRING`, or `NURBSCURVE`.

This method supports Circular Strings and Curves.

□□

End M value of a LineString

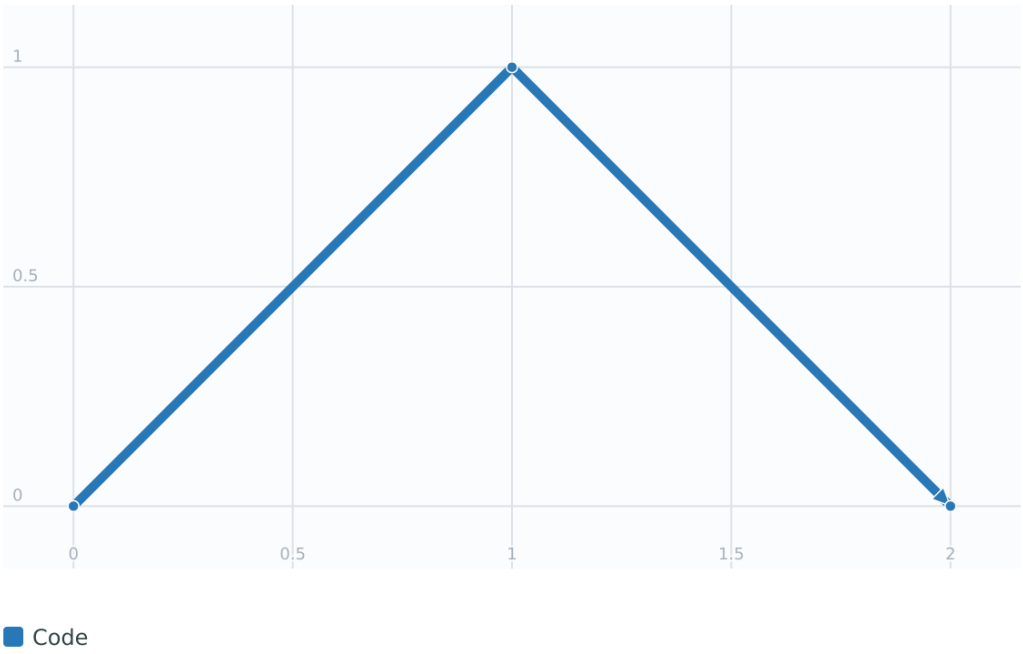
Code

```
SELECT ST_EndM('LINESTRING M(0 0 10,1 1 20,2 0 30)'::geometry);
```

Output

```
30
```

Figure



End M value of a CircularString

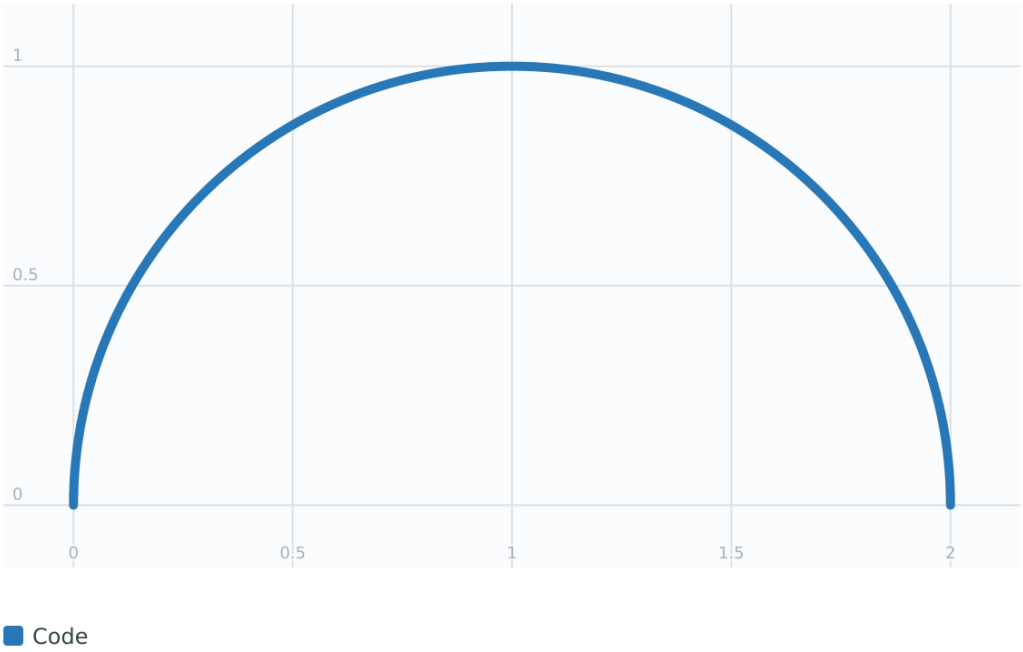
Code

```
SELECT ST_EndM('CIRCULARSTRING M(0 0 5,1 1 10,2 0 15)'::geometry);
```

Output

```
15
```

Figure



End M value of a NURBSCurve

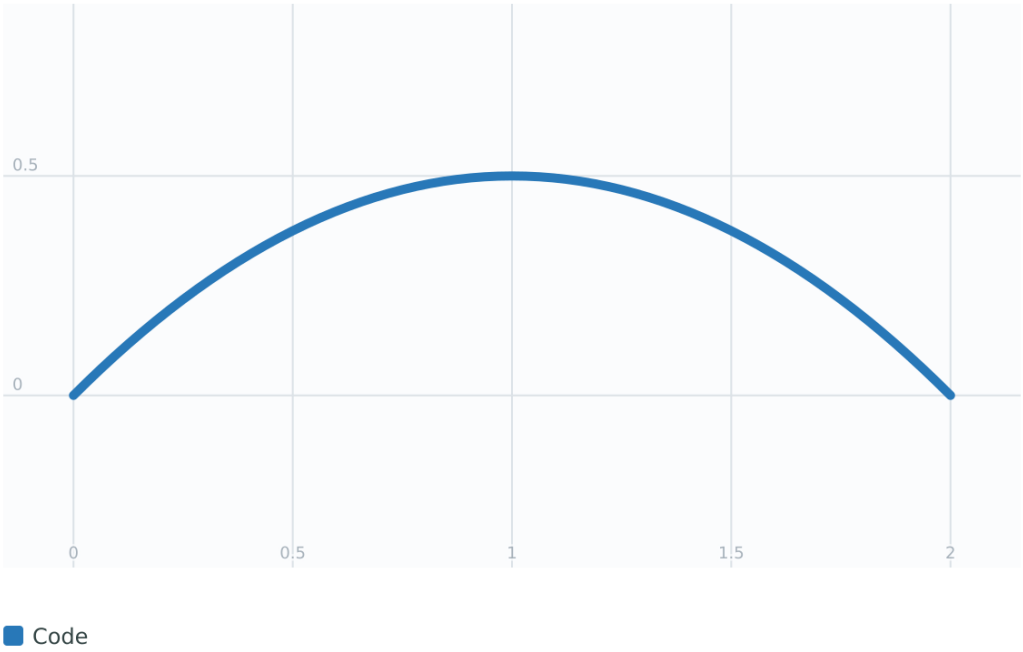
Code

```
SELECT ST_EndM('NURBSCURVE M(2, (0 0 100, 1 1 200, 2 0 300))'::geometry);
```

Output

```
300
```

Figure



Geometry without M dimension returns NULL

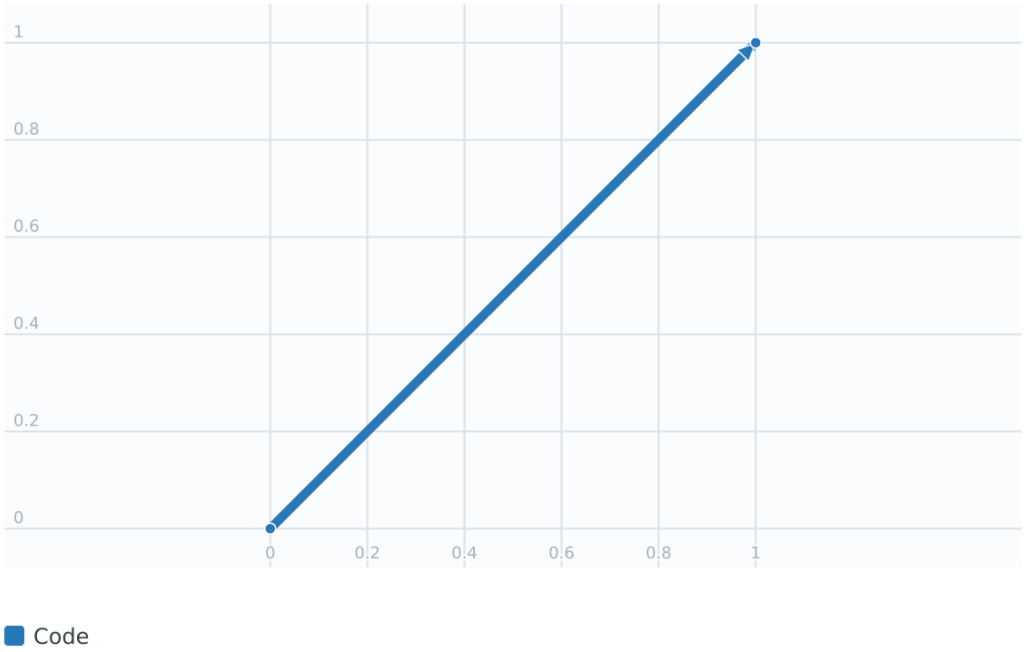
Code

```
SELECT ST_EndM('LINESTRING(0 0,1 1)'::geometry) IS NULL AS is_null;
```

Output

```
t
```

Figure



□□

ST_StartM, ST_SetEndM, ST_EndPoint

7.4.43 ST_SetStartM

ST_SetStartM — Sets the M coordinate of the first point of a LineString, CircularLineString, or NURBSCurve.

□□

geometry **ST_SetStartM**(geometry geom, float8 m);

□□

Returns a copy of the input `LINESTRING`, `CIRCULARLINESTRING`, or `NURBSCURVE` geometry with the M coordinate of the first point set to the specified value. If the input geometry does not have an M dimension, it will be added. Returns `NULL` if the input is not a `LINESTRING`, `CIRCULARLINESTRING`, or `NURBSCURVE`.

This method supports Circular Strings and Curves.

□□

Set start M value of a LineString

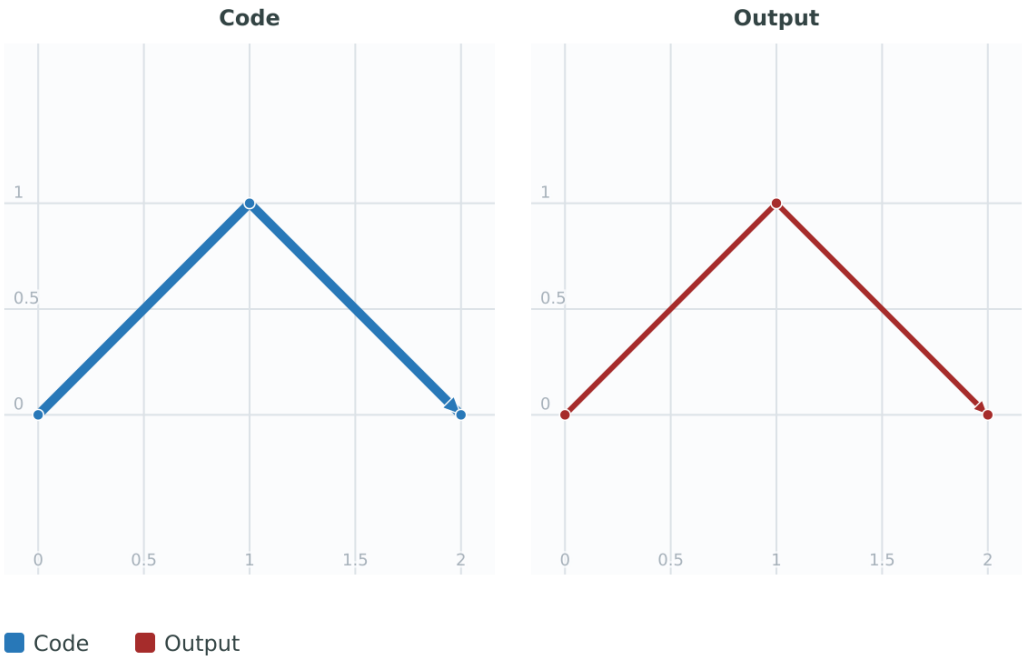
Code

```
SELECT ST_SetStartM('LINESTRING M(0 0 10,1 1 20,2 0 30)'::geometry, 100);
```

Output

```
LINESTRING M (0 0 100,1 1 20,2 0 30)
```

Figure



Add M dimension and set start M value

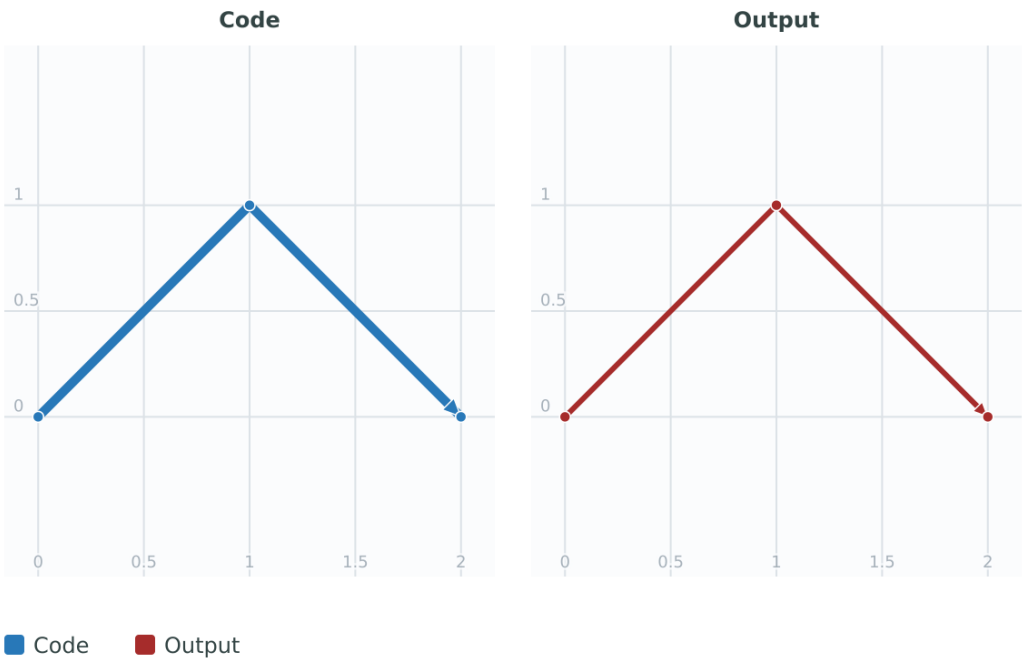
Code

```
SELECT ST_SetStartM('LINESTRING(0 0,1 1,2 0)'::geometry, 50);
```

Output

```
LINESTRING M (0 0 50,1 1 0,2 0 0)
```

Figure



Set start M value of a CircularString

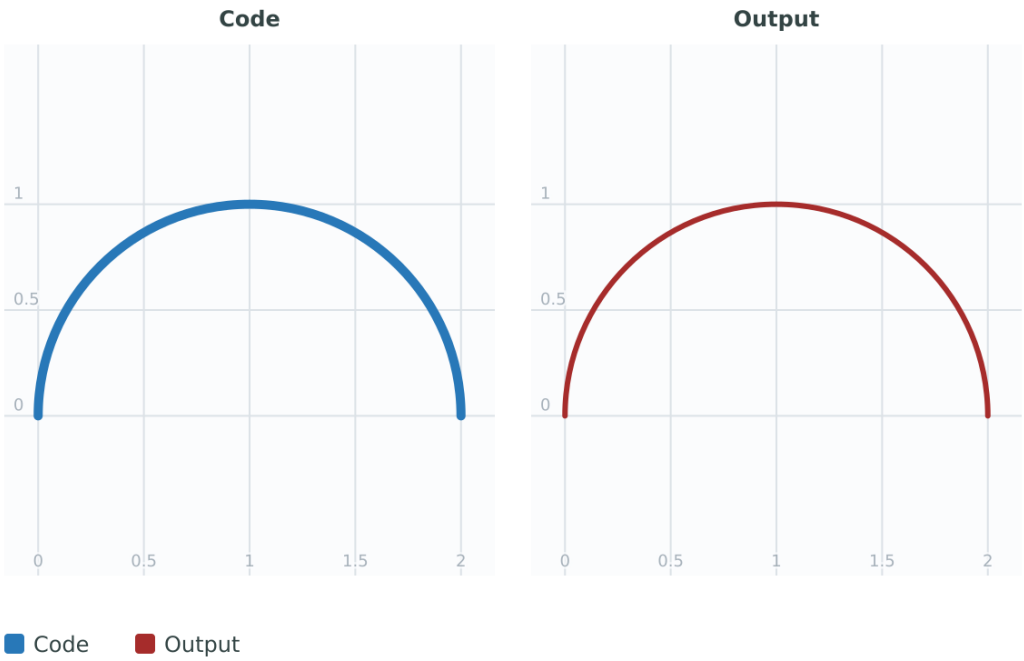
Code

```
SELECT ST_SetStartM('CIRCULARSTRING M(0 0 5,1 1 10,2 0 15)'::geometry, 99);
```

Output

```
CIRCULARSTRING M (0 0 99,1 1 10,2 0 15)
```

Figure



Set start M value of a NURBSCurve

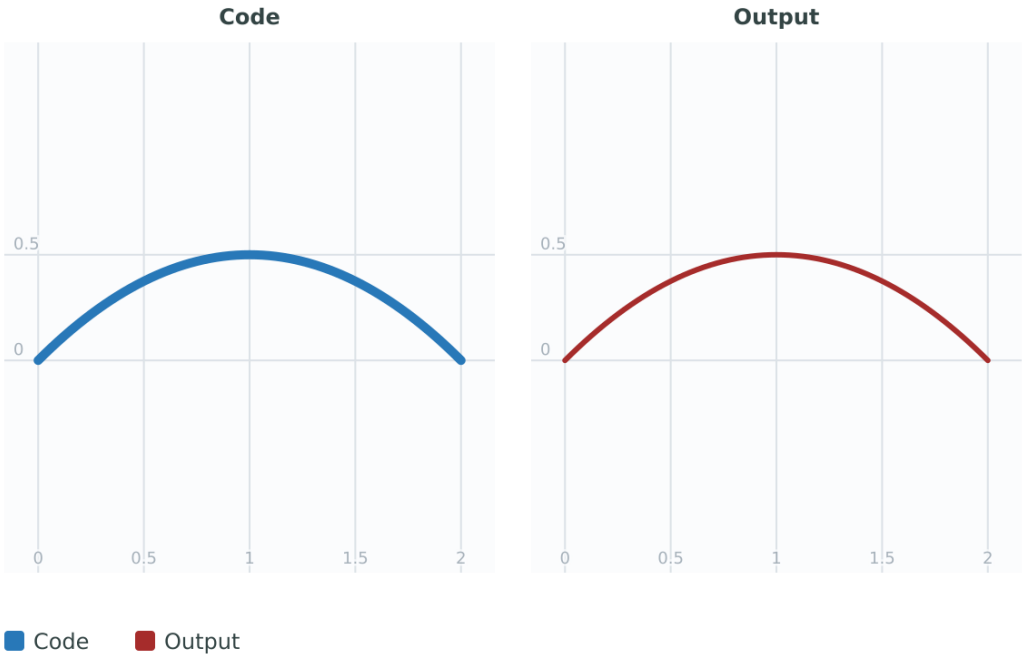
Code

```
SELECT ST_SetStartM('NURBSCURVE M(2, (0 0 100, 1 1 200, 2 0 300))'::geometry, 500);
```

Output

```
NURBSCURVE M (DEGREE 2,CONTROLPOINTS M (NURBSPOINT(WEIGHTEDPOINT M (0 0 500),WEIGHT 1), NURBSPOINT(WEIGHTEDPOINT M (1 ↩
    1 200),WEIGHT 1), NURBSPOINT(WEIGHTEDPOINT M (2 0 300),WEIGHT 1)),KNOTS (KNOT(0,3), KNOT(1,3)))
```

Figure



□□

ST_SetEndM, ST_StartM, ST_StartPoint

7.4.44 ST_SetEndM

ST_SetEndM — Sets the M coordinate of the last point of a LineString, CircularLineString, or NURBSCurve.

□□

geometry **ST_SetEndM**(geometry geom, float8 m);

□□

Returns a copy of the input LINESTRING, CIRCULARLINESTRING, or NURBSCURVE geometry with the M coordinate of the last point set to the specified value. If the input geometry does not have an M dimension, it will be added. Returns NULL if the input is not a LINESTRING, CIRCULARLINESTRING, or NURBSCURVE.

This method supports Circular Strings and Curves.

□□

Set end M value of a LineString

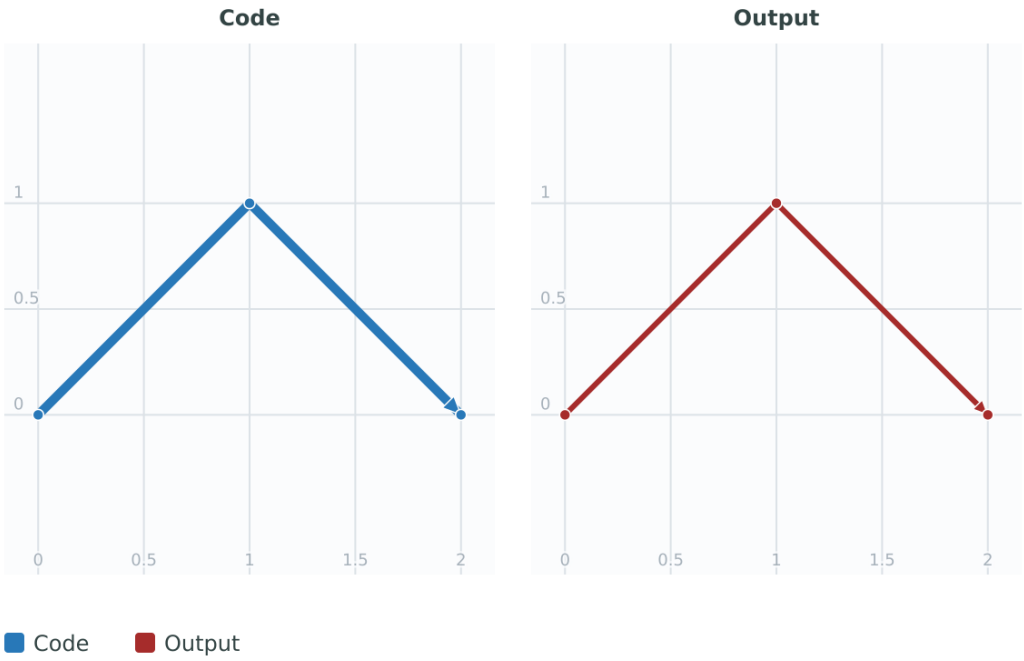
Code

```
SELECT ST_SetEndM('LINESTRING M(0 0 10,1 1 20,2 0 30)'::geometry, 999);
```

Output

```
LINESTRING M (0 0 10,1 1 20,2 0 999)
```

Figure



Add M dimension and set end M value

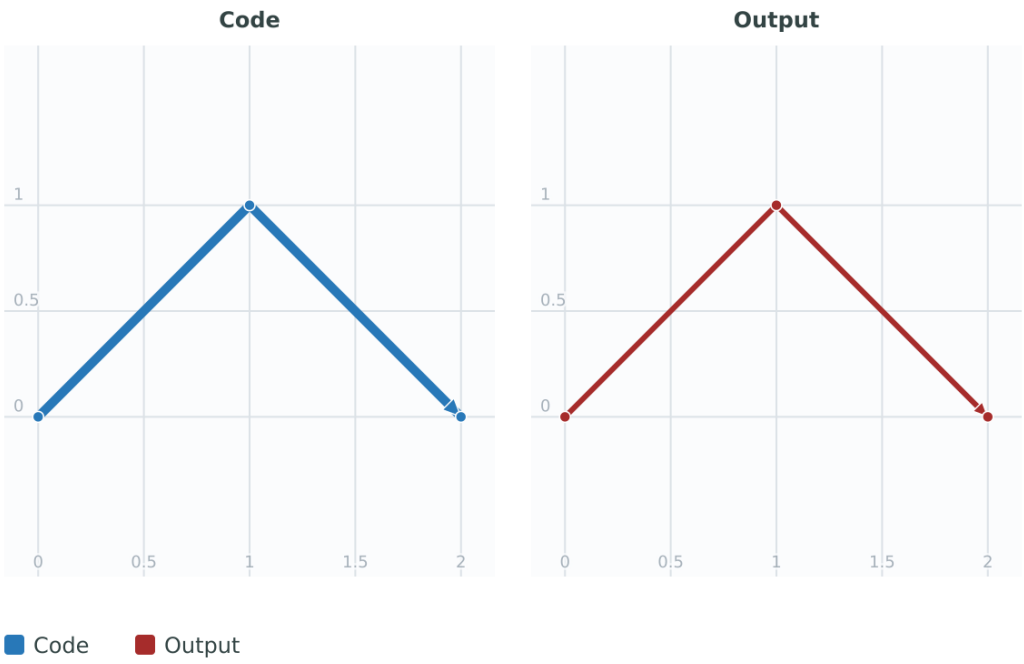
Code

```
SELECT ST_SetEndM('LINESTRING(0 0,1 1,2 0)'::geometry, 75);
```

Output

```
LINESTRING M (0 0 0,1 1 0,2 0 75)
```

Figure



Set end M value of a CircularString

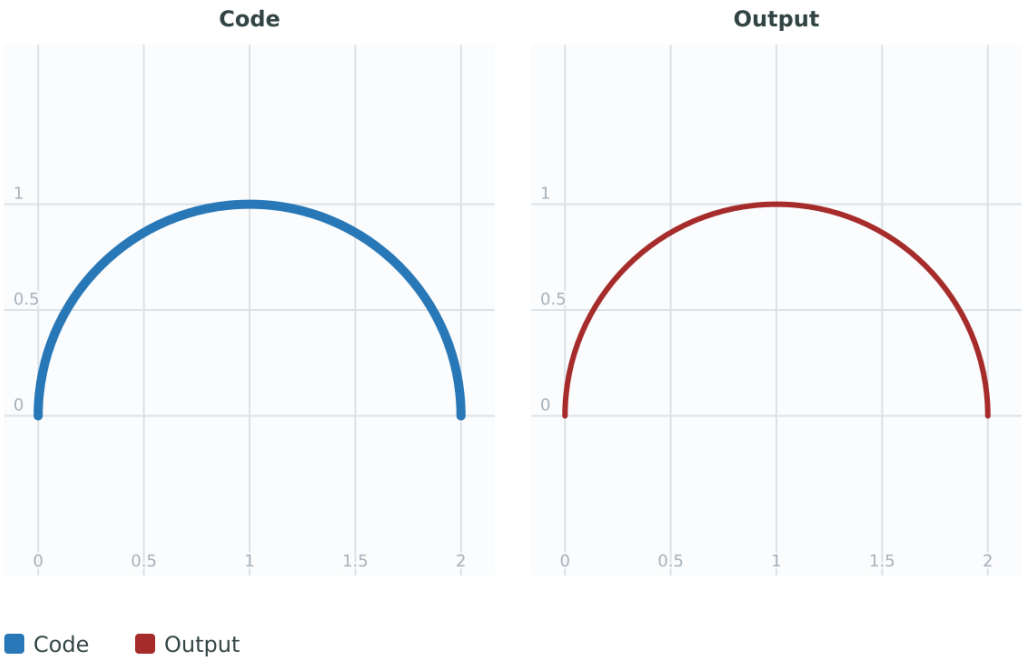
Code

```
SELECT ST_SetEndM('CIRCULARSTRING M(0 0 5,1 1 10,2 0 15)'::geometry, 88);
```

Output

```
CIRCULARSTRING M (0 0 5,1 1 10,2 0 88)
```

Figure



Set end M value of a NURBSCurve

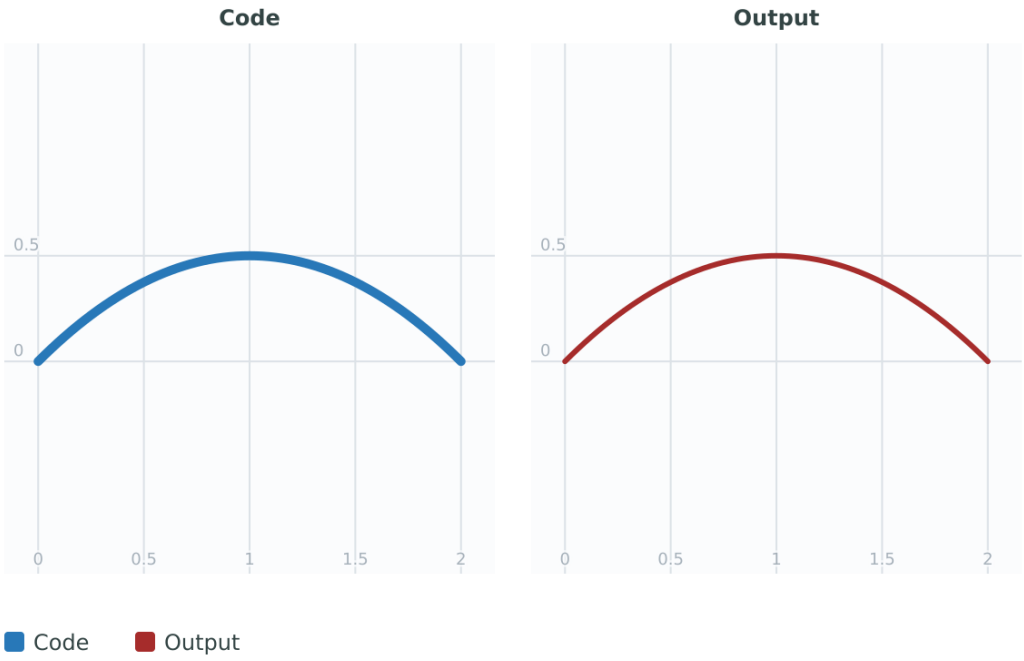
Code

```
SELECT ST_SetEndM('NURBSCURVE M(2, (0 0 100, 1 1 200, 2 0 300))'::geometry, 999);
```

Output

```
NURBSCURVE M (DEGREE 2,CONTROLPOINTS M (NURBSPOINT(WEIGHTEDPOINT M (0 0 100),WEIGHT 1), NURBSPOINT(WEIGHTEDPOINT M (1 ↩
     1 200),WEIGHT 1), NURBSPOINT(WEIGHTEDPOINT M (2 0 999),WEIGHT 1)),KNOTS (KNOT(0,3), KNOT(1,3)))
```

Figure



☒☒

ST_SetStartM, ST_EndM, ST_EndPoint

7.4.45 ST_Summary

ST_Summary — ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.

☒☒

text **ST_Summary**(geometry g);
text **ST_Summary**(geography g);

☒☒

☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.
☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒

- Z: Z □□□□□□□□.
- B: □□□□□□□□□□□□□□□□.
- G: □□□ (□□□) □□□.
- S: □□□□□□□□□□□□□□□□.

This method supports Circular Strings and Curves.

This function supports Polyhedral surfaces.

This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).

1.2.2 □□□□□□□□□□□□.

□□□□: 2.0.0 □□□□□□□□□□□□□□□□□□□□.

□□□□: 2.1.0 □□. □□□□□□□□□□□□□□□□□□□□□ S □□□□□□□□□□.

□□□□: 2.2.0 □□□□ TIN □□□□□ (curve) □□□□□□□□□□□.

□□

Code

```
SELECT ST_Summary('LINESTRING(0 0,1 1)'::geometry) as geom,
ST_Summary('POLYGON((0 0,1 1,1 2,1 1,0 0))'::geography) geog;
```

Output

```
geom             |          geog
------------------------------+--------------------------
 LineString[B] with 2 points | Polygon[BGS] with 1 rings
                              | ring 0 has 5 points
                              :
(1 row)
```

Code

```
SELECT ST_Summary('LINESTRING(0 0 1,1 1 1)'::geography) As geog_line,
ST_Summary('SRID=4326;POLYGON((0 0 1,1 1 2,1 2 3,1 1 1,0 0 1))'::geometry) As geom_poly;
```

Output

```
geog_line             |        geom_poly
------------------------------- +-------------------------
 LineString[ZBGS] with 2 points | Polygon[ZBS] with 1 rings
                                :    ring 0 has 5 points
                                :
(1 row)
```

□□

PostGIS_DropBBox, PostGIS_AddBBox, ST_Force3DM, ST_Force3DZ, ST_Force2D, geography
ST_IsValid, ST_IsValid, ST_IsValidReason, ST_IsValidDetail

7.4.46 ST_X

ST_X — Returns the X coordinate of a Point.

□□

float **ST_X**(geometry a_point);

□□

□□□□ X □□□□□□□□□. X □□□□□□□□ NULL □□□□□□□. □□□□□□□□□□□□□□□.

> **Note!** □□
> To get the minimum and maximum X value of geometry coordinates use the functions ST_XMin and ST_XMax.

This method implements the SQL/MM specification. SQL-MM 3: 6.1.3

This function supports 3d and will not drop the z-index.

□□

Code

```
SELECT ST_X('POINT(1 2 3 4)');
```

Output

```
1
```

Code

```
SELECT ST_Y(ST_Centroid('LINESTRING(1 2 3 4,1 1 1 1)'::geometry));
```

Output

```
1.5
```

□□

ST_Centroid, ST_GeomFromEWKT, ST_M, ST_XMax, ST_XMin, ST_Y, ST_Z

7.4.47 ST_Y

ST_Y — Returns the Y coordinate of a Point.

□□

float **ST_Y**(geometry a_point);

□□

□□□□ Y □□□□□□□□□. Y □□□□□□□□ NULL □□□□□□□. □□□□□□□□□□□□□□□.



□□
To get the minimum and maximum Y value of geometry coordinates use the functions ST_YMin and ST_YMax.

This method implements the OGC Simple Features Implementation Specification for SQL 1.1.

This method implements the SQL/MM specification. SQL-MM 3: 6.1.4

This function supports 3d and will not drop the z-index.

□□

Code

```
SELECT ST_Y('POINT(1 2 3 4)');
```

Output

```
2
```

Code

```
SELECT ST_Y(ST_Centroid('LINESTRING(1 2 3 4,1 1 1 1)'::geometry));
```

Output

```
1.5
```

□□

ST_Centroid, ST_GeomFromEWKT, ST_M, ST_X, ST_YMax, ST_YMin, ST_Z

7.4.48 ST_Z

ST_Z — Returns the Z coordinate of a Point.

□□

float **ST_Z**(geometry a_point);

□□

□□□□ Z □□□□□□□□□. Z □□□□□□□□ NULL □□□□□□□. □□□□□□□□□□□□□□□.



□□
To get the minimum and maximum Z value of geometry coordinates use the functions ST_ZMin and ST_ZMax.

This method implements the SQL/MM specification.

This function supports 3d and will not drop the z-index.

□□

Code

```
SELECT ST_Z('POINT(1 2 3 4)');
```

Output

```
3
```

□□

ST_GeomFromEWKT, ST_M, ST_X, ST_Y, ST_ZMax, ST_ZMin

7.4.49 ST_ZMFlag

ST_ZMFlag — ST_Geometry □□□□□□□□□□□□□□.

□□

smallint **ST_ZMFlag**(geometry geomA);

□□

ST_Geometry □□□□□□□□□□□□□□.
Values are: 0 = 2D, 1 = 3D-M, 2 = 3D-Z, 3 = 4D.

This function supports 3d and will not drop the z-index.

This method supports Circular Strings and Curves.

□□

Code

```
SELECT ST_ZMFlag('LINESTRING(1 2,3 4)');
```

Output

```
0
```

Code

```
SELECT ST_ZMFlag('LINESTRINGM(1 2 3, 3 4 3)');
```

Output

```
1
```

Code

```
SELECT ST_ZMFlag('CIRCULARSTRING(1 2 3,3 4 3,5 6 3)');
```

Output

```
2
```

Code

```
SELECT ST_ZMFlag('POINT(1 2 3 4)');
```

Output

```
3
```

□□

ST_CoordDim, ST_NDims, ST_Dimension

7.4.50 ST_HasZ

ST_HasZ — Checks if a geometry has a Z dimension.

□□

boolean **ST_HasZ**(geometry geom);

□□

Checks if the input geometry has a Z dimension and returns a boolean value. If the geometry has a Z dimension, it returns true; otherwise, it returns false.

Geometry objects with a Z dimension typically represent three-dimensional (3D) geometries, while those without it are two-dimensional (2D) geometries.

This function is useful for determining if a geometry has elevation or height information.

Availability: 3.5.0

This function supports 3d and will not drop the z-index.

This function supports M coordinates.

□□

Code

```
SELECT ST_HasZ('POINT(1 2 3)');
```

Output

```
t
```

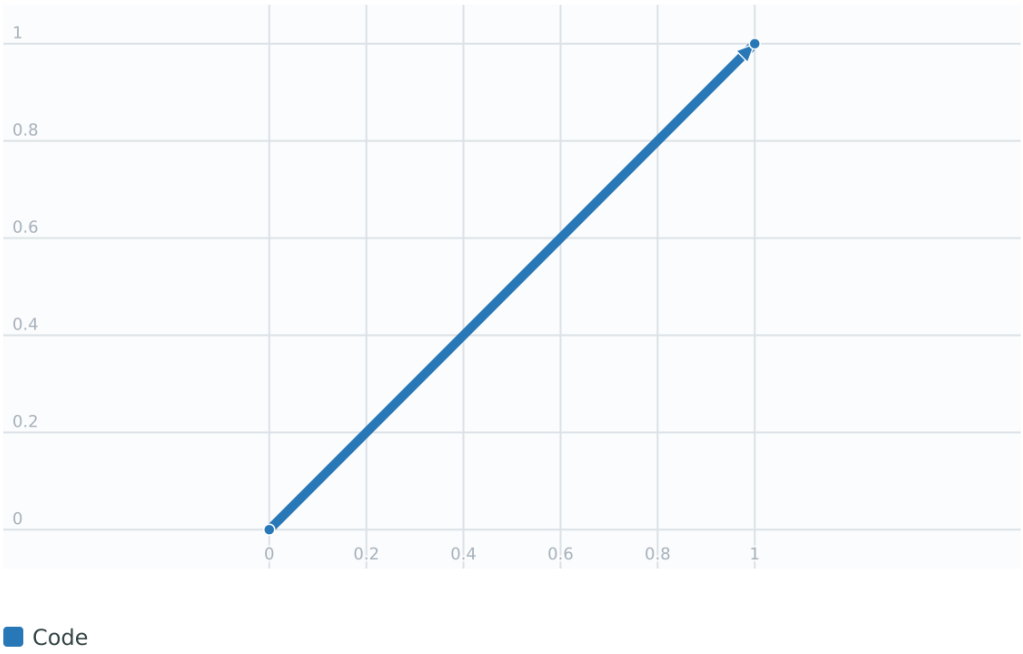
Code

```
SELECT ST_HasZ('LINESTRING(0 0,1 1)');
```

Output

```
f
```

Figure



□□

ST_ZMFlag
ST_HasM

7.4.51 ST_HasM

ST_HasM — Checks if a geometry has an M (measure) dimension.

□□

boolean **ST_HasM**(geometry geom);

□□

Checks if the input geometry has an M (measure) dimension and returns a boolean value. If the geometry has an M dimension, it returns true; otherwise, it returns false.

Geometry objects with an M dimension typically represent measurements or additional data associated with spatial features.

This function is useful for determining if a geometry includes measure information.

Availability: 3.5.0

This function supports 3d and will not drop the z-index.

This function supports M coordinates.

□□

Code

```
SELECT ST_HasM('POINTM(1 2 3)');
```

Output

```
t
```

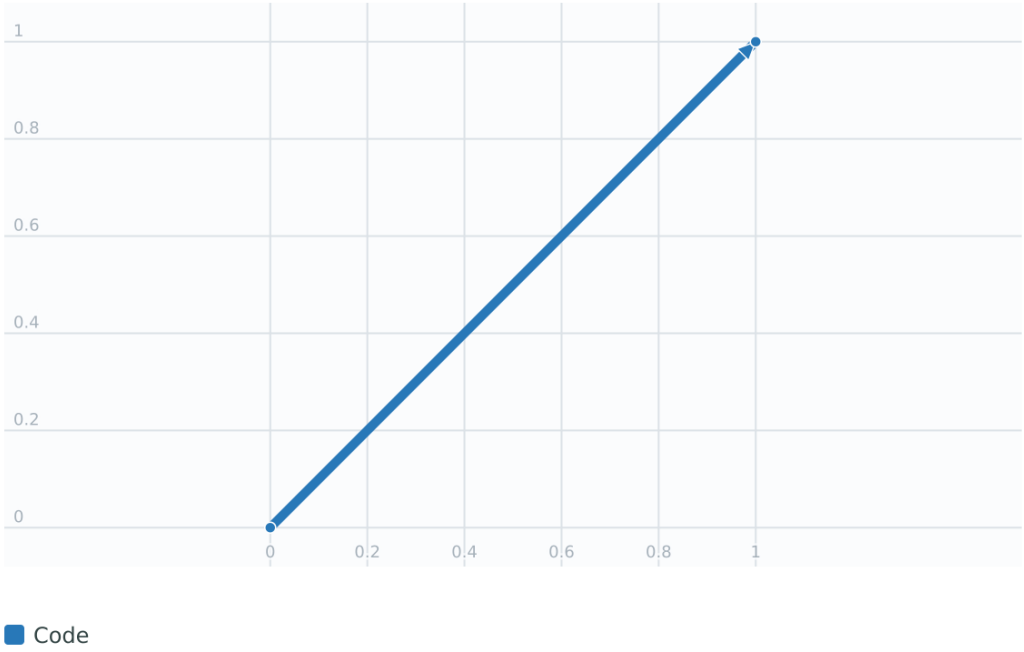
Code

```
SELECT ST_HasM('LINESTRING(0 0,1 1)');
```

Output

```
f
```

Figure



□□

ST_ZMFlag
ST_HasZ

7.4.52 ST_ControlPoints

ST_ControlPoints — Returns the control points of a NURBS curve as a MULTIPOINT geometry.

□□

geometry **ST_ControlPoints**(geometry nurbscurve);

□□

Extracts the control points from a NURBS curve and returns them as a MULTIPOINT geometry. Control points define the control polygon that influences the shape of the NURBS curve.

The function raises an error if the input is not a NURBS curve. Returns NULL if the input is NULL.

Availability: 3.7.0

□□

This example extracts control points from a NURBS curve.

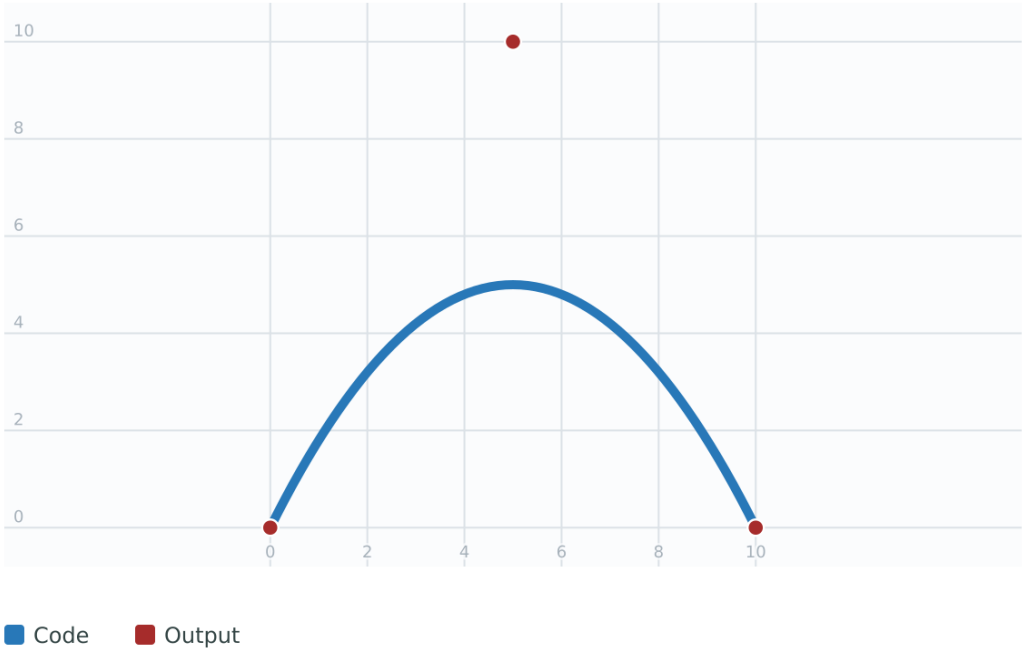
Code

```
SELECT ST_ControlPoints('NURBSCURVE(2, (0 0, 5 10, 10 0))'::geometry);
```

Output

```
MULTIPOINT((0 0),(5 10),(10 0))
```

Figure



This example counts the control points.

Code

```
SELECT ST_NPoints(ST_ControlPoints('NURBSCURVE(2, (0 0, 5 10, 10 0))'::geometry));
```

Output

```
3
```

□□

ST_MakeNurbsCurve, ST_NumControlPoints, ST_Weights

7.4.53 ST_Degree

ST_Degree — Returns the polynomial degree of a NURBS curve.

□□

integer **ST_Degree**(geometry nurbscurve);

□□

Returns the polynomial degree of a NURBS curve as an integer. The degree determines the curve's smoothness and flexibility. Higher degrees produce smoother curves but require more control points.

The function raises an error if the input is not a NURBS curve. Returns NULL if the input is NULL.

Availability: 3.7.0

□□

This example returns the degree of a NURBS curve.

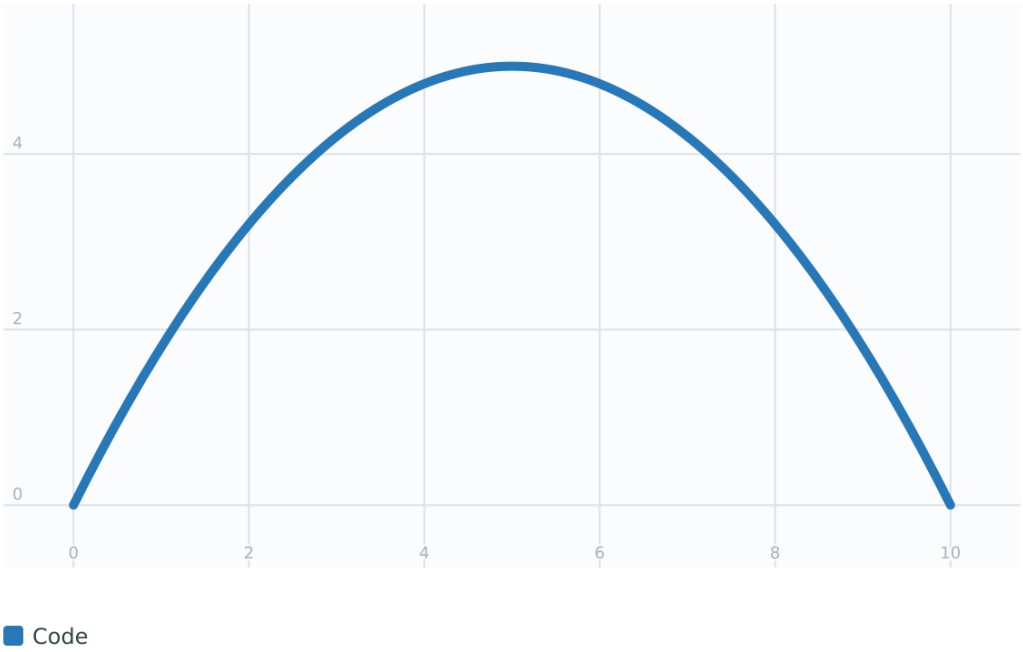
Code

```
SELECT ST_Degree('NURBSCURVE(2, (0 0, 5 10, 10 0))'::geometry);
```

Output

```
2
```

Figure



This example checks whether a NURBS curve is cubic.

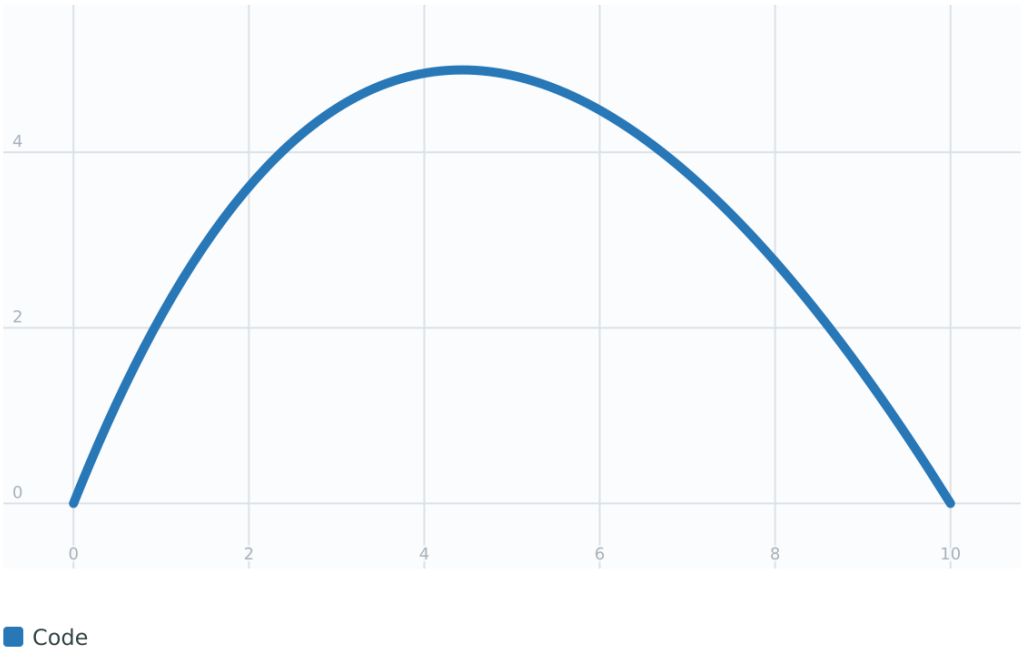
Code

```
SELECT ST_Degree('NURBSCURVE(3, (0 0, 2 5, 5 8, 10 0))'::geometry) = 3;
```

Output

```
t
```

Figure



□□

ST_MakeNurbsCurve, ST_NumControlPoints

7.4.54 ST_Weights

ST_Weights — Returns the weight array of a rational NURBS curve.

□□

float8[] **ST_Weights**(geometry nurbscurve);

□□

Returns the weight values of a rational NURBS curve as an array of double precision values. Weights control the influence of each control point on the curve shape. Higher weights pull the curve closer to the corresponding control point.

Returns NULL if the curve has no weights (non-rational B-spline) or if the input is NULL. Raises an error if the input is not a NURBS curve.

Availability: 3.7.0

□□

This example returns weights from a rational NURBS curve.

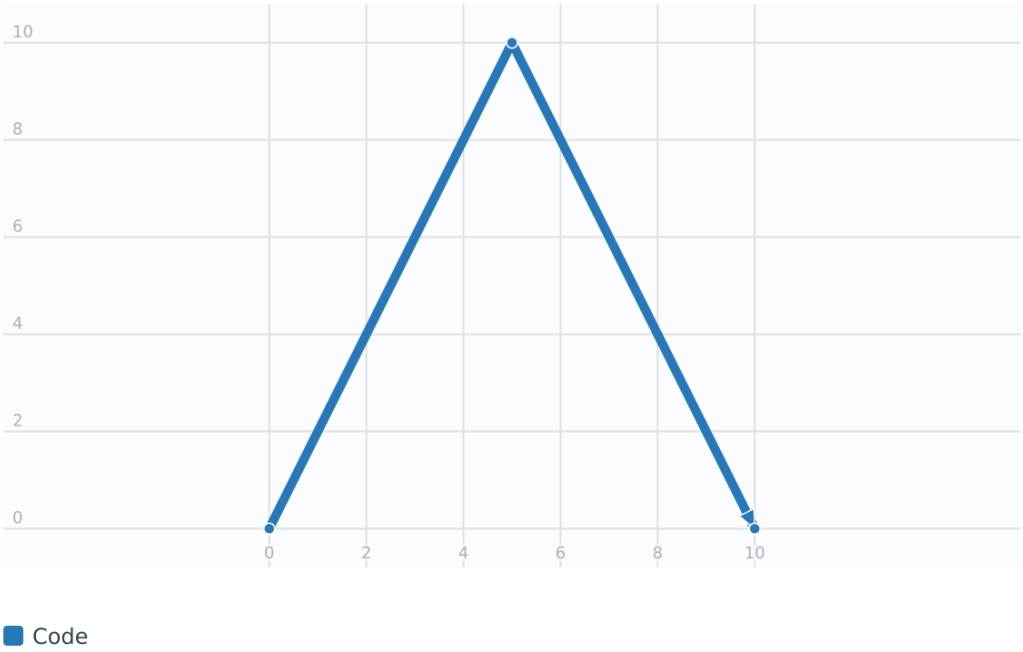
Code

```
SELECT ST_Weights(ST_MakeNurbsCurve(2, 'LINESTRING(0 0,5 10,10 0)'::geometry, ARRAY[1.0, ↩
    2.0, 1.0]));
```

Output

```
{1,2,1}
```

Figure



This example checks the weights of a non-rational curve.

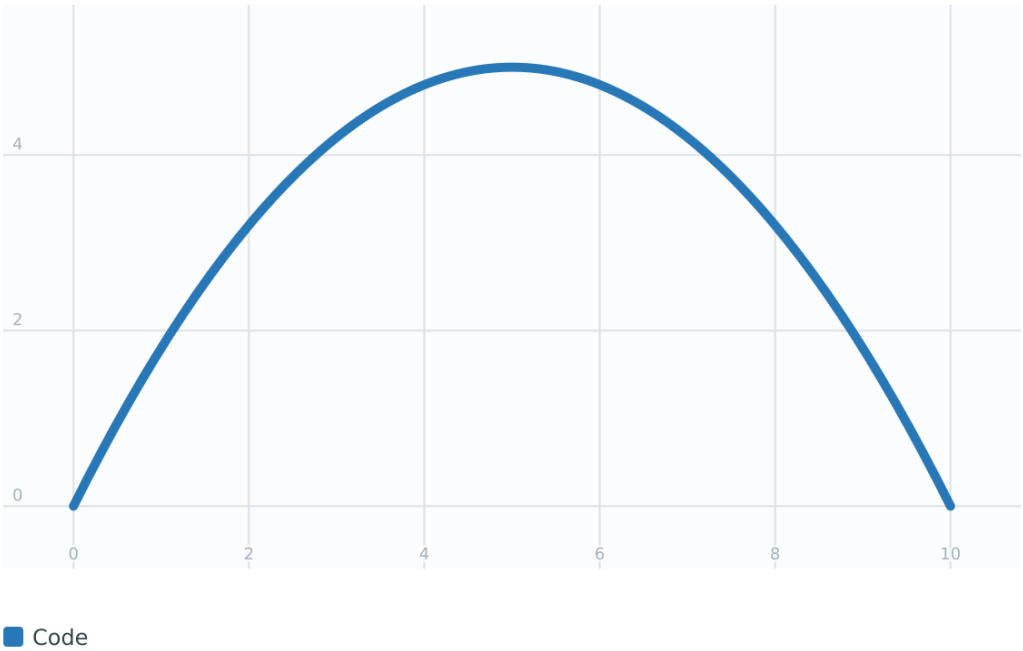
Code

```
SELECT ST_Weights('NURBSCURVE(2, (0 0, 5 10, 10 0))'::geometry);
```

Output

```
null
```

Figure



□□

ST_MakeNurbsCurve, ST_NurbsCurveIsRational, ST_ControlPoints

7.4.55 ST_Knots

ST_Knots — Returns the knot vector of a NURBS curve.

□□

float8[] **ST_Knots**(geometry nurbscurve);

□□

Returns the knot vector of a NURBS curve as an array of double precision values. The knot vector defines the parameter space and controls where and how the control points affect the curve. It must be a non-decreasing sequence with length equal to (number of control points + degree + 1).

Returns NULL if the curve uses the default uniform clamped knot vector or if the input is NULL. Raises an error if the input is not a NURBS curve.

Availability: 3.7.0

□□

This example returns knots from a NURBS curve with a custom knot vector.

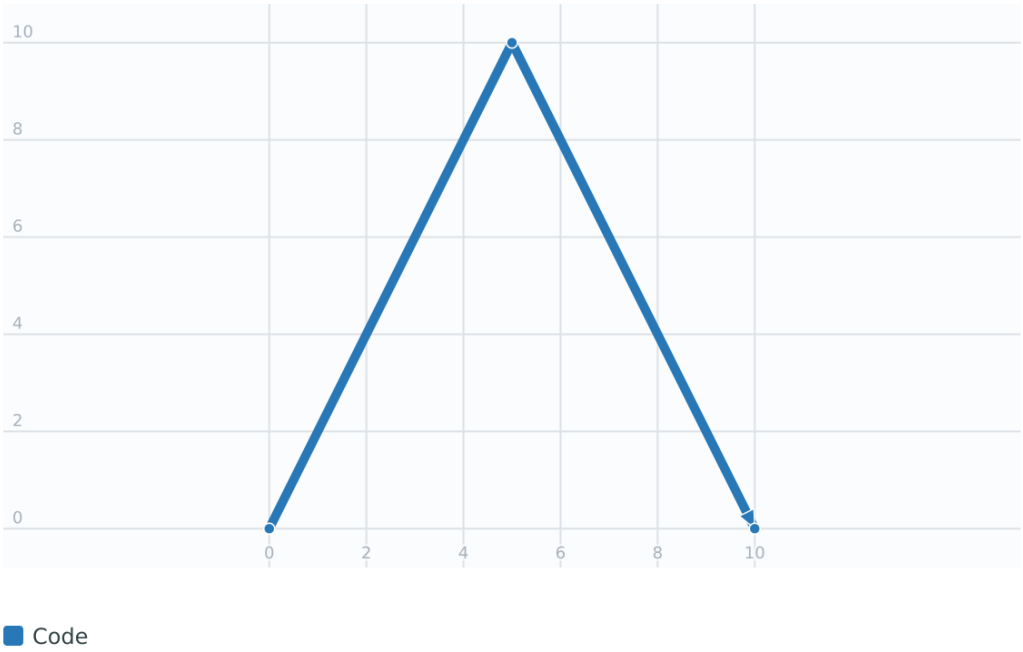
Code

```
SELECT ST_Knots(ST_MakeNurbsCurve(
        2,
        'LINESTRING(0 0,5 10,10 0)'::geometry,
        ARRAY[1.0, 2.0, 1.0],
        ARRAY[0, 0, 0, 1, 1, 1]));
```

Output

```
{0,0,0,1,1,1}
```

Figure



This example checks the knots of a curve using the default knot vector.

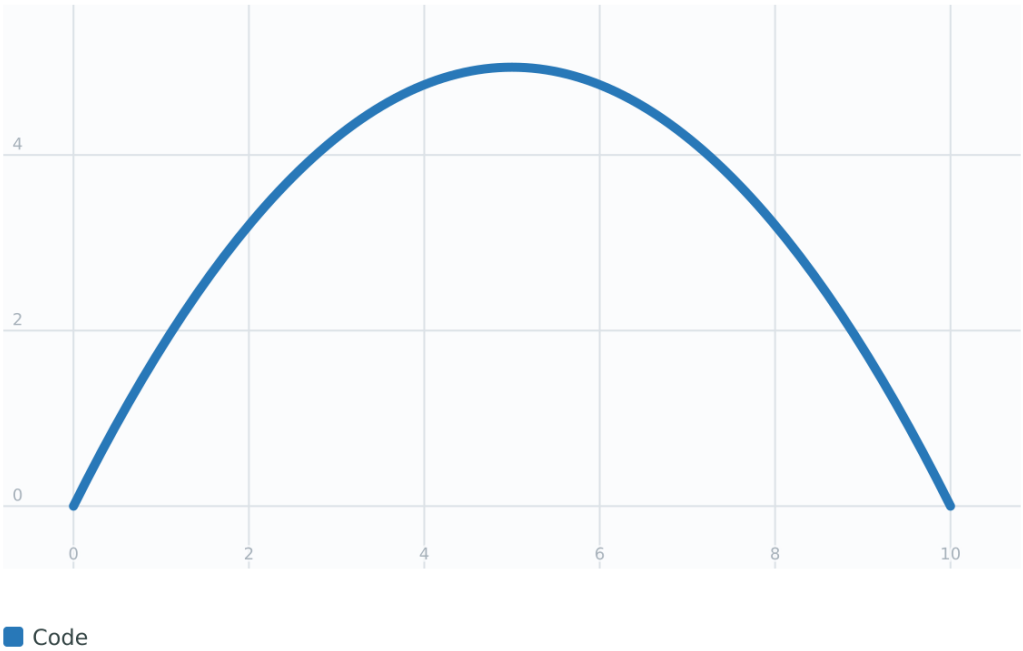
Code

```
SELECT ST_Knots('NURBSCURVE(2, (0 0, 5 10, 10 0))'::geometry);
```

Output

```
null
```

Figure



□□

ST_MakeNurbsCurve, ST_Degree, ST_NumControlPoints

7.4.56 ST_NumControlPoints

ST_NumControlPoints — Returns the number of control points in a NURBS curve.

□□

integer **ST_NumControlPoints**(geometry nurbscurve);

□□

Returns the number of control points in a NURBS curve. The number of control points must be at least (degree + 1) for a valid NURBS curve.

Raises an error if the input is not a NURBS curve. Returns NULL if the input is NULL.

Availability: 3.7.0

□□

This example returns the number of control points.

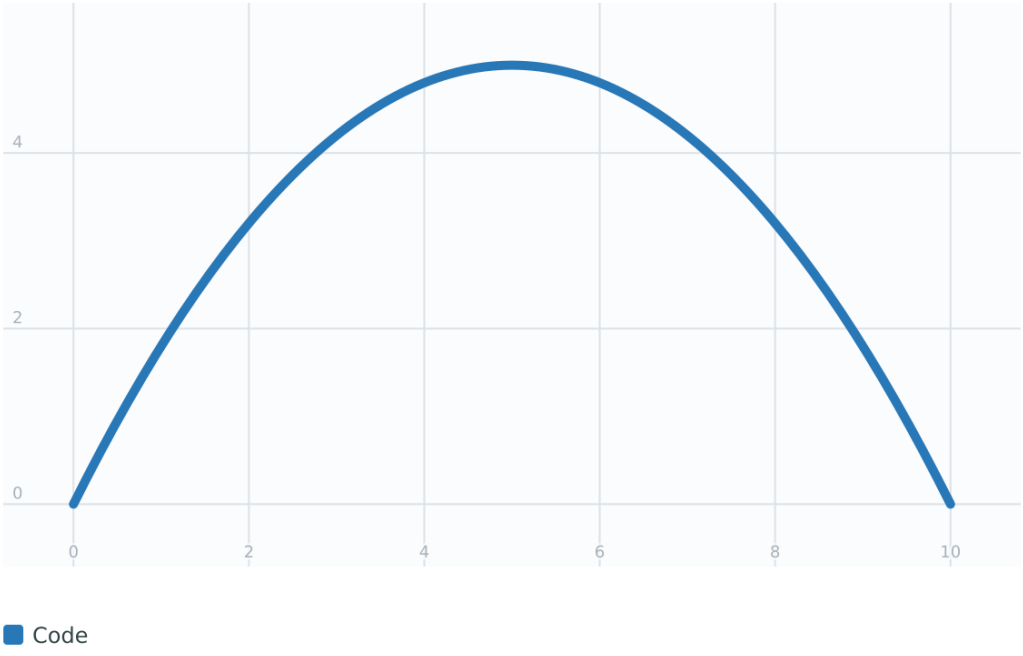
Code

```
SELECT ST_NumControlPoints('NURBSCURVE(2, (0 0, 5 10, 10 0))'::geometry);
```

Output

```
3
```

Figure



This example verifies that a curve has enough control points for its degree.

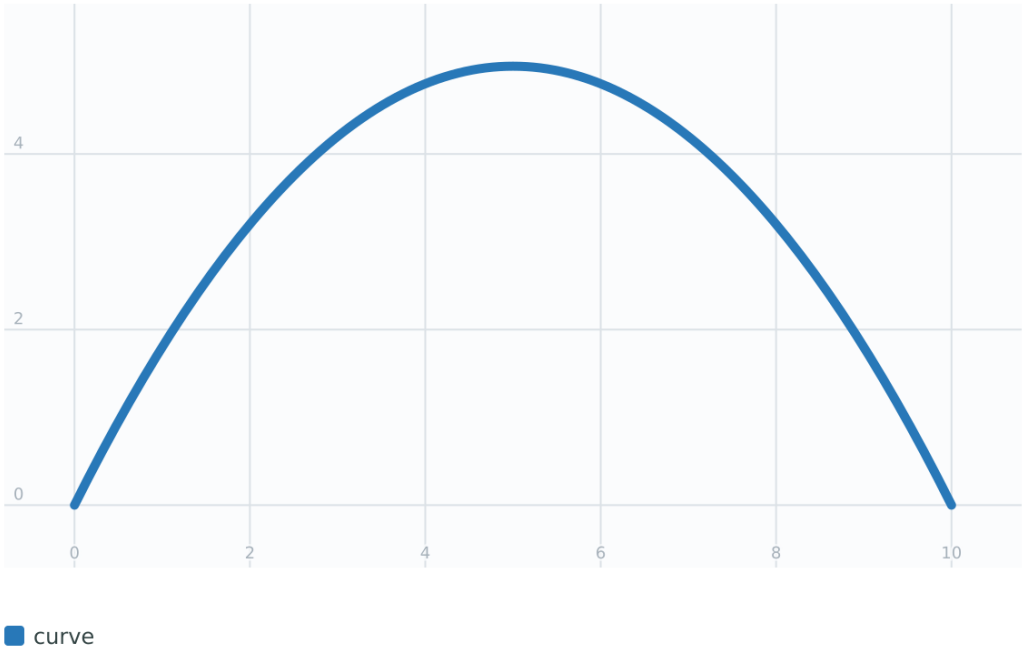
Code

```
SELECT ST_NumControlPoints(curve)
>= ST_Degree(curve) + 1 AS valid
FROM (SELECT 'NURBSCURVE(2, (0 0, 5 10, 10 0))'::geometry AS curve) AS t;
```

Output

```
t
```

Figure



☒☒

ST_MakeNurbsCurve, ST_ControlPoints, ST_Degree

7.4.57 ST_NurbsCurveIsRational

ST_NurbsCurveIsRational — Checks if a NURBS curve is rational (has weights).

☒☒

boolean **ST_NurbsCurveIsRational**(geometry nurbscurve);

☒☒

Returns TRUE if the NURBS curve has weights assigned to its control points (rational NURBS), FALSE if it uses uniform weights (non-rational B-spline).

Rational NURBS curves provide more flexibility in curve shape control through variable weights. Non-rational B-splines use uniform weights and are a special case of NURBS curves.

Raises an error if the input is not a NURBS curve. Returns NULL if the input is NULL.

Availability: 3.7.0

□□

This example checks whether a curve is rational.

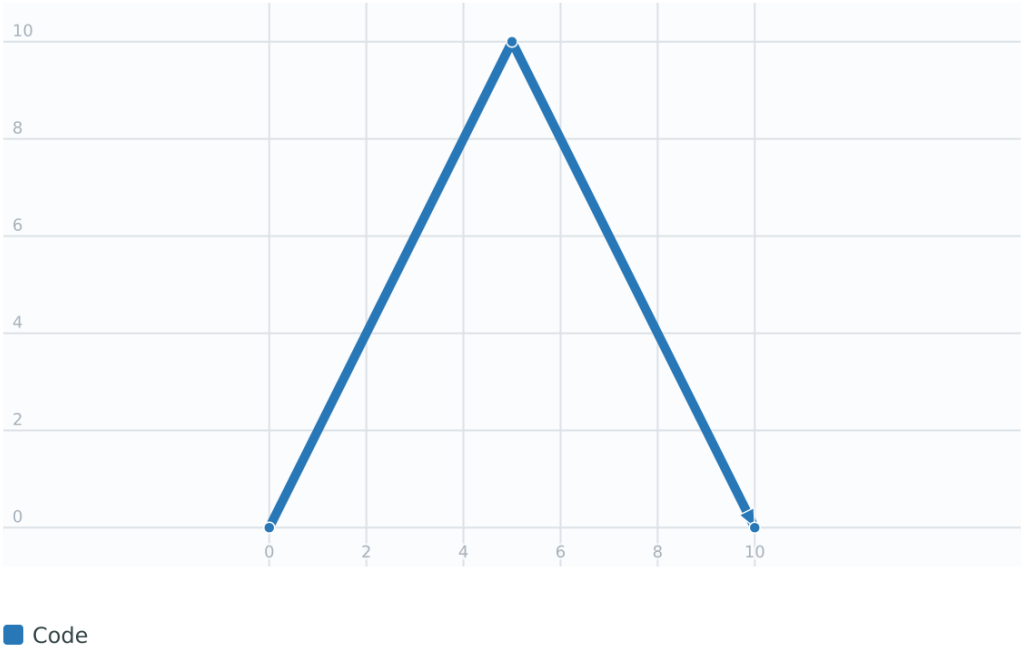
Code

```
SELECT ST_NurbsCurveIsRational(ST_MakeNurbsCurve(
        2,
        'LINESTRING(0 0,5 10,10 0)'::geometry,
        ARRAY[1.0, 2.0, 1.0]));
```

Output

```
t
```

Figure



This example checks a non-rational curve.

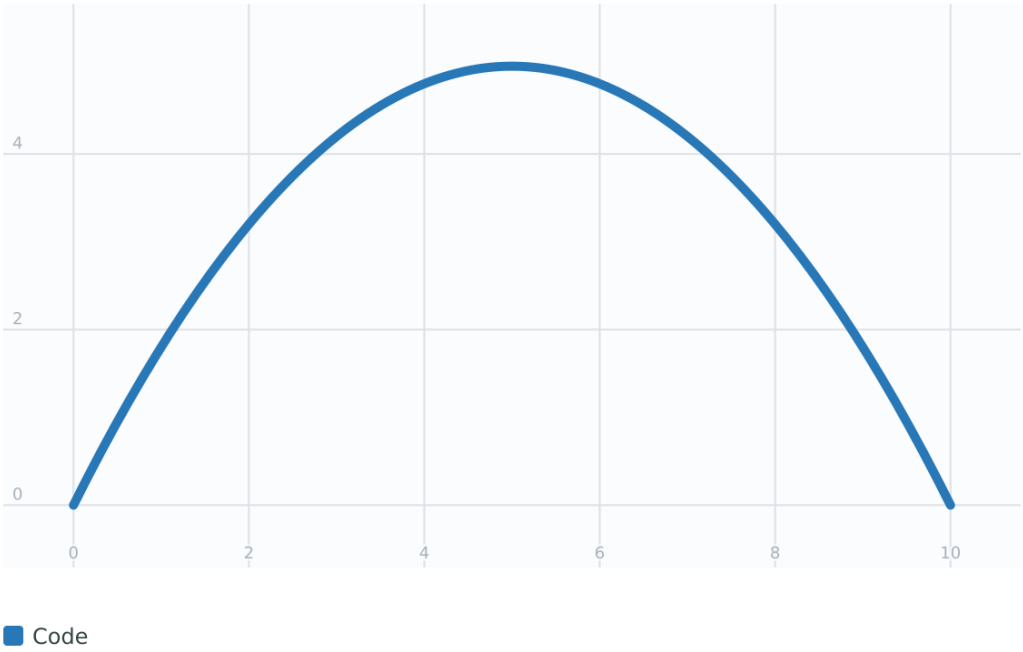
Code

```
SELECT ST_NurbsCurveIsRational('NURBSCURVE(2, (0 0, 5 10, 10 0))'::geometry);
```

Output

```
f
```

Figure



☒☒

ST_MakeNurbsCurve, ST_Weights, ST_IsNurbsCurve

7.4.58 ST_IsNurbsCurve

ST_IsNurbsCurve — Checks if a geometry is a NURBS curve.

☒☒

boolean **ST_IsNurbsCurve**(geometry geom);

☒☒

Returns TRUE if the geometry is a NURBS curve, FALSE otherwise. This function can be used to test geometry types before applying NURBS-specific functions.

Returns NULL if the input is NULL.

Availability: 3.7.0

☒☒

This example checks whether a geometry is a NURBS curve.

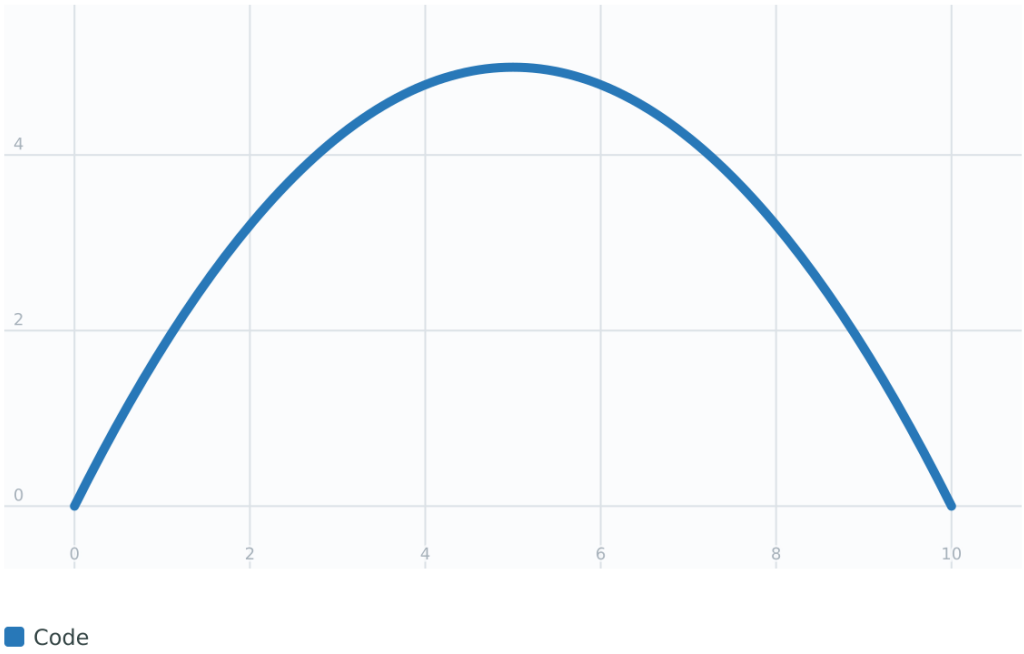
Code

```
SELECT ST_IsNurbsCurve('NURBSCURVE(2, (0 0, 5 10, 10 0))'::geometry);
```

Output

```
t
```

Figure



This example tests a regular LineString.

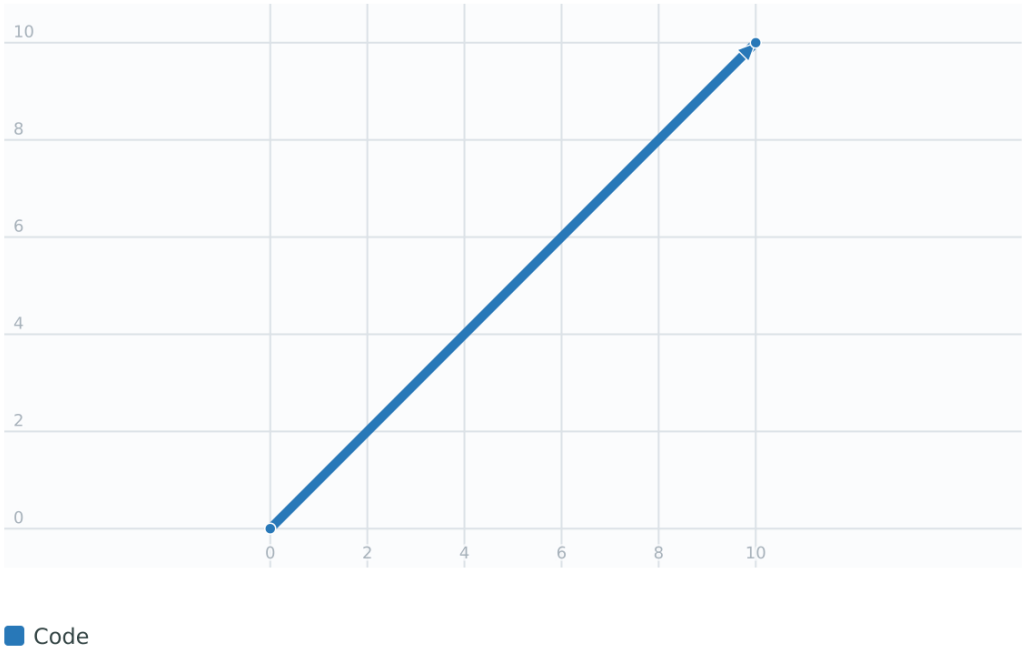
Code

```
SELECT ST_IsNurbsCurve('LINESTRING(0 0,10 10)'::geometry);
```

Output

```
f
```

Figure



□□

ST_MakeNurbsCurve, ST_GeometryType, ST_NurbsCurveIsRational

7.4.59 ST_Evaluate

ST_Evaluate — Evaluates a NURBS curve at a specific parameter value and returns the resulting point.

□□

geometry **ST_Evaluate**(geometry nurbscurve, float8 parameter);

□□

Evaluates a NURBS curve at the specified parameter value and returns a POINT geometry representing the position on the curve. The parameter typically ranges from 0.0 (start of curve) to 1.0 (end of curve), though the actual range depends on the knot vector.

Raises an error if the input is not a NURBS curve or if evaluation fails. Returns NULL if either input is NULL.

Availability: 3.7.0

□□

This example returns the midpoint of a NURBS curve.

Code

```
SELECT ST_Evaluate('NURBSCURVE(2, (0 0, 5 10, 10 0))'::geometry, 0.5);
```

This example samples multiple points along a curve.

Code

```
SELECT t, ST_Evaluate(curve, t) AS point
FROM (SELECT 'NURBSCURVE(2, (0 0, 5 10, 10 0))'::geometry AS curve) AS c,
         generate_series(0, 1, 0.25) AS t;
```

☒☒

ST_MakeNurbsCurve, ST_NurbsToLineString

7.4.60 ST_NurbsToLineString

ST_NurbsToLineString — Converts a NURBS curve to a LINESTRING by uniform sampling.

☒☒

geometry **ST_NurbsToLineString**(geometry nurbscurve, integer num_segments=32);

☒☒

Converts a NURBS curve to a LINESTRING geometry by creating a piecewise linear approximation. The curve is sampled at uniformly distributed parameter values and the resulting points are connected with straight line segments.

The optional `num_segments` parameter specifies the number of line segments in the output (default: 32, range: 2-10000). Higher values produce smoother approximations but increase geometry complexity.

Raises an error if the input is not a NURBS curve. Returns NULL if the input is NULL.

Availability: 3.7.0

☒☒

This example converts a NURBS curve to a LineString using the default quality.

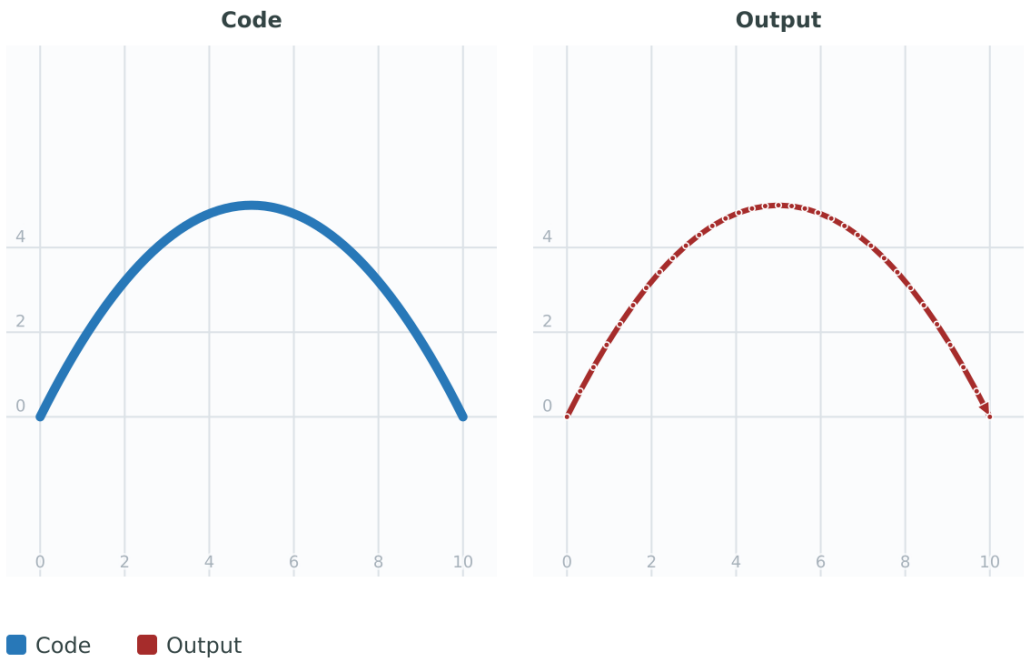
Code

```
SELECT ST_NurbsToLineString('NURBSCURVE(2, (0 0, 5 10, 10 0))'::geometry);
```

Output

```
LINESTRING(0 0,0.3125 0.60546875,0.625 1.171875,0.9375 1.69921875,1.25 2.1875,1.5625 2.63671875,1.875 3.046875,2.1875 ↩
    3.41796875,2.5 3.75,2.8125 4.04296875,3.125 4.296875,3.4375 4.51171875,3.75 4.6875,4.0625 4.82421875,4.375 ↩
    4.921875,4.6875 4.98046875,5 5,5.3125 4.98046875,5.625 4.921875,5.9375 4.82421875,6.25 4.6875,6.5625 ↩
    4.51171875,6.875 4.296875,7.1875 4.04296875,7.5 3.75,7.8125 3.41796875,8.125 3.046875,8.4375 2.63671875,8.75 ↩
    2.1875,9.0625 1.69921875,9.375 1.171875,9.6875 0.60546875,10 0)
```

Figure

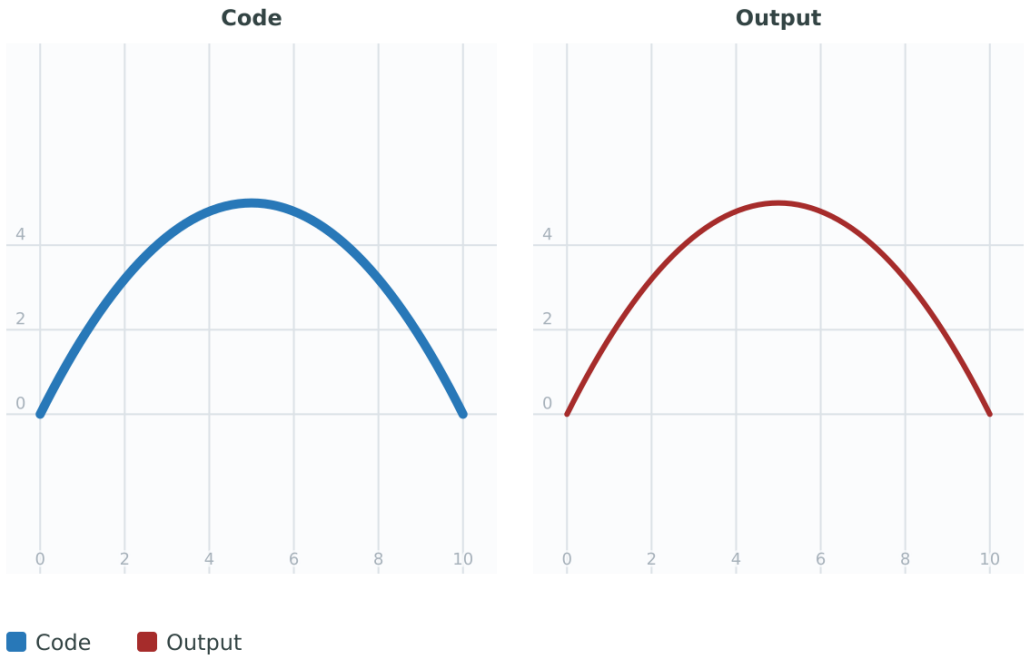


This example creates a high-quality approximation with 64 segments.

Code

```
SELECT ST_NurbsToLineString('NURBSCURVE(2, (0 0, 5 10, 10 0))'::geometry, 64);
```

Figure



This example creates a lower-quality, faster approximation with 16 segments.

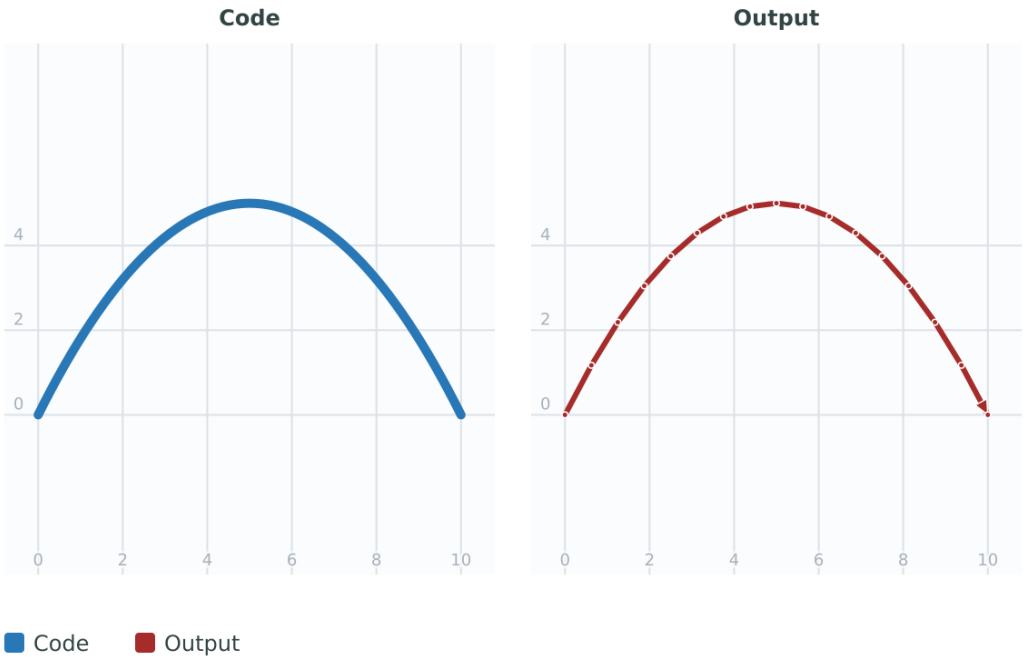
Code

```
SELECT ST_NurbsToLineString('NURBSCURVE(2, (0 0, 5 10, 10 0))'::geometry, 16);
```

Output

```
LINESTRING(0 0,0.625 1.171875,1.25 2.1875,1.875 3.046875,2.5 3.75,3.125 4.296875,3.75 4.6875,4.375 4.921875,5 5,5.625 ↩
    4.921875,6.25 4.6875,6.875 4.296875,7.5 3.75,8.125 3.046875,8.75 2.1875,9.375 1.171875,10 0)
```

Figure



☒☒

ST_MakeNurbsCurve, ST_Evaluate, ST_Segmentize

7.5 ☒☒☒☒☒ (editor)

7.5.1 ST_AddPoint

ST_AddPoint — ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.

☒☒

geometry **ST_AddPoint**(geometry linestring, geometry point);
geometry **ST_AddPoint**(geometry linestring, geometry point, integer position = -1);

□□

Adds a point to a LineString before the index *position* (using a 0-based index). If the *position* parameter is omitted or is -1 the point is appended to the end of the LineString.

1.1.0 □□□□□□□□□□□□.

This function supports 3d and will not drop the z-index.

□□

Add a point to the end of a 3D line

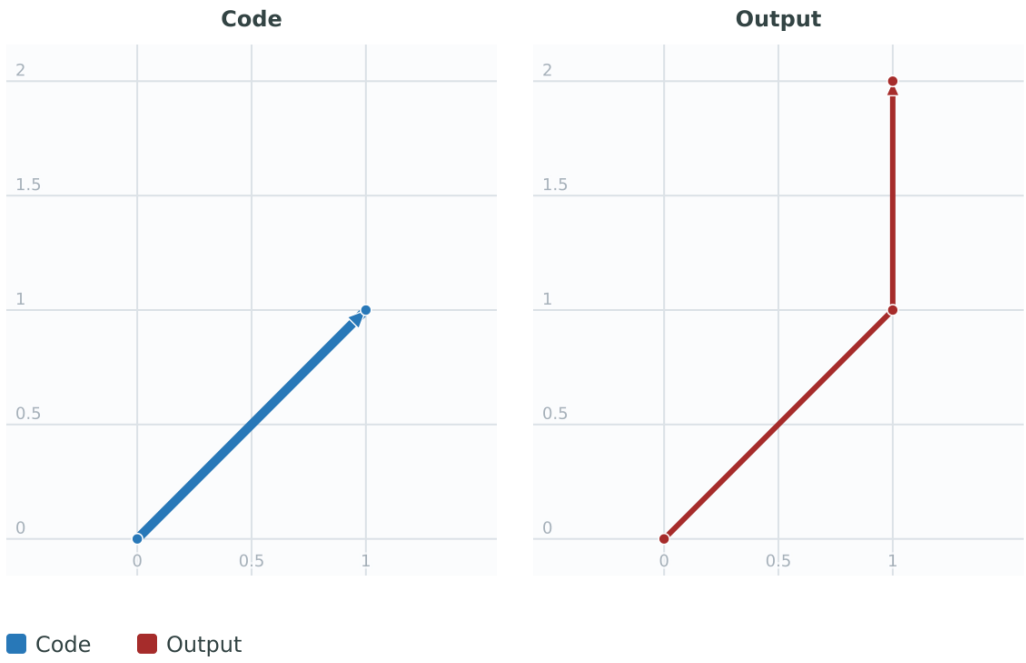
Code

```
SELECT ST_AddPoint('LINESTRING(0 0 1,1 1 1)', ST_MakePoint(1, 2, 3));
```

Output

```
LINESTRING(0 0 1,1 1 1,1 2 3)
```

Figure



Guarantee all lines in a table are closed by adding the start point of each line to the end of the line only for those that are not closed.

Code

```
UPDATE sometable
SET geom = ST_AddPoint(geom, ST_StartPoint(geom))
FROM sometable
WHERE ST_IsClosed(geom) = false;
```

□□

ST_RemovePoint, ST_SetPoint

7.5.2 ST_CollectionExtract

ST_CollectionExtract — Given a geometry collection, returns a multi-geometry containing only elements of a specified type.

□□

geometry **ST_CollectionExtract**(geometry collection);

geometry **ST_CollectionExtract**(geometry collection, integer type);

□□

Given a geometry collection, returns a homogeneous multi-geometry.

If the *type* is not specified, returns a multi-geometry containing only geometries of the highest dimension. So polygons are preferred over lines, which are preferred over points.

If the *type* is specified, returns a multi-geometry containing only that type. If there are no sub-geometries of the right type, an EMPTY geometry is returned. Only points, lines and polygons are supported. The type numbers are:

- 1 == POINT
- 2 == LINESTRING
- 3 == POLYGON

For atomic geometry inputs, the geometry is returned unchanged if the input type matches the requested type. Otherwise, the result is an EMPTY geometry of the specified type. If required, these can be converted to multi-geometries using ST_Multi.



□□
MultiPolygon results are not checked for validity. If the polygon components are adjacent or overlapping the result will be invalid. (For example, this can occur when applying this function to an ST_Split result.) This situation can be checked with ST_IsValid and repaired with ST_MakeValid.

1.5.0 □□□□□□□□□□□□.



□□
Prior to 1.5.3 this function returned atomic inputs unchanged, no matter type. In 1.5.3 non-matching single geometries returned a NULL result. In 2.0.0 non-matching single geometries return an EMPTY result of the requested type.

□□

Extract highest-dimension type:

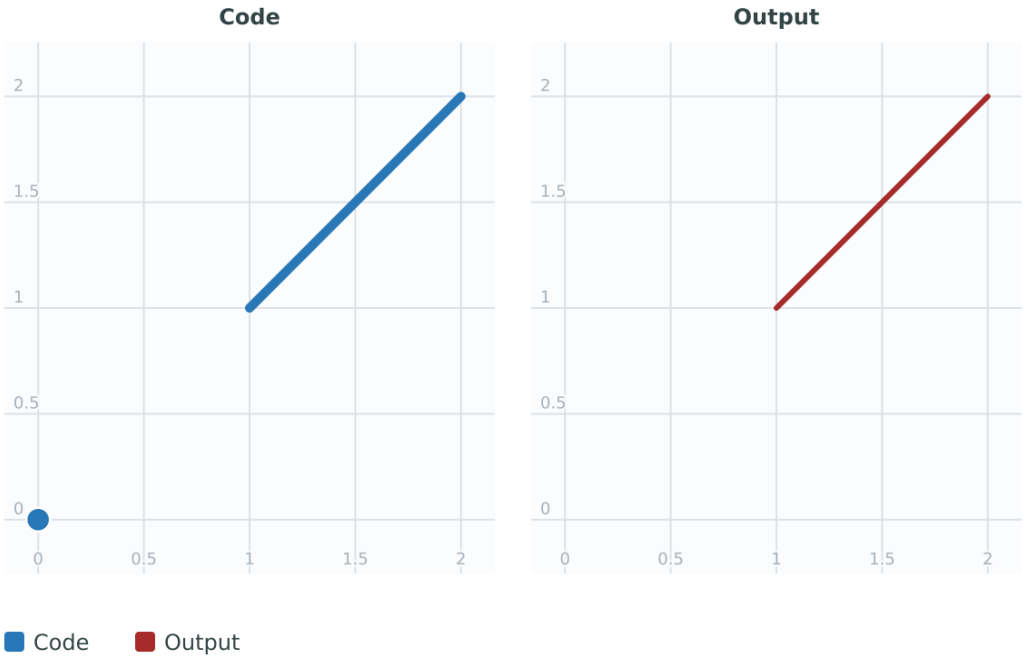
Code

```
SELECT ST_CollectionExtract('GEOMETRYCOLLECTION( POINT(0 0), LINESTRING(1 1,2 2) )');
```

Output

```
MULTILINESTRING((1 1, 2 2))
```

Figure



Extract points (type 1 == POINT):

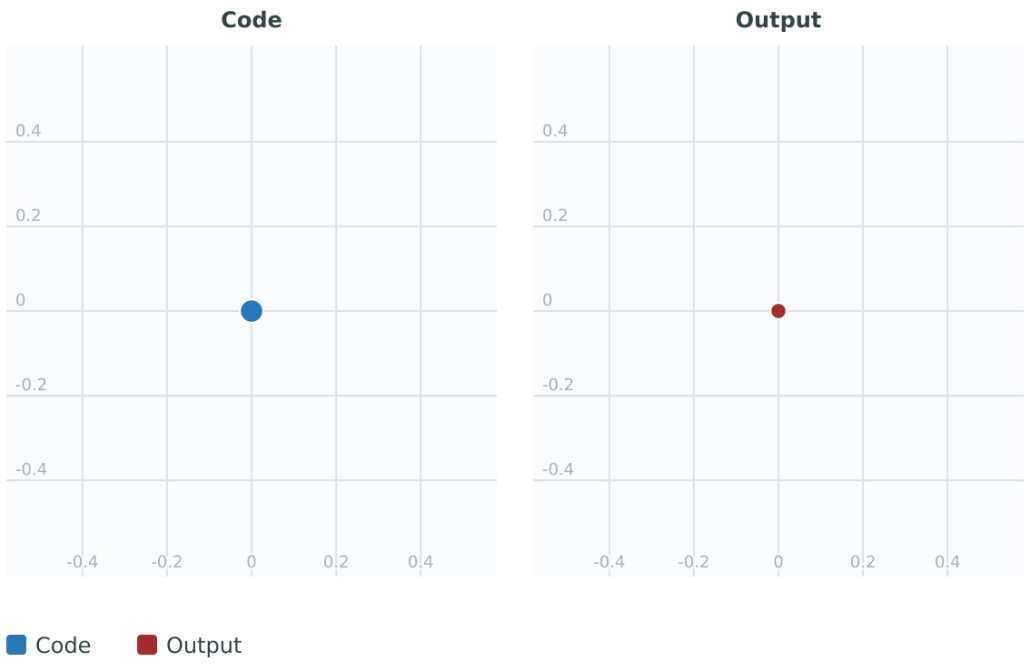
Code

```
SELECT ST_CollectionExtract('GEOMETRYCOLLECTION(GEOMETRYCOLLECTION(POINT(0 0)))',
        1);
```

Output

```
MULTIPOINT((0 0))
```

Figure



Extract lines (type 2 == LINESTRING):

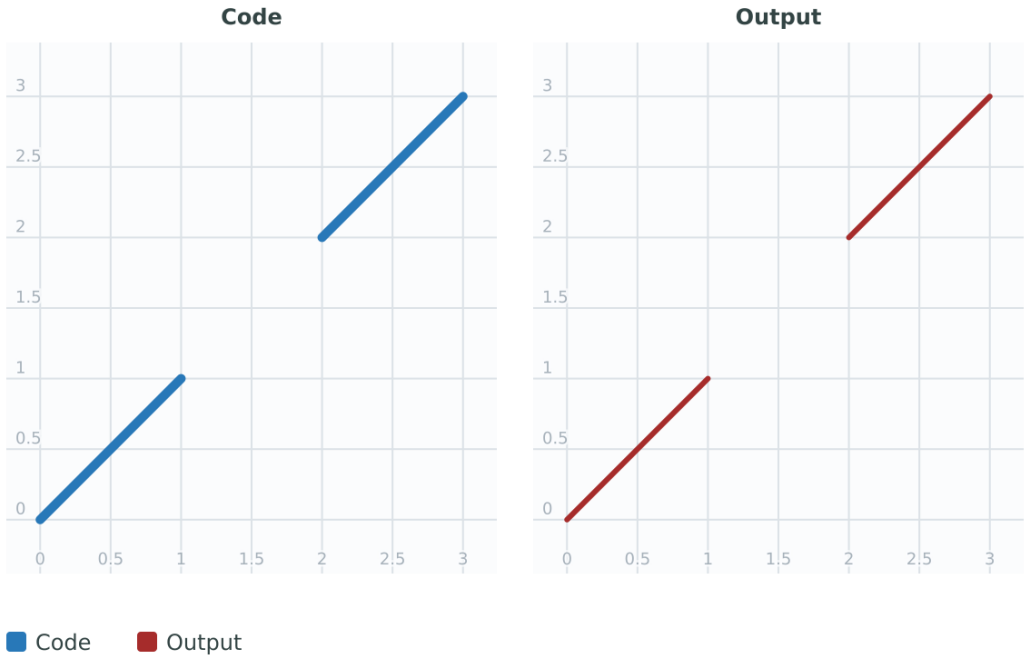
Code

```
SELECT ST_CollectionExtract('GEOMETRYCOLLECTION(GEOMETRYCOLLECTION(LINESTRING(0 0,1 1)),  ↩
    LINESTRING(2 2,3 3))',
        2);
```

Output

```
MULTILINESTRING((0 0, 1 1), (2 2, 3 3))
```

Figure



□□

ST_CollectionHomogenize, ST_Multi, ST_IsValid, ST_MakeValid

7.5.3 ST_CollectionHomogenize

ST_CollectionHomogenize — Returns the simplest representation of a geometry collection.

□□

geometry **ST_CollectionHomogenize**(geometry collection);

□□

□□□□□□□□□□□□□□□□” □□□□□□” □□□□□□□□□□.

- Homogeneous (uniform) collections are returned as the appropriate multi-geometry.
- Heterogeneous (mixed) collections are flattened into a single GeometryCollection.
- Collections containing a single atomic element are returned as that element.
- Atomic geometries are returned unchanged. If required, these can be converted to a multi-geometry using ST_Multi.



□□
This function does not ensure that the result is valid. In particular, a collection containing adjacent or overlapping Polygons will create an invalid MultiPolygon. This situation can be checked with ST_IsValid and repaired with ST_MakeValid.

2.0.0 □□□□□□□□□□□□.

□□

Single-element collection converted to an atomic geometry

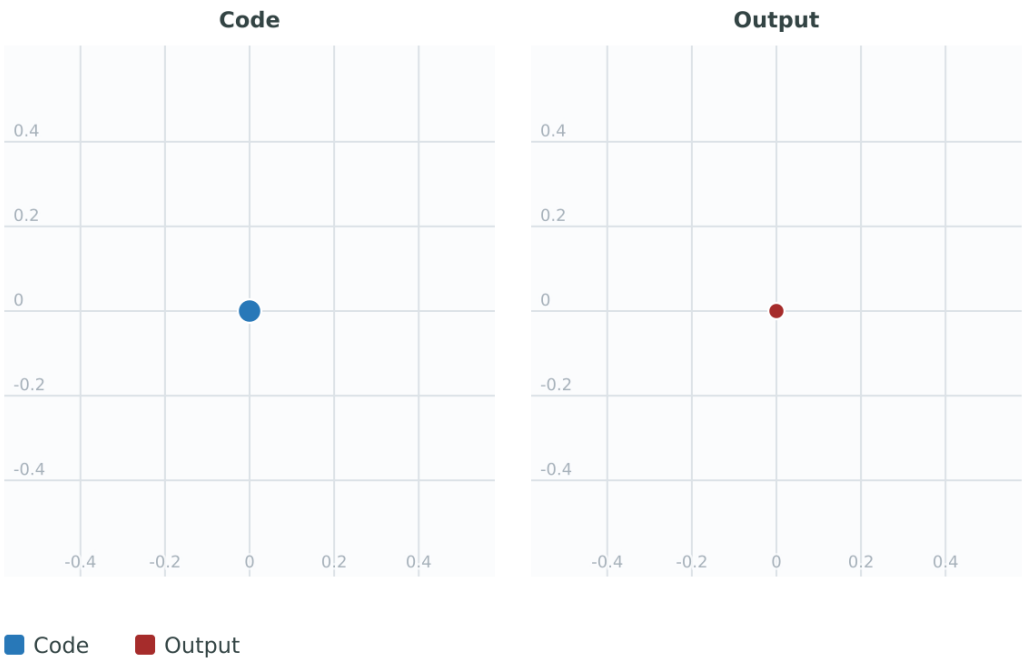
Code

```
SELECT ST_CollectionHomogenize('GEOMETRYCOLLECTION(POINT(0 0))');
```

Output

```
POINT(0 0)
```

Figure



Nested single-element collection converted to an atomic geometry:

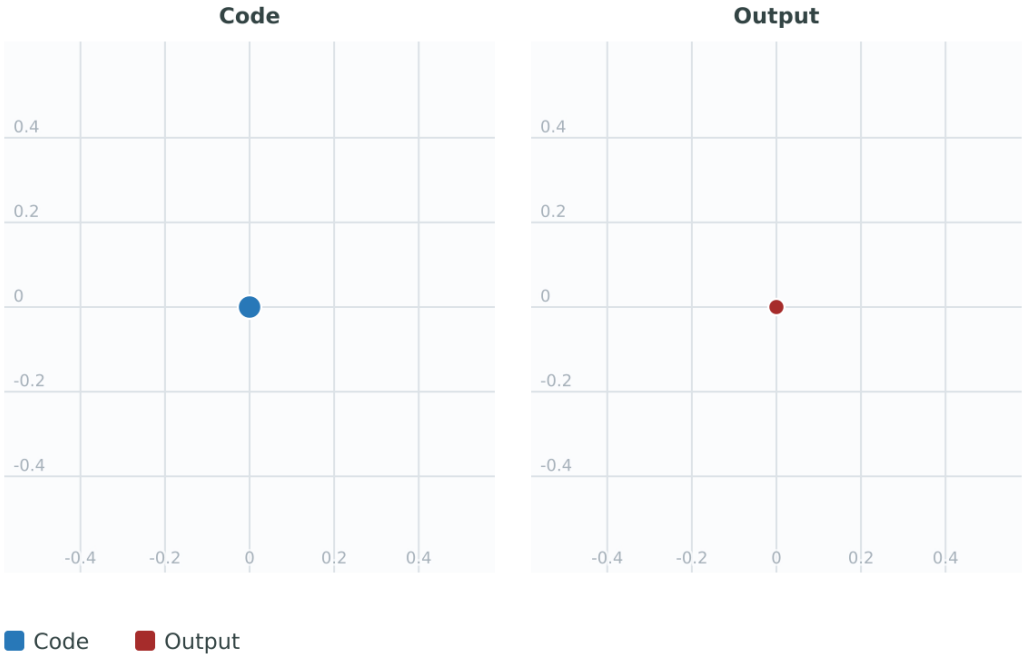
Code

```
SELECT ST_CollectionHomogenize('GEOMETRYCOLLECTION(MULTIPOINT((0 0)))');
```

Output

```
POINT(0 0)
```

Figure



Collection converted to a multi-geometry:

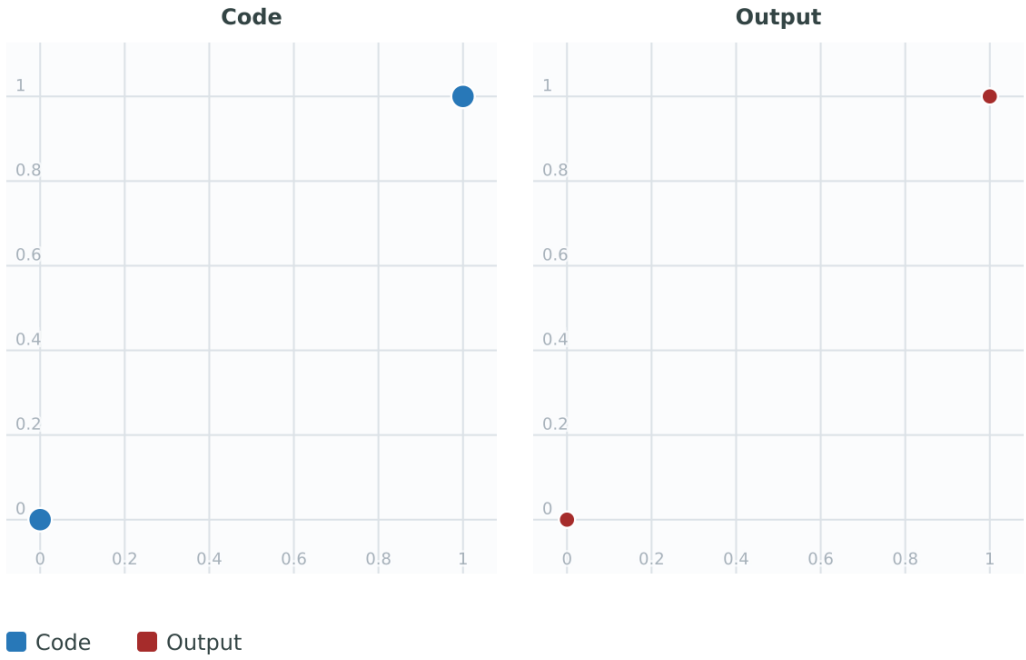
Code

```
SELECT ST_CollectionHomogenize('GEOMETRYCOLLECTION(POINT(0 0), POINT(1 1))');
```

Output

```
MULTIPOINT((0 0),(1 1))
```

Figure



Nested heterogeneous collection flattened to a GeometryCollection:

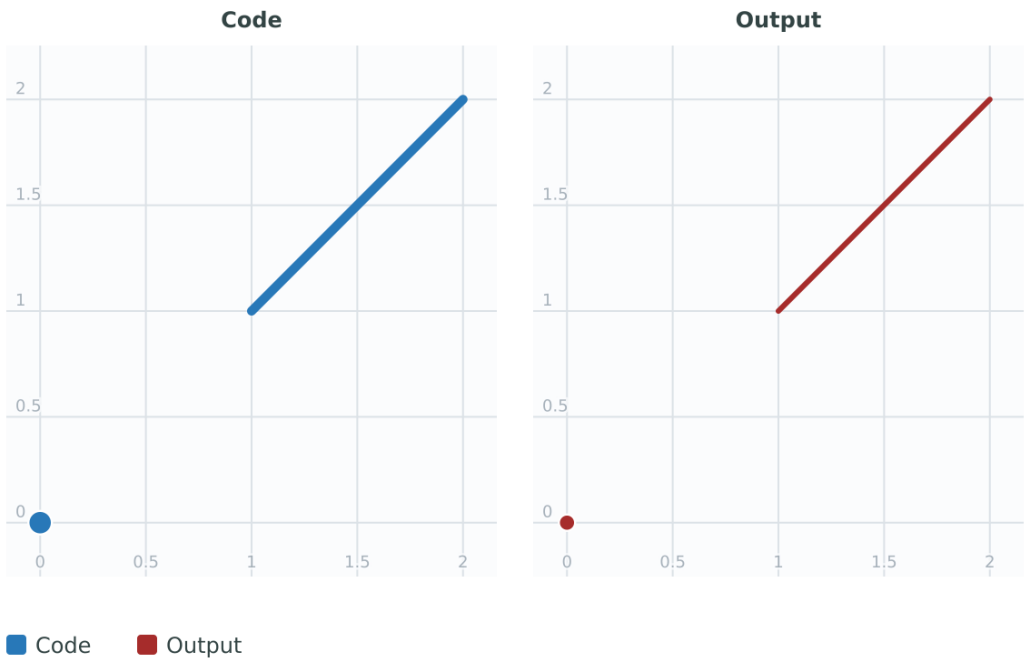
Code

```
SELECT ST_CollectionHomogenize('GEOMETRYCOLLECTION(POINT(0 0), GEOMETRYCOLLECTION(
    LINESTRING(1 1,2 2)))');
```

Output

```
GEOMETRYCOLLECTION(POINT(0 0), LINESTRING(1 1,2 2))
```

Figure



Collection of Polygons converted to an (invalid) MultiPolygon:

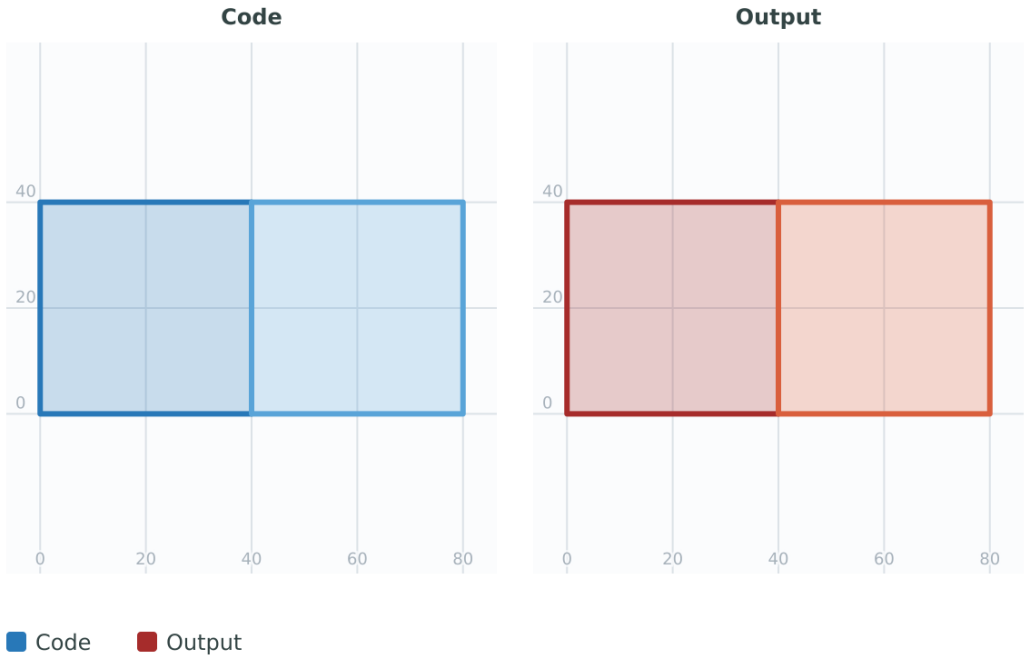
Code

```
SELECT ST_CollectionHomogenize('GEOMETRYCOLLECTION (POLYGON ((10 50,50 50,50 10,10 10,10 ↩
    50)),POLYGON ((90 50,90 10,50 10,50 50,90 50)))');
```

Output

```
MULTIPOLYGON(((10 50,50 50,50 10,10 10,10 50)),((90 50,90 10,50 10,50 50,90 50)))
```

Figure



□□

ST_CollectionExtract, ST_Multi, ST_IsValid, ST_MakeValid

7.5.4 ST_CurveToLine

ST_CurveToLine — Converts a geometry containing curves to a linear geometry.

□□

geometry **ST_CurveToLine**(geometry curveGeom, float tolerance, integer tolerance_type, integer flags);

□□

Converts a CIRCULAR STRING to regular LINESTRING or CURVEPOLYGON to POLYGON or MULTISURFACE to MULTIPOLYGON. Useful for outputting to devices that can't support CIRCULARSTRING geometry types

Converts a given geometry to a linear geometry. Each curved geometry or segment is converted into a linear approximation using the given `tolerance` and options (32 segments per quadrant and no options by default).

The 'tolerance_type' argument determines interpretation of the `tolerance` argument. It can take the following values:

- 0 (default): Tolerance is max segments per quadrant.
- 1: Tolerance is max-deviation of line from curve, in source units.
- 2: Tolerance is max-angle, in radians, between generating radii.

The 'flags' argument is a bitfield. 0 by default. Supported bits are:

- 1: Symmetric (orientation independent) output.
- 2: Retain angle, avoids reducing angles (segment lengths) when producing symmetric output. Has no effect when Symmetric flag is off.

Availability: 1.3.0

Enhanced: 2.4.0 added support for max-deviation and max-angle tolerance, and for symmetric output.

Enhanced: 3.0.0 implemented a minimum number of segments per linearized arc to prevent topological collapse.

This method implements the OGC Simple Features Implementation Specification for SQL 1.1.

This method implements the SQL/MM specification. SQL-MM 3: 7.1.7

This function supports 3d and will not drop the z-index.

This method supports Circular Strings and Curves.

□□

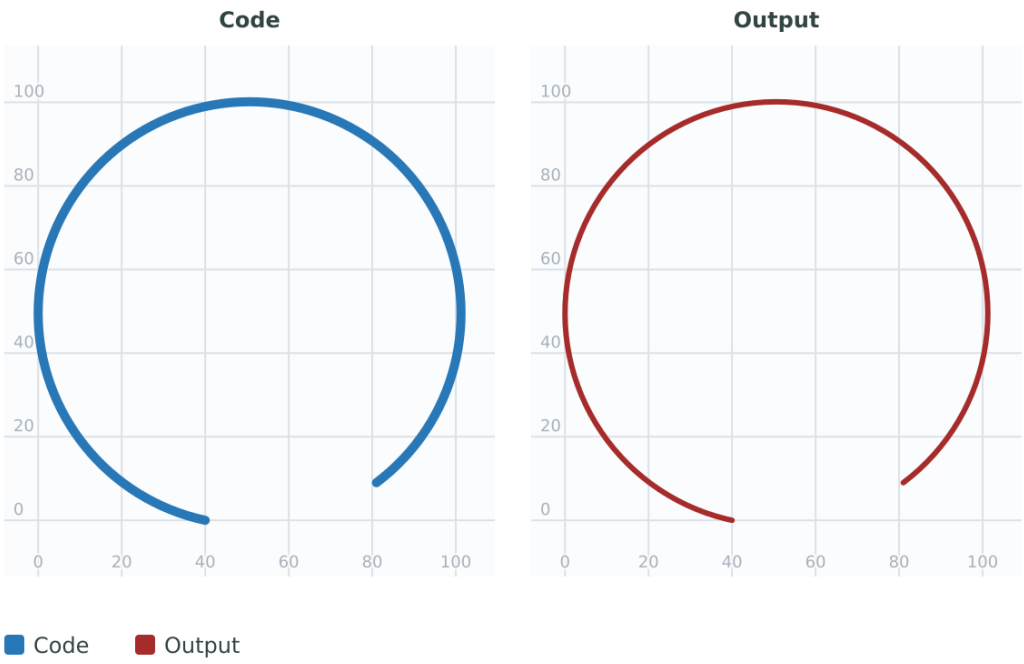
Code

```
SELECT ST_CurveToLine('CIRCULARSTRING(220268 150415,220227 150505,220227 150406)');
```

Output

```
LINESTRING(220268 150415,220269.95064912 150416.539364228,220271.823415575 150418.17258804,220273.613787707 ↩
     150419.895736857,
220275.317452352 150421.704659462,220276.930305234 150423.594998003,220278.448460847 150425.562198489,
220279.868261823 150427.60152176,220281.186287736 150429.708054909,220282.399363347 150431.876723113,
220283.50456625 150434.10230186,220284.499233914 150436.379429536,220285.380970099 150438.702620341,220286.147650624 ↩
     150441.066277505,
220286.797428488 150443.464706771,220287.328738321 150445.892130112,220287.740300149 150448.342699654,
220288.031122486 150450.810511759,220288.200504713 150453.289621251,220288.248038775 150455.77405574,
220288.173610157 150458.257830005,220287.977398166 150460.734960415,220287.659875492 150463.199479347,
220287.221807076 150465.64544956,220286.664248262 150468.066978495,220285.988542259 150470.458232479,220285.196316903 ↩
     150472.81345077,
220284.289480732 150475.126959442,220283.270218395 150477.39318505,220282.140985384 150479.606668057,
220280.90450212 150481.762075989,220279.5637474 150483.85421628,220278.12195122 150485.87804878,
220276.582586992 150487.828697901,220274.949363179 150489.701464356,220273.226214362 150491.491836488,
220271.417291757 150493.195501133,220269.526953216 150494.808354014,220267.559752731 150496.326509628,
220265.520429459 150497.746310603,220263.41389631 150499.064336517,220261.245228106 150500.277412127,
220259.019649359 150501.38261503,220256.742521683 150502.377282695,220254.419330878 150503.259018879,
220252.055673714 150504.025699404,220249.657244448 150504.675477269,220247.229821107 150505.206787101,
220244.779251566 150505.61834893,220242.311439461 150505.909171266,220239.832329968 150506.078553494,
220237.347895479 150506.126087555,220234.864121215 150506.051658938,220232.386990804 150505.855446946,
220229.922471872 150505.537924272,220227.47650166 150505.099855856,220225.054972724 150504.542297043,
220222.663718741 150503.86659104,220220.308500449 150503.074365683,
220217.994991777 150502.167529512,220215.72876617 150501.148267175,
220213.515283163 150500.019034164,220211.35987523 150498.7825509,
220209.267734939 150497.441796181,220207.243902439 150496,
220205.293253319 150494.460635772,220203.420486864 150492.82741196,220201.630114732 150491.104263143,
220199.926450087 150489.295340538,220198.313597205 150487.405001997,220196.795441592 150485.437801511,
220195.375640616 150483.39847824,220194.057614703 150481.291945091,220192.844539092 150479.123276887,220191.739336189 ↩
     150476.89769814,
220190.744668525 150474.620570464,220189.86293234 150472.297379659,220189.096251815 150469.933722495,
220188.446473951 150467.535293229,220187.915164118 150465.107869888,220187.50360229 150462.657300346,
220187.212779953 150460.189488241,220187.043397726 150457.710378749,220186.995863664 150455.22594426,
220187.070292282 150452.742169995,220187.266504273 150450.265039585,220187.584026947 150447.800520653,
220188.022095363 150445.35455044,220188.579654177 150442.933021505,220189.25536018 150440.541767521,
220190.047585536 150438.18654923,220190.954421707 150435.873040558,220191.973684044 150433.60681495,
220193.102917055 150431.393331943,220194.339400319 150429.237924011,220195.680155039 150427.14578372,220197.12195122 ↩
     150425.12195122,
220198.661315447 150423.171302099,220200.29453926 150421.298535644,220202.017688077 150419.508163512,220203.826610682 ↩
     150417.804498867,
220205.716949223 150416.191645986,220207.684149708 150414.673490372,220209.72347298 150413.253689397,220211.830006129 ↩
     150411.935663483,
220213.998674333 150410.722587873,220216.22425308 150409.61738497,220218.501380756 150408.622717305,220220.824571561 ↩
     150407.740981121,
220223.188228725 150406.974300596,220225.586657991 150406.324522731,220227 150406)
```

Figure



This example uses 3D input.

Code

```
SELECT ST_CurveToLine('CIRCULARSTRING(220268 150415 1,220227 150505 2,220227 150406 3)');
```

Output

```
LINESTRING(220268 150415 1,220269.95064912 150416.539364228 1.0181172856673,
220271.823415575 150418.17258804 1.03623457133459,220273.613787707 150419.895736857 1.05435185700189,....AD INFINITUM ↩
    ....
   220225.586657991 150406.324522731 1.32611114201132,220227 150406 3)
```

Use only 2 segments to approximate quarter circle.

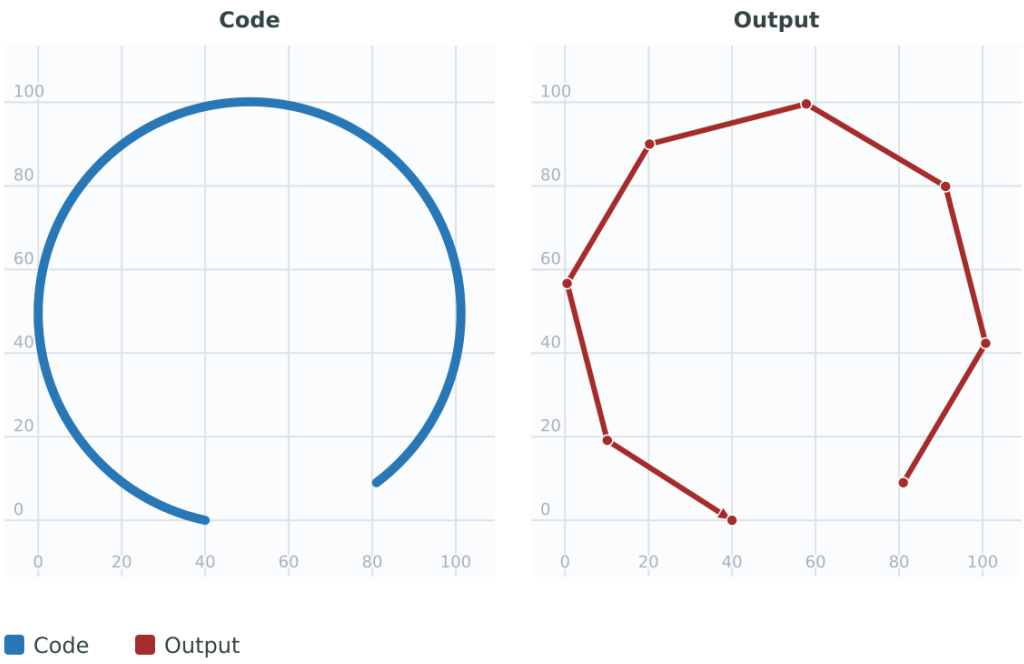
Code

```
SELECT ST_CurveToLine('CIRCULARSTRING(220268 150415,220227 150505,220227 150406)', 2);
```

Output

```
LINESTRING(220268 150415,220287.740300149 150448.342699654,220278.12195122 150485.87804878,
220244.779251566 150505.61834893,220207.243902439 150496,220187.50360229 150462.657300346,
220197.12195122 150425.12195122,220227 150406)
```

Figure



Ensure the approximated line is no more than 20 units away from the original curve, and make the result direction-neutral.

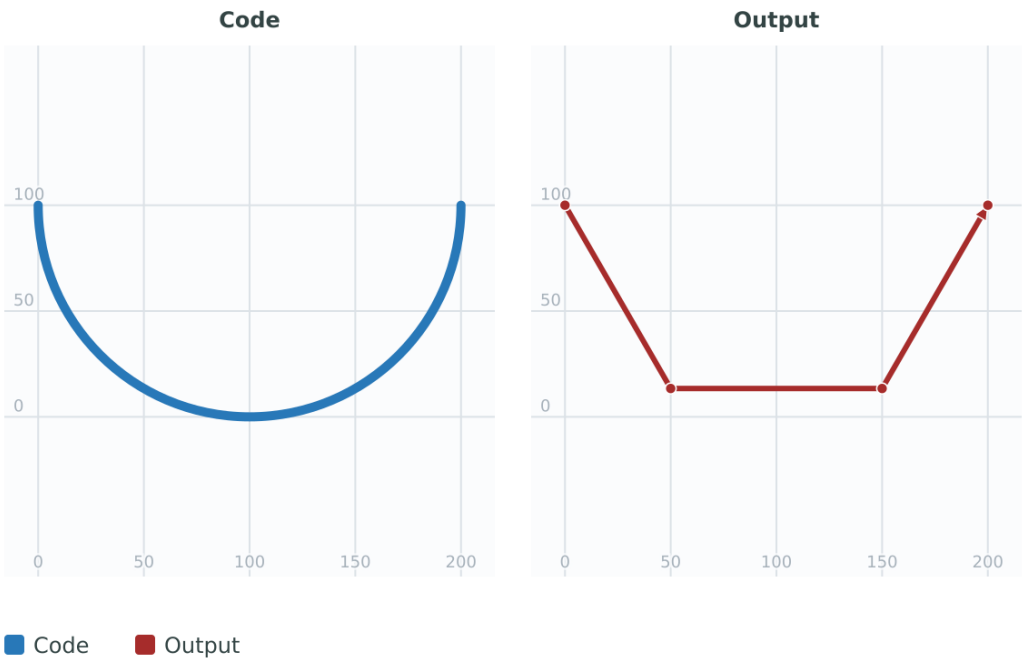
Code

```
SELECT ST_CurveToLine(
    'CIRCULARSTRING(0 0,100 -100,200 0)'::geometry,
    20,
    1,
    1
  );
```

Output

```
LINESTRING(0 0,49.99999999999996 -86.60254037844383,149.99999999999994 -86.6025403784439,200 0)
```

Figure



□□

ST_LineToCurve

7.5.5 ST_Scroll

ST_Scroll — Change start point of a closed LineString.

□□

geometry **ST_Scroll**(geometry linestring, geometry point);

□□

Changes the start/end point of a closed LineString to the given vertex *point*.
Availability: 3.2.0

This function supports 3d and will not drop the z-index.

This function supports M coordinates.

□□

Make a closed line start at its 3rd vertex.

Code

```
SELECT ST_Scroll('SRID=4326;LINESTRING(0 0 0 1,10 0 2 0,5 5 4 2,0 0 0 1)', 'POINT(5 5 4 2)' ↩
    );
```

Output

```
SRID=4326;LINESTRING(5 5 4 2,0 0 0 1,10 0 2 0,5 5 4 2)
```

Figure



□□

ST_Normalize

7.5.6 ST_FlipCoordinates

ST_FlipCoordinates — Returns a version of a geometry with X and Y axis flipped.

□□

geometry **ST_FlipCoordinates**(geometry geom);

□□

Returns a version of the given geometry with X and Y axis flipped. Useful for fixing geometries which contain coordinates expressed as latitude/longitude (Y,X).

2.0.0 □□□□□□□□□□□□.

This method supports Circular Strings and Curves.

✅ This function supports 3d and will not drop the z-index.

✅ This function supports M coordinates.

✅ This function supports Polyhedral surfaces.

✅ This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).

□□

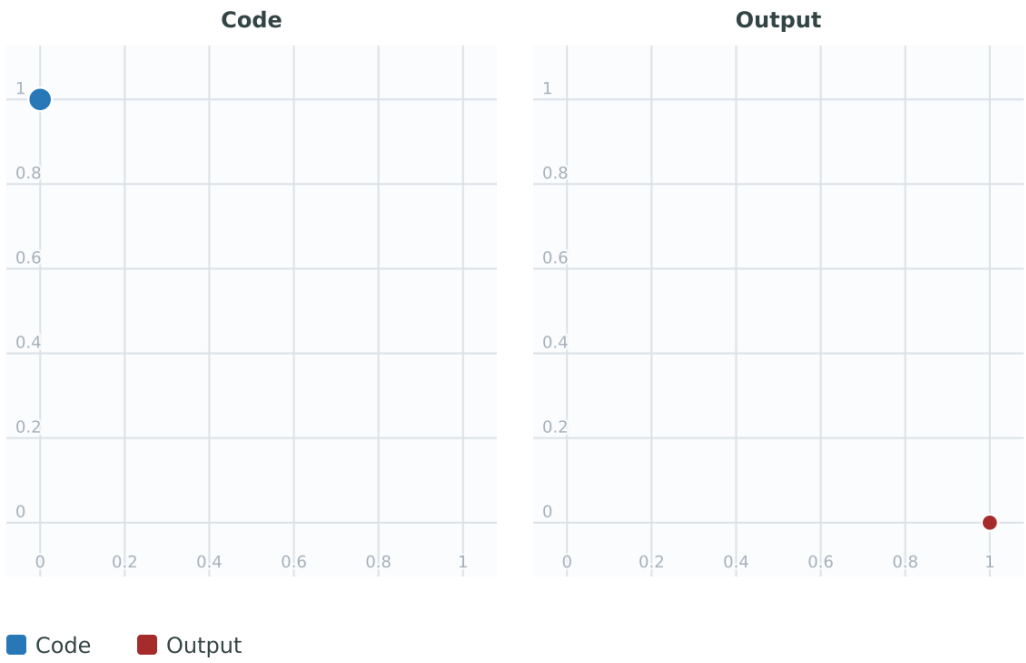
Code

```
SELECT ST_FlipCoordinates(GeomFromEWKT('POINT(1 2)'));
```

Output

```
POINT(2 1)
```

Figure



□□

ST_SwapOrdinates

7.5.7 ST_Force2D

ST_Force2D — □□□"2 □□□□" □□□□□□.

□□

geometry **ST_Force2D**(geometry geomA);

⊠⊠

⊠⊠⊠”2 ⊠⊠⊠⊠” ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ X ⊠ Y ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠ (OGC ⊠⊠⊠⊠ 2 ⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠) OGC ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠⊠⊠: 2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (polyhedral surface) ⊠⊠⊠⊠⊠⊠⊠.

⊠⊠⊠⊠: 2.1.0 ⊠⊠⊠⊠, ⊠ 2.0.x ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ST_Force_2D ⊠⊠⊠⊠.

This method supports Circular Strings and Curves.

This function supports Polyhedral surfaces.

This function supports 3d and will not drop the z-index.

⊠⊠

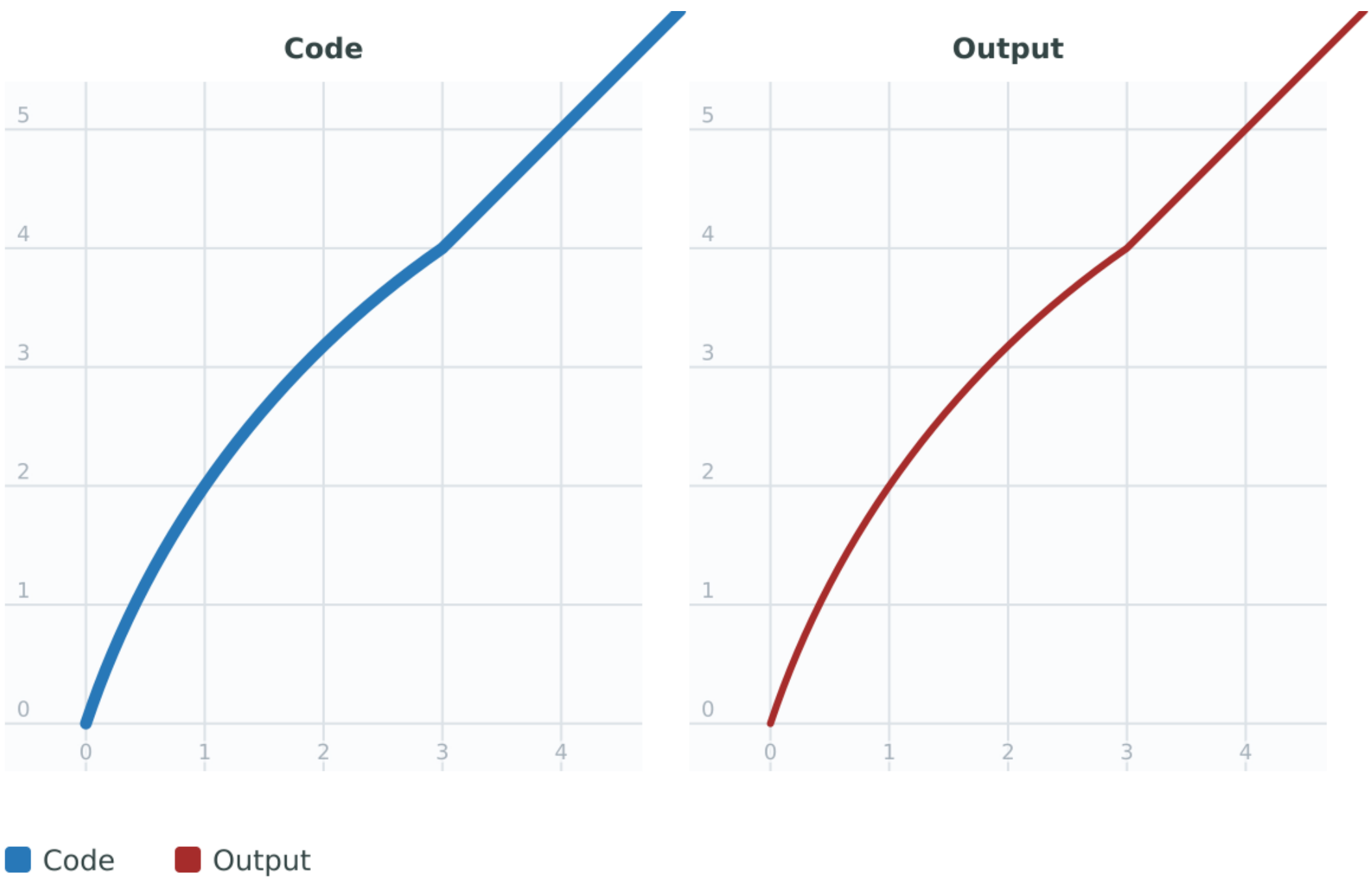
Code

```
SELECT ST_Force2D('CIRCULARSTRING(1 1 2,2 3 2,4 5 2,6 7 2,5 6 2)');
```

Output

```
CIRCULARSTRING(1 1,2 3,4 5,6 7,5 6)
```

Figure



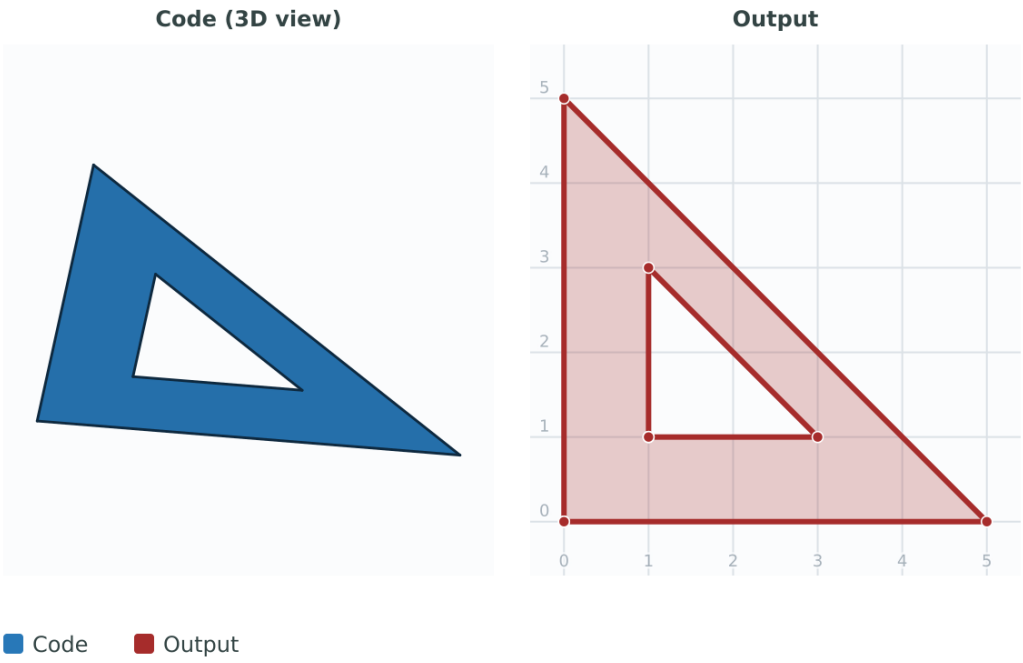
Code

```
SELECT ST_Force2D('POLYGON((0 0 2,0 5 2,5 0 2,0 0 2),(1 1 2,3 1 2,1 3 2,1 1 2))');
```

Output

```
POLYGON((0 0,0 5,5 0,0 0),(1 1,3 1,1 3,1 1))
```

Figure



□□

ST_Force3D

7.5.8 ST_Force3D

ST_Force3D — □□□ XYZ □□□□□□□□□. ST_Force3DZ □□□□□□□.

□□

geometry **ST_Force3D**(geometry geomA, float Zvalue = 0.0);

□□

Forces the geometries into XYZ mode. This is an alias for ST_Force3DZ. If a geometry has no Z component, then a *Zvalue* Z coordinate is tacked on.

□□□□: 2.0.0 □□□□□□□□□□ (polyhedral surface) □□□□□□.

□□□□: 2.1.0 □□□□, □ 2.0.x □□□□□□□□□□□ ST_Force_3D □□□□.

Changed: 3.1.0. Added support for supplying a non-zero Z value.

This function supports Polyhedral surfaces.

This method supports Circular Strings and Curves.

This function supports 3d and will not drop the z-index.

□□

Nothing happens to an already 3D geometry.

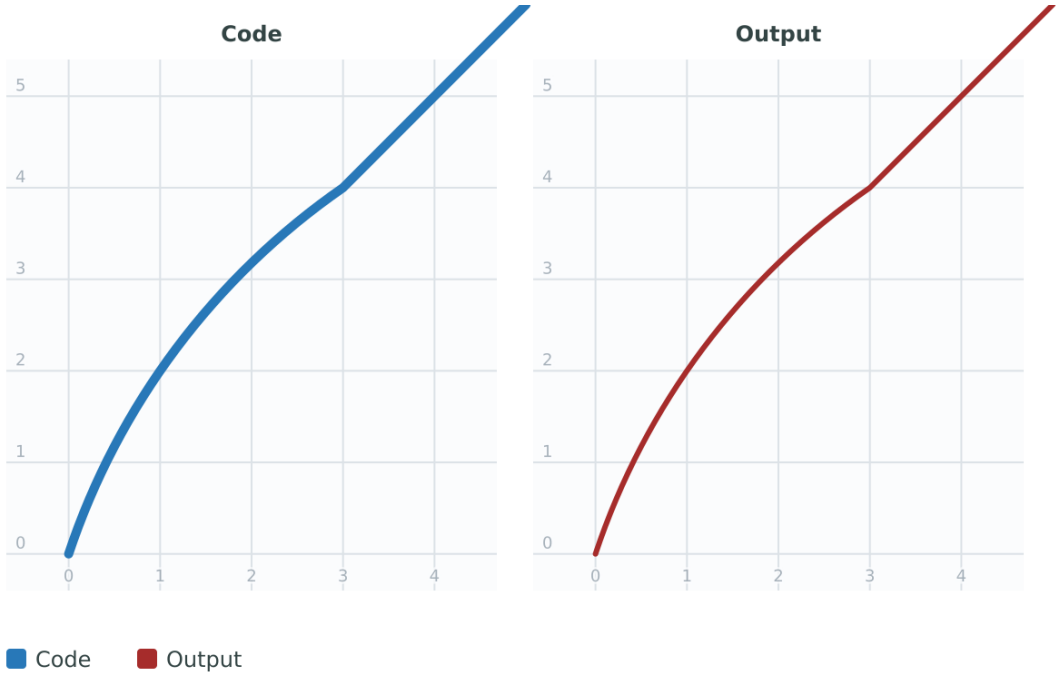
Code

```
SELECT ST_Force3D('CIRCULARSTRING(1 1 2,2 3 2,4 5 2,6 7 2,5 6 2)');
```

Output

```
CIRCULARSTRING(1 1 2,2 3 2,4 5 2,6 7 2,5 6 2)
```

Figure



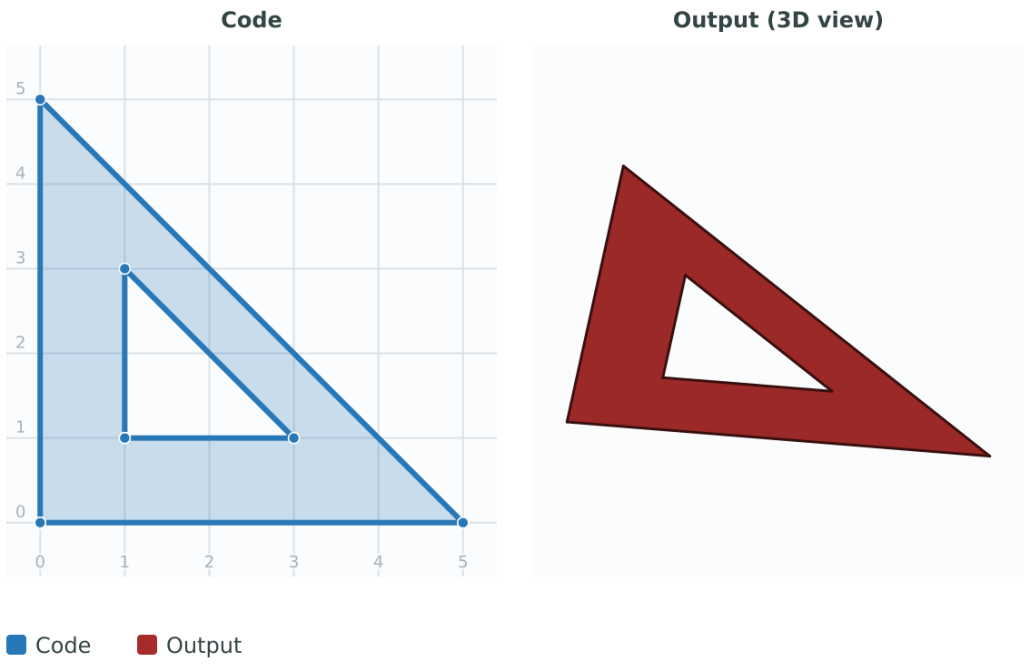
Code

```
SELECT ST_Force3D('POLYGON((0 0,0 5,5 0,0 0),(1 1,3 1,1 3,1 1))');
```

Output

```
POLYGON((0 0 0,0 5 0,5 0 0,0 0 0),(1 1 0,3 1 0,1 3 0,1 1 0))
```

Figure



□□

ST_AsEWKT, ST_Force2D, ST_Force3DM, ST_Force3DZ

7.5.9 ST_Force3DZ

ST_Force3DZ — □□□ XYZ □□□□□□□□□.

□□

geometry **ST_Force3DZ**(geometry geomA, float Zvalue = 0.0);

□□

Forces the geometries into XYZ mode. If a geometry has no Z component, then a *Zvalue* Z coordinate is tacked on.

□□□□: 2.0.0 □□□□□□□□□□ (polyhedral surface) □□□□□□.

□□□□: 2.1.0 □□□□□, □ 2.0.x □□□□□□□□□□□ ST_Force_3DZ □□□□.

Changed: 3.1.0. Added support for supplying a non-zero Z value.

This function supports Polyhedral surfaces.

This function supports 3d and will not drop the z-index.

This method supports Circular Strings and Curves.

☒☒

Nothing happens to an already 3D geometry.

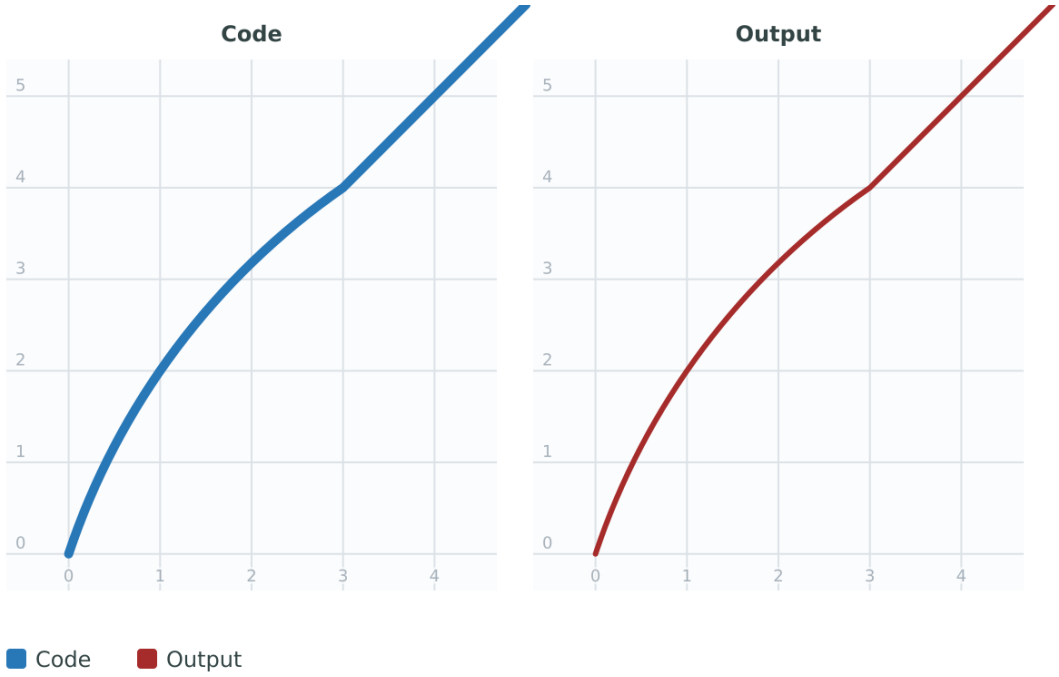
Code

```
SELECT ST_Force3DZ('CIRCULARSTRING(1 1 2,2 3 2,4 5 2,6 7 2,5 6 2)');
```

Output

```
CIRCULARSTRING(1 1 2,2 3 2,4 5 2,6 7 2,5 6 2)
```

Figure



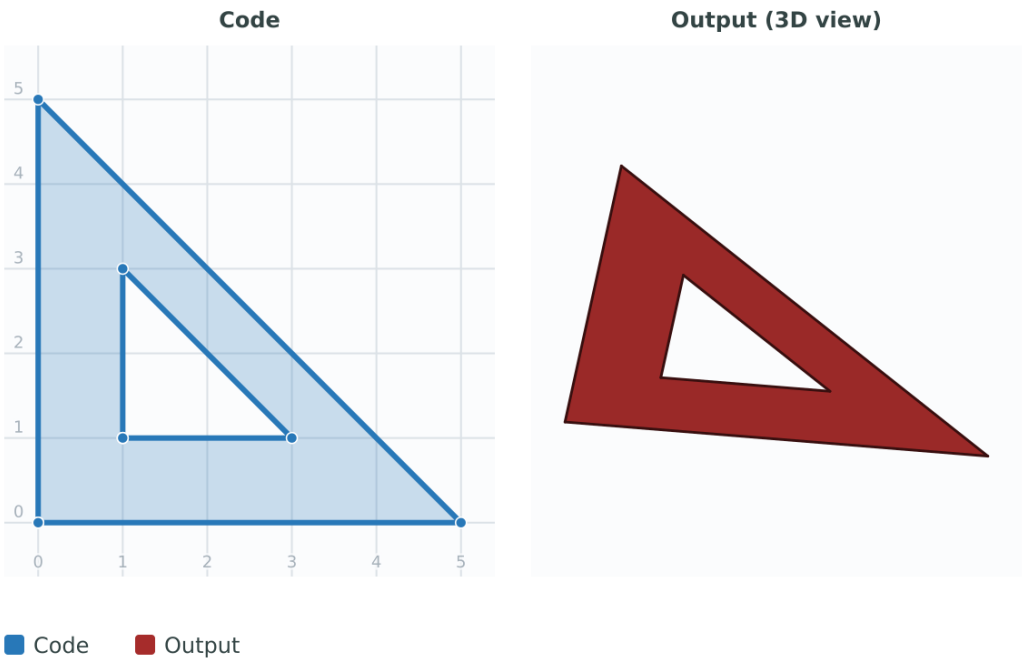
Code

```
SELECT ST_Force3DZ('POLYGON((0 0,0 5,5 0,0 0),(1 1,3 1,1 3,1 1))');
```

Output

```
POLYGON((0 0 0,0 5 0,5 0 0,0 0 0),(1 1 0,3 1 0,1 3 0,1 1 0))
```

Figure



□□

ST_AsEWKT, ST_Force2D, ST_Force3DM, ST_Force3D

7.5.10 ST_Force3DM

ST_Force3DM — □□□ XYM □□□□□□□□.

□□

geometry **ST_Force3DM**(geometry geomA, float Mvalue = 0.0);

□□

Forces the geometries into XYM mode. If a geometry has no M component, then a *Mvalue* M coordinate is tacked on. If it has a Z component, then Z is removed

□□□□: 2.1.0 □□□□, □ 2.0.x □□□□□□□□□□□ ST_Force_3DM □□□□□.

Changed: 3.1.0. Added support for supplying a non-zero M value.

This method supports Circular Strings and Curves.

□□

Nothing happens to an already 3D geometry.

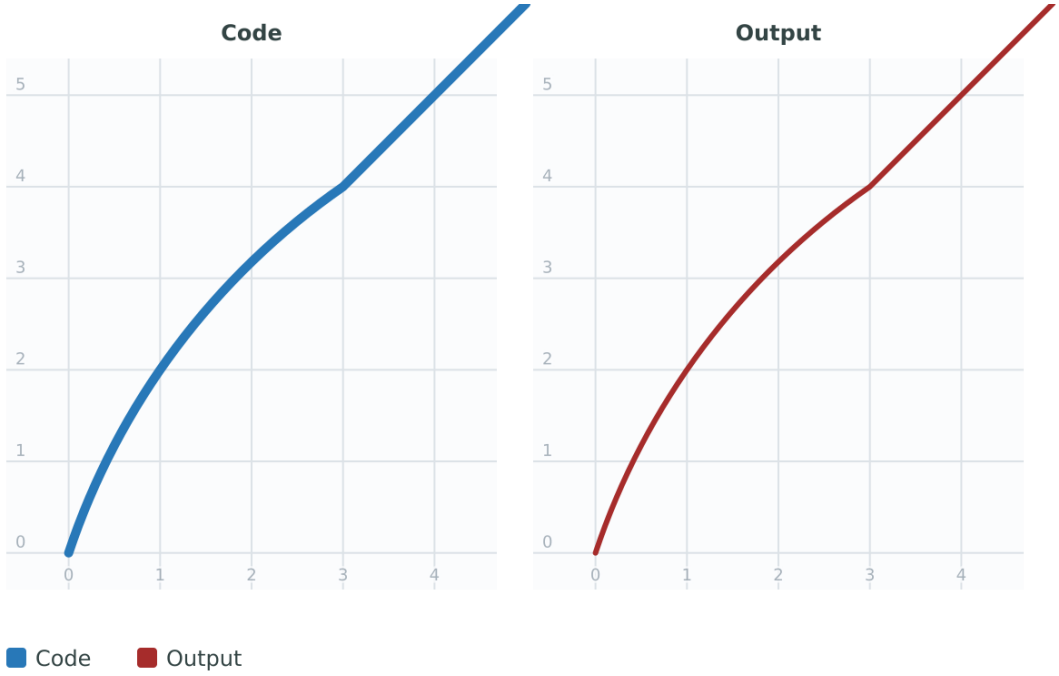
Code

```
SELECT ST_Force3DM('CIRCULARSTRING(1 1 2,2 3 2,4 5 2,6 7 2,5 6 2)');
```

Output

```
CIRCULARSTRINGM(1 1 0,2 3 0,4 5 0,6 7 0,5 6 0)
```

Figure



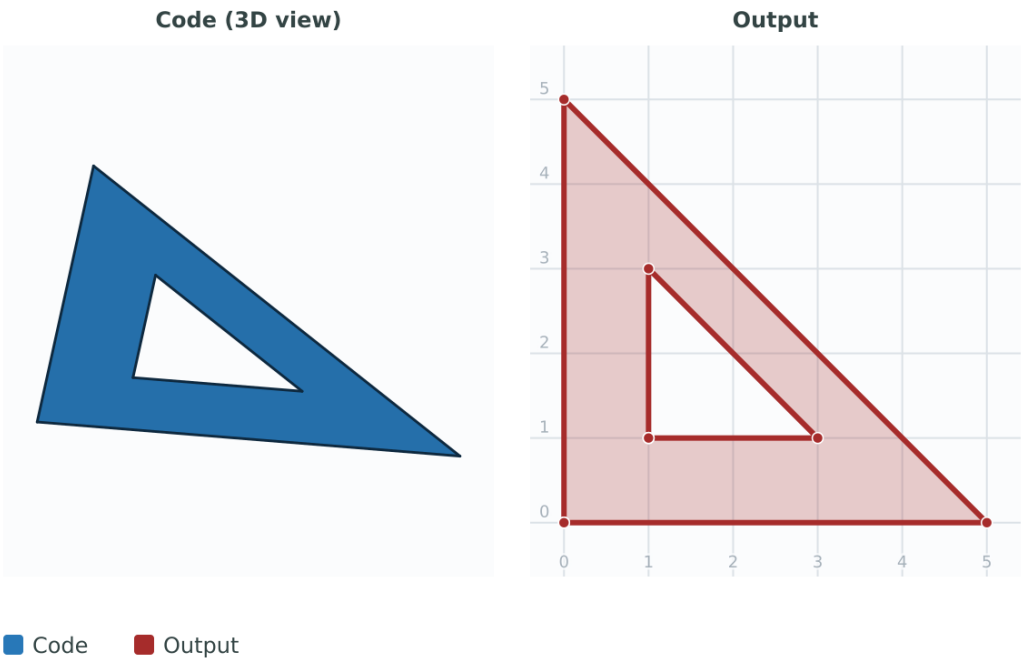
Code

```
SELECT ST_Force3DM('POLYGON((0 0 1,0 5 1,5 0 1,0 0 1),(1 1 1,3 1 1,1 3 1,1 1 1))');
```

Output

```
POLYGONM((0 0 0,0 5 0,5 0 0,0 0 0),(1 1 0,3 1 0,1 3 0,1 1 0))
```

Figure



□□

ST_AsEWKT, ST_Force2D, ST_Force3DM, ST_Force3D, ST_GeomFromEWKT

7.5.11 ST_Force4D

ST_Force4D — □□□ XYZM □□□□□□□□.

□□

geometry **ST_Force4D**(geometry geomA, float Zvalue = 0.0, float Mvalue = 0.0);

□□

Forces the geometries into XYZM mode. *Zvalue* and *Mvalue* is tacked on for missing Z and M dimensions, respectively.

□□□□: 2.1.0 □□□□, □ 2.0.x □□□□□□□□□□ ST_Force_4D □□□□.

Changed: 3.1.0. Added support for supplying non-zero Z and M values.

This function supports 3d and will not drop the z-index.

This method supports Circular Strings and Curves.

□□

Nothing happens to an already 3D geometry.

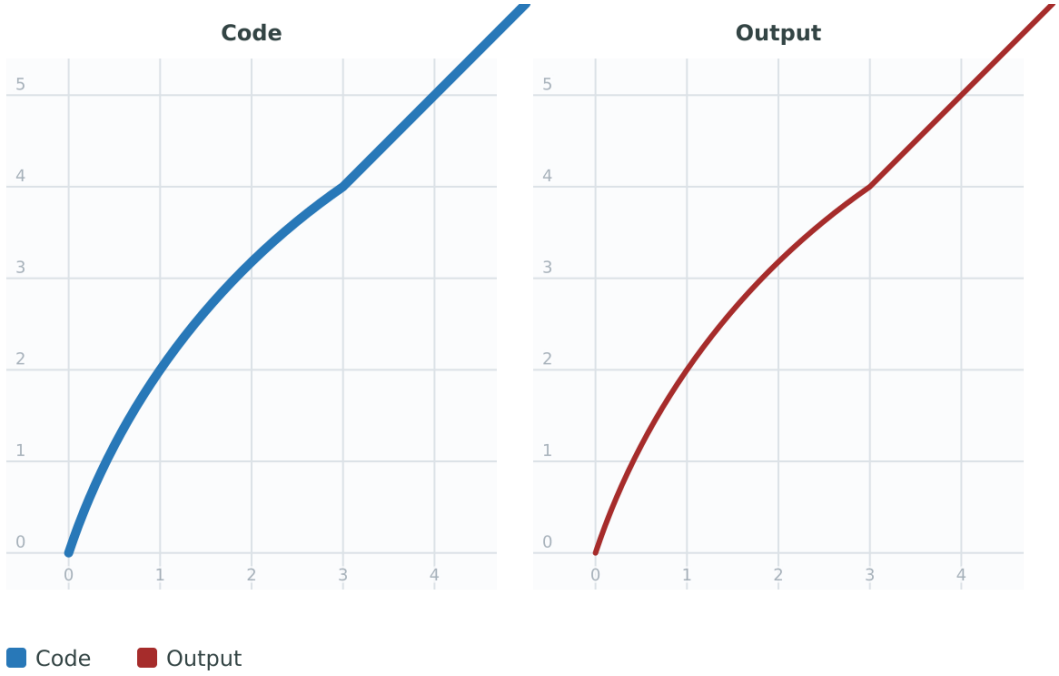
Code

```
SELECT ST_Force4D('CIRCULARSTRING(1 1 2,2 3 2,4 5 2,6 7 2,5 6 2)');
```

Output

```
CIRCULARSTRING(1 1 2 0,2 3 2 0,4 5 2 0,6 7 2 0,5 6 2 0)
```

Figure



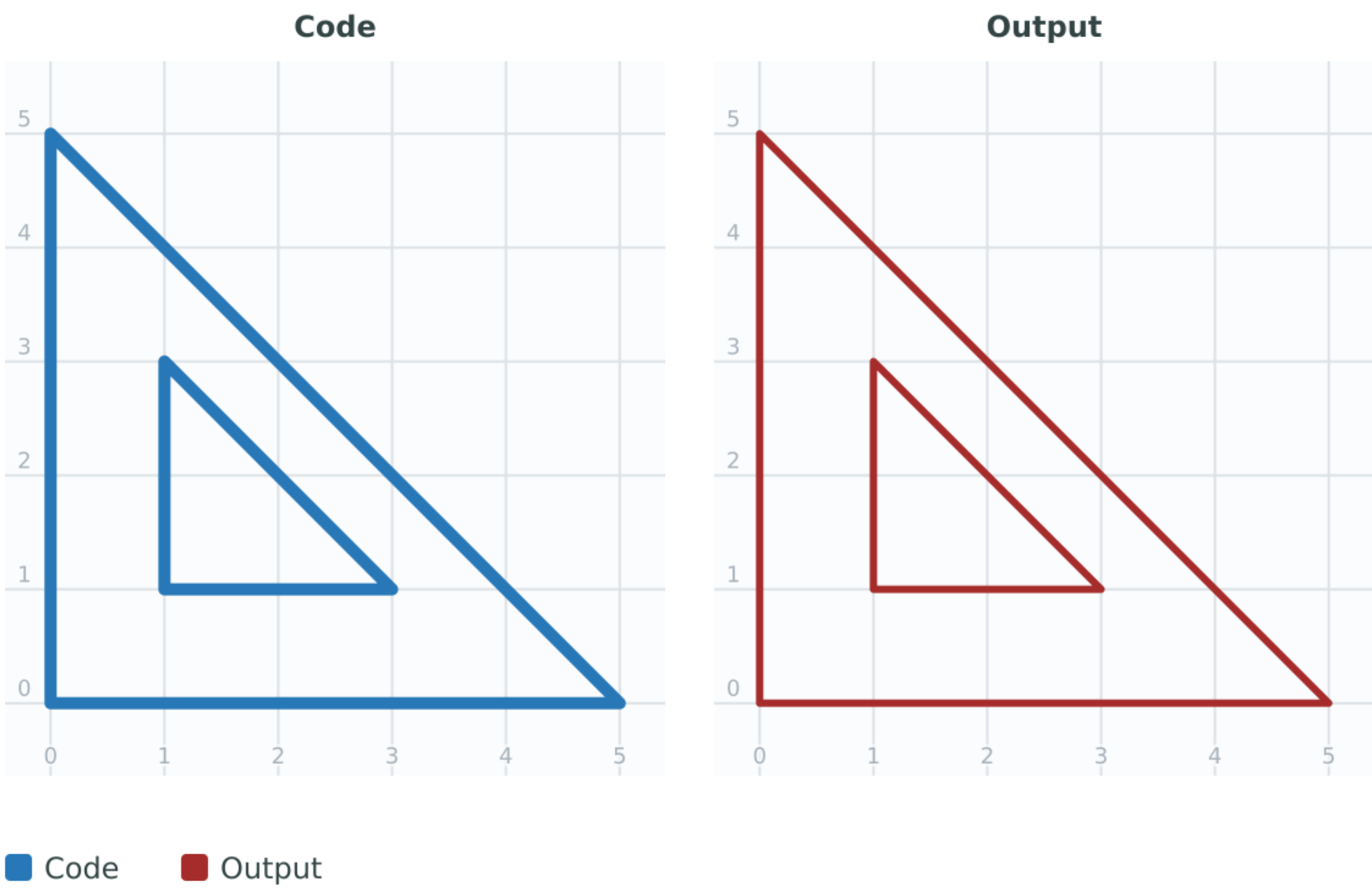
Code

```
SELECT ST_Force4D('MULTILINESTRINGM((0 0 1,0 5 2,5 0 3,0 0 4),(1 1 1,3 1 1,1 3 1,1 1 1))');
```

Output

```
MULTILINESTRING((0 0 0 1,0 5 0 2,5 0 0 3,0 0 0 4),(1 1 0 1,3 1 0 1,1 3 0 1,1 1 0 1))
```

Figure



⊠⊠

ST_AsEWKT, ST_Force2D, ST_Force3DM, ST_Force3D

7.5.12 ST_ForceCollection

ST_ForceCollection — ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

geometry **ST_ForceCollection**(geometry geomA);

⊠⊠

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠ WKB ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠⊠⊠: 2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠ (polyhedral surface) ⊠⊠⊠⊠⊠⊠.

1.2.2 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. 1.3.4 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (curve) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠. 1.3.4 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠⊠⊠: 2.1.0 ⊠⊠⊠⊠, ⊠ 2.0.x ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ST_Force_Collection ⊠⊠⊠⊠⊠.

This function supports Polyhedral surfaces.

This function supports 3d and will not drop the z-index.

This method supports Circular Strings and Curves.

⊠⊠

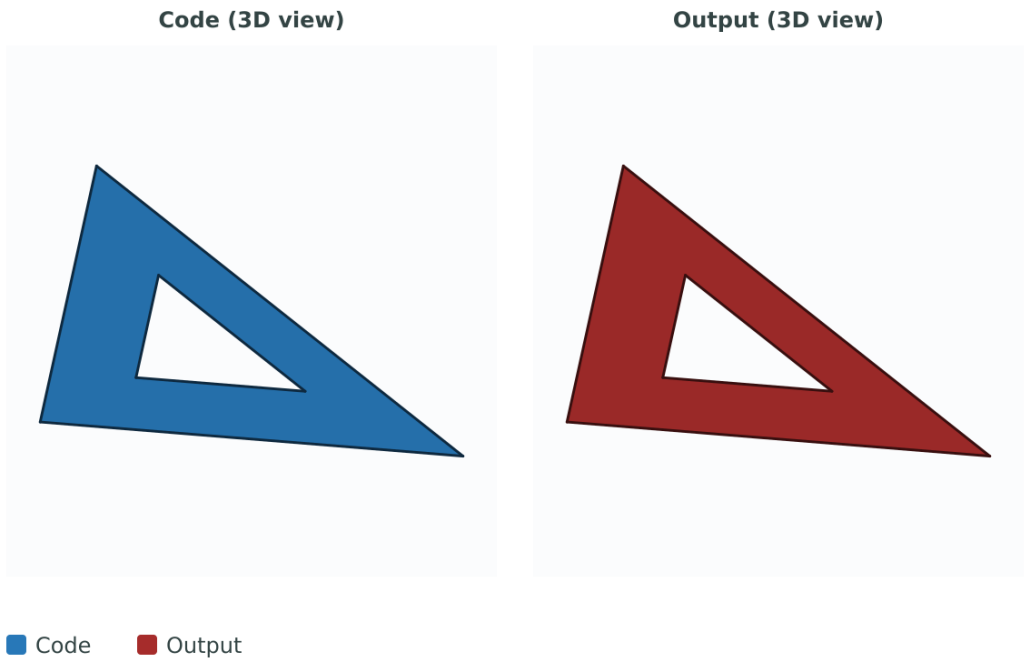
Code

```
SELECT ST_ForceCollection('POLYGON((0 0 1,0 5 1,5 0 1,0 0 1),(1 1 1,3 1 1,1 3 1,1 1 1))');
```

Output

```
GEOMETRYCOLLECTION(POLYGON((0 0 1,0 5 1,5 0 1,0 0 1),(1 1 1,3 1 1,1 3 1,1 1 1)))
```

Figure



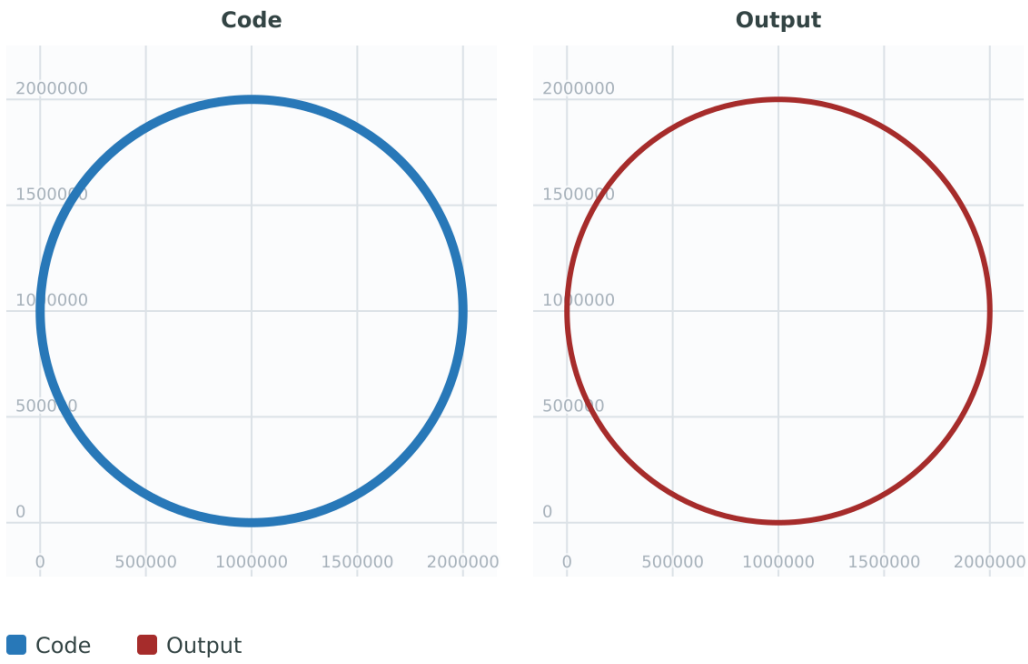
Code

```
SELECT ST_ForceCollection('CIRCULARSTRING(220227 150406,2220227 150407,220227 150406)');
```

Output

```
GEOMETRYCOLLECTION(CIRCULARSTRING(220227 150406,2220227 150407,220227 150406))
```

Figure



This example uses a PolyhedralSurface.

Code

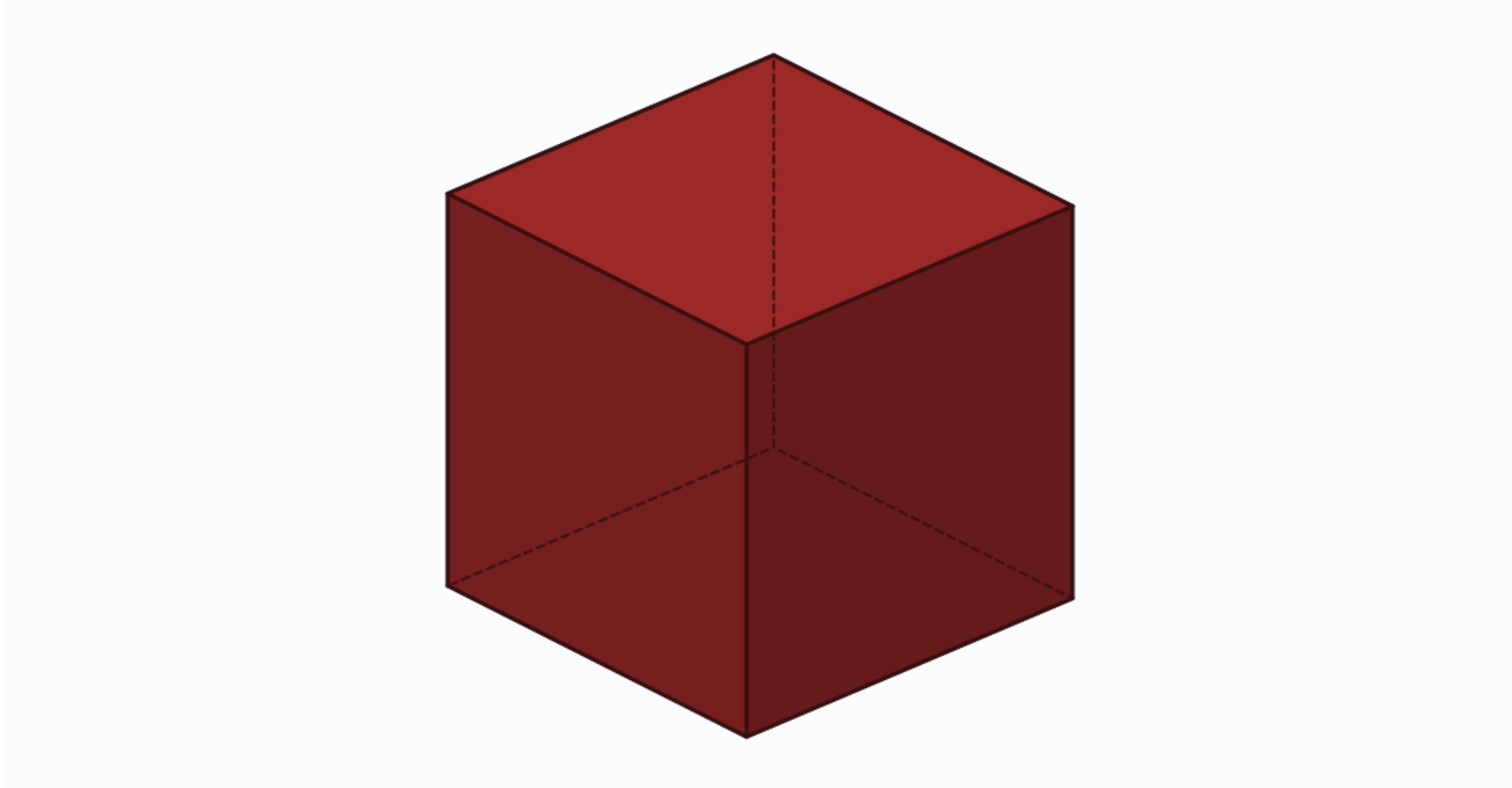
```
SELECT ST_ForceCollection('POLYHEDRALSURFACE(((0 0 0,0 0 1,0 1 1,0 1 0,0 0 0)),
 ((0 0 0,0 1 0,1 1 0,1 0 0,0 0 0)),
 ((0 0 0,1 0 0,1 0 1,0 0 1,0 0 0)),
 ((1 1 0,1 1 1,1 0 1,1 0 0,1 1 0)),
 ((0 1 0,0 1 1,1 1 1,1 1 0,0 1 0)),
 ((0 0 1,1 0 1,1 1 1,0 1 1,0 0 1)))');
```

Output

```
GEOMETRYCOLLECTION(
  POLYGON((0 0 0,0 0 1,0 1 1,0 1 0,0 0 0)),
  POLYGON((0 0 0,0 1 0,1 1 0,1 0 0,0 0 0)),
  POLYGON((0 0 0,1 0 0,1 0 1,0 0 1,0 0 0)),
  POLYGON((1 1 0,1 1 1,1 0 1,1 0 0,1 1 0)),
  POLYGON((0 1 0,0 1 1,1 1 1,1 1 0,0 1 0)),
  POLYGON((0 0 1,1 0 1,1 1 1,0 1 1,0 0 1))
)
```

Figure

3D view



Code Output

⊠⊠

ST_AsEWKT, ST_Force2D, ST_Force3DM, ST_Force3D, ST_GeomFromEWKT

7.5.13 ST_ForceCurve

ST_ForceCurve — ⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (upcast) ⊠⊠⊠.

⊠⊠

geometry **ST_ForceCurve**(geometry g);

⊠⊠

⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠ (compoundcurve) ⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (multisurface) ⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

2.2.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

This function supports 3d and will not drop the z-index.

This method supports Circular Strings and Curves.

⊠⊠

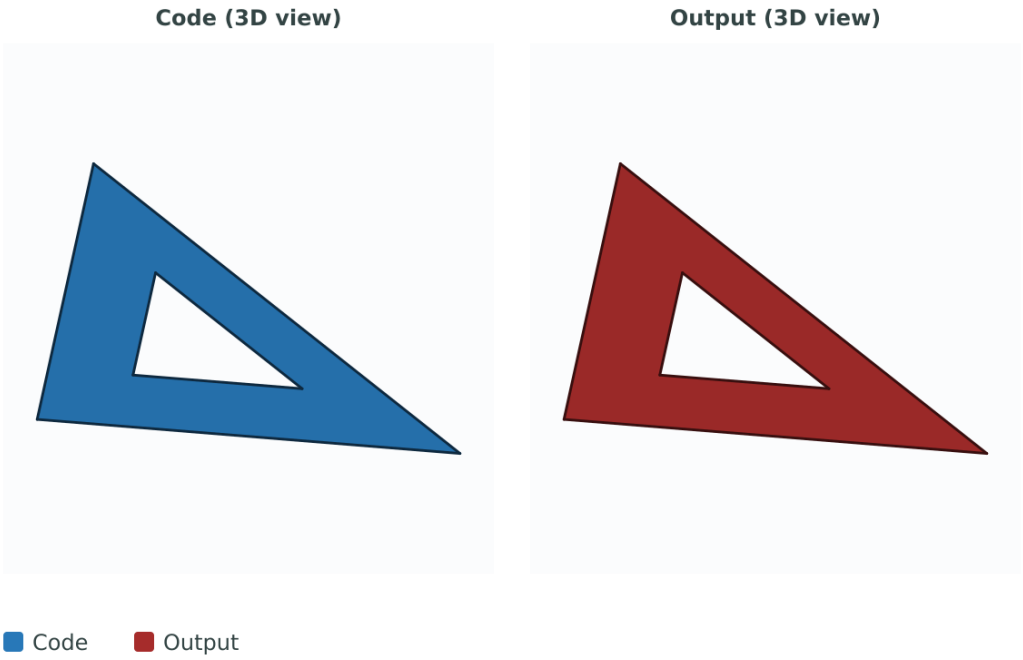
Code

```
SELECT ST_ForceCurve('POLYGON((0 0 2,5 0 2,0 5 2,0 0 2),(1 1 2,1 3 2,3 1 2,1 1 2))':: ↩
    geometry
  );
```

Output

```
CURVEPOLYGON Z ((0 0 2,5 0 2,0 5 2,0 0 2),(1 1 2,1 3 2,3 1 2,1 1 2))
```

Figure



□□

ST_LineToCurve

7.5.14 ST_ForcePolygonCCW

ST_ForcePolygonCCW — Orients all exterior rings counter-clockwise and all interior rings clockwise.

□□

geometry **ST_ForcePolygonCCW**(geometry geom);

□□

Forces (Multi)Polygons to use a counter-clockwise orientation for their exterior ring, and a clockwise orientation for their interior rings. Non-polygonal geometries are returned unchanged.
2.2.0 □□□□□□□□□□□□.

This function supports 3d and will not drop the z-index.

This function supports M coordinates.

□□

Force a polygon and its hole to use counter-clockwise exterior rings and clockwise interior rings.

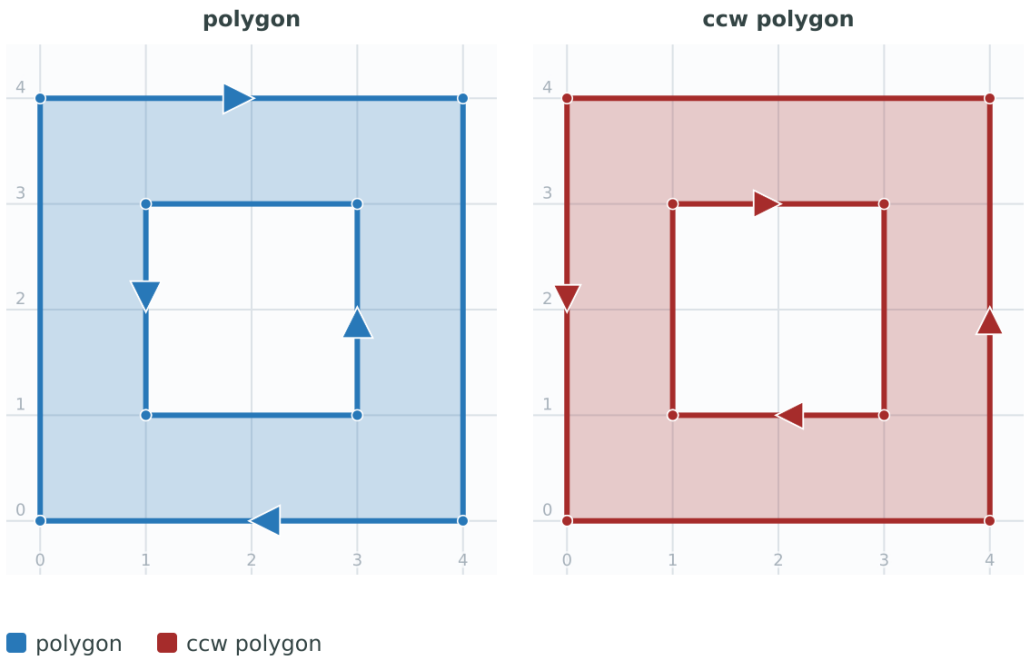
Code

```
WITH data AS (
  SELECT 'POLYGON((0 0,0 4,4 4,4 0,0 0),
                  (1 1,3 1,3 3,1 3,1 1))'::geometry AS geom
)
SELECT geom AS input_polygon,
       ST_ForcePolygonCCW(geom) AS ccw_polygon
FROM data;
```

Output

```
POLYGON((0 0,0 4,4 4,4 0,0 0),(1 1,3 1,3 3,1 3,1 1)) | POLYGON((0 0,4 0,4 4,0 4,0 0),(1 1,1 3,3 3,3 1,1 1))
```

Figure



□□

ST_ForcePolygonCW, ST_IsPolygonCCW, ST_IsPolygonCW

7.5.15 ST_ForcePolygonCW

ST_ForcePolygonCW — Orients all exterior rings clockwise and all interior rings counter-clockwise.

□□

geometry **ST_ForcePolygonCW**(geometry geom);

□□

Forces (Multi)Polygons to use a clockwise orientation for their exterior ring, and a counter-clockwise orientation for their interior rings. Non-polygonal geometries are returned unchanged.

2.2.0 □□□□□□□□□□□□.

✅ This function supports 3d and will not drop the z-index.

✅ This function supports M coordinates.

□□

Force a polygon and its hole to use clockwise exterior rings and counter-clockwise interior rings.

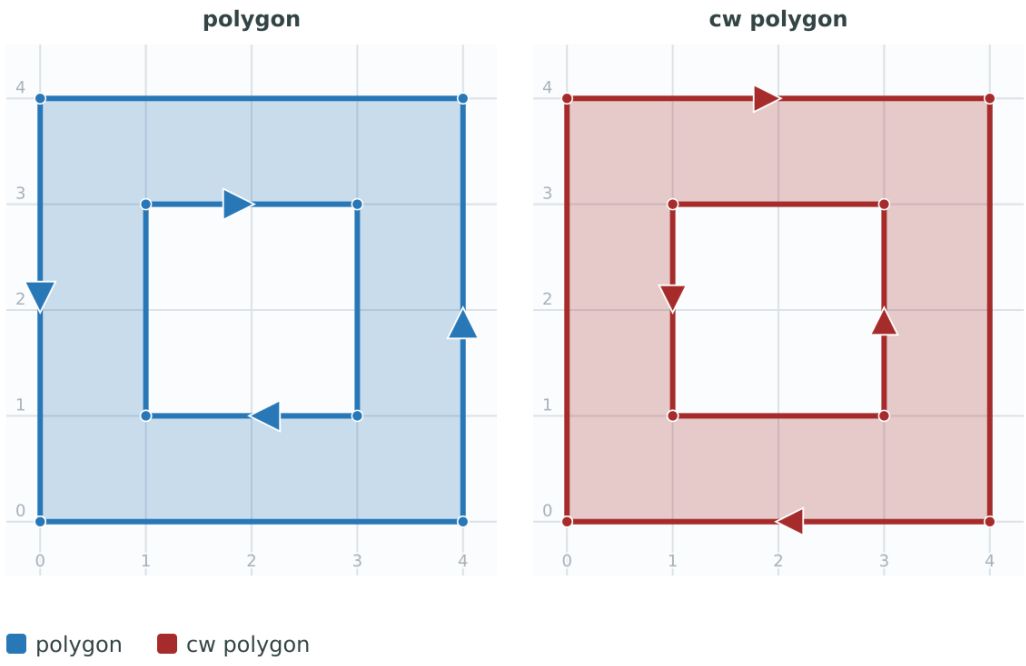
Code

```
WITH data AS (
  SELECT 'POLYGON((0 0,4 0,4 4,0 4,0 0),
                  (1 1,1 3,3 3,3 1,1 1))'::geometry AS geom
)
SELECT geom AS input_polygon,
       ST_ForcePolygonCW(geom) AS cw_polygon
FROM data;
```

Output

```
POLYGON((0 0,4 0,4 4,0 4,0 0),(1 1,1 3,3 3,3 1,1 1)) | POLYGON((0 0,0 4,4 4,4 0,0 0),(1 1,3 1,3 3,1 3,1 1))
```

Figure



□□

ST_ForcePolygonCCW, ST_IsPolygonCCW, ST_IsPolygonCW

7.5.16 ST_ForceSFS

ST_ForceSFS — Force geometries to use SFS 1.1 or 1.2 geometry types.

☒☒

geometry **ST_ForceSFS**(geometry geomA);
geometry **ST_ForceSFS**(geometry geomA, text version);

☒☒

Forces a geometry to use the geometry types defined by the requested version of the OGC Simple Features Specification. The default is SFS 1.1. Curved geometries are converted to linear representations, `TRIANGLE` is converted to `POLYGON`, and `TIN` and `POLYHEDRALSURFACE` are converted to `GEOMETRYCOLLECTION` values containing `POLYGON` elements.

If *`version`* begins with `1.2`, curved geometries are still converted to linear representations, but `TRIANGLE`, `TIN`, and `POLYHEDRALSURFACE` are preserved. Other version values use the default SFS 1.1 behavior. Geometry collections are processed recursively, and the SRID and Z/M coordinates are preserved.

2.1.0 ☒☒☒☒☒☒☒☒☒☒.

This function supports Polyhedral surfaces.

This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).

This method supports Circular Strings and Curves.

This function supports 3d and will not drop the z-index.

☒☒

Compare the geometry types produced for SFS 1.1 (the default) and SFS 1.2. Read each row from the input type in the direction of the result columns.

Code

```
WITH inputs(geom) AS (VALUES
    ('CIRCULARSTRING(0 0, 1 1, 2 0)'::geometry),
    ('TRIANGLE((0 0, 0 2, 2 0, 0 0))'::geometry),
    ('TIN(((0 0, 0 2, 2 0, 0 0)))'::geometry),
    ('POLYHEDRALSURFACE(((0 0, 0 2, 2 0, 0 0)))'::geometry)
)
SELECT ST_GeometryType(geom) || ' b''→b''' AS input,
       ST_GeometryType(ST_ForceSFS(geom)) AS "SFS 1.1 (default)",
       ST_GeometryType(ST_ForceSFS(geom, '1.2')) AS "SFS 1.2"
FROM inputs;
```

Output

```
input          |   SFS 1.1 (default)   |       SFS 1.2
-----------------------+-----------------------+---------------------
 ST_CircularString b''→b''    | ST_LineString         | ST_LineString
 ST_Triangle b''→b''          | ST_Polygon            | ST_Triangle
 ST_Tin b''→b''               | ST_GeometryCollection | ST_Tin
 ST_PolyhedralSurface b''→b'' | ST_GeometryCollection | ST_PolyhedralSurface
(4 rows)
```

□□

ST_CurveToLine, ST_ForceCurve

7.5.17 ST_ForceRHR

ST_ForceRHR — □□□□□□□□□□□□□□□ (orientation) □□□□□□ (Right-Hand Rule) □□□□□ □□□□□.

□□

geometry **ST_ForceRHR**(geometry g);

□□

Forces the orientation of the vertices in a polygon to follow a Right-Hand-Rule, in which the area that is bounded by the polygon is to the right of the boundary. In particular, the exterior ring is orientated in a clockwise direction and the interior rings in a counter-clockwise direction. This function is a synonym for ST_ForcePolygonCW



□□
The above definition of the Right-Hand-Rule conflicts with definitions used in other contexts. To avoid confusion, it is recommended to use ST_ForcePolygonCW.

□□□□: 2.0.0 □□□□□□□□□□ (polyhedral surface) □□□□□□.

 This function supports 3d and will not drop the z-index.

 This function supports Polyhedral surfaces.

□□

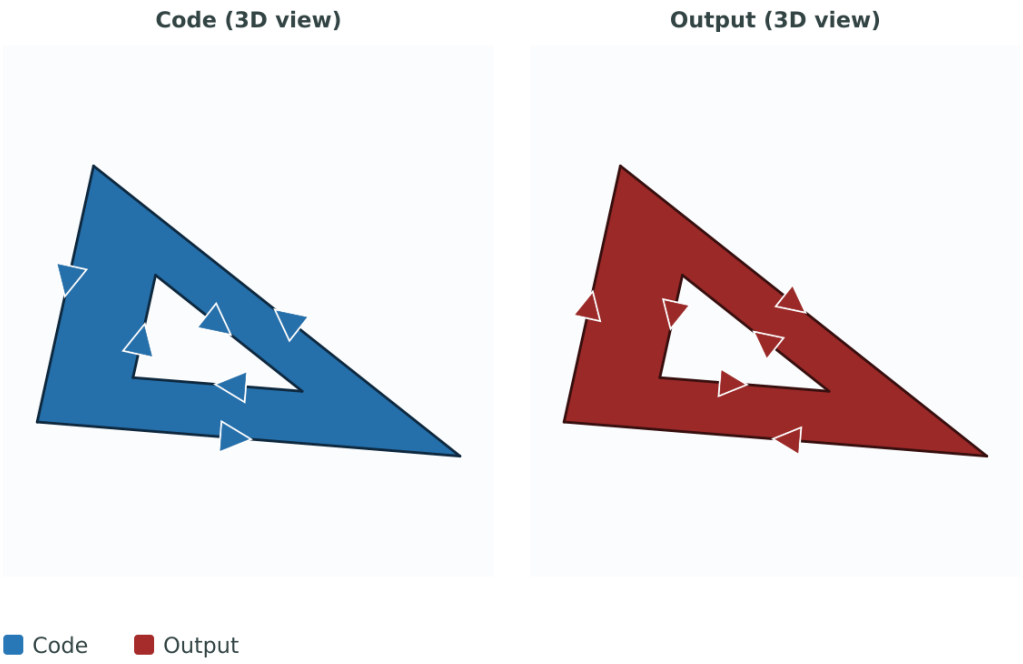
Code

```
SELECT ST_ForceRHR('POLYGON((0 0 2,5 0 2,0 5 2,0 0 2),(1 1 2,1 3 2,3 1 2,1 1 2))'
  );
```

Output

```
POLYGON((0 0 2,0 5 2,5 0 2,0 0 2),(1 1 2,3 1 2,1 3 2,1 1 2))
```

Figure



☒☒

ST_ForcePolygonCCW, ST_ForcePolygonCW, ST_IsPolygonCCW, ST_IsPolygonCW, ST_BuildArea, ST_Polygo
ST_Reverse

7.5.18 ST_LineExtend

ST_LineExtend — Returns a line extended forwards and backwards by specified distances.

☒☒

geometry **ST_LineExtend**(geometry line, float distance_forward, float distance_backward=0.0);

☒☒

Returns a line extended forwards and backwards by adding new start (and end) points at the given distance(s). A distance of zero does not add a point. Only non-negative distances are allowed. The direction(s) of the added point(s) is determined by the first (and last) two distinct points of the line. Duplicate points are ignored.

Availability: 3.4.0

Example: Extends a line 5 units forward and 6 units backward

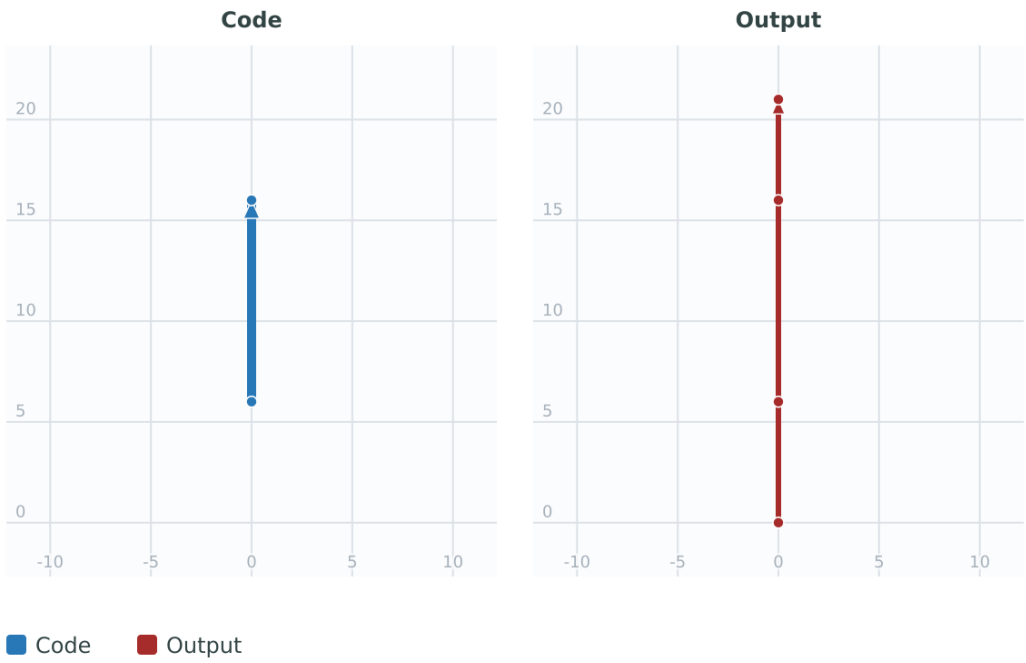
Code

```
SELECT ST_LineExtend('LINESTRING(0 0,0 10)'::geometry, 5, 6);
```

Output

```
LINESTRING(0 -6,0 0,0 10,0 15)
```

Figure



☒☒

ST_LineSubstring, ST_LocateAlong, ST_Project

7.5.19 ST_LineToCurve

ST_LineToCurve — Converts a linear geometry to a curved geometry.

☒☒

geometry **ST_LineToCurve**(geometry geomANoncircular);

☒☒

Converts plain LINESTRING/POLYGON to CIRCULAR STRINGs and Curved Polygons. Note much fewer points are needed to describe the curved equivalent.



☒☒
If the input LINESTRING/POLYGON is not curved enough to clearly represent a curve, the function will return the same input geometry.

Availability: 1.3.0

This function supports 3d and will not drop the z-index.

This method supports Circular Strings and Curves.

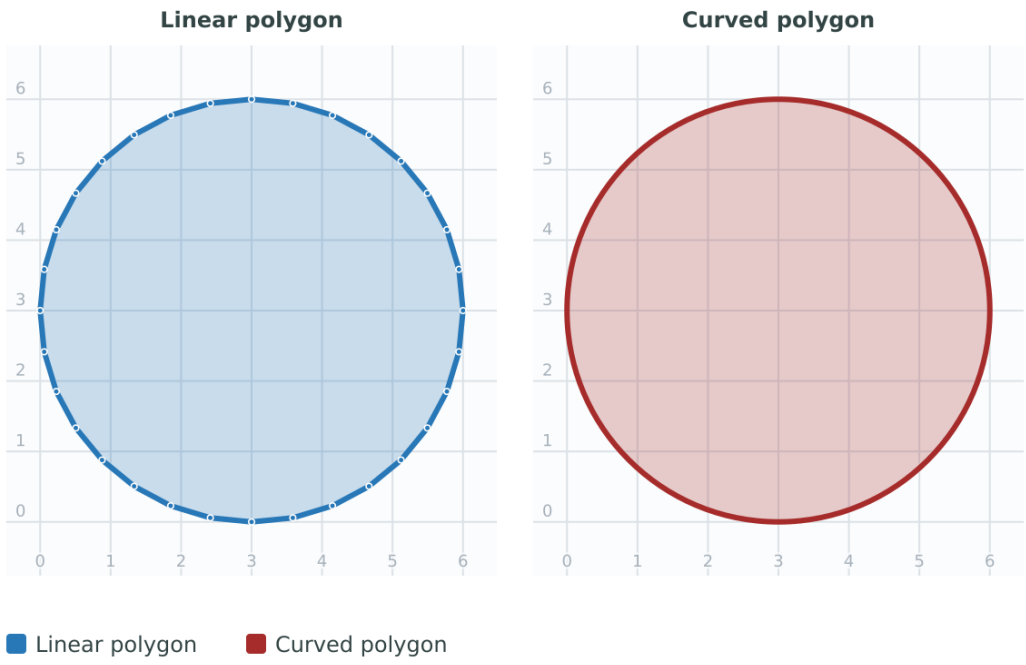
□□

This example converts a linear Polygon approximating a circle to a CurvePolygon.

Code

```
WITH data AS (
    SELECT ST_Buffer('POINT(1 3)'::geometry, 3) AS geom
)
SELECT geom AS "input_Linear polygon",
       ST_LineToCurve(geom) AS "Curved polygon"
FROM data;
```

Figure

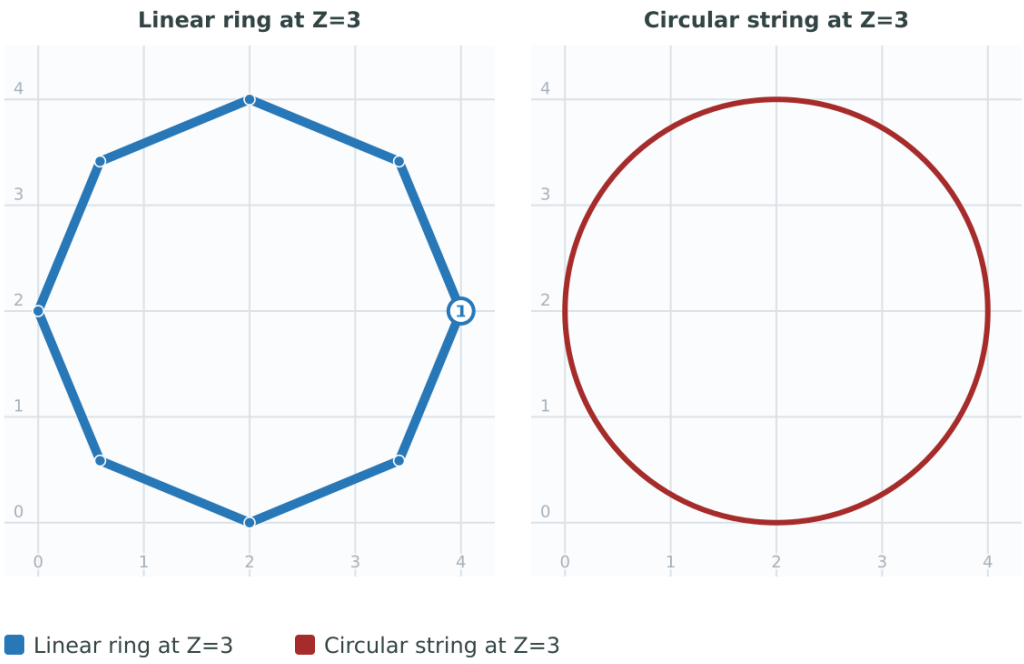


This example converts a 3D linear ring to a CircularString while preserving Z.

Code

```
WITH data AS (
    SELECT ST_Translate(
        ST_Force3D(ST_Boundary(ST_Buffer(ST_Point(1, 3), 2, 2))),
        0, 0, 3
    ) AS geom
)
SELECT geom AS "input_Linear ring at Z=3",
       ST_LineToCurve(geom) AS "Circular string at Z=3"
FROM data;
```

Figure



□□

ST_CurveToLine

7.5.20 ST_Multi

ST_Multi — □□□□□□□□□□□□□□□□□□□□.

□□

geometry **ST_Multi**(geometry geom);

□□

Returns the geometry as a MULTI* geometry collection. If the geometry is already a collection, it is returned unchanged.

□□

A Polygon is converted to a single-element MultiPolygon. The coordinates are unchanged.

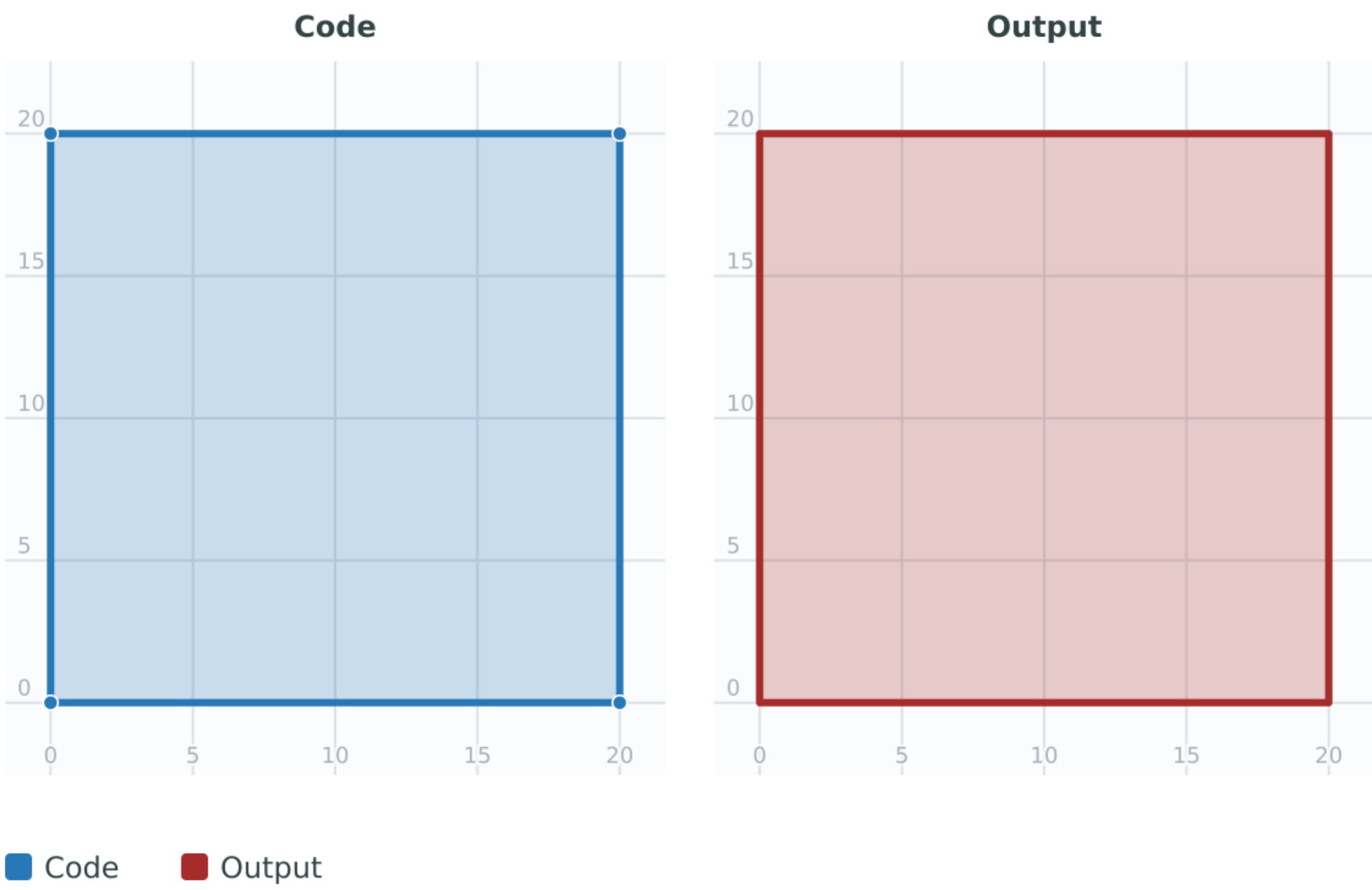
Code

```
SELECT ST_Multi('POLYGON ((10 30,30 30,30 10,10 10,10 30))');
```

Output

```
MULTIPOLYGON(((10 30,30 30,30 10,10 10,10 30)))
```

Figure



☒☒

ST_AsText

7.5.21 ST_Normalize

ST_Normalize — ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.

☒☒

geometry **ST_Normalize**(geometry geom);

☒☒

☒☒☒☒☒☒☒☒☒☒☒☒/☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒, ☒☒☒☒☒☒☒☒☒☒☒☒☒☒, ☒☒☒☒☒☒☒☒☒ ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.

☒☒☒☒☒☒☒, ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒ (☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒).

2.3.0 ☒☒☒☒☒☒☒☒☒☒☒☒.

☒☒

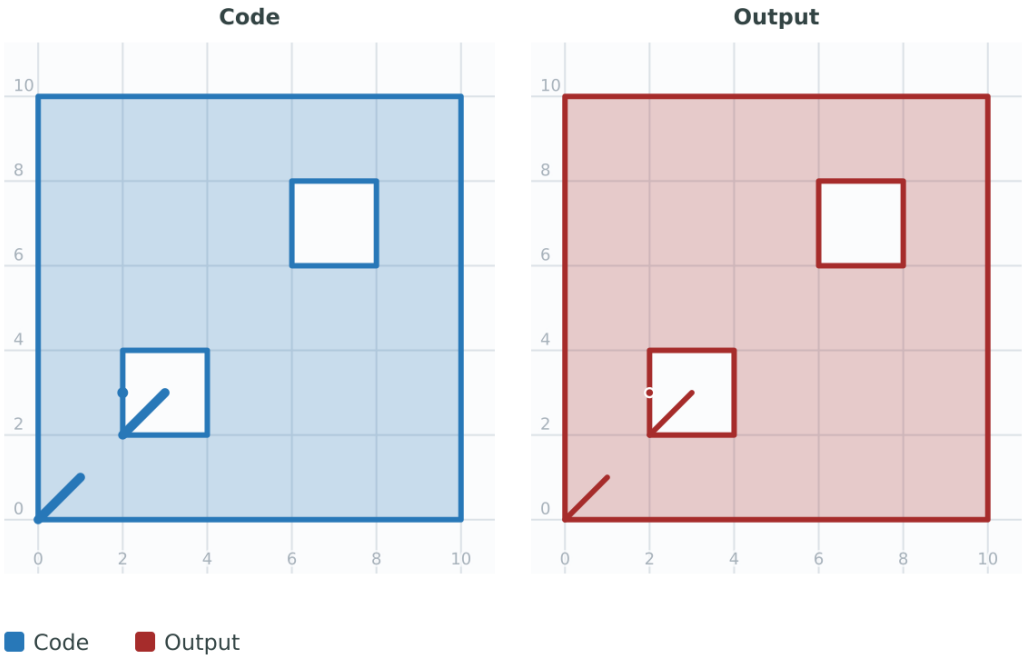
Code

```
SELECT ST_Normalize('GEOMETRYCOLLECTION(
    POINT(2 3),
    MULTILINESTRING((0 0,1 1),(2 2,3 3)),
    POLYGON(
      (0 10,0 0,10 0,10 10,0 10),
      (4 2,2 2,2 4,4 4,4 2),
      (6 8,8 8,8 6,6 6,6 8)
    )
  )');
```

Output

```
GEOMETRYCOLLECTION(POLYGON((0 0,0 10,10 10,10 0,0 0),(6 6,8 6,8 8,6 8,6 6),(2 2,4 2,4 4,2 4,2 2)), MULTILINESTRING((2 ↩
    2,3 3),(0 0,1 1)), POINT(2 3))
```

Figure



□□

ST_Equals,

7.5.22 ST_Project

ST_Project — Returns a point projected from a start point by a distance and bearing (azimuth).

□□

geometry **ST_Project**(geometry g1, float distance, float azimuth);
geometry **ST_Project**(geometry g1, geometry g2, float distance);
geography **ST_Project**(geography g1, float distance, float azimuth);
geography **ST_Project**(geography g1, geography g2, float distance);

□□

Returns a point projected from a point along a geodesic using a given distance and azimuth (bearing). This is known as the direct geodesic problem.

The two-point version uses the path from the first to the second point to implicitly define the azimuth and uses the distance as before.

The distance is given in meters. Negative values are supported.

The azimuth (also known as heading or bearing) is given in radians. It is measured clockwise from true north.

- North is azimuth zero (0 degrees)
- East is azimuth π/2 (90 degrees)
- South is azimuth π (180 degrees)
- West is azimuth 3π/2 (270 degrees)

Negative azimuth values and values greater than 2π (360 degrees) are supported.

2.0.0 □□□□□□□□□□□□.

Enhanced: 2.4.0 Allow negative distance and non-normalized azimuth.

Enhanced: 3.4.0 Allow geometry arguments and two-point form omitting azimuth.

Example: Projected point at 100,000 meters and bearing 45 degrees

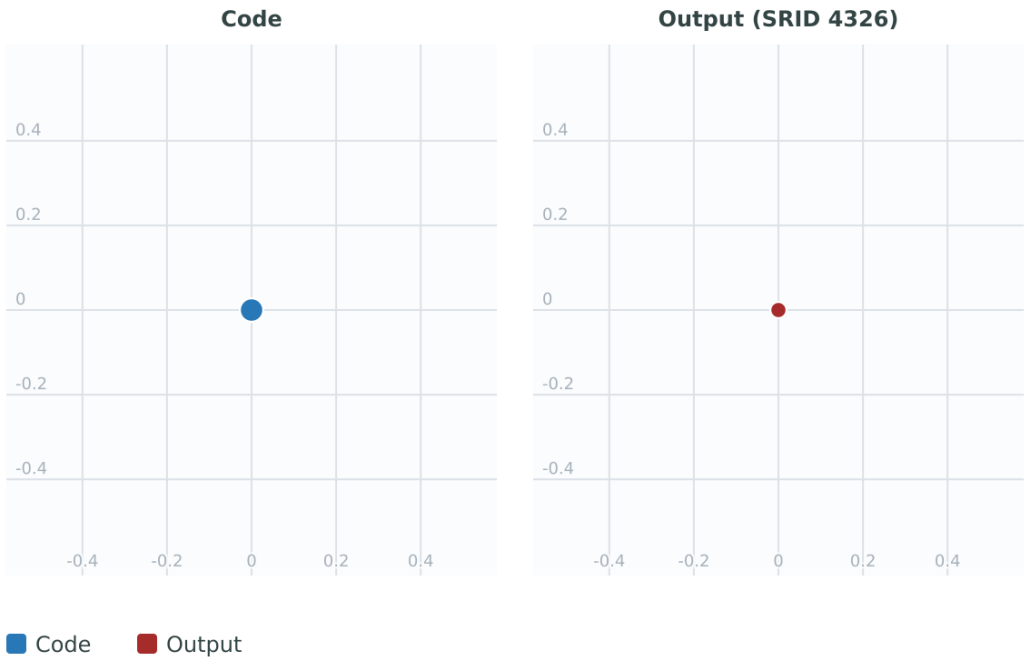
Code

```
SELECT ST_Project('POINT(0 0)'::geography, 100000, radians(45.0));
```

Output

```
POINT(0.635231029125537 0.639472334729198)
```

Figure



□□

ST_Azimuth, ST_Distance, PostgreSQL function radians()

7.5.23 ST_QuantizeCoordinates

ST_QuantizeCoordinates — Sets least significant bits of coordinates to zero

□□

geometry **ST_QuantizeCoordinates**(geometry g, int prec_x, int prec_y = NULL, int prec_z = NULL, int prec_m = NULL);

□□

`ST_QuantizeCoordinates` determines the number of bits (`N`) required to represent a coordinate value with a specified number of digits after the decimal point, and then sets all but the `N` most significant bits to zero. The resulting coordinate value will still round to the original value, but will have improved compressiblity. This can result in a significant disk usage reduction provided that the geometry column is using a compressible storage type. The function allows specification of a different number of digits after the decimal point in each dimension; unspecified dimensions are assumed to have the precision of the `x` dimension. Negative digits are interpreted to refer digits to the left of the decimal point, (i.e., `prec_x=-2` will preserve coordinate values to the nearest 100.

The coordinates produced by `ST_QuantizeCoordinates` are independent of the geometry that contains those coordinates and the relative position of those coordinates within the geometry. As a result, existing topological relationships between geometries are unaffected by use of this function. The function may produce invalid geometry when it is called with a number of digits lower than the intrinsic precision of the geometry.

Availability: 2.5.0

Technical Background

PostGIS stores all coordinate values as double-precision floating point integers, which can reliably represent 15 significant digits. However, PostGIS may be used to manage data that intrinsically has fewer than 15 significant digits. An example is TIGER data, which is provided as geographic coordinates with six digits of precision after the decimal point (thus requiring only nine significant digits of longitude and eight significant digits of latitude.)

When 15 significant digits are available, there are many possible representations of a number with 9 significant digits. A double precision floating point number uses 52 explicit bits to represent the significand (mantissa) of the coordinate. Only 30 bits are needed to represent a mantissa with 9 significant digits, leaving 22 insignificant bits; we can set their value to anything we like and still end up with a number that rounds to our input value. For example, the value 100.123456 can be represented by the floating point numbers closest to 100.123456000000, 100.123456000001, and 100.123456432199. All are equally valid, in that `ST_AsText(geom, 6)` will return the same result with any of these inputs. As we can set these bits to any value, `ST_QuantizeCoordinates` sets the 22 insignificant bits to zero. For a long coordinate sequence this creates a pattern of blocks of consecutive zeros that is compressed by PostgreSQL more efficiently.



□□
Only the on-disk size of the geometry is potentially affected by `ST_QuantizeCoordinates`. ST_MemSize, which reports the in-memory usage of the geometry, will return the the same value regardless of the disk space used by a geometry.

□□

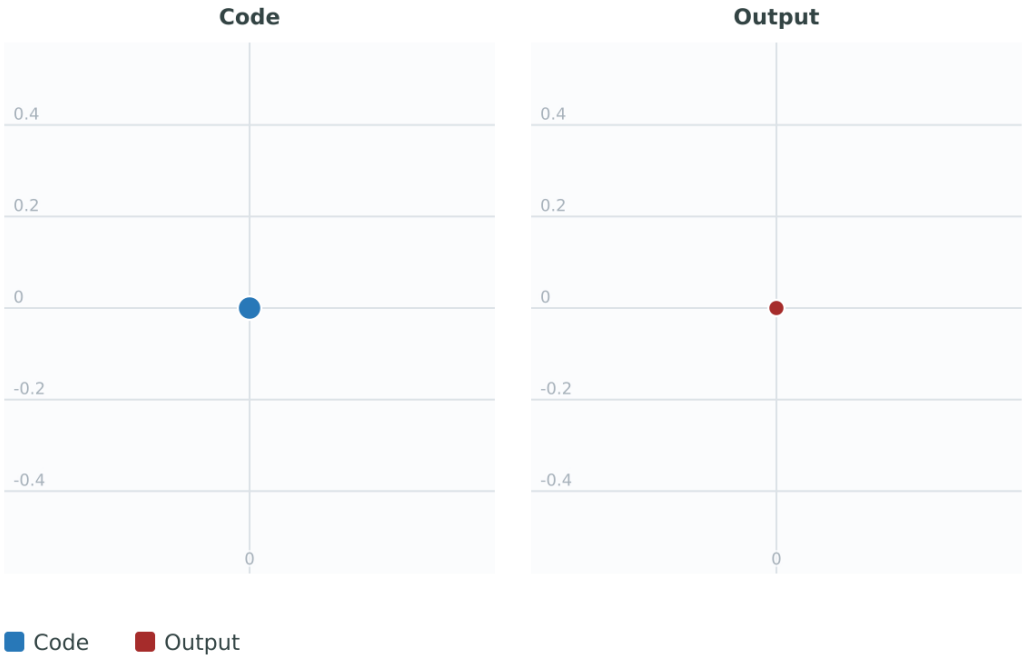
Code

```
SELECT ST_QuantizeCoordinates('POINT (100.123456 0)'::geometry, 4);
```

Output

```
POINT(100.12344360351562 0)
```

Figure



Code

```
WITH test AS (SELECT 'POINT (123.456789123456 123.456789123456)'::geometry AS geom)
SELECT
  digits,
  encode(ST_QuantizeCoordinates(geom, digits), 'hex'),
  ST_QuantizeCoordinates(geom, digits)
FROM test, generate_series(15, -15, -1) AS digits;
```

Output

```
digits |                  encode                  |                 st_astext
--------+------------------------------------------+-------------------------------------------
     15 | 01010000005f9a72083cdd5e405f9a72083cdd5e40 | POINT(123.456789123456 123.456789123456)
     14 | 01010000005f9a72083cdd5e405f9a72083cdd5e40 | POINT(123.456789123456 123.456789123456)
     13 | 01010000005e9a72083cdd5e405e9a72083cdd5e40 | POINT(123.45678912345599 123.45678912345599)
     12 | 0101000000409a72083cdd5e40409a72083cdd5e40 | POINT(123.45678912345556 123.45678912345556)
     11 | 0101000000009a72083cdd5e40009a72083cdd5e40 | POINT(123.45678912345466 123.45678912345466)
     10 | 0101000000009872083cdd5e40009872083cdd5e40 | POINT(123.45678912344738 123.45678912344738)
      9 | 0101000000008072083cdd5e40008072083cdd5e40 | POINT(123.45678912336007 123.45678912336007)
      8 | 0101000000000070083cdd5e40000070083cdd5e40 | POINT(123.45678912103176 123.45678912103176)
      7 | 0101000000000060083cdd5e40000060083cdd5e40 | POINT(123.4567891061306 123.4567891061306)
      6 | 0101000000000000083cdd5e40000000083cdd5e40 | POINT(123.45678901672363 123.45678901672363)
      5 | 0101000000000000003cdd5e40000000003cdd5e40 | POINT(123.456787109375 123.456787109375)
      4 | 0101000000000000003cdd5e40000000003cdd5e40 | POINT(123.456787109375 123.456787109375)
      3 | 0101000000000000038dd5e400000000038dd5e40 | POINT(123.45654296875 123.45654296875)
      2 | 0101000000000000000dd5e400000000000dd5e40 | POINT(123.453125 123.453125)
      1 | 0101000000000000000dc5e400000000000dc5e40 | POINT(123.4375 123.4375)
      0 | 0101000000000000000c05e400000000000c05e40 | POINT(123 123)
     -1 | 0101000000000000000000005e4000000000000005e40 | POINT(120 120)
```

```
     -2 | 0101000000000000000000058400000000000005840 | POINT(96 96)
     -3 | 0101000000000000000000058400000000000005840 | POINT(96 96)
     -4 | 0101000000000000000000058400000000000005840 | POINT(96 96)
     -5 | 0101000000000000000000058400000000000005840 | POINT(96 96)
     -6 | 0101000000000000000000058400000000000005840 | POINT(96 96)
     -7 | 0101000000000000000000058400000000000005840 | POINT(96 96)
     -8 | 0101000000000000000000058400000000000005840 | POINT(96 96)
     -9 | 0101000000000000000000058400000000000005840 | POINT(96 96)
    -10 | 0101000000000000000000058400000000000005840 | POINT(96 96)
    -11 | 0101000000000000000000058400000000000005840 | POINT(96 96)
    -12 | 0101000000000000000000058400000000000005840 | POINT(96 96)
    -13 | 0101000000000000000000058400000000000005840 | POINT(96 96)
    -14 | 0101000000000000000000058400000000000005840 | POINT(96 96)
    -15 | 0101000000000000000000058400000000000005840 | POINT(96 96)
(31 rows)
```

☒☒

ST_SnapToGrid

7.5.24 ST_RemovePoint

ST_RemovePoint — Remove a point from a linestring.

☒☒

geometry **ST_RemovePoint**(geometry linestring, integer offset);

☒☒

Removes a point from a LineString, given its index (0-based). Useful for turning a closed line (ring) into an open linestring.

Enhanced: 3.2.0

1.1.0 ☒☒☒☒☒☒☒☒☒☒☒.

This function supports 3d and will not drop the z-index.

☒☒

Guarantees no lines are closed by removing the end point of closed lines (rings). Assumes geom is of type LINESTRING

Code

```
UPDATE sometable
    SET geom = ST_RemovePoint(geom, ST_NPoints(geom) - 1)
    FROM sometable
    WHERE ST_IsClosed(geom);
```

☒☒

ST_AddPoint, ST_NPoints, ST_NumPoints

7.5.25 ST_RemoveRepeatedPoints

ST_RemoveRepeatedPoints — Returns a version of a geometry with duplicate points removed.

□□

geometry **ST_RemoveRepeatedPoints**(geometry geom, float8 tolerance = 0.0);

□□

Returns a version of the given geometry with duplicate consecutive points removed. The function processes only (Multi)LineStrings, (Multi)Polygons and MultiPoints but it can be called with any kind of geometry. Elements of GeometryCollections are processed individually. The endpoints of LineStrings are preserved.

If a non-zero *tolerance* parameter is provided, vertices within the tolerance distance of one another are considered to be duplicates. The distance is computed in 2D (XY plane).

Enhanced: 3.2.0

2.2.0 □□□□□□□□□□□□.

This function supports Polyhedral surfaces.

This function supports 3d and will not drop the z-index.

□□

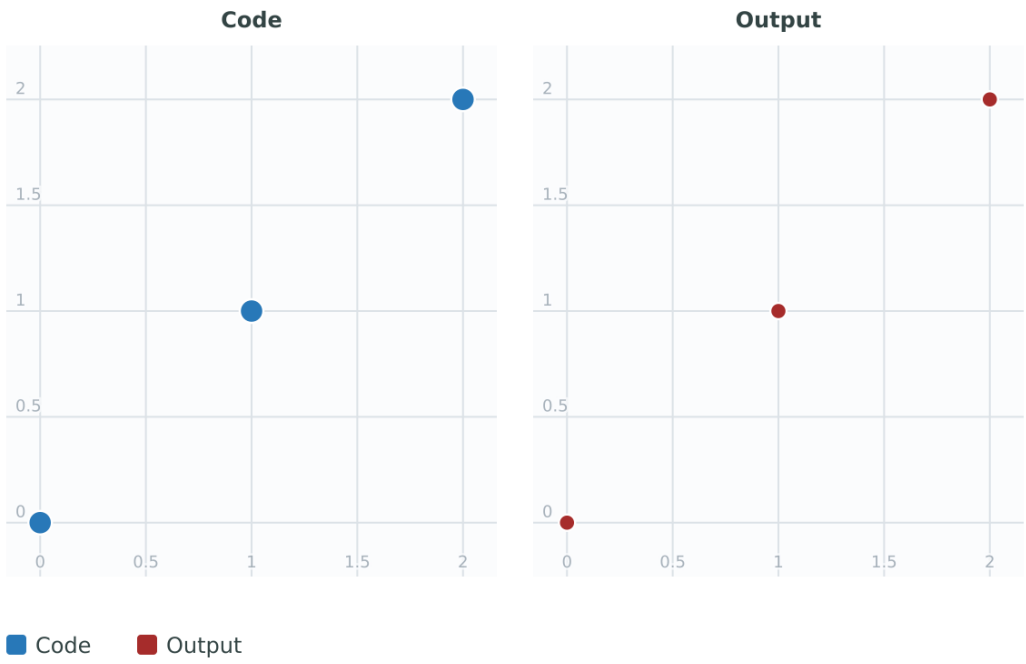
Code

```
SELECT ST_RemoveRepeatedPoints('MULTIPOINT ((1 1),(2 2),(3 3),(2 2))');
```

Output

```
MULTIPOINT((1 1),(2 2),(3 3))
```

Figure



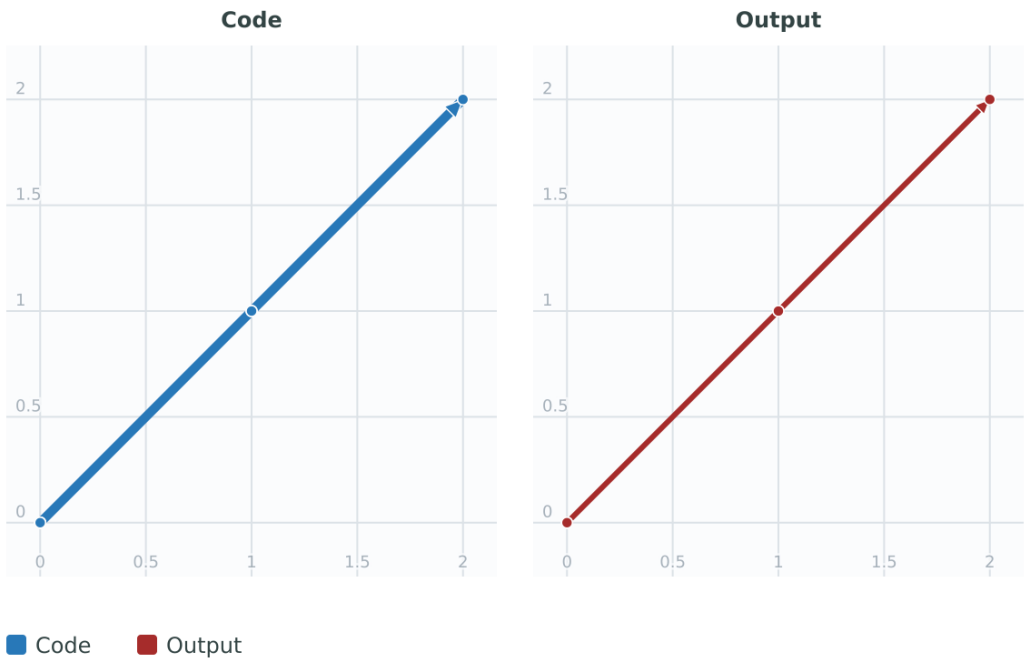
Code

```
SELECT ST_RemoveRepeatedPoints('LINESTRING (0 0,0 0,1 1,0 0,1 1,2 2)');
```

Output

```
LINESTRING(0 0,1 1,0 0,1 1,2 2)
```

Figure



Example: Collection elements are processed individually.

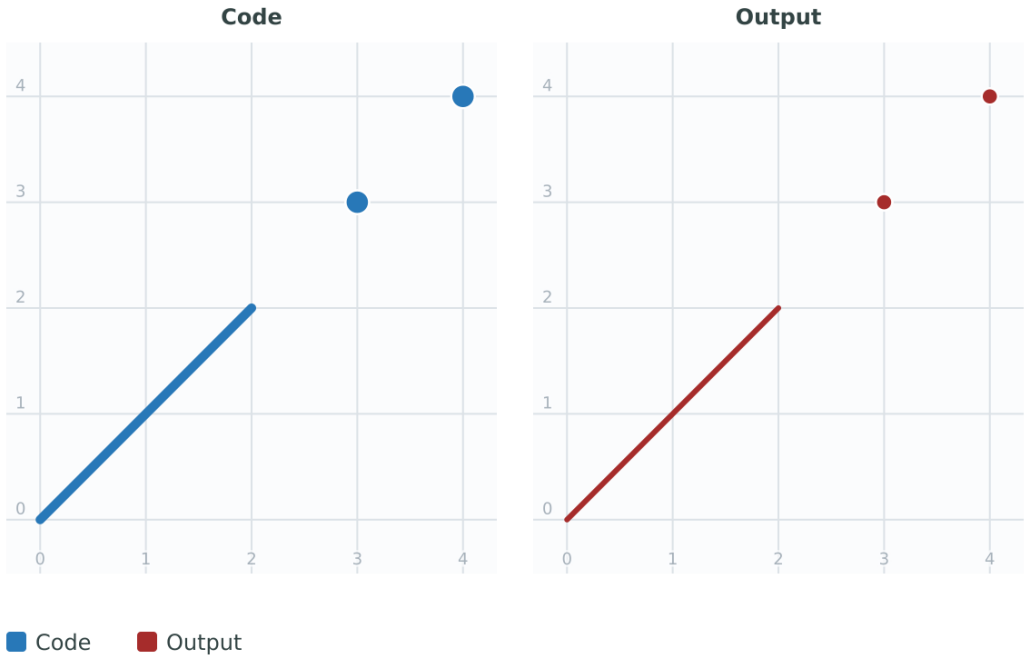
Code

```
SELECT ST_RemoveRepeatedPoints('GEOMETRYCOLLECTION (LINESTRING (1 1,2 2,2 2,3 3),POINT (4 ↩
    4),POINT (4 4),POINT (5 5))');
```

Output

```
GEOMETRYCOLLECTION(LINESTRING(1 1,2 2,3 3), POINT(4 4), POINT(4 4), POINT(5 5))
```

Figure



Example: Repeated point removal with a distance tolerance.

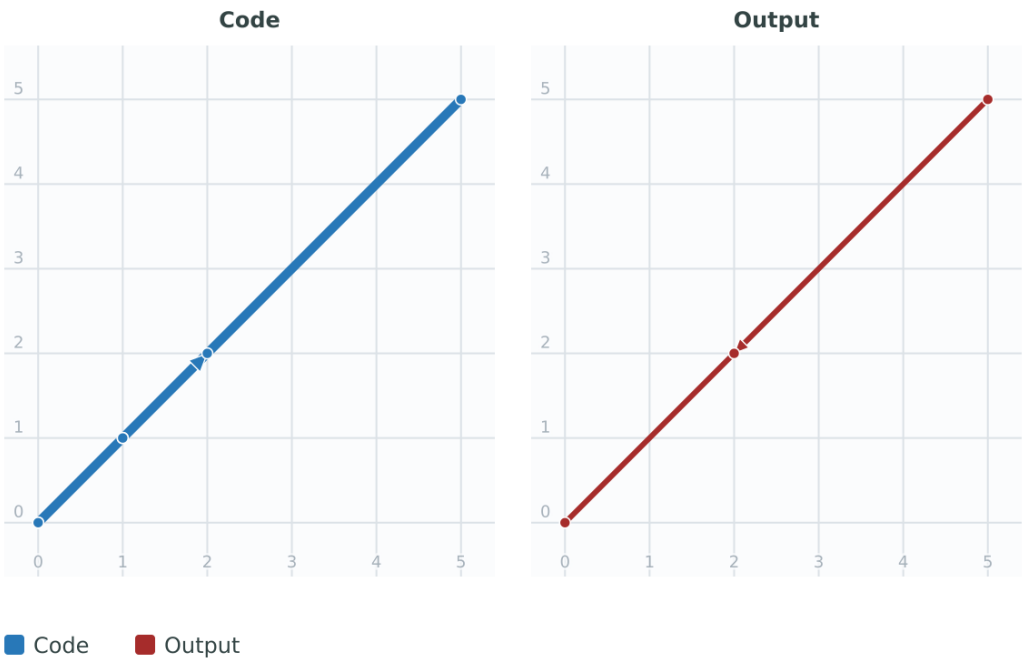
Code

```
SELECT ST_RemoveRepeatedPoints('LINESTRING (0 0,0 0,1 1,5 5,1 1,2 2)', 2);
```

Output

```
LINESTRING(0 0,5 5,2 2)
```

Figure



☒☒

ST_Simplify

7.5.26 ST_RemoveIrrelevantPointsForView

ST_RemoveIrrelevantPointsForView — Removes points that are irrelevant for rendering a specific rectangular view of a geometry.

☒☒

geometry **ST_RemoveIrrelevantPointsForView**(geometry geom, box2d bounds, boolean cartesian_hint = false);

☒☒

Returns a geometry without points being irrelevant for rendering the geometry within a given rectangular view.

This function can be used to quickly preprocess geometries that should be rendered only within certain bounds.

Only geometries of type (MULTI)POLYGON and (MULTI)LINESTRING are evaluated. Other geometries keep unchanged.

In contrast to `ST_ClipByBox2D()` this function

- sorts out points without computing new intersection points which avoids rounding errors and usually increases performance,

- returns a geometry with equal or similar point number,
- leads to the same rendering result within the specified view, and
- may introduce self-intersections which would make the resulting geometry invalid (see example below).

If `cartesian_hint` is set to `true`, the algorithm applies additional optimizations involving cartesian math to further reduce the resulting point number. Please note that using this option might introduce rendering artifacts if the resulting coordinates are projected into another (non-cartesian) coordinate system before rendering.



☒☒
For polygons, this function does currently not ensure that the result is valid. This situation can be checked with ST_IsValid and repaired with ST_MakeValid.

Availability: 3.5.0

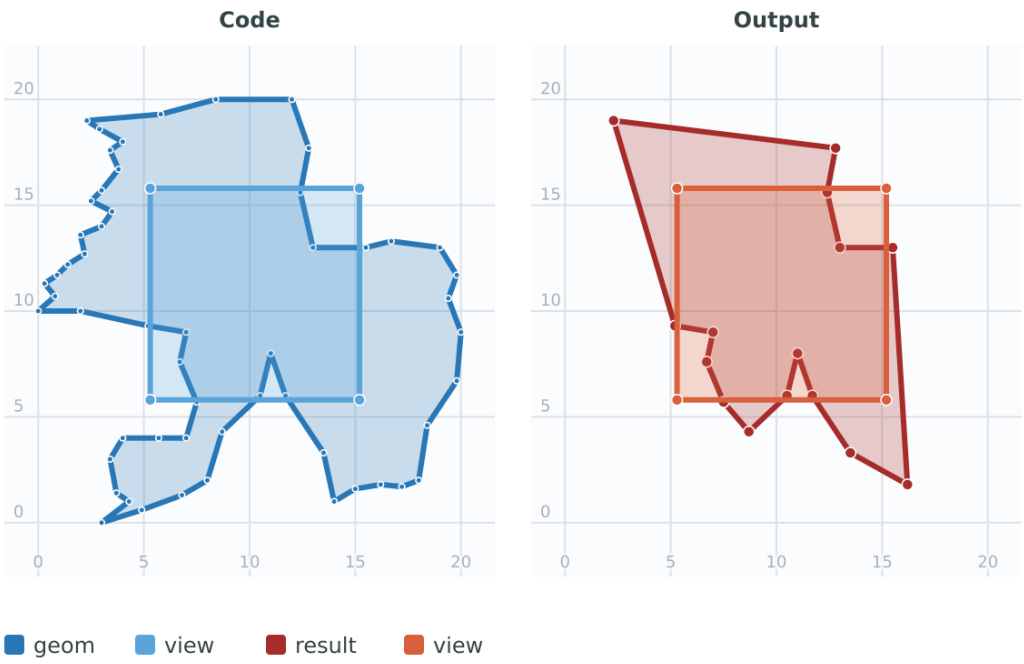
☒☒

The input polygon has 55 points. The result retains 15 points, while rendering the same shape inside the view envelope.

Code

```
WITH data(geom, view) AS (VALUES (
  'POLYGON((3.2 14.2,6.8 14.2,7.6 11.9,7.2 9.8,7.8 7.2,10.3 7.2,11.5 7.5,13.8 7.2,14.6 ↩
   5.9,14.2 4.8,14.8 3.2,14.6 0.9,13.2 -1.2,12.8 -3.8,12 -4.1,11 -4,9.8 -4.2,8.8 -4.8,8.3 ↩
   -2.5,6.5 0.2,5.8 2.2,5.3 0.2,3.5 -1.5,2.8 -3.8,1.6 -4.5,-0.3 -5.2,-2.2 -5.8,-0.9 ↩
   -4.8,-1.5 -4.4,-1.8 -2.8,-1.2 -1.8,0.5 -1.8,1.8 -1.8,2.3 -0.1,1.5 1.8,1.8 3.2,0 3.5,-3.2 ↩
    4.2,-5.2 4.2,-4.4 4.9,-4.9 5.5,-4.3 5.9,-3.8 6.4,-3 6.9,-3.2 7.8,-2.2 8.2,-1.7 8.9,-2.7 ↩
    9.4,-2.2 9.9,-1.4 10.9,-1.8 11.8,-1.2 12.2,-2.3 12.8,-2.9 13.2,0.6 13.5,3.2 14.2))'':: ↩
   geometry,
  ST_MakeEnvelope(0.1, 0, 10, 10)
))
SELECT geom AS input_geom,
       view AS input_view,
       ST_RemoveIrrelevantPointsForView(geom, view, true) AS result,
       view AS view
FROM data;
```

Figure

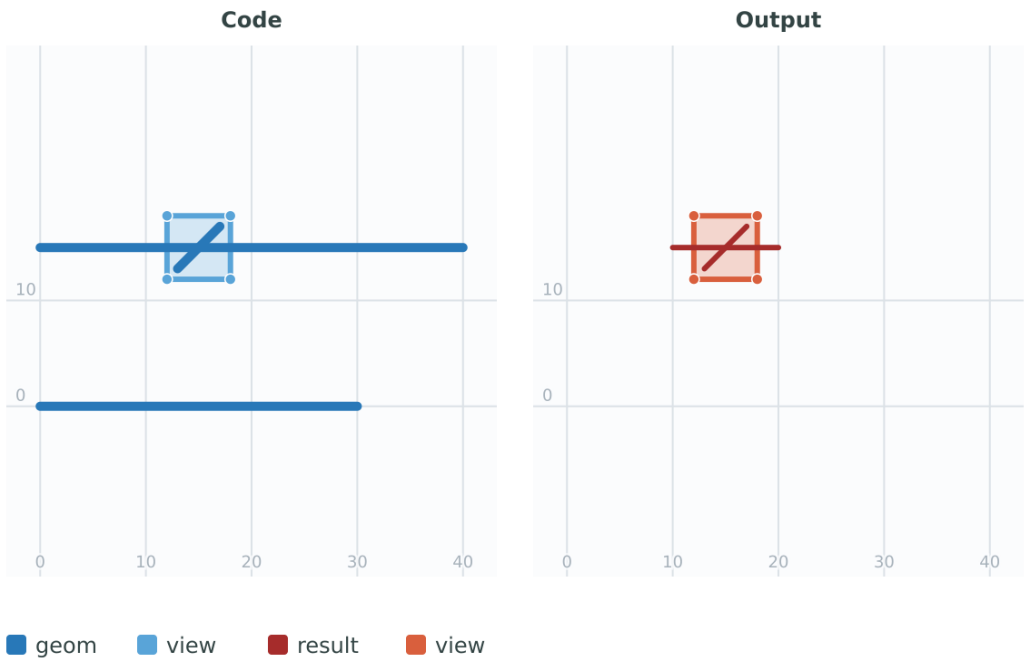


Only linework affecting the view is retained; existing vertices are reused.

Code

```
WITH data(geom, view) AS (VALUES (
  'MULTILINESTRING((0 0,10 0,20 0,30 0),(0 15,5 15,10 15,15 15,20 15,25 15,30 15,40 15),(13 ↩
     13,15 15,17 17))'::geometry,
  ST_MakeEnvelope(12, 12, 18, 18)
))
SELECT geom AS input_geom,
       view AS input_view,
       ST_RemoveIrrelevantPointsForView(geom, view, true) AS result,
       view AS view
FROM data;
```

Figure



A line with no segment relevant to the view becomes empty.

Code

```
SELECT ST_RemoveIrrelevantPointsForView('LINESTRING(0 0,10 0,20 0,30 0)',
ST_MakeEnvelope(12, 12, 18, 18), true);
```

Output

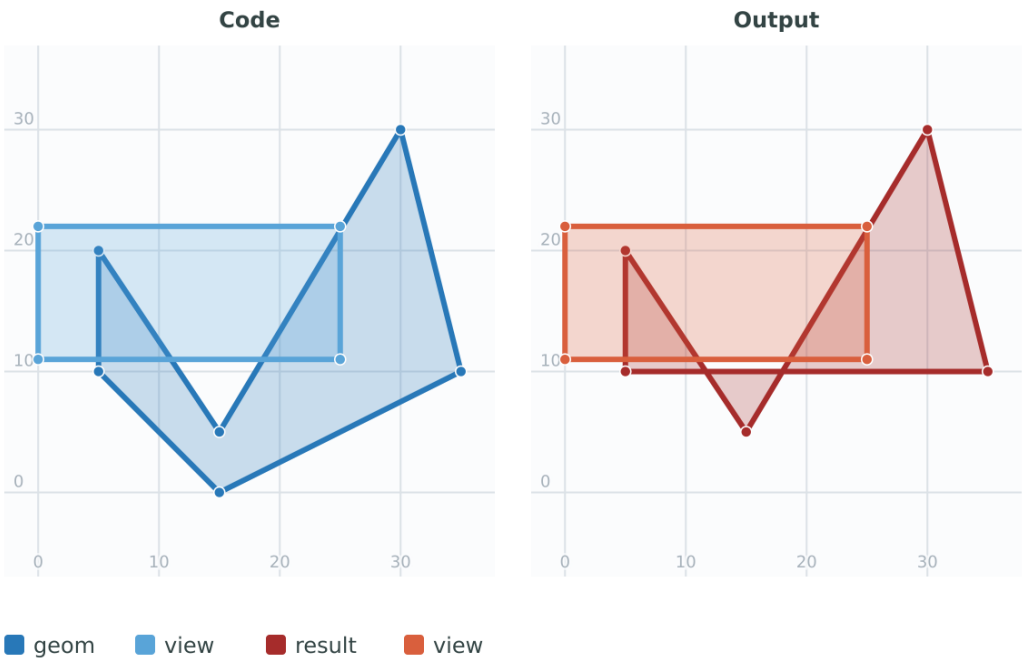
```
LINESTRING
```

Removing vertices without computing intersections can introduce self-intersections in the result.

Code

```
WITH data(geom, view) AS (VALUES (
  'POLYGON((0 10,0 0,10 -10,30 0,25 20,10 -5,0 10))'::geometry,
  ST_MakeEnvelope(-5, 1, 20, 12)
))
SELECT geom AS input_geom,
       view AS input_view,
       ST_RemoveIrrelevantPointsForView(geom, view, true) AS result,
       view AS view
FROM data;
```

Figure

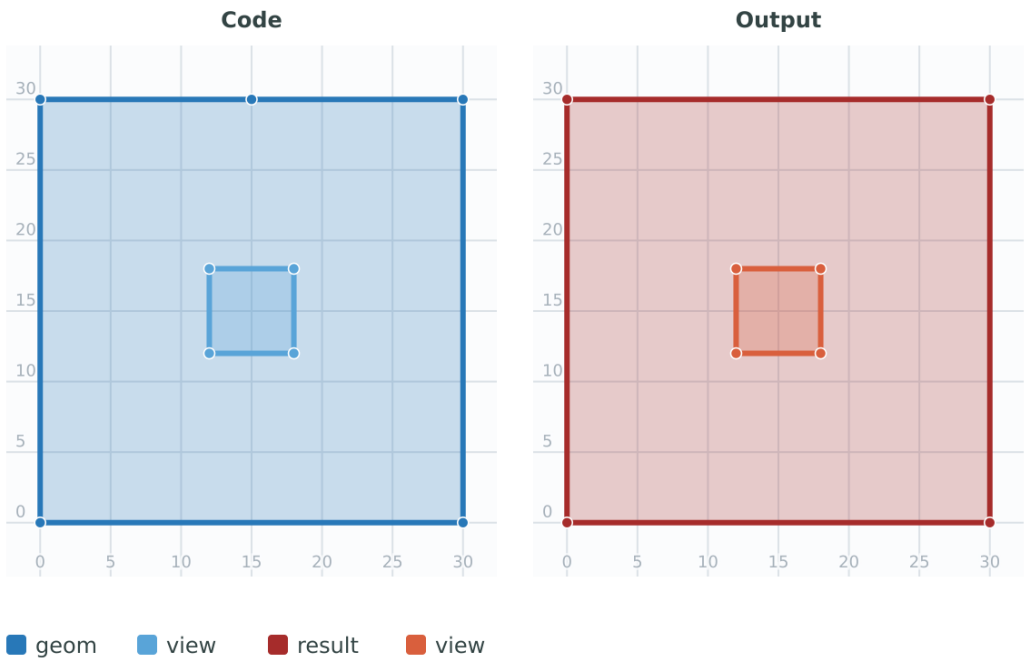


With the default *cartesian_hint*, the function removes only points known to be irrelevant to the view.

Code

```
WITH data(geom, view) AS (VALUES (
  'POLYGON((0 30,15 30,30 30,30 0,0 0,0 30))'::geometry,
  ST_MakeEnvelope(12, 12, 18, 18)
))
SELECT geom AS input_geom,
       view AS input_view,
       ST_RemoveIrrelevantPointsForView(geom, view) AS result,
       view AS view
FROM data;
```

Figure



☒☒

ST_ClipByBox2D, ST_Intersection

7.5.27 ST_RemoveSmallParts

ST_RemoveSmallParts — Removes small parts (polygon rings or linestrings) of a geometry.

☒☒

geometry **ST_RemoveSmallParts**(geometry geom, double precision minSizeX, double precision minSizeY);

☒☒

Returns a geometry without small parts (exterior or interior polygon rings, or linestrings).

This function can be used as preprocessing step for creating simplified maps, e. g. to remove small islands or holes.

It evaluates only geometries of type (MULTI)POLYGON and (MULTI)LINESTRING. Other geometries remain unchanged.

If *minSizeX* is greater than 0, parts are sorted out if their width is smaller than *minSizeX*.

If *minSizeY* is greater than 0, parts are sorted out if their height is smaller than *minSizeY*.

Both *minSizeX* and *minSizeY* are measured in coordinate system units of the geometry.

For polygon types, evaluation is done separately for each ring which can lead to one of the following results:

- the original geometry,
- a POLYGON with all rings with less vertices,
- a POLYGON with a reduced number of interior rings (having possibly less vertices),
- a POLYGON EMPTY, or
- a MULTIPOLYGON with a reduced number of polygons (having possibly less interior rings or vertices), or
- a MULTIPOLYGON EMPTY.

For linestring types, evaluation is done for each linestring which can lead to one of the following results:

- the original geometry,
- a LINESTRING with a reduced number of vertices,
- a LINESTRING EMPTY,
- a MULTILINESTRING with a reduced number of linestrings (having possibly less vertices), or
- a MULTILINESTRING EMPTY.

Availability: 3.5.0

□□

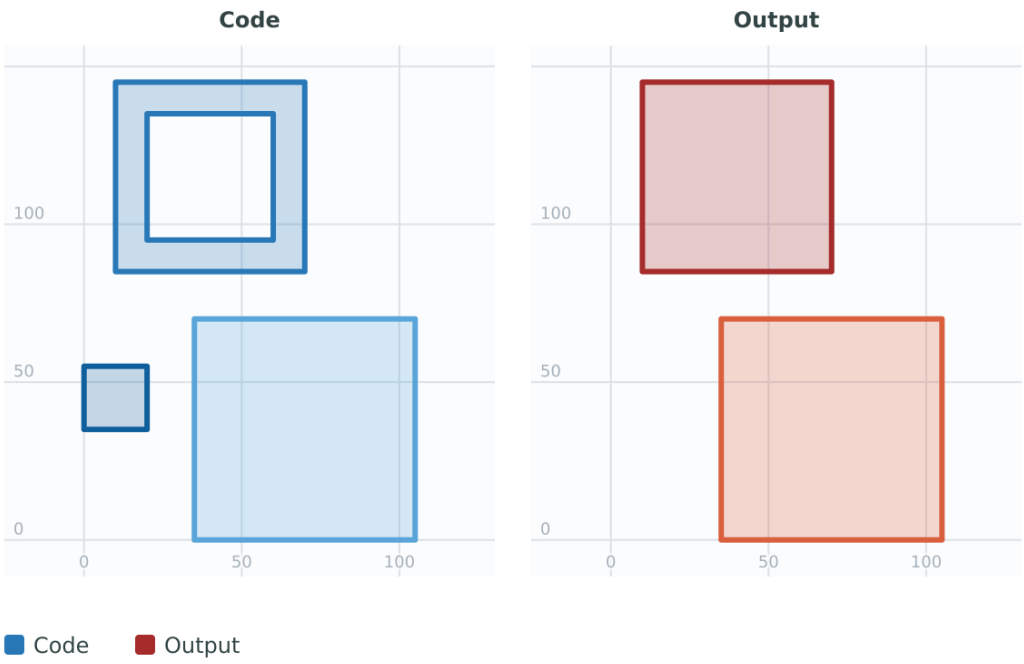
Code

```
SELECT ST_RemoveSmallParts('MULTIPOLYGON(
    ((60 160,120 160,120 220,60 220,60 160),(70 170,70 210,110 210,110 170,70 170)),
    ((85 75,155 75,155 145,85 145,85 75)),
    ((50 110,70 110,70 130,50 130,50 110)))',
    50, 50);
```

Output

```
MULTIPOLYGON(((60 160,120 160,120 220,60 220,60 160)),((85 75,155 75,155 145,85 145,85 75)))
```

Figure



Code

```
SELECT ST_RemoveSmallParts('LINESTRING(10 10,20 20)',
    50, 50);
```

Output

```
LINESTRING
```

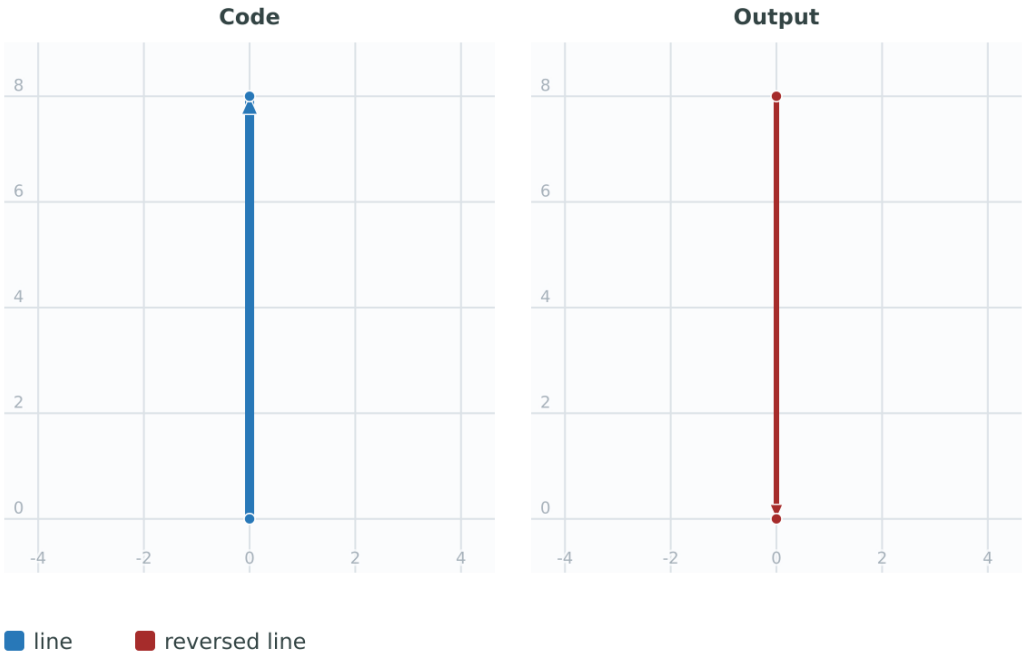
7.5.28 ST_Reverse

ST_Reverse — ⊠⊠⊠⊠⊠⊠

Code

```
WITH data(geom) AS (VALUES (
  ST_MakeLine(ST_Point(1, 2), ST_Point(1, 10))
))
SELECT geom AS input_line,
       ST_Reverse(geom) AS reversed_line
FROM data;
```

Figure



7.5.29 ST_Segmentize

ST_Segmentize — Returns a modified geometry/geography having no segment longer than a given distance.

☒☒

geometry **ST_Segmentize**(geometry geom, float max_segment_length);
geography **ST_Segmentize**(geography geog, float max_segment_length);

☒☒

Returns a modified geometry/geography having no segment longer than `max_segment_length`. Length is computed in 2D. Segments are always split into equal-length subsegments.

- For geometry, the maximum length is in the units of the spatial reference system.
- For geography, the maximum length is in meters. Distances are computed on the sphere. Added vertices are created along the spherical great-circle arcs defined by segment endpoints.



□□
This only shortens long segments. It does not lengthen segments shorter than the maximum length.



□□
For inputs containing long segments, specifying a relatively short `max_segment_length` can cause a very large number of vertices to be added. This can happen unintentionally if the argument is specified accidentally as a number of segments, rather than a maximum length.

1.2.2 □□□□□□□□□□□□.

Enhanced: 3.0.0 Segmentize geometry now produces equal-length subsegments

Enhanced: 2.3.0 Segmentize geography now produces equal-length subsegments

□□□□: 2.1.0 □□□□□□□□□□□□□□□□□□□□.

Changed: 2.1.0 As a result of the introduction of geography support, the usage `ST_Segmentize('LINESTRING 2, 3 4)', 0.5)` causes an ambiguous function error. The input needs to be properly typed as a geometry or geography. Use ST_GeomFromText, ST_GeogFromText or a cast to the required type (e.g. `ST_Segmentize('LINESTRING(1 2, 3 4)'::geometry, 0.5) )`

□□

Segmentizing a line. Long segments are split evenly, and short segments are not split.

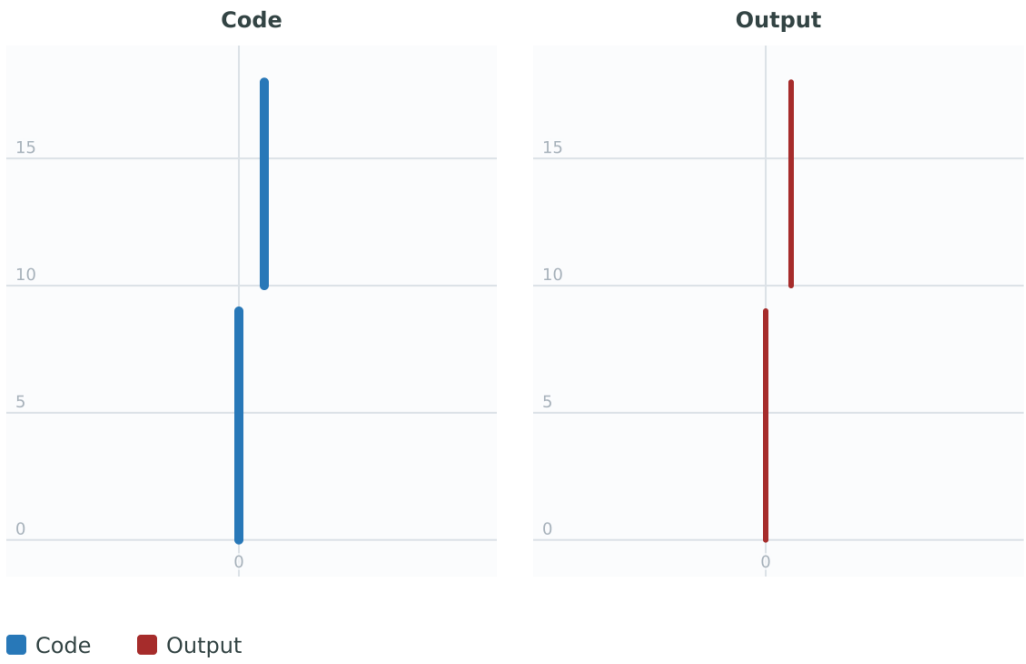
Code

```
SELECT ST_Segmentize('MULTILINESTRING((0 0,0 1,0 9),(1 10,1 18))'::geometry,
5);
```

Output

```
MULTILINESTRING((0 0,0 1,0 5,0 9),(1 10,1 14,1 18))
```

Figure



Segmentizing a polygon:

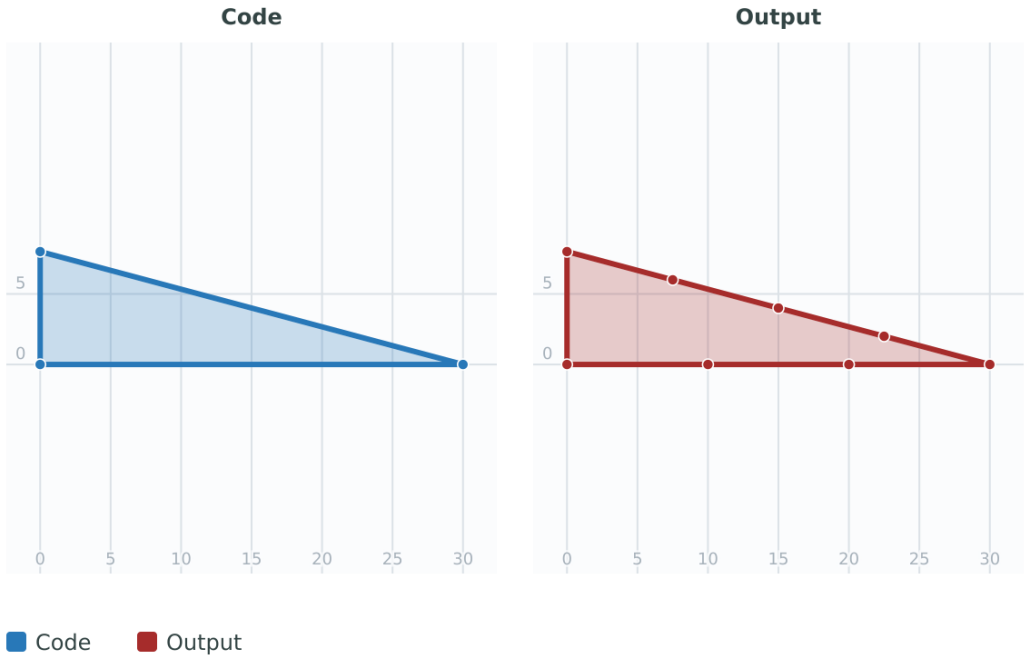
Code

```
SELECT ST_Segmentize('POLYGON((0 0,0 8,30 0,0 0))'::geometry,
        10);
```

Output

```
POLYGON((0 0,0 8,7.5 6,15 4,22.5 2,30 0,20 0,10 0,0 0))
```

Figure



Segmentizing a geographic line, using a maximum segment length of 2000 kilometers. Vertices are added along the great-circle arc connecting the endpoints.

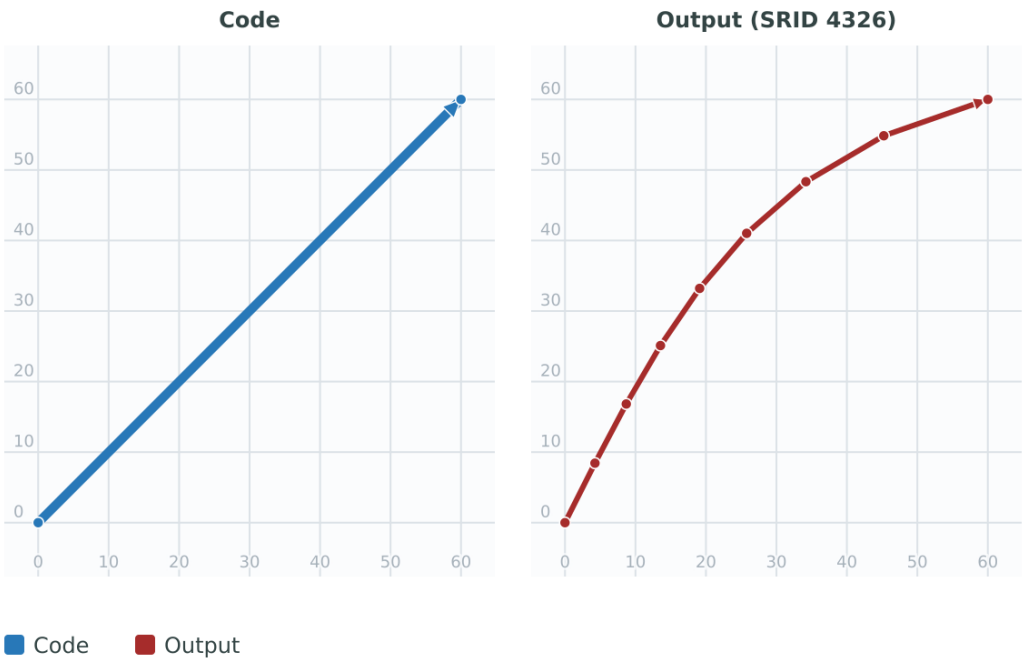
Code

```
SELECT ST_Segmentize(('LINESTRING (0 0,60 60)'::geography), 2000000);
```

Output

```
LINESTRING(0 0,4.252632294621186 8.43596525986862,8.69579947419404 16.824093489701564,13.550465473227048 ↩
    25.107950473646188,19.1066053508691 33.21091076089908,25.779290201459894 41.01711439406505,34.188839517966954 ↩
    48.337222885886,45.238153936612264 54.84733442373889,60 60)
```

Figure



☒☒

ST_LineSubstring

7.5.30 ST_SetPoint

ST_SetPoint — ☒☒☒☒☒☒

1.1.0 □□□□□□□□□□□□.
□□□□□□: 2.3.0 □□□□□□□□□□□□□□□□□□□□□□□□.

This function supports 3d and will not drop the z-index.

□□

Change first point in line string from -1 3 to -1 1.

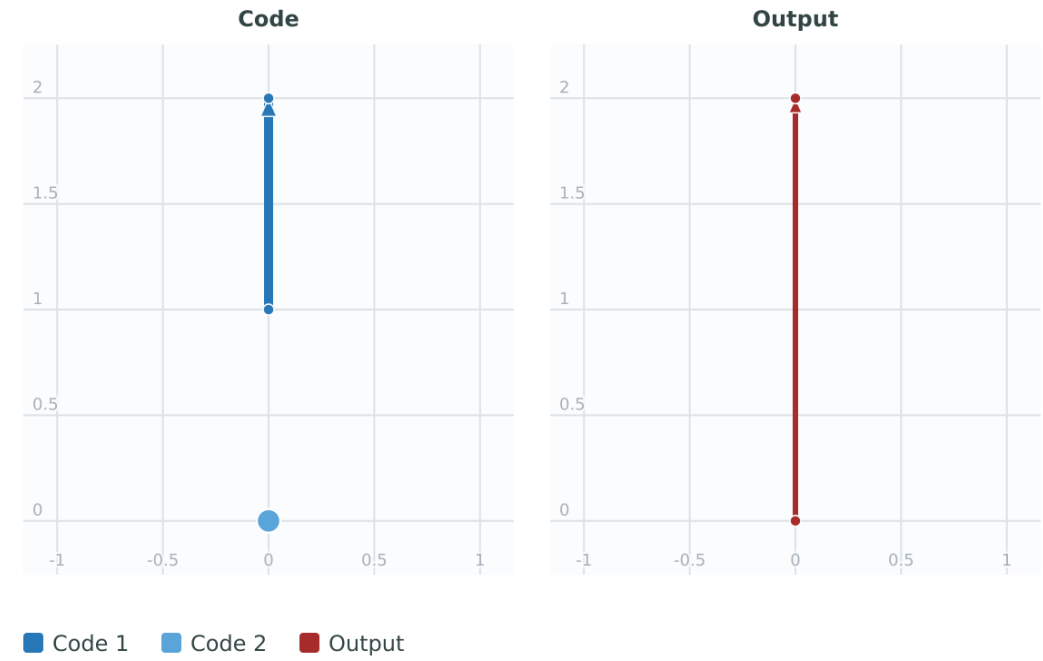
Code

```
SELECT ST_SetPoint('LINESTRING(-1 2,-1 3)', 0, 'POINT(-1 1)');
```

Output

```
LINESTRING(-1 1,-1 3)
```

Figure



Change last point in a line string (let's play with 3D linestring this time)

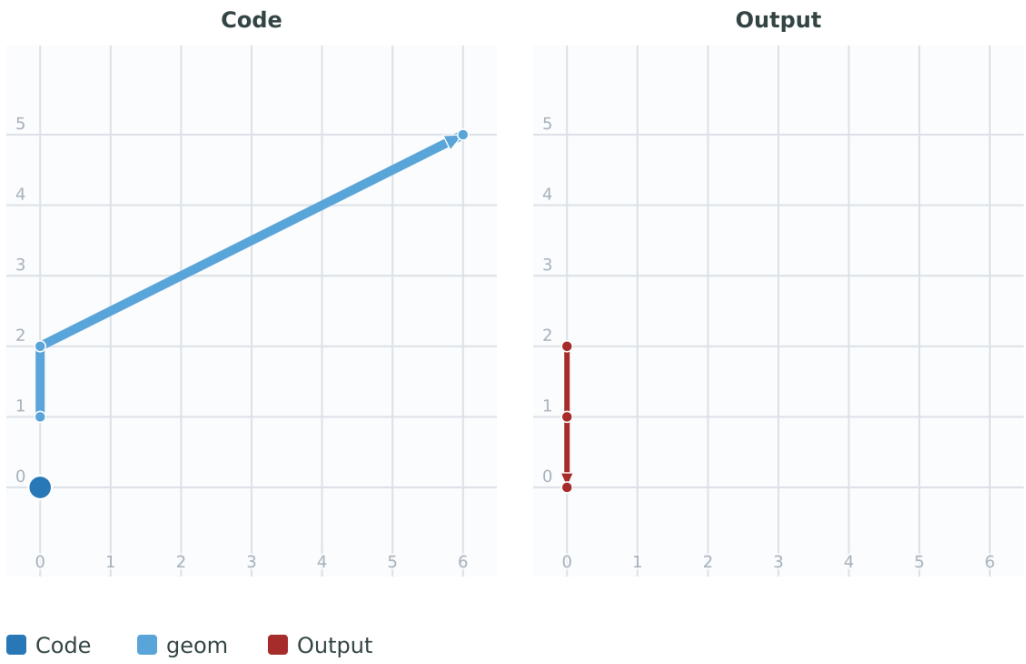
Code

```
SELECT ST_SetPoint(foo.geom, ST_NumPoints(foo.geom) - 1, 'POINT(-1 1 3)')
FROM (SELECT 'LINESTRING(-1 2 3,-1 3 4,5 6 7)'::geometry As geom) As foo;
```

Output

```
LINESTRING(-1 2 3,-1 3 4,-1 1 3)
```

Figure



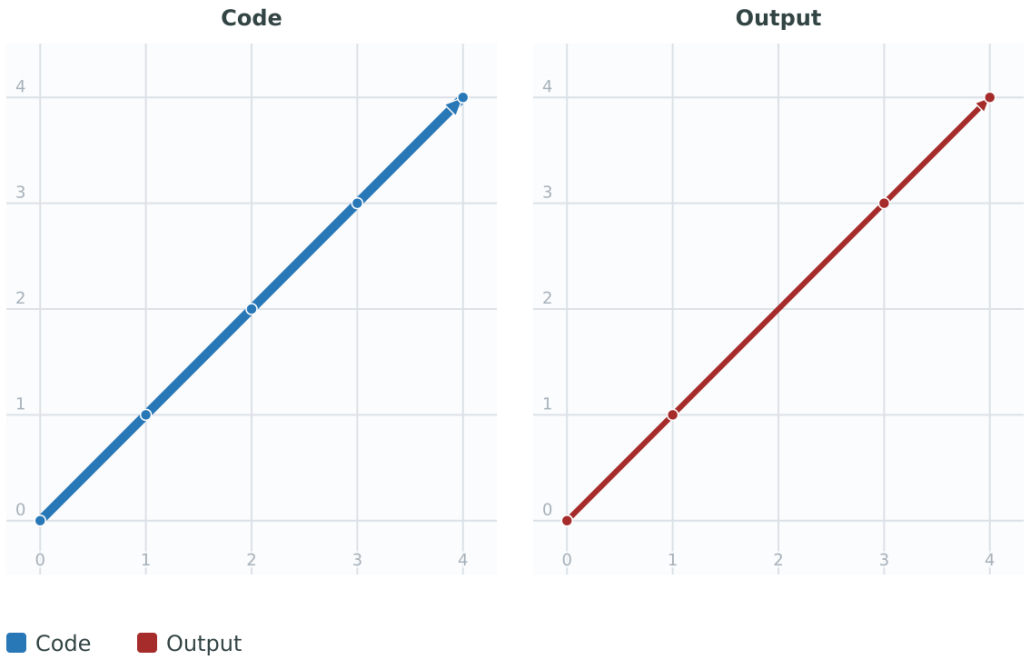
Code

```
SELECT ST_SetPoint(g, -3, p)
FROM ST_GeomFromText('LINESTRING(0 0,1 1,2 2,3 3,4 4)') AS g
    CROSS JOIN ST_PointN(g, 1) as p;
```

Output

```
LINESTRING(0 0,1 1,0 0,3 3,4 4)
```

Figure



□□

ST_AddPoint, ST_NPoints, ST_NumPoints, ST_PointN, ST_RemovePoint

7.5.31 ST_ShiftLongitude

ST_ShiftLongitude — Shifts the longitude coordinates of a geometry between -180..180 and 0..360.

□□

geometry **ST_ShiftLongitude**(geometry geom);

□□

Reads every point/vertex in a geometry, and shifts its longitude coordinate from -180..0 to 180..360 and vice versa if between these ranges. This function is symmetrical so the result is a 0..360 representation of a -180..180 data and a -180..180 representation of a 0..360 data.



□□
This is only useful for data with coordinates in longitude/latitude; e.g. SRID 4326 (WGS 84 geographic)



□□
1.3.4 □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□. 1.3.4 □□□□□□□□□□□□□□□□□□□□□□.

This function supports 3d and will not drop the z-index.

□□□□: 2.0.0 □□□□□□□□□□ (polyhedral surface) □ TIN □□□□□□.

□□: 2.2.0 □□□□□□□, □□□□□□□□"ST_Shift_Longitude" □□□□.

This function supports Polyhedral surfaces.

This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).

□□

This example performs a forward longitude shift on a single point.

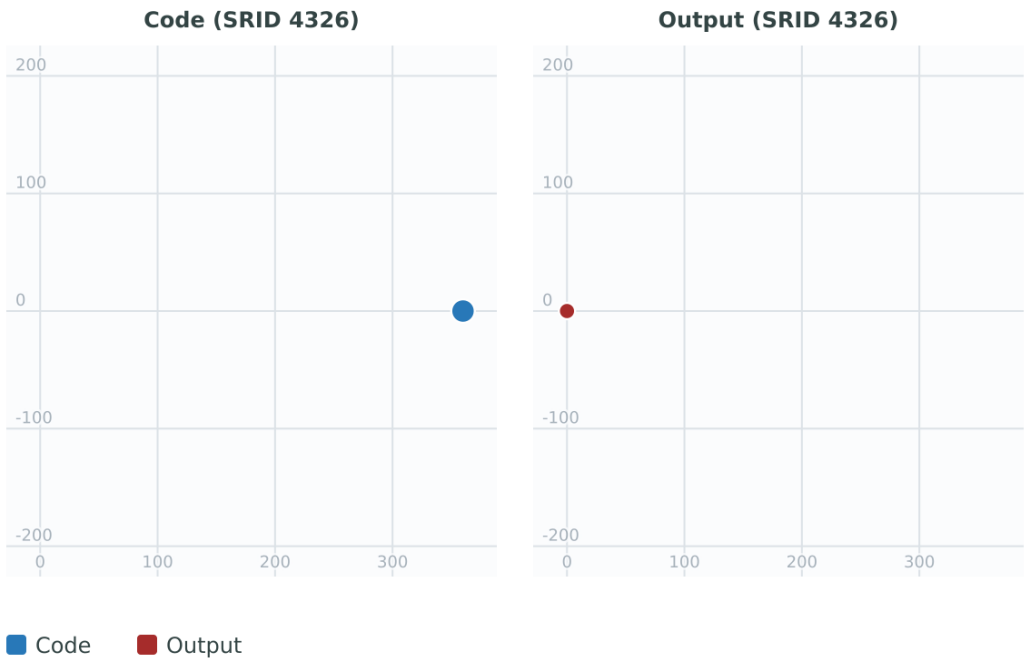
Code

```
SELECT ST_ShiftLongitude('SRID=4326;POINT(270 0)'::geometry)
```

Output

```
POINT(-90 0)
```

Figure



This example performs a reverse longitude shift on a single point.

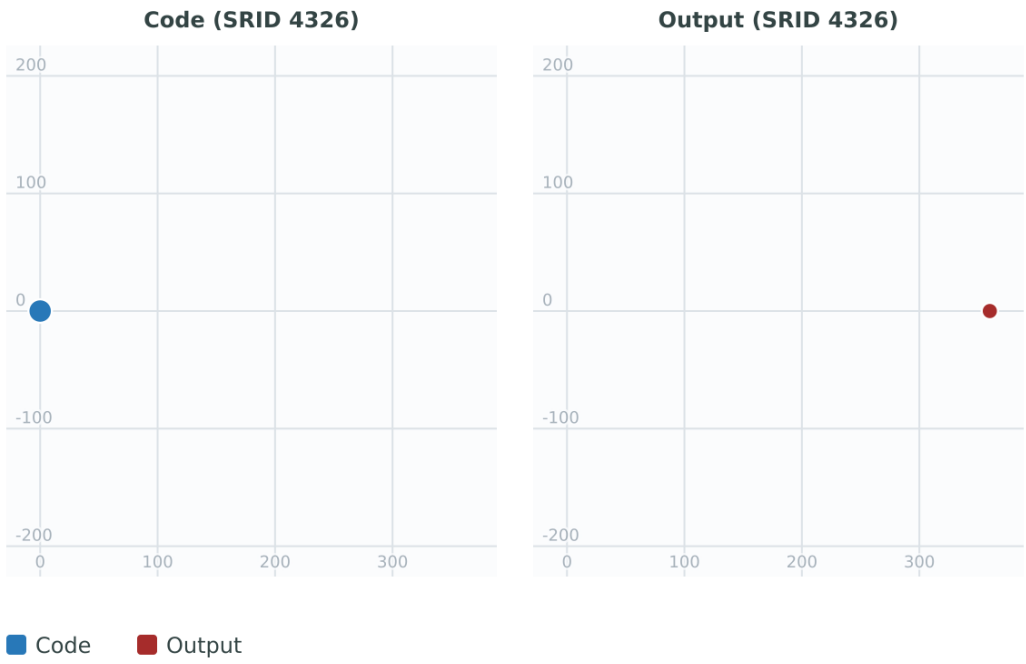
Code

```
SELECT ST_ShiftLongitude('SRID=4326;POINT(-90 0)'::geometry)
```

Output

```
POINT(270 0)
```

Figure



This example shows that LineStrings are shifted only where coordinates require it.

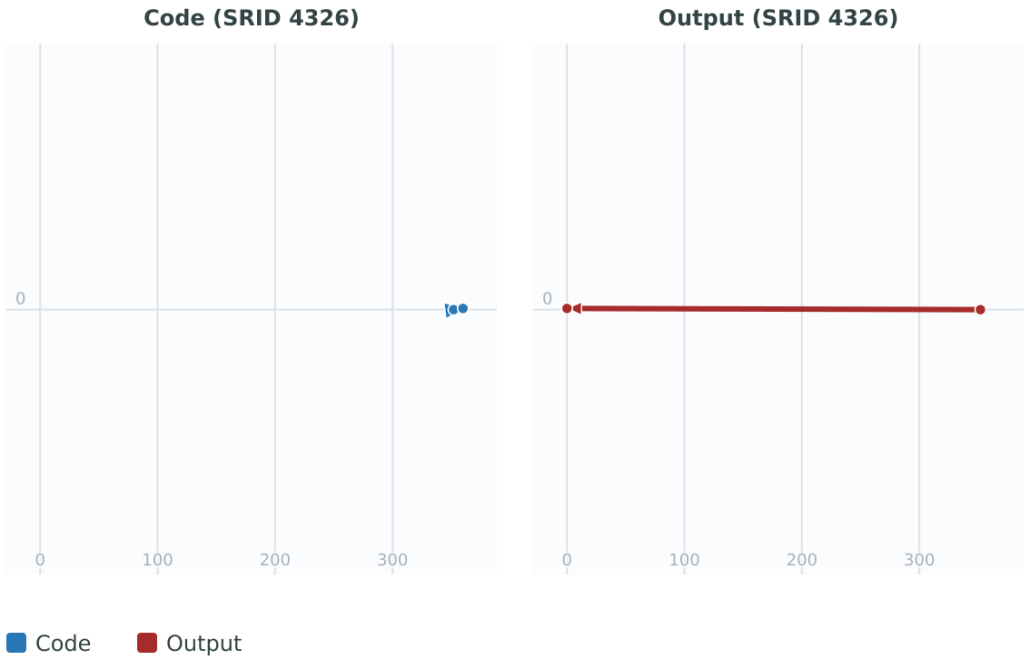
Code

```
SELECT ST_ShiftLongitude('SRID=4326;LINESTRING(174 12,182 13)'::geometry)
```

Output

```
LINESTRING(174 12,-178 13)
```

Figure



⊠⊠

ST_WrapX

7.5.32 ST_WrapX

ST_WrapX — X ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

geometry **ST_WrapX**(geometry geom, float8 wrap, float8 move);

⊠⊠

This function splits the input geometries and then moves every resulting component falling on the right (for negative 'move') or on the left (for positive 'move') of given 'wrap' line in the direction specified by the 'move' parameter, finally re-unioning the pieces together.

> **Note**
> ⊠⊠
> ⊠⊠.

Availability: 2.3.0 requires GEOS

This function supports 3d and will not drop the z-index.

⊠⊠

Move all components whose bounding boxes fall completely to the left of `x = 0` by +360.

Code

```
SELECT ST_WrapX(geom, 0, 360);
```

Move all components whose bounding boxes fall completely to the left of `x = -30` by +360.

Code

```
SELECT ST_WrapX(geom, -30, 360);
```

⊠⊠

ST_ShiftLongitude

7.5.33 ST_SnapToGrid

ST_SnapToGrid — ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (snap) ⊠⊠⊠⊠.

□□

geometry **ST_SnapToGrid**(geometry geomA, float originX, float originY, float sizeX, float sizeY);
geometry **ST_SnapToGrid**(geometry geomA, float sizeX, float sizeY);
geometry **ST_SnapToGrid**(geometry geomA, float size);
geometry **ST_SnapToGrid**(geometry geomA, geometry pointOrigin, float sizeX, float sizeY, float sizeZ, float sizeM);

□□

□□ 1, 2, 3: □□□□□□□□□□□□□□□ (cell) □□□□□□□□□□□□ (snap) □□□□. □□□□□□□□□□□□□□□□□□□□□, □□□□□□□□□□□□□□□□□□□□□□□□□□□□ NULL □□□□□□. □□□□□□□□□□□□□□□□□□□□□□□□□. □□□□□□□□□□□□□.

□□ 4: 1.1.0 □□□□□□□□□□□. □□□□□□□□□□□□□ (□□□□□, □□□□□□□□) □□□□□□□□□□□□□□□□□□. □□□□□□□□□□□□□□□□□□□□□□, □□□□□ 0 □□□□□□□□.

> **Note**
> □□
> □□□□□□□□□□□□□□□□□□□ (ST_IsSimple □□).

> **Note**
> □□
> 1.1.0 □□□□□□□□□□□□□ 2 □□□□□□□□□□□. 1.1.0 □□□□□□□□□□□, □□□□□□□□□□□□□, □□□□□□□□□□□□□□□□□□□. □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□.

1.0.0RC1 □□□□□□□□□□□□.

1.1.0 □□□□ Z □ M □□□□□□.

This function supports 3d and will not drop the z-index.

□□

Snap a geometry column to a precision grid of 10^-3.

Code

```
UPDATE mytable
   SET geom = ST_SnapToGrid(geom, 0.001);
```

Display how the same grid size rounds a LineString.

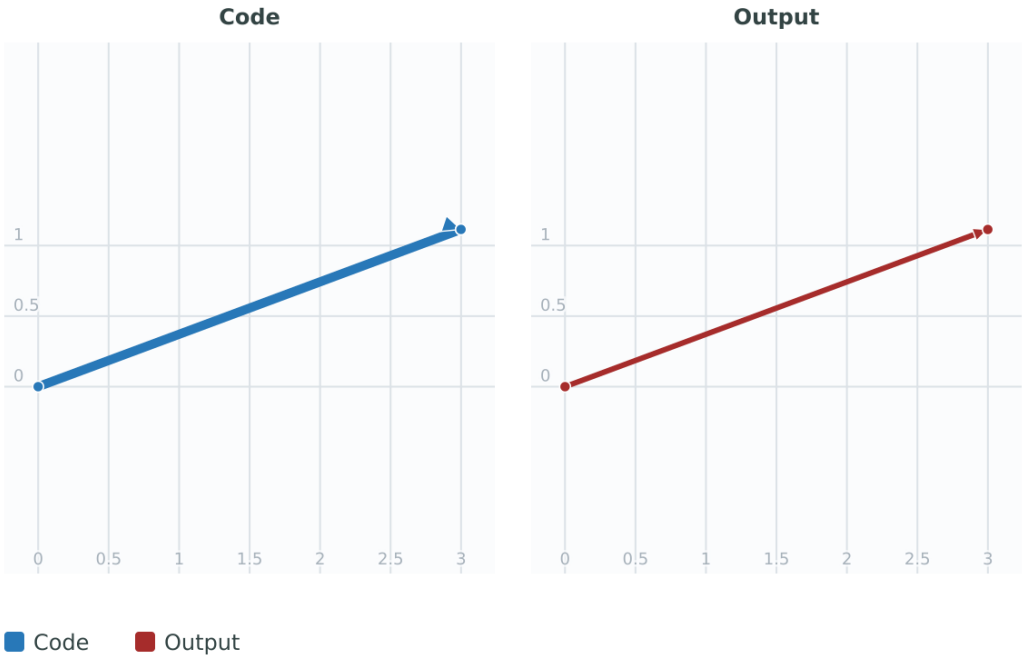
Code

```
SELECT ST_SnapToGrid('LINESTRING(1.1115678 2.123,4.111111 3.2374897,4.11112 3.23748667)':: ↩
    geometry,
        0.001);
```

Output

```
LINESTRING(1.112 2.123,4.111 3.237)
```

Figure



This example snaps a 4D geometry.

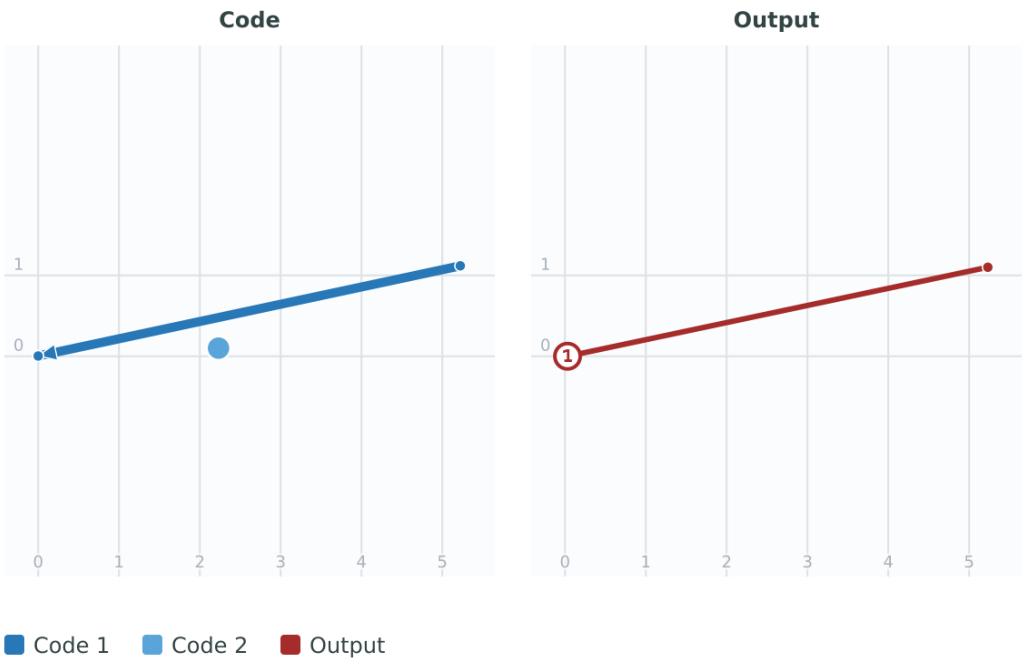
Code

```
SELECT ST_SnapToGrid('LINESTRING(-1.1115678 2.123 2.3456 1.11111,
        4.111111 3.2374897 3.1234 1.1111,-1.11111112 2.123 2.3456 1.1111112)'::geometry,
        'POINT(1.12 2.22 3.2 4.4444)',
        0.1, 0.1, 0.1, 0.01);
```

Output

```
LINESTRING(-1.08 2.12 2.3 1.1144,4.12 3.22 3.1 1.1144,-1.08 2.12 2.3 1.1144)
```

Figure



With a 4D geometry - the ST_SnapToGrid(geom,size) only touches x and y coords but keeps m and z the same.

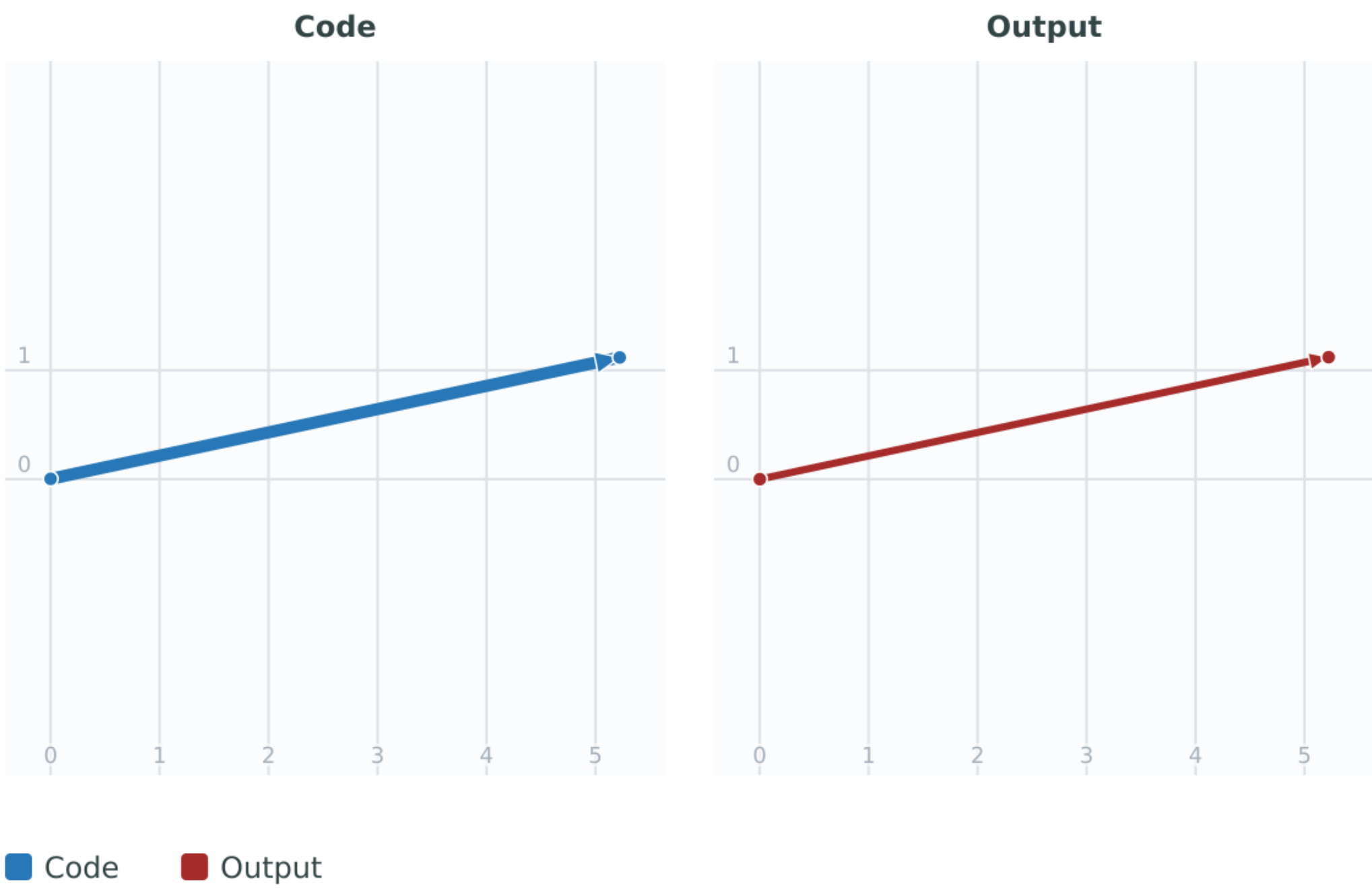
Code

```
SELECT ST_SnapToGrid('LINESTRING(-1.1115678 2.123 3 2.3456,
        4.111111 3.2374897 3.1234 1.1111)'::geometry,
       0.01);
```

Output

```
LINESTRING(-1.11 2.12 3 2.3456,4.11 3.24 3.1234 1.1111)
```

Figure



⊠⊠

ST_Snap, ST_AsEWKT, ST_AsText, ST_GeomFromText, ST_GeomFromEWKT, ST_Simplify

7.5.34 ST_Snap

ST_Snap — ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

geometry **ST_Snap**(geometry input, geometry reference, float tolerance);

⊠⊠

Snaps the vertices and segments of a geometry to another Geometry's vertices. A snap distance tolerance is used to control where snapping is performed. The result geometry is the input geometry with the vertices snapped. If no snapping occurs then the input geometry is returned unchanged.

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, (⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠ (heuristics) ⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

> **Note!**
> ⊠⊠
> ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (ST_IsSimple ⊠⊠) ⊠⊠⊠⊠ (ST_IsValid ⊠⊠) ⊠⊠⊠⊠⊠⊠⊠⊠⊠.

GEOS □□□□□
2.0.0 □□□□□□□□□□□□.

□□

A MultiPolygon snapped to a LineString at 1.01 times their distance.

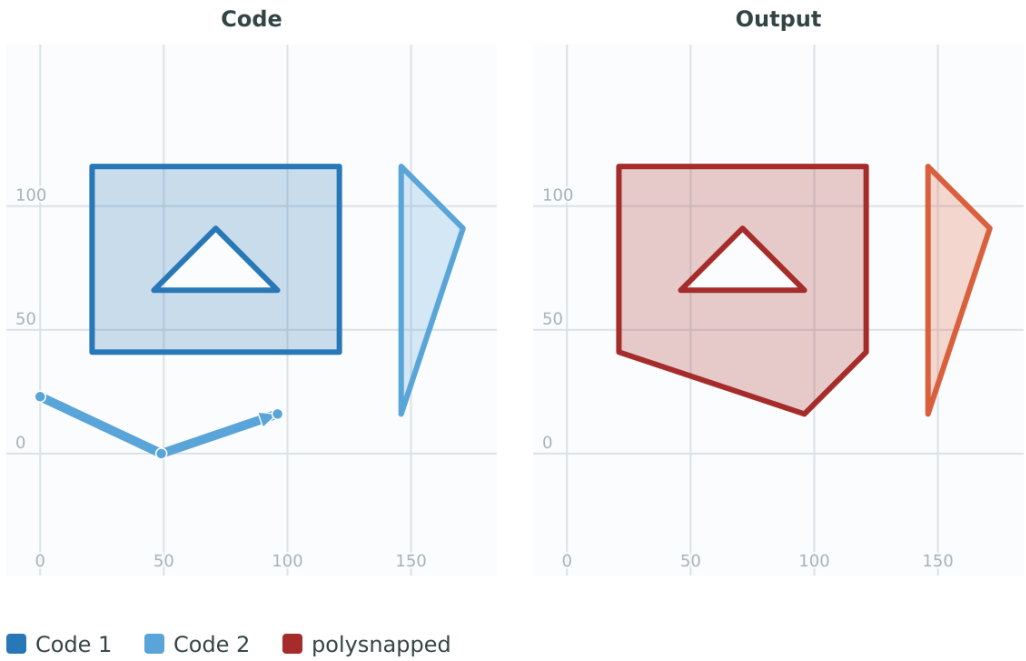
Code

```
SELECT ST_Snap(poly, line, ST_Distance(poly, line)*1.01) AS polysnapped
FROM (SELECT
   ST_GeomFromText('MULTIPOLYGON(
     ((26 125,26 200,126 200,126 125,26 125 ),
      ( 51 150,101 150,76 175,51 150 )),
      (( 151 100,151 200,176 175,151 100 )))') As poly,
       ST_GeomFromText('LINESTRING (5 107,54 84,101 100)') As line
    ) As foo;
```

Output

```
MULTIPOLYGON(((26 125,26 200,126 200,126 125,101 100,26 125),
(51 150,101 150,76 175,51 150)),((151 100,151 200,176 175,151 100)))
```

Figure



A MultiPolygon snapped to a LineString at 1.25 times their distance.

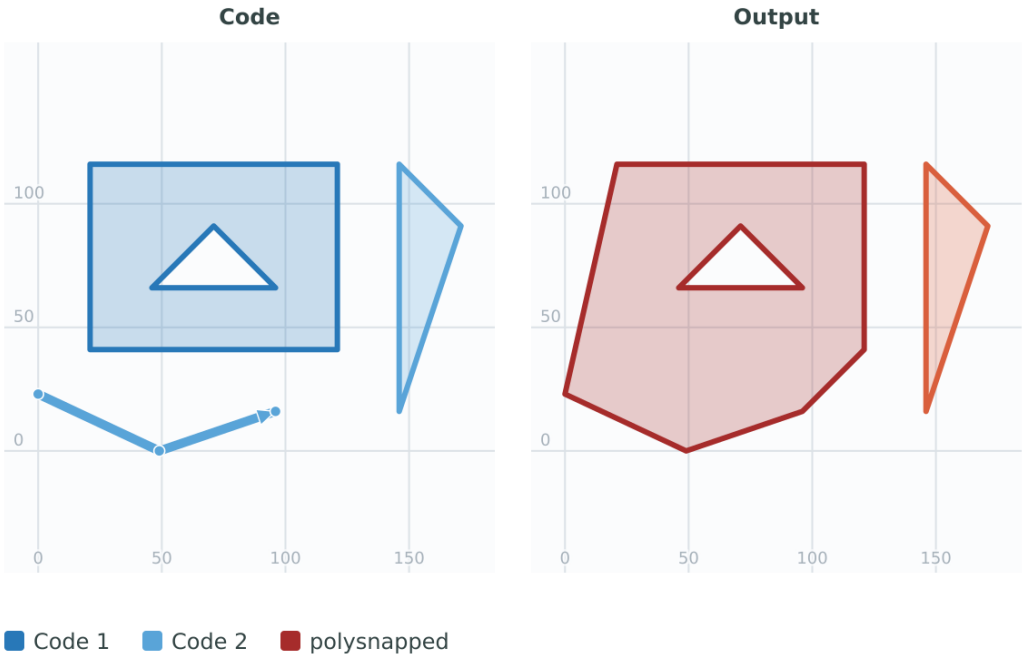
Code

```
SELECT ST_Snap(poly, line, ST_Distance(poly, line)*1.25) AS polysnapped
FROM (SELECT
  ST_GeomFromText('MULTIPOLYGON(
    (( 26 125,26 200,126 200,126 125,26 125 ),
      ( 51 150,101 150,76 175,51 150 )),
      (( 151 100,151 200,176 175,151 100 )))') As poly,
       ST_GeomFromText('LINESTRING (5 107,54 84,101 100)') As line
    ) As foo;
```

Output

```
MULTIPOLYGON(((5 107,26 200,126 200,126 125,101 100,54 84,5 107),
(51 150,101 150,76 175,51 150)),((151 100,151 200,176 175,151 100)))
```

Figure



The LineString snapped to the original MultiPolygon at 1.01 times their distance.

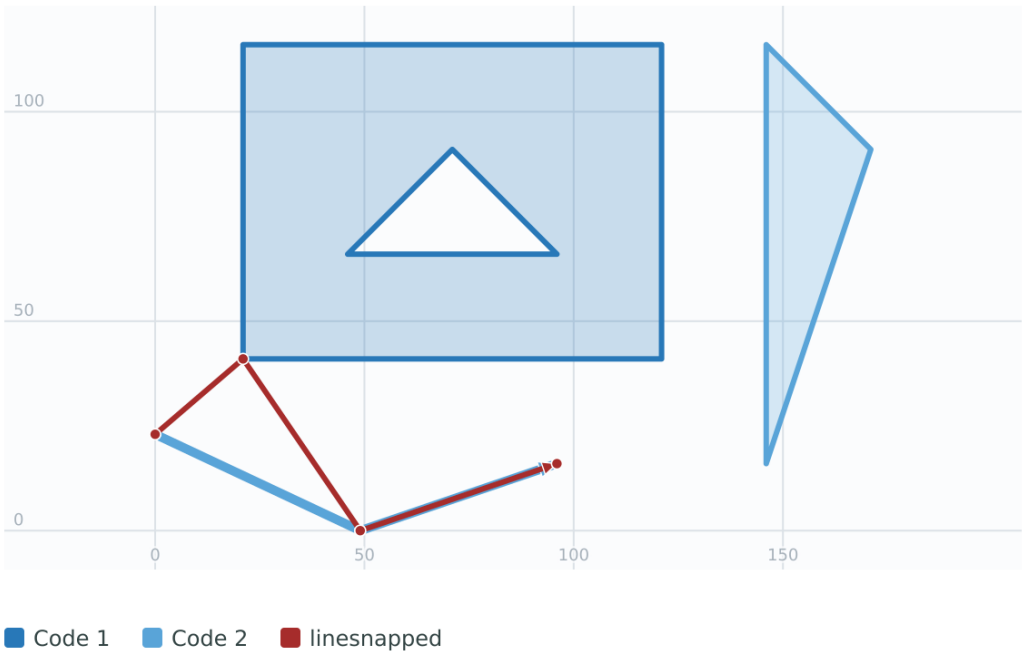
Code

```
SELECT ST_Snap(line, poly, ST_Distance(poly, line)*1.01) AS linesnapped
FROM (SELECT
  ST_GeomFromText('MULTIPOLYGON(
     ((26 125,26 200,126 200,126 125,26 125),
      (51 150,101 150,76 175,51 150 )),
      ((151 100,151 200,176 175,151 100)))') As poly,
       ST_GeomFromText('LINESTRING (5 107,54 84,101 100)') As line
    ) As foo;
```

Output

```
LINESTRING(5 107,26 125,54 84,101 100)
```

Figure



The LineString snapped to the original MultiPolygon at 1.25 times their distance.

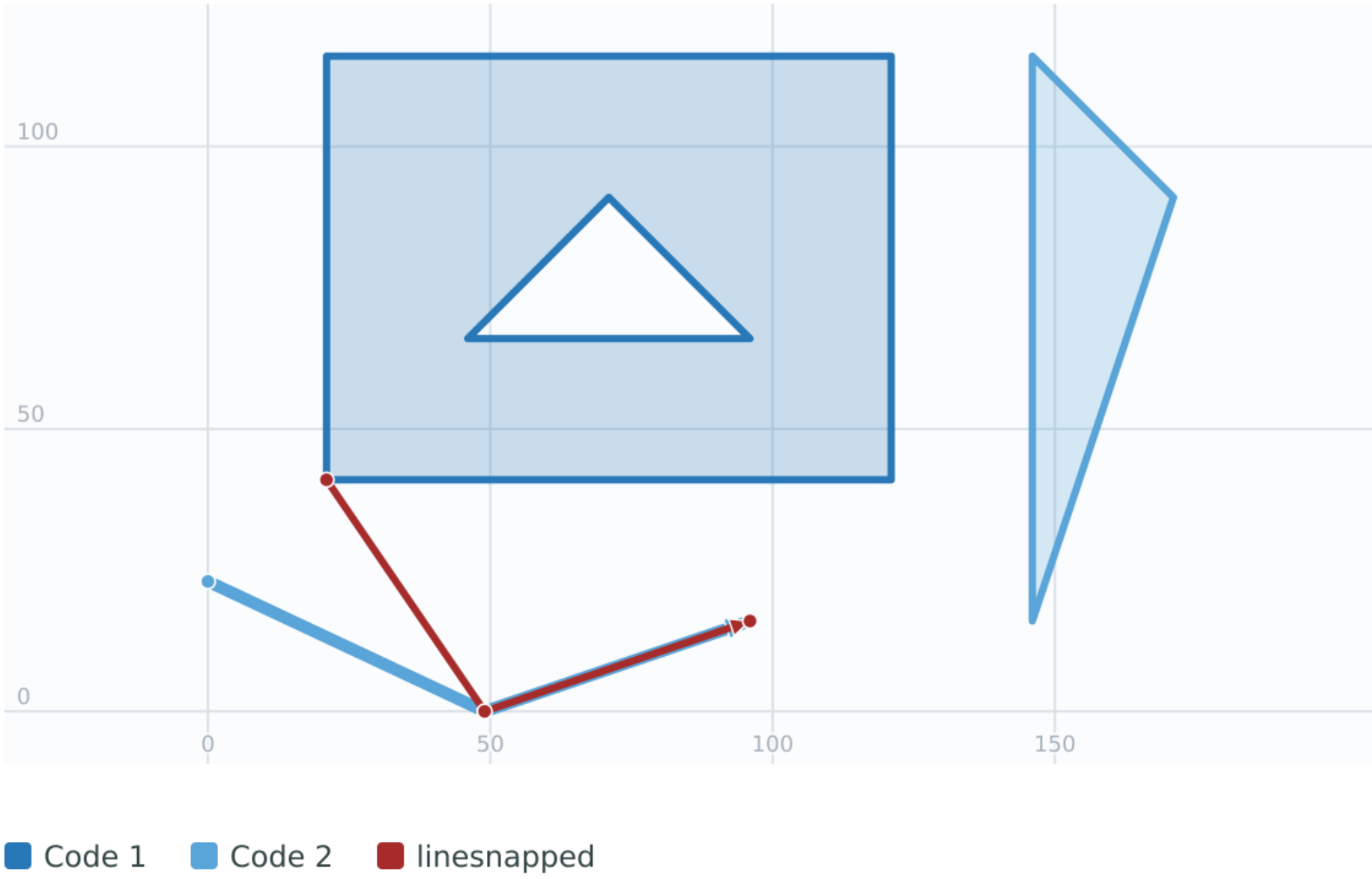
Code

```
SELECT ST_Snap(line, poly, ST_Distance(poly, line)*1.25) AS linesnapped
FROM (SELECT
  ST_GeomFromText('MULTIPOLYGON(
     (( 26 125,26 200,126 200,126 125,26 125 ),
      (51 150,101 150,76 175,51 150 )),
      ((151 100,151 200,176 175,151 100 )))') As poly,
       ST_GeomFromText('LINESTRING (5 107,54 84,101 100)') As line
    ) As foo;
```

Output

```
LINESTRING(26 125,54 84,101 100)
```

Figure



⊠⊠

ST_SnapToGrid

7.5.35 ST_SwapOrdinates

ST_SwapOrdinates — ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

geometry **ST_SwapOrdinates**(geometry geom, cstring ords);

⊠⊠

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
`ords` ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ 2 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠ x, y, z, ⊠⊠⊠ m ⊠⊠⊠.
2.2.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

- This method supports Circular Strings and Curves.
- This function supports 3d and will not drop the z-index.
- This function supports M coordinates.
- This function supports Polyhedral surfaces.
- This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).

□□

This example scales the M value by 2.

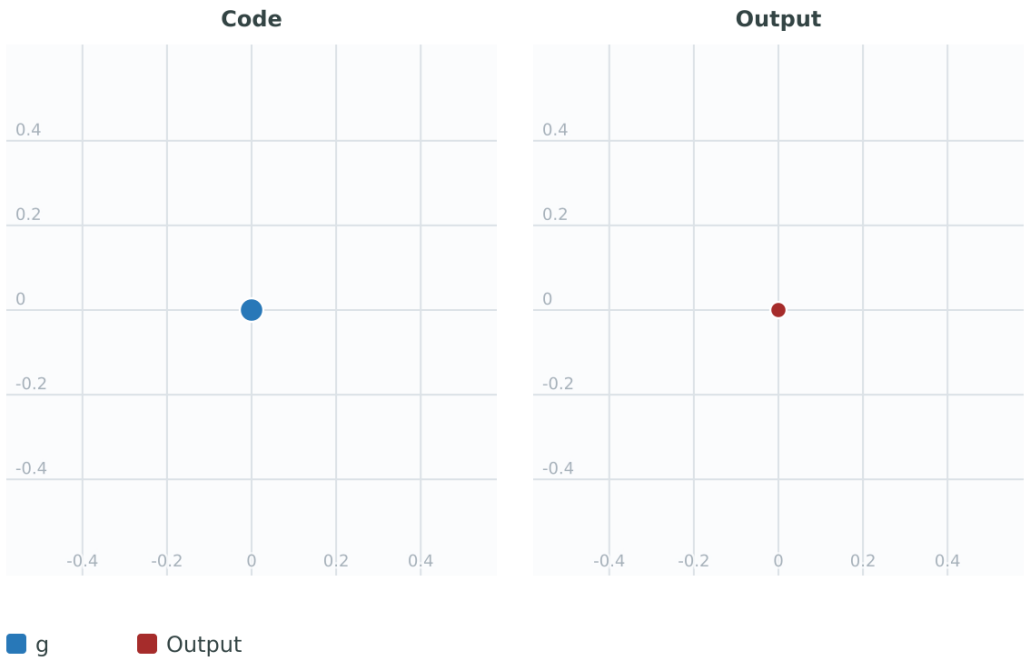
Code

```
SELECT ST_SwapOrdinates(ST_Scale(ST_SwapOrdinates(g, 'xm'),
      2, 1
    ),
  'xm') FROM ( SELECT 'POINT ZM (0 0 0 2)'::geometry g ) foo;
```

Output

```
POINT ZM (0 0 0 4)
```

Figure



□□

ST_FlipCoordinates

7.6 Geometry Validation

7.6.1 ST_IsValid

ST_IsValid — Tests if a geometry is well-formed in 2D.

□□

boolean **ST_IsValid**(geometry g);
boolean **ST_IsValid**(geometry g, integer flags);

□□

Tests if an ST_Geometry value is well-formed and valid in 2D according to the OGC rules. For geometries with 3 and 4 dimensions, the validity is still only tested in 2 dimensions. For geometries that are invalid, a PostgreSQL NOTICE is emitted providing details of why it is not valid.

For the version with the `flags` parameter, supported values are documented in ST_IsValidDetail This version does not print a NOTICE explaining invalidity.

For more information on the definition of geometry validity, refer to 4.4□



□□
SQL-MM defines the result of ST_IsValid(NULL) to be 0, while PostGIS returns NULL.

GEOS □□□□□

The version accepting flags is available starting with 2.0.0.



This method implements the OGC Simple Features Implementation Specification for SQL 1.1.



This method implements the SQL/MM specification. SQL-MM 3: 5.1.9



□□
Neither OGC-SFS nor SQL-MM specifications include a flag argument for ST_IsValid. The flag is a PostGIS extension.

□□

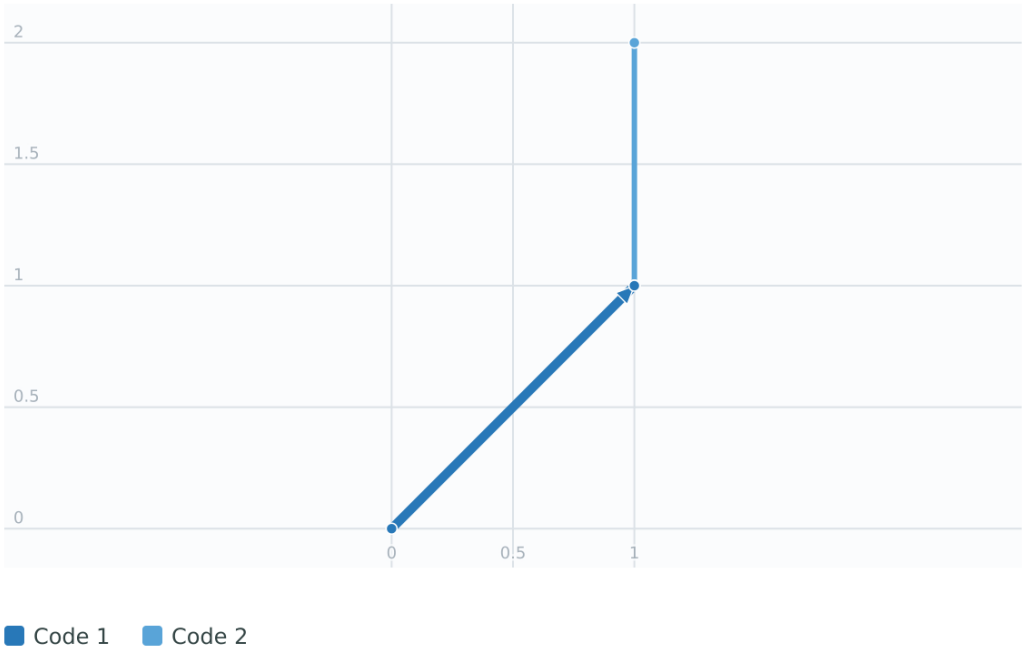
Code

```
SELECT ST_IsValid('LINESTRING(0 0,1 1)') As good_line,
ST_IsValid('POLYGON((0 0,1 1,1 2,1 1,0 0))') As bad_poly;
```

Output

```
NOTICE:  Self-intersection at or near point 0 0
 good_line | bad_poly
-----------+----------
 t         | f
```

Figure



Find tables with invalid geometries in registered geometry columns.

Code

```
CREATE TEMP TABLE invalid_geometry_columns (
    table_name text,
    column_name text,
    invalid_count bigint
);

DO $$
DECLARE
    rec record;
    invalid_count bigint;
BEGIN
    FOR rec IN
        SELECT f_table_schema, f_table_name, f_geometry_column
        FROM geometry_columns
        ORDER BY f_table_schema, f_table_name, f_geometry_column
    LOOP
        EXECUTE format(
            'SELECT count(*) FROM %I.%I WHERE NOT ST_IsValid(%I, 0)',
            rec.f_table_schema,
            rec.f_table_name,
            rec.f_geometry_column
        ) INTO invalid_count;

        IF invalid_count
> 0 THEN
            INSERT INTO invalid_geometry_columns
            VALUES (
                format('%I.%I', rec.f_table_schema, rec.f_table_name),
                rec.f_geometry_column,
                invalid_count
            );
        END IF;
    END LOOP;
```

```
END;
$$;

SELECT * FROM invalid_geometry_columns;
```

□□

ST_IsSimple, ST_IsValidReason, ST_IsValidDetail,

7.6.2 ST_IsValidDetail

ST_IsValidDetail — Returns a `valid_detail` row stating if a geometry is valid or if not a reason and a location.

□□

valid_detail **ST_IsValidDetail**(geometry geom, integer flags);

□□

Returns a `valid_detail` row, containing a boolean (`valid`) stating if a geometry is valid, a varchar (`reason`) stating a reason why it is invalid and a geometry (`location`) pointing out where it is invalid.

Useful to improve on the combination of ST_IsValid and ST_IsValidReason to generate a detailed report of invalid geometries.

The optional `flags` parameter is a bitfield. It can have the following values:

- 0: Use usual OGC SFS validity semantics.
- 1: Consider certain kinds of self-touching rings (inverted shells and exverted holes) as valid. This is also known as "the ESRI flag", since this is the validity model used by those tools. Note that this is invalid under the OGC model.

GEOS □□□□□

2.0.0 □□□□□□□□□□□□.

□□

This example returns the first three rejects from a successful quintuplet experiment.

Code

```
SELECT gid, reason(ST_IsValidDetail(geom)), location(ST_IsValidDetail(geom)) as location
FROM
(SELECT ST_MakePolygon(ST_ExteriorRing(e.buff), array_agg(f.line)) As geom, gid
FROM (SELECT ST_Buffer(ST_Point(x1*10,y1), z1) As buff, x1*10 + y1*100 + z1*1000 As gid
    FROM generate_series(-4,6) x1
    CROSS JOIN generate_series(2,5) y1
    CROSS JOIN generate_series(1,8) z1
    WHERE x1
> y1*0.5 AND z1 < x1*y1) As e
```

```
    INNER JOIN (SELECT ST_Translate(ST_ExteriorRing(ST_Buffer(ST_Point(x1*10,y1), z1)),y1 ↩
    *1, z1*2) As line
    FROM generate_series(-3,6) x1
    CROSS JOIN generate_series(2,5) y1
    CROSS JOIN generate_series(1,10) z1
    WHERE x1
> y1*0.75 AND z1 < x1*y1) As f
ON (ST_Area(e.buff)
> 78 AND ST_Contains(e.buff, f.line))
GROUP BY gid, e.buff) As quintuplet_experiment
WHERE ST_IsValid(geom) = false
ORDER BY gid
LIMIT 3;
```

Output

```
 gid  |      reason       |  location
------+-------------------+-------------
 5330 | Self-intersection | POINT(32 5)
 5340 | Self-intersection | POINT(42 5)
 5350 | Self-intersection | POINT(52 5)
```

This example reports a valid LineString.

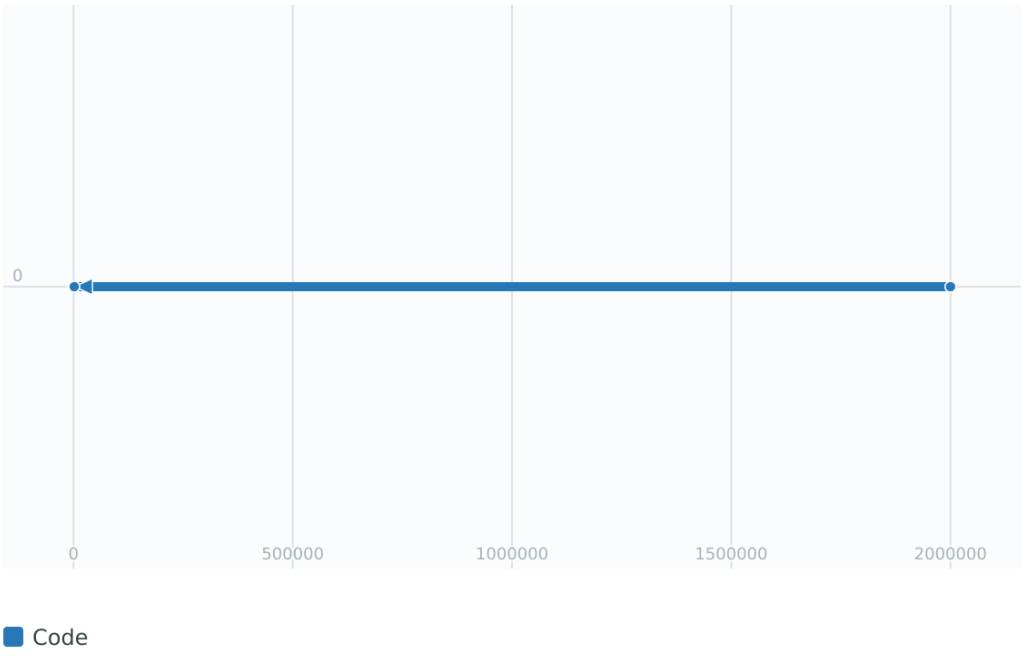
Code

```
SELECT * FROM ST_IsValidDetail('LINESTRING(220227 150406,2220227 150407,222020 150410)');
```

Output

```
 valid | reason | location
-------+--------+----------
 t     |        |
```

Figure



□□

ST_IsValid, ST_IsValidReason

7.6.3 ST_IsValidReason

ST_IsValidReason — Returns text stating if a geometry is valid, or a reason for invalidity.

□□

text **ST_IsValidReason**(geometry geomA);
text **ST_IsValidReason**(geometry geomA, integer flags);

□□

Returns text stating if a geometry is valid, or if invalid a reason why.

Useful in combination with ST_IsValid to generate a detailed report of invalid geometries and reasons.

Allowed `flags` are documented in ST_IsValidDetail.

GEOS □□□□□

Availability: 1.4

Availability: 2.0 version taking flags.

□□

This example uses an invalid bow-tie polygon.

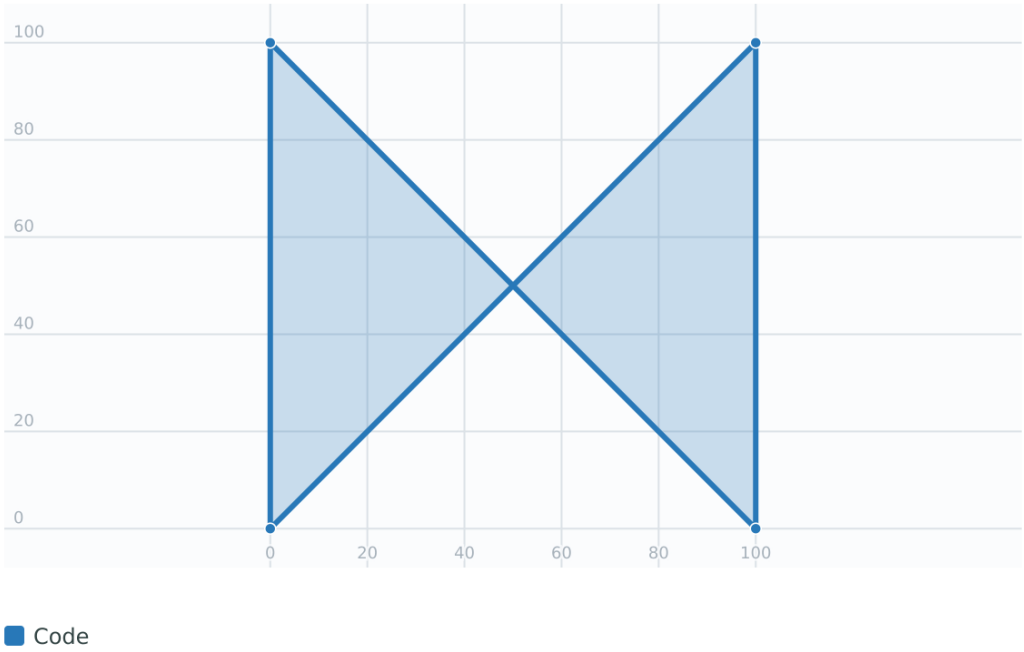
Code

```
SELECT ST_IsValidReason(
    'POLYGON ((100 200,100 100,200 200,
     200 100,100 200))'::geometry) as validity_info;
```

Output

```
Self-intersection[150 150]
```

Figure



This example returns the first three rejects from a successful quintuplet experiment.

Code

```
SELECT gid, ST_IsValidReason(geom) as validity_info
FROM
(SELECT ST_MakePolygon(ST_ExteriorRing(e.buff), array_agg(f.line)) As geom, gid
FROM (SELECT ST_Buffer(ST_Point(x1*10,y1), z1) As buff, x1*10 + y1*100 + z1*1000 As gid
    FROM generate_series(-4,6) x1
    CROSS JOIN generate_series(2,5) y1
    CROSS JOIN generate_series(1,8) z1
    WHERE x1
> y1*0.5 AND z1 < x1*y1) As e
    INNER JOIN (SELECT ST_Translate(ST_ExteriorRing(ST_Buffer(ST_Point(x1*10,y1), z1)),y1 ↩
    *1, z1*2) As line
    FROM generate_series(-3,6) x1
    CROSS JOIN generate_series(2,5) y1
    CROSS JOIN generate_series(1,10) z1
    WHERE x1
> y1*0.75 AND z1 < x1*y1) As f
ON (ST_Area(e.buff)
> 78 AND ST_Contains(e.buff, f.line))
GROUP BY gid, e.buff) As quintuplet_experiment
WHERE ST_IsValid(geom) = false
ORDER BY gid
LIMIT 3;
```

Output

```
 gid  |      validity_info
------+-------------------------
 5330 | Self-intersection [32 5]
 5340 | Self-intersection [42 5]
 5350 | Self-intersection [52 5]
```

Simple example.

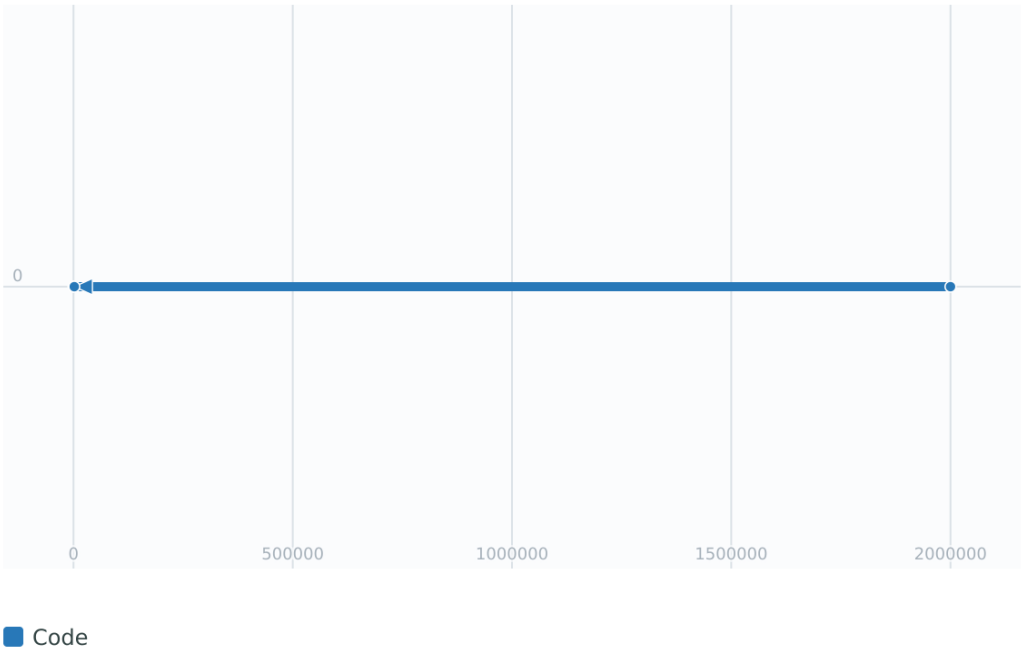
Code

```
SELECT ST_IsValidReason('LINESTRING(220227 150406,2220227 150407,222020 150410)');
```

Output

```
 st_isvalidreason
------------------
 Valid Geometry
```

Figure



□□

ST_IsValid, ST_Summary

7.6.4 ST_MakeValid

ST_MakeValid — Attempts to make an invalid geometry valid without losing vertices.

□□

geometry **ST_MakeValid**(geometry input);
geometry **ST_MakeValid**(geometry input, text params);

□□

The function attempts to create a valid representation of a given invalid geometry without losing any of the input vertices. Valid geometries are returned unchanged.

Supported inputs are: POINTS, MULTIPOINTS, LINESTRINGS, MULTILINESTRINGS, POLYGONS, MULTIPOLYGONS and GEOMETRYCOLLECTIONS containing any mix of them.

In case of full or partial dimensional collapses, the output geometry may be a collection of lower-to-equal dimension geometries, or a geometry of lower dimension.

Single polygons may become multi-geometries in case of self-intersections.

The `params` argument can be used to supply an options string to select the method to use for building valid geometry. The options string is in the format "method=linework|structure keepcollapsed=true|false". If no "params" argument is provided, the "linework" algorithm will be used as the default.

The "method" key has two values.

- "linework" is the default algorithm, and builds valid geometries by first extracting all lines, noding that linework together, then building a value output from the linework. The requirement that no vertices are lost can generate complex collections as outputs.
- "structure" is an algorithm that distinguishes between interior and exterior rings, building new geometry by unioning exterior rings, and then differencing all interior rings. The results tend to be more intuitive, where collapsed lines are discarded in the output.

The "keepcollapsed" key is only valid for the "structure" algorithm, and takes a value of "true" or "false".

- Keep collapsed of "false" is the default, and means that geometry components that collapse to a lower dimensionality, for example a one-point linestring will be dropped.
- Keep collapsed of "true" means geometry components that collapse to a lower dimensionality will be retained.

GEOS □□□□□

2.0.0 □□□□□□□□□□□□.

Enhanced: 2.0.1, speed improvements

Enhanced: 2.1.0, added support for GEOMETRYCOLLECTION and MULTIPOINT.

Enhanced: 3.1.0, added removal of Coordinates with NaN values.

Enhanced: 3.2.0, added algorithm options, 'linework' and 'structure' which requires GEOS >= 3.10.0.

This function supports 3d and will not drop the z-index.

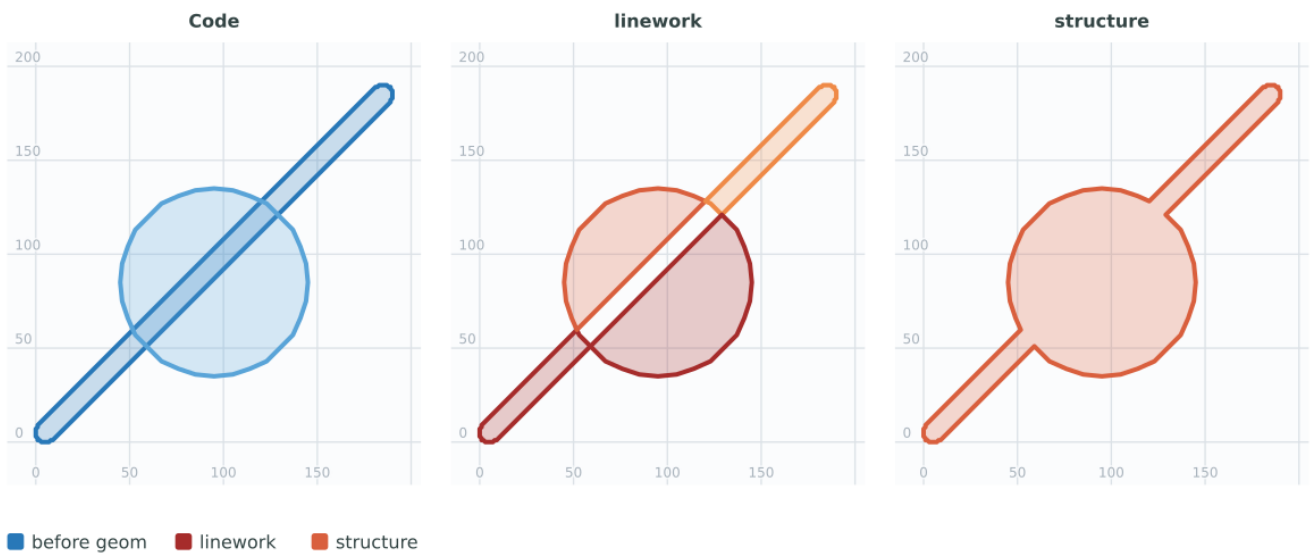
□□

Make valid a MultiPolygon of two overlapping polygons using the linework and structure methods.

Code

```
SELECT ST_MakeValid(before_geom) AS linework,
  ST_MakeValid(before_geom, 'method=structure') AS structure
FROM (SELECT 'MULTIPOLYGON(((186 194,187 194,188 195,189 195,190 195,
191 195,192 195,193 194,194 194,194 193,195 192,195 191,
195 190,195 189,195 188,194 187,194 186,14 6,13 6,12 5,11 5,
10 5,9 5,8 5,7 6,6 6,6 7,5 8,5 9,5 10,5 11,5 12,6 13,6 14,186 194)),
((150 90,149 80,146 71,142 62,135 55,128 48,119 44,110 41,100 40,
90 41,81 44,72 48,65 55,58 62,54 71,51 80,50 90,51 100,
54 109,58 118,65 125,72 132,81 136,90 139,100 140,110 139,
119 136,128 132,135 125,142 118,146 109,149 100,150 90)))'::geometry AS before_geom) AS f;
```

Figure

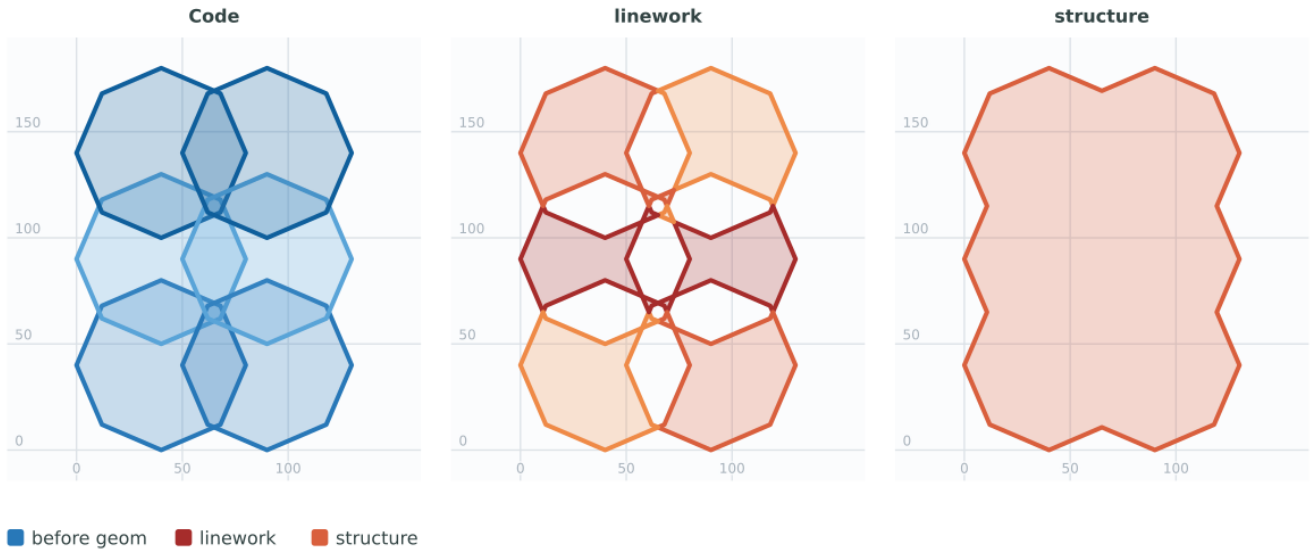


Make valid a MultiPolygon of six overlapping polygons using the linework and structure methods.

Code

```
SELECT ST_MakeValid(before_geom) AS linework,
  ST_MakeValid(before_geom, 'method=structure') AS structure
FROM (SELECT 'MULTIPOLYGON(((91 50,79 22,51 10,23 22,11 50,23 78,51 90,79 78,91 50)),
      ((91 100,79 72,51 60,23 72,11 100,23 128,51 140,79 128,91 100)),
      ((91 150,79 122,51 110,23 122,11 150,23 178,51 190,79 178,91 150)),
      ((141 50,129 22,101 10,73 22,61 50,73 78,101 90,129 78,141 50)),
      ((141 100,129 72,101 60,73 72,61 100,73 128,101 140,129 128,141 100)),
      ((141 150,129 122,101 110,73 122,61 150,73 178,101 190,129 178,141 150)))'::geometry ↩
    AS before_geom) AS c;
```

Figure



Collapsed Geometry Examples

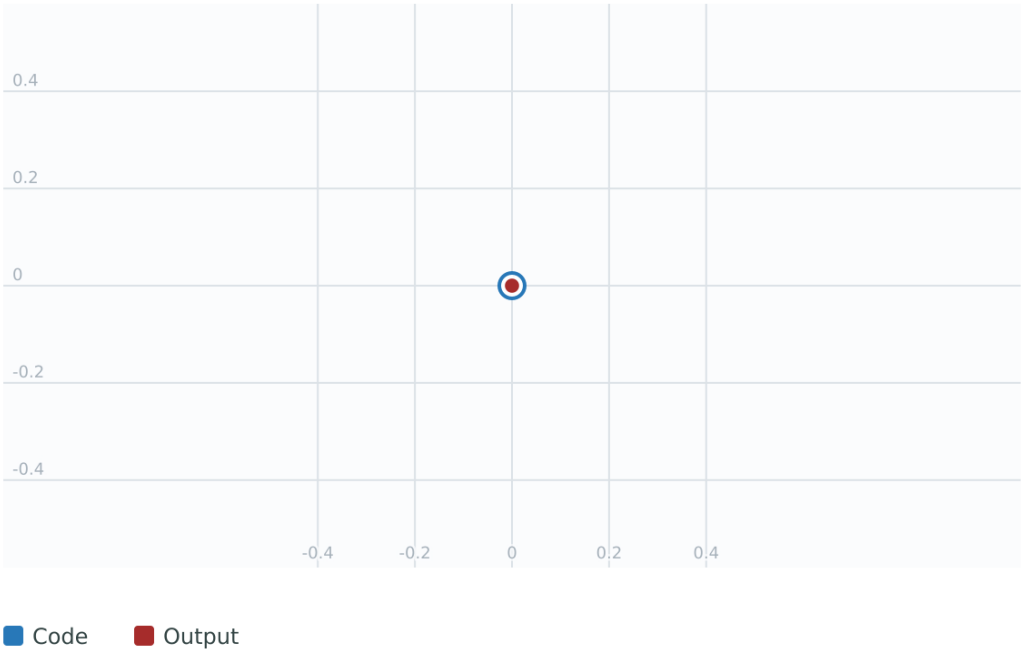
Code

```
SELECT ST_MakeValid(
'LINESTRING(0 0,0 0)',
'method=structure keepcollapsed=true'
);
```

Output

```
POINT(0 0)
```

Figure



Code

```
SELECT ST_MakeValid(
'LINESTRING(0 0,0 0)',
'method=structure keepcollapsed=false'
);
```

Output

```
LINESTRING
```

□□

ST_IsValid, ST_Collect, ST_CollectionExtract

7.7 Spatial Reference System Functions

7.7.1 ST_InverseTransformPipeline

ST_InverseTransformPipeline — Return a new geometry with coordinates transformed to a different spatial reference system using the inverse of a defined coordinate transformation pipeline.

□□

geometry **ST_InverseTransformPipeline**(geometry geom, text pipeline, integer to_srid);

□□

Return a new geometry with coordinates transformed to a different spatial reference system using a defined coordinate transformation pipeline to go in the inverse direction.

Refer to ST_TransformPipeline for details on writing a transformation pipeline.

Availability: 3.4.0

The SRID of the input geometry is ignored, and the SRID of the output geometry will be set to zero unless a value is provided via the optional `to_srid` parameter. When using ST_TransformPipeline the pipeline is executed in a forward direction. Using `ST_InverseTransformPipeline()` the pipeline is executed in the inverse direction.

Transforms using pipelines are a specialised version of ST_Transform. In most cases `ST_Transform` will choose the correct operations to convert between coordinate systems, and should be preferred.

□□

Change WGS 84 long lat to UTM 31N using the EPSG:16031 conversion

This example transforms in the inverse direction.

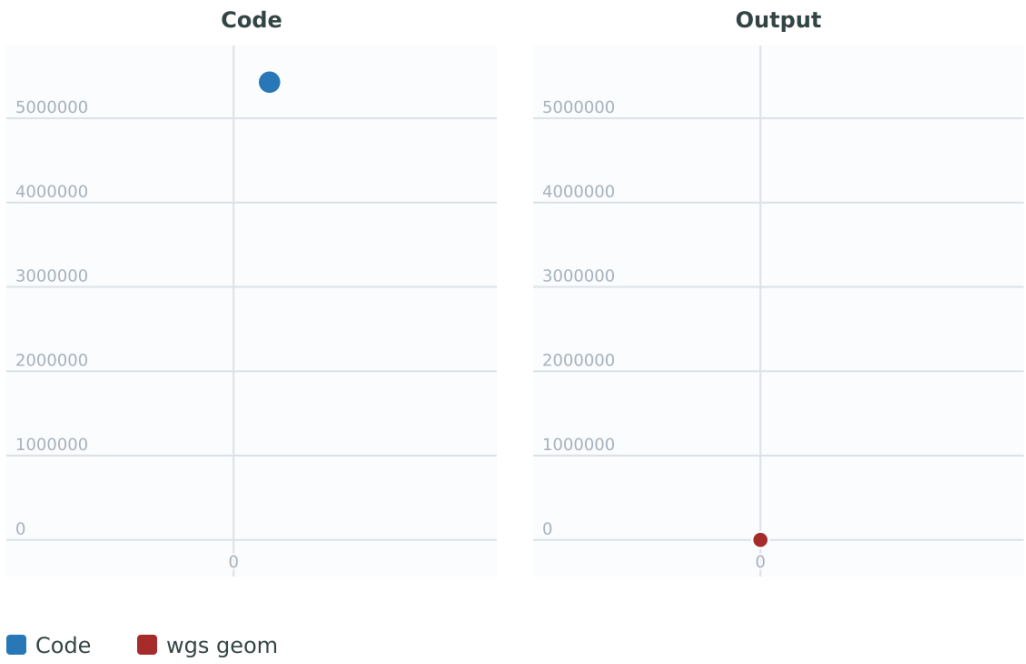
Code

```
SELECT ST_InverseTransformPipeline(
        'POINT(426857.9877165967 5427937.523342293)'::geometry,
        'urn:ogc:def:coordinateOperation:EPSG::16031'
    ) AS wgs_geom;
```

Output

```
POINT(2 48.99999999999999)
```

Figure



GDA2020 example.

This example uses ST_Transform with automatic selection of a conversion pipeline.

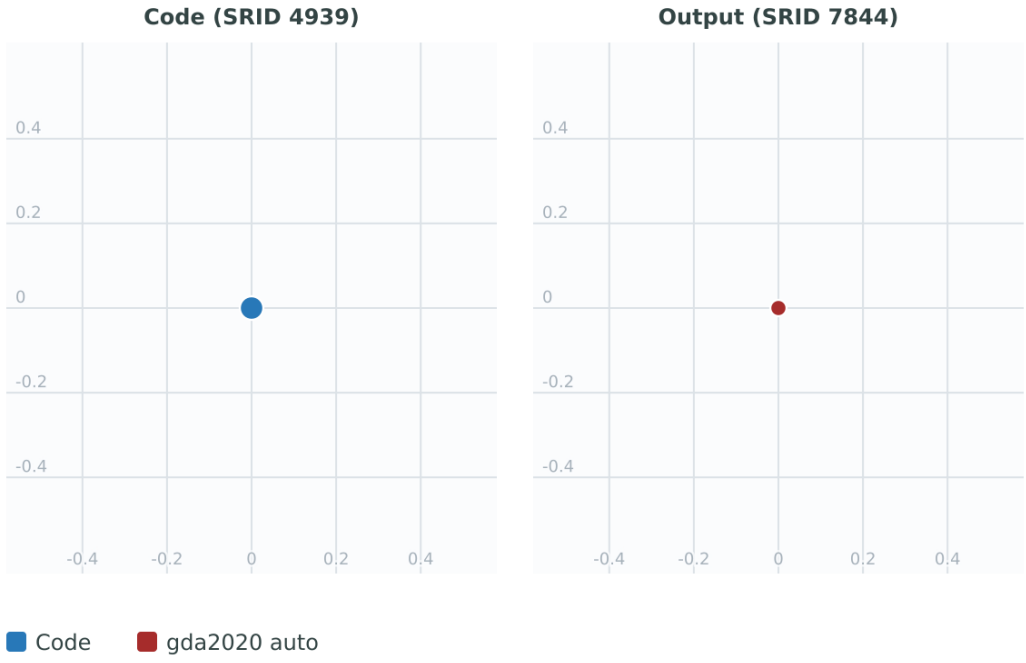
Code

```
SELECT ST_Transform('SRID=4939;POINT(143.0 -37.0)'::geometry, 7844) AS gda2020_auto;
```

Output

```
POINT(143.00000635638918 -36.999986706128176)
```

Figure



□□

ST_Transform, ST_TransformPipeline

7.7.2 ST_SetSRID

ST_SetSRID — Set the SRID on a geometry.

□□

geometry **ST_SetSRID**(geometry geom, integer srid);

□□

Sets the SRID on a geometry to a particular integer value. Useful in constructing bounding boxes for queries.



□□
This function does not transform the geometry coordinates in any way - it simply sets the meta data defining the spatial reference system the geometry is assumed to be in. Use ST_Transform if you want to transform the geometry into a new projection.


This method implements the OGC Simple Features Implementation Specification for SQL 1.1.


This method supports Circular Strings and Curves.

□□

Mark a point as WGS 84 longitude/latitude.

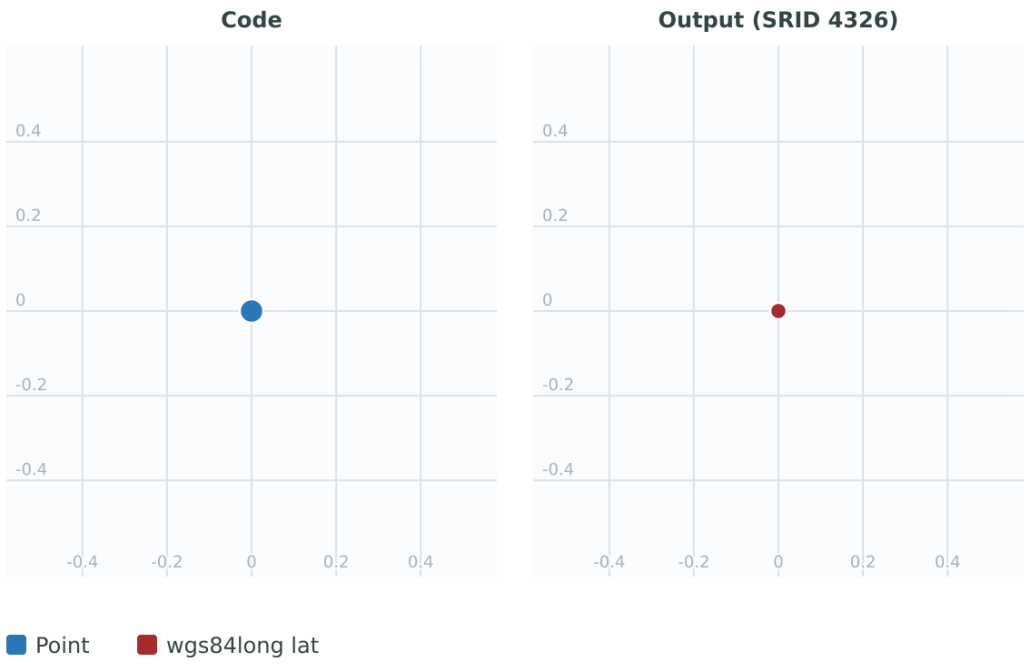
Code

```
SELECT ST_SetSRID(ST_Point(-123.365556, 48.428611), 4326) AS wgs84long_lat;
```

Output

```
SRID=4326;POINT(-123.365556 48.428611)
```

Figure



Mark a point as WGS 84 longitude/latitude and then transform it to Web Mercator.

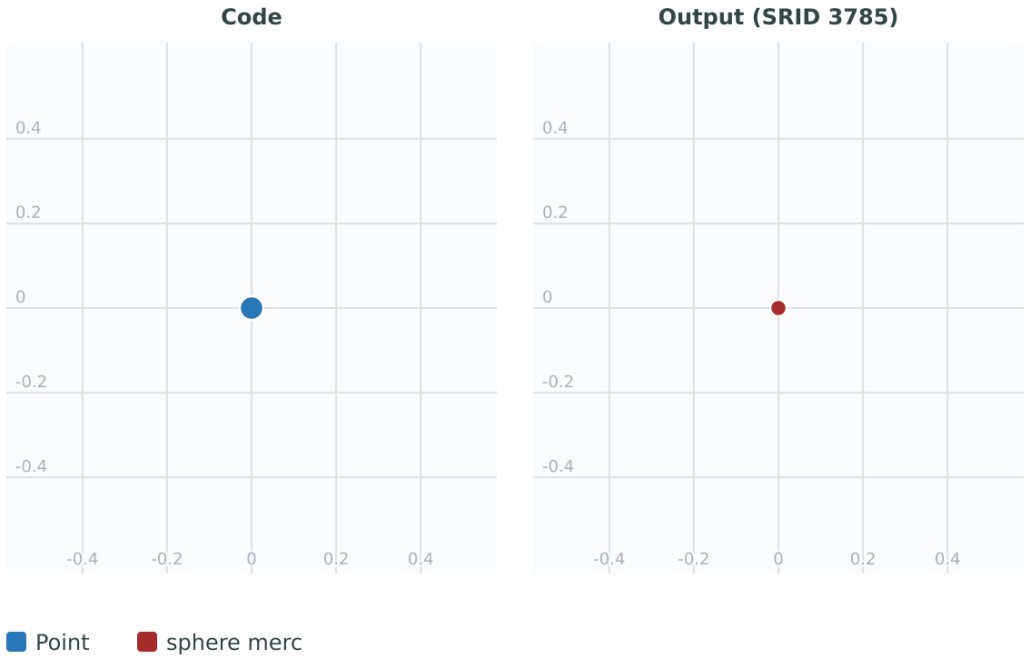
Code

```
SELECT ST_Transform(ST_SetSRID(ST_Point(-123.365556, 48.428611), 4326), 3785) AS ↩
    sphere_merc;
```

Output

```
SRID=3785;POINT(-13732990.8753491 6178458.96425423)
```

Figure



□□

4.5□, ST_SRID, ST_Transform, UpdateGeometrySRID

7.7.3 ST_SRID

ST_SRID — Returns the spatial reference identifier for a geometry.

□□

integer **ST_SRID**(geometry g1);

□□

Returns the spatial reference identifier for the ST_Geometry as defined in spatial_ref_sys table. 4.5□



□□
spatial_ref_sys table is a table that catalogs all spatial reference systems known to PostGIS and is used for transformations from one spatial reference system to another. So verifying you have the right spatial reference system identifier is important if you plan to ever transform your geometries.

 This method implements the OGC Simple Features Implementation Specification for SQL 1.1. s2.1.1.1

 This method implements the SQL/MM specification. SQL-MM 3: 5.1.5

 This method supports Circular Strings and Curves.

□□

Code

```
SELECT ST_SRID(ST_GeomFromText('POINT(-71.1043 42.315)', 4326));
```

Output

```
st_srid
---------
    4326
(1 row)
```

□□

4.5□, ST_SetSRID, ST_Transform, ST_SRID, ST_SRID

7.7.4 ST_Transform

ST_Transform — Return a new geometry with coordinates transformed to a different spatial reference system.

□□

geometry **ST_Transform**(geometry g1, integer srid);
geometry **ST_Transform**(geometry geom, text to_proj);
geometry **ST_Transform**(geometry geom, text from_proj, text to_proj);
geometry **ST_Transform**(geometry geom, text from_proj, integer to_srid);

□□

Returns a new geometry with its coordinates transformed to a different spatial reference system. The destination spatial reference `to_srid` may be identified by a valid SRID integer parameter (i.e. it must exist in the `spatial_ref_sys` table). Alternatively, a spatial reference defined as a PROJ.4 string can be used for `to_proj` and/or `from_proj`, however these methods are not optimized. If the destination spatial reference system is expressed with a PROJ.4 string instead of an SRID, the SRID of the output geometry will be set to zero. With the exception of functions with `from_proj`, input geometries must have a defined SRID.

ST_Transform is often confused with ST_SetSRID. ST_Transform actually changes the coordinates of a geometry from one spatial reference system to another, while ST_SetSRID() simply changes the SRID identifier of the geometry.

ST_Transform automatically selects a suitable conversion pipeline given the source and target spatial reference systems. To use a specific conversion method, use ST_TransformPipeline.



□□
Requires PostGIS be compiled with PROJ support. Use PostGIS_Full_Version to confirm you have PROJ support compiled in.



□□
If using more than one transformation, it is useful to have a functional index on the commonly used transformations to take advantage of index usage.



□□
1.3.4 □□□□□□□□□□□□□□□□ (curve) □□□□□□□□□□□□□□□□□□□□□□□□. 1.3.4 □□□□□□□□□□□□□□□.

□□□□: 2.0.0 □□□□□□□□□ (polyhedral surface) □□□□□□.

Enhanced: 2.3.0 support for direct PROJ.4 text was introduced.

This method implements the SQL/MM specification. SQL-MM 3: 5.1.6

This method supports Circular Strings and Curves.

This function supports Polyhedral surfaces.

□□

Change Massachusetts state plane US feet geometry to WGS 84 long lat

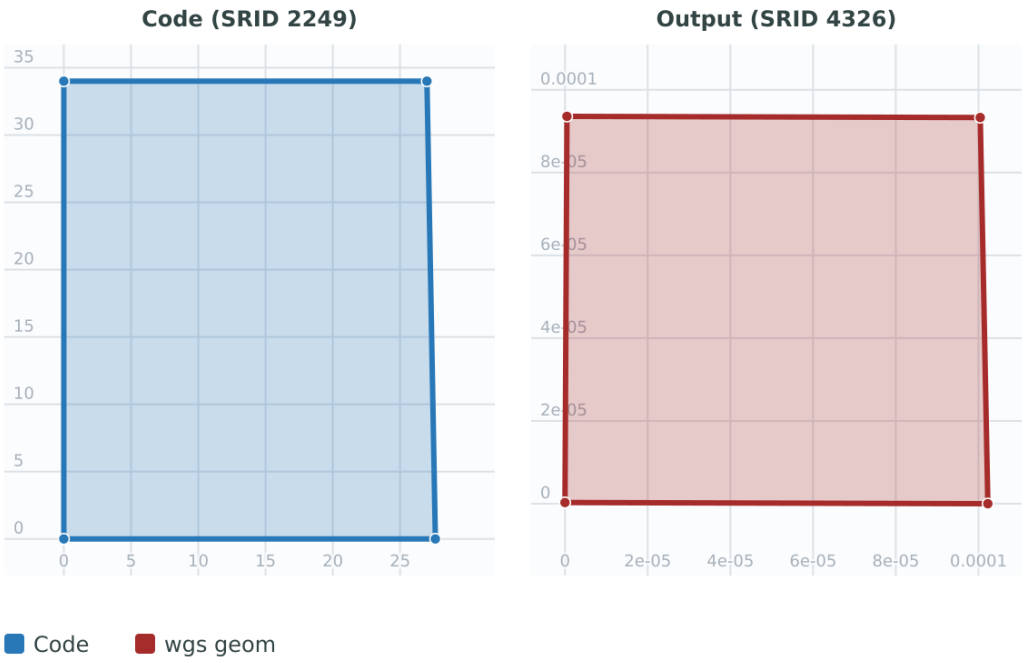
Code

```
SELECT ST_Transform(
  'SRID=2249;POLYGON((743238 2967416,743238 2967450,
    743265 2967450,743265.625 2967416,743238 2967416))'::geometry,
  4326) AS wgs_geom;
```

Output

```
SRID=4326;POLYGON((-71.1776848522251 42.3902896512903,-71.17768437663261 42.390382947801015,-71.17758443054647 ↩
    42.390382667791854,-71.17758259272304 42.390289364798825,-71.1776848522251 42.3902896512903))
```

Figure



3D Circular String example.

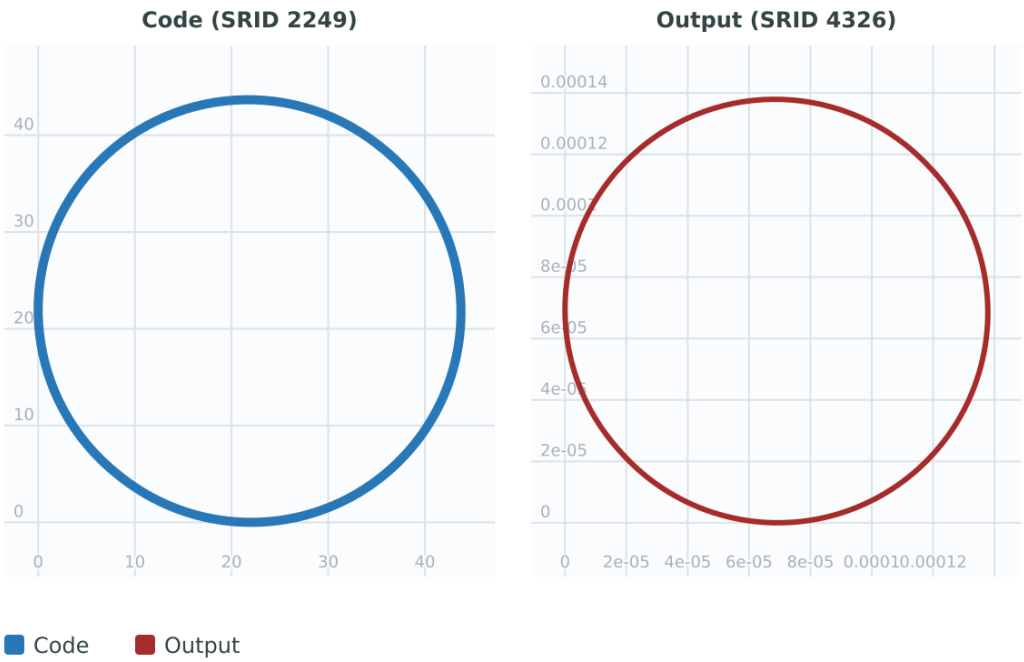
Code

```
SELECT ST_Transform('SRID=2249;CIRCULARSTRING(743238 2967416 1,743238 2967450 2,743265 ↩
    2967450 3,743265.625 2967416 3,743238 2967416 4)'::geometry,
    4326);
```

Output

```
SRID=4326;CIRCULARSTRING(-71.1776848522251 42.3902896512903 1,-71.17768437663261 42.390382947801015 2,
-71.17758443054647 42.390382667791854 3,
-71.17758259272304 42.390289364798825 3,-71.1776848522251 42.3902896512903 4)
```

Figure



Example of creating a partial functional index. For tables where you are not sure all the geometries will be filled in, its best to use a partial index that leaves out null geometries which will both conserve space and make your index smaller and more efficient.

Code

```
CREATE INDEX idx_geom_26986_parcels
  ON parcels
  USING gist
  (ST_Transform(geom, 26986))
  WHERE geom IS NOT NULL;
```

Examples of using PROJ.4 text to transform with custom spatial references.

Find intersection of two polygons near the North pole, using a custom Gnomic projection. See http://boundles
the-peel/.

Code

```
WITH data AS (
  SELECT
    ST_GeomFromText('POLYGON((170 50,170 72,-130 72,-130 50,170 50))', 4326) AS p1,
    ST_GeomFromText('POLYGON((-170 68,-170 90,-141 90,-141 68,-170 68))', 4326) AS p2,
    '+proj=gnom +ellps=WGS84 +lat_0=70 +lon_0=-160 +no_defs'::text AS gnom
)
SELECT ST_Transform(ST_Intersection(ST_Transform(p1, gnom), ST_Transform(p2, gnom)),
  gnom, 4326)
FROM data;
```

Output

```
POLYGON((-140.9999840290591 73.42768865981984,-141 68,-170 68,-170.0000026915361 74.05492179637055,-140.9999840290591 ↩
    73.42768865981984))
```

Figure



Configuring transformation behavior

Sometimes coordinate transformation involving a grid-shift can fail, for example if PROJ.4 has not been built with grid-shift files or the coordinate does not lie within the range for which the grid shift is defined. By default, PostGIS will throw an error if a grid shift file is not present, but this behavior can be configured on a per-SRID basis either by testing different `to_proj` values of PROJ.4 text, or altering the `proj4text` value within the `spatial_ref_sys` table.

For example, the proj4text parameter +datum=NAD87 is a shorthand form for the following +nadgrids parameter:

Code

```
+nadgrids=@conus, @alaska, @ntv2_0.gsb, @ntv1_can.dat
```

The @ prefix means no error is reported if the files are not present, but if the end of the list is reached with no file having been appropriate (ie. found and overlapping) then an error is issued.

If, conversely, you wanted to ensure that at least the standard files were present, but that if all files were scanned without a hit a null transformation is applied you could use:

Code

```
+nadgrids=@conus, @alaska, @ntv2_0.gsb, @ntv1_can.dat, null
```

The null grid shift file is a valid grid shift file covering the whole world and applying no shift. So for a complete example, if you wanted to alter PostGIS so that transformations to SRID 4267 that didn't lie within the correct range did not throw an ERROR, you would use the following:

Code

```
UPDATE spatial_ref_sys SET proj4text = '+proj=longlat +ellps=clrk66 +nadgrids=@conus, ↩
    @alaska,@ntv2_0.gsb,@ntv1_can.dat,null +no_defs' WHERE srid = 4267;
```

□□

4.5□, ST_SetSRID, ST_SRID, UpdateGeometrySRID, ST_TransformPipeline

7.7.5 ST_TransformPipeline

ST_TransformPipeline — Return a new geometry with coordinates transformed to a different spatial reference system using a defined coordinate transformation pipeline.

□□

geometry **ST_TransformPipeline**(geometry g1, text pipeline, integer to_srid);

□□

Return a new geometry with coordinates transformed to a different spatial reference system using a defined coordinate transformation pipeline.

Transformation pipelines are defined using any of the following string formats:

- `urn:ogc:def:coordinateOperation:AUTHORITY::CODE`. Note that a simple `EPSG:CODE` string does not uniquely identify a coordinate operation: the same EPSG code can be used for a CRS definition.
- A PROJ pipeline string of the form: `+proj=pipeline ...`. Automatic axis normalisation will not be applied, and if necessary the caller will need to add an additional pipeline step, or remove `axisswap` steps.
- Concatenated operations of the form: `urn:ogc:def:coordinateOperation,coordinateOperation:EPSG:`

Availability: 3.4.0

The SRID of the input geometry is ignored, and the SRID of the output geometry will be set to zero unless a value is provided via the optional `to_srid` parameter. When using `ST_TransformPipeline()` the pipeline is executed in a forward direction. Using ST_InverseTransformPipeline the pipeline is executed in the inverse direction.

Transforms using pipelines are a specialised version of ST_Transform. In most cases `ST_Transform` will choose the correct operations to convert between coordinate systems, and should be preferred.

□□

Change WGS 84 long lat to UTM 31N using the EPSG:16031 conversion

This example transforms in the forward direction.

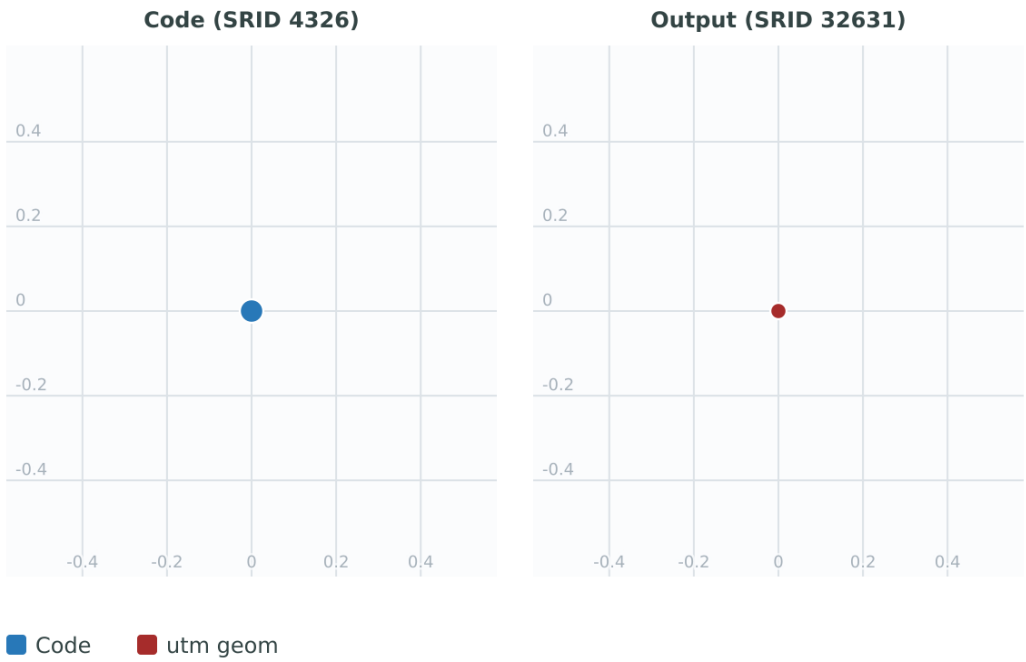
Code

```
SELECT
    ST_SetSRID(ST_TransformPipeline(
        'SRID=4326;POINT(2 49)'::geometry,
        'urn:ogc:def:coordinateOperation:EPSG::16031'
    ), 32631) AS utm_geom;
```

Output

```
SRID=32631;POINT(426857.9877165967 5427937.523342293)
```

Figure



This example transforms in the inverse direction.

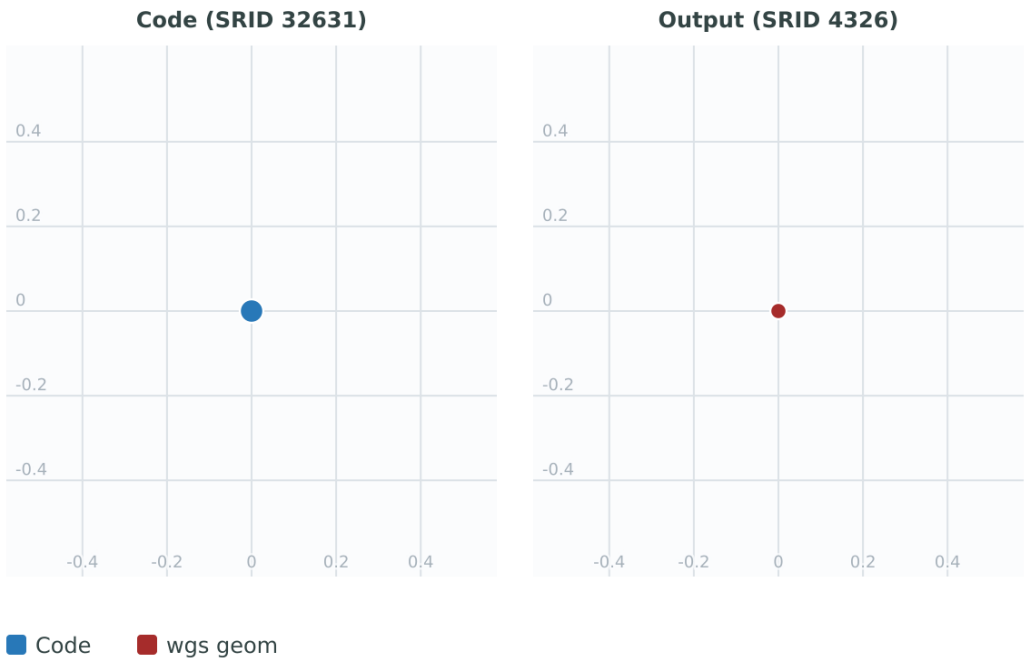
Code

```
SELECT
    ST_SetSRID(ST_InverseTransformPipeline(
        'SRID=32631;POINT(426857.9877165967 5427937.523342293)'::geometry,
        'urn:ogc:def:coordinateOperation:EPSG::16031'
    ), 4326) AS wgs_geom;
```

Output

```
SRID=4326;POINT(2 48.99999999999999)
```

Figure



GDA2020 example.

This example uses ST_Transform with automatic selection of a conversion pipeline.

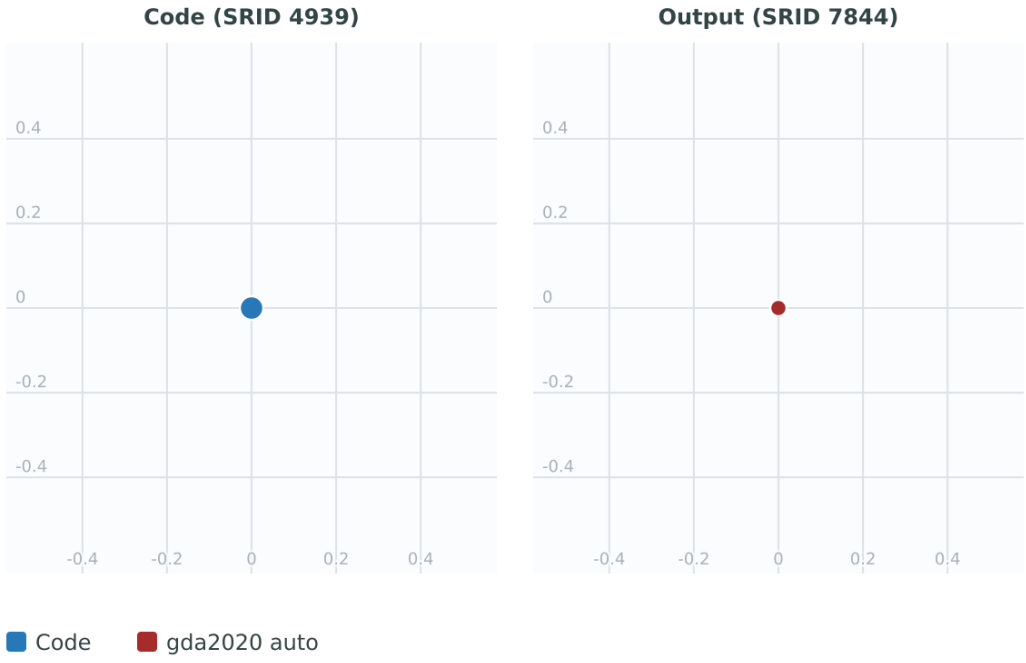
Code

```
SELECT ST_Transform('SRID=4939;POINT(143.0 -37.0)'::geometry, 7844) AS gda2020_auto;
```

Output

```
POINT(143.00000635638918 -36.999986706128176)
```

Figure



Using a defined conversion (EPSG:8447)

Code

```
SELECT
    ST_TransformPipeline(
        'SRID=4939;POINT(143.0 -37.0)'::geometry,
        'urn:ogc:def:coordinateOperation:EPSG::8447'
    ) AS gda2020_code;
```

Output

```
POINT(143.0000063280214 -36.999986718287545)
```

This example uses a PROJ pipeline definition matching EPSG:8447, as returned from projinfo. Any axisswap steps must be removed.

Code

```
SELECT
    ST_TransformPipeline(
        'SRID=4939;POINT(143.0 -37.0)'::geometry,
        '+proj=pipeline
   +step +proj=unitconvert +xy_in=deg +xy_out=rad
   +step +proj=hgridshift +grids=au_icsm_GDA94_GDA2020_conformal_and_distortion.tif
   +step +proj=unitconvert +xy_in=rad +xy_out=deg'
    ) AS gda2020_pipeline;
```

Output

```
POINT(143.0000063280214 -36.999986718287545)
```

☒☒

ST_Transform, ST_InverseTransformPipeline

7.7.6 postgis_srs_codes

postgis_srs_codes — Return the list of SRS codes associated with the given authority.

☒☒

setof text **postgis_srs_codes**(text auth_name);

☒☒

Returns a set of all `auth_srid` for the given `auth_name`.

Availability: 3.4.0

Proj version 6+

☒☒

List the first ten codes associated with the EPSG authority.

Code

```
SELECT * FROM postgis_srs_codes('EPSG') ORDER BY 1 LIMIT 10;
```

Output

```
10156
10157
10158
10160
10162
10163
10164
10165
10166
10167
```

☒☒

postgis_srs, postgis_srs_all, postgis_srs_search

7.7.7 postgis_srs

postgis_srs — Return a metadata record for the requested authority and srid.

☒☒

table(auth_name text, auth_srid text, srname text, srtext text, proj4text text, point_sw geometry, point_ne geometry) **postgis_srs**(text auth_name, text auth_srid);

☒☒

Returns a metadata record for the requested `auth_srid` for the given `auth_name`. The record will have the `auth_name`, `auth_srid`, `srname`, `srtext`, `proj4text`, and the corners of the area of usage, `point_sw` and `point_ne`.

Availability: 3.4.0

Proj version 6+

☒☒

Get the metadata for EPSG:3005.

Code

```
SELECT auth_name, auth_srid, srname,point_sw AS point_sw,point_ne AS point_ne
FROM postgis_srs('EPSG', '3005');
```

Output

```
-[ RECORD 1 ]------------------
auth_name | EPSG
auth_srid | 3005
srname    | NAD83 / BC Albers
point_sw  | POINT(-139.04 48.25)
point_ne  | POINT(-114.08 60.01)
```

☒☒

postgis_srs_codes, postgis_srs_all, postgis_srs_search

7.7.8 postgis_srs_all

postgis_srs_all — Return metadata records for every spatial reference system in the underlying Proj database.

☒☒

table(auth_name text, auth_srid text, srname text, srtext text, proj4text text, point_sw geometry, point_ne geometry) **postgis_srs_all**(void);

☒☒

Returns a set of all metadata records in the underlying Proj database. The records will have the `auth_name`, `auth_srid`, `srname`, `srtext`, `proj4text`, and the corners of the area of usage, `point_sw` and `point_ne`.

Availability: 3.4.0

Proj version 6+

☒☒

Get the first 10 metadata records from the Proj database.

Code

```
SELECT auth_name, auth_srid, srname FROM postgis_srs_all() ORDER BY auth_name, auth_srid ↩
    LIMIT 10;
```

Output

```
auth_name | auth_srid |                                srname
-----------+-----------+-------------------------------------------------------------------------
 EPSG      | 10156     | ETRS89 + MSL UK & Ireland VORF08 depth
 EPSG      | 10157     | ETRS89 + CD UK & Ireland VORF08 depth
 EPSG      | 10158     | S34J-IRF
 EPSG      | 10160     | S34J reconstruction east-orientated
 EPSG      | 10162     | JGD2011 / Japan Plane Rectangular CS I + JGD2011 (vertical) height
 EPSG      | 10163     | JGD2011 / Japan Plane Rectangular CS II + JGD2011 (vertical) height
 EPSG      | 10164     | JGD2011 / Japan Plane Rectangular CS III + JGD2011 (vertical) height
 EPSG      | 10165     | JGD2011 / Japan Plane Rectangular CS IV + JGD2011 (vertical) height
 EPSG      | 10166     | JGD2011 / Japan Plane Rectangular CS V + JGD2011 (vertical) height
 EPSG      | 10167     | JGD2011 / Japan Plane Rectangular CS VI + JGD2011 (vertical) height
```

□□

postgis_srs_codes, postgis_srs, postgis_srs_search

7.7.9 postgis_srs_search

postgis_srs_search — Return metadata records for projected coordinate systems that have areas of usage that fully contain the bounds parameter.

□□

table(auth_name text, auth_srid text, srname text, srtext text, proj4text text, point_sw geometry, point_ne geometry) **postgis_srs_search**(geometry bounds, text auth_name=EPSG);

□□

Return a set of metadata records for projected coordinate systems that have areas of usage that fully contain the bounds parameter. Each record will have the `auth_name`, `auth_srid`, `srname`, `srtext`, `proj4text`, and the corners of the area of usage, `point_sw` and `point_ne`.

The search only looks for projected coordinate systems, and is intended for users to explore the possible systems that work for the extent of their data.

Availability: 3.4.0

Proj version 6+

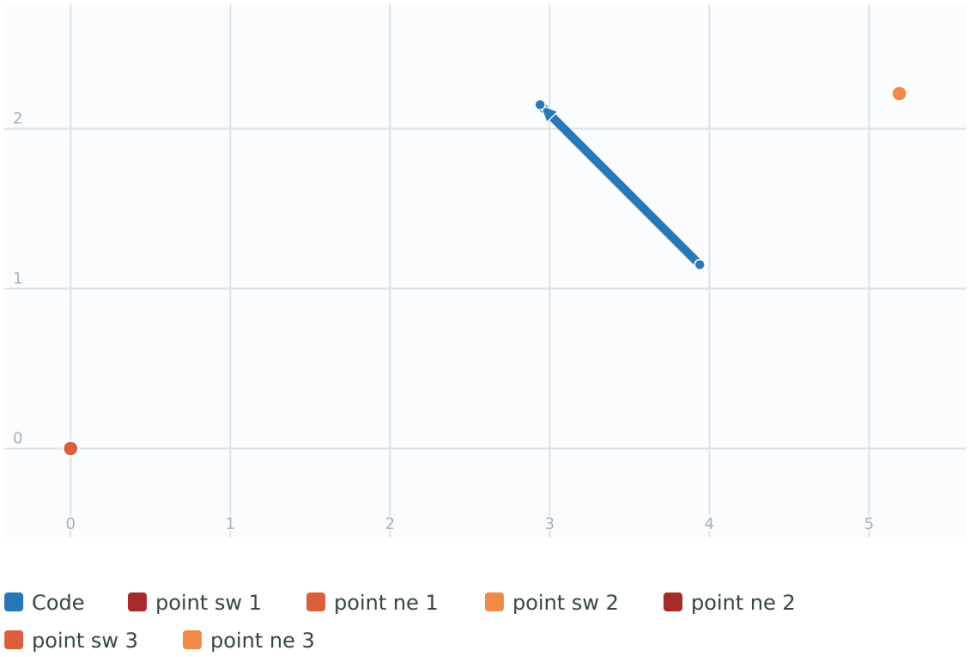
□□

Search for projected coordinate systems in Louisiana.

Code

```
SELECT auth_name, auth_srid, srname,point_sw AS point_sw,point_ne AS point_ne
FROM postgis_srs_search('SRID=4326;LINESTRING(-90 30,-91 31)')
WHERE auth_srid IN ('2801', '3452', '3457')
ORDER BY auth_name, auth_srid::integer
```

Figure



Scan a table for max extent and find projected coordinate systems that might suit.

Code

```
WITH ext AS (
  SELECT ST_Extent(geom) AS geom, Max(ST_SRID(geom)) AS srid
  FROM foo
)
SELECT auth_name, auth_srid, srname,
  point_sw AS point_sw,
  point_ne AS point_ne
FROM ext
CROSS JOIN postgis_srs_search(ST_SetSRID(ext.geom, ext.srid))
LIMIT 3;
```

☒☒

postgis_srs_codes, postgis_srs_all, postgis_srs

7.8 Geometry Input

7.8.1 Well-Known Text (WKT)

7.8.1.1 ST_BdPolyFromText

ST_BdPolyFromText — ☒☒☒☒☒☒☒ WKT ☒☒☒☒☒☒☒

⊠⊠

⊠⊠⊠⊠⊠⊠⊠⊠ WKT ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

> **⊠⊠**
> WKT ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠ ST_BdMPolyFromText ⊠⊠⊠⊠⊠⊠, ⊠⊠ PostGIS ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ST_BuildArea() ⊠⊠ ⊠⊠⊠⊠⊠.

This method implements the OGC Simple Features Implementation Specification for SQL 1.1. s3.2.6.2

GEOS ⊠⊠⊠⊠⊠

1.1.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

ST_BuildArea, ST_BdMPolyFromText

7.8.1.2 ST_BdMPolyFromText

ST_BdMPolyFromText — ⊠⊠⊠⊠⊠⊠⊠ WKT ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

geometry **ST_BdMPolyFromText**(text WKT, integer srid);

⊠⊠

⊠⊠⊠⊠⊠⊠⊠⊠ WKT ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

> **⊠⊠**
> WKT ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ST_BdPolyFromText ⊠⊠⊠ ⊠⊠⊠, ⊠⊠ PostGIS ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ST_BuildArea() ⊠⊠⊠⊠⊠⊠⊠.

This method implements the OGC Simple Features Implementation Specification for SQL 1.1. s3.2.6.2

GEOS ⊠⊠⊠⊠⊠

1.1.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

ST_BuildArea, ST_BdPolyFromText

7.8.1.3 ST_GeogFromText

ST_GeogFromText — WKT (□□) □□□□□□□□□□□□□□□□□□□□□□□□□□□□□.

□□

geography **ST_GeogFromText**(text EWKT);

□□

WKT □□□□□□□□ WKT □□□□□□□□□□□□□□□□□□□□□□□. □□□□□□□□□□□□□□□□ SRID 4326 □□□□□□□□. □□□□□ ST_GeographyFromText □□□□□□□. □□□□□□□□□□□□□□□□□□□□□□□□.

□□

Add a WGS 84 geography point column to a table.

Code

```
ALTER TABLE sometable ADD COLUMN geog geography(POINT, 4326);
```

Populate the geography column from longitude and latitude columns.

Code

```
UPDATE sometable
SET geog = ST_GeogFromText('SRID=4326;POINT(' || lon || ' ' || lat || ')');
```

Specify a geography point using EPSG:4267, NAD27.

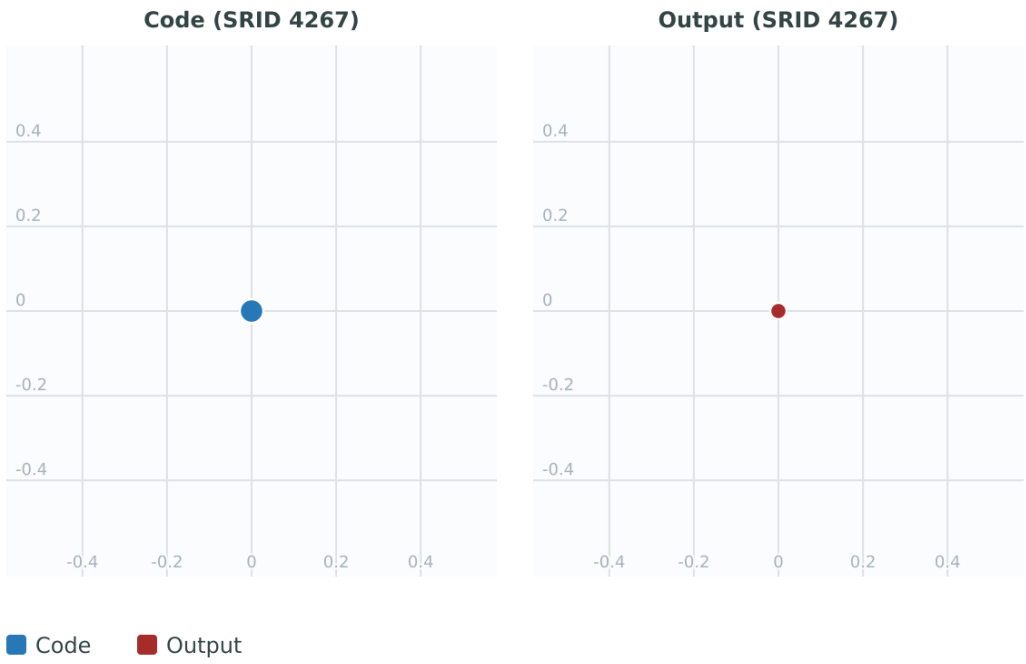
Code

```
SELECT ST_GeogFromText('SRID=4267;POINT(-77.0092 38.889588)');
```

Output

```
SRID=4267;POINT(-77.0092 38.889588)
```

Figure



☒☒

ST_AsText, ST_GeographyFromText

7.8.1.4 ST_GeographyFromText

ST_GeographyFromText — WKT (☒☒) ☒☒☒☒☒☒

□□

geometry **ST_GeomCollFromText**(text WKT, integer srid);
geometry **ST_GeomCollFromText**(text WKT);

□□

Makes a collection Geometry from the Well-Known-Text (WKT) representation with the given SRID. If SRID is not given, it defaults to 0.

OGC □□ 3.2.6.2 - □□□□ SRID □□□□□□□ (conformance suite) □□□□□□□□.

WKT □□□□□ (GEOMETRYCOLLECTION) □□□□□ null □□□□□□.



□□
□□□ WKT □□□□□□□□□□□□□□□, □□□□□□□□□□□□. □□□□□□□□□□□□
□□□□□□□□ ST_GeomFromText □□□□□□.

 This method implements the OGC Simple Features Implementation Specification for SQL 1.1. s3.2.6.2

 This method implements the SQL/MM specification.

□□

Code

```
SELECT ST_GeomCollFromText('GEOMETRYCOLLECTION(POINT(1 2), LINESTRING(1 2,3 4))');
```

□□

ST_GeomFromText, ST_SRID

7.8.1.6 ST_GeomFromEWKT

ST_GeomFromEWKT — EWKT(Extended Well-Known Text) □□□□□□ ST_Geometry □□□□□□□.

□□

geometry **ST_GeomFromEWKT**(text EWKT);

□□

OGC EWKT(Extended Well-Known Text) □□□□□□□ PostGIS ST_Geometry □□□□□□□□.



□□
EWKT □□□ OGC □□□□□□□, SRID(□□□□□□□□□□) □□□□□ PostGIS □□□□□□
□.

□□□□: 2.0.0 □□□□□□□□□□ (polyhedral surface) □ TIN □□□□□□.

This function supports 3d and will not drop the z-index.

This method supports Circular Strings and Curves.

This function supports Polyhedral surfaces.

This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).

□□

Create a LineString with SRID 4269.

Code

```
SELECT ST_GeometryType(geom) AS geometry_type, ST_SRID(geom) AS srid
FROM ST_GeomFromEWKT('SRID=4269;LINESTRING(-71.160281 42.258729,-71.160837 ↩
    42.259113,-71.161144 42.25932)') AS geom;
```

Output

```
geometry_type | srid
--------------+------
 ST_LineString | 4269
(1 row)
```

Create a MultiLineString with SRID 4269.

Code

```
SELECT ST_GeometryType(geom) AS geometry_type, ST_SRID(geom) AS srid
FROM ST_GeomFromEWKT('SRID=4269;MULTILINESTRING((-71.160281 42.258729,-71.160837 ↩
    42.259113,-71.161144 42.25932))') AS geom;
```

Output

```
geometry_type    | srid
--------------------+------
 ST_MultiLineString  | 4269
(1 row)
```

Create a Point with SRID 4269.

Code

```
SELECT ST_GeometryType(geom) AS geometry_type, ST_SRID(geom) AS srid
FROM ST_GeomFromEWKT('SRID=4269;POINT(-71.064544 42.28787)') AS geom;
```

Output

```
geometry_type | srid
--------------+------
 ST_Point      | 4269
(1 row)
```

Create a Polygon with SRID 4269.

Code

```
SELECT ST_GeometryType(geom) AS geometry_type, ST_SRID(geom) AS srid
FROM ST_GeomFromEWKT('SRID=4269;POLYGON((-71.1776585052917 ↩
    42.3902909739571,-71.1776820268866 42.3903701743239,
-71.1776063012595 42.3903825660754,-71.1775826583081 42.3903033653531,-71.1776585052917 ↩
    42.3902909739571))') AS geom;
```

Output

```
geometry_type | srid
--------------+------
 ST_Polygon    | 4269
(1 row)
```

Create a MultiPolygon with SRID 4269.

Code

```
SELECT ST_GeometryType(geom) AS geometry_type, ST_SRID(geom) AS srid
FROM ST_GeomFromEWKT('SRID=4269;MULTIPOLYGON(((-71.1031880899493 42.3152774590236,
-71.1031627617667 42.3152960829043,-71.102923838298 42.3149156848307,
-71.1023097974109 42.3151969047397,-71.1019285062273 42.3147384934248,
-71.102505233663 42.3144722937587,-71.10277487471 42.3141658254797,
-71.103113945163 42.3142739188902,-71.10324876416 42.31402489987,
-71.1033002961013 42.3140393340215,-71.1033488797549 42.3139495090772,
-71.103396240451 42.3138632439557,-71.1041521907712 42.3141153348029,
-71.1041411411543 42.3141545014533,-71.1041287795912 42.3142114839058,
-71.1041188134329 42.3142693656241,-71.1041112482575 42.3143272556118,
-71.1041072845732 42.3143851580048,-71.1041057218871 42.3144430686681,
-71.1041065602059 42.3145009876017,-71.1041097995362 42.3145589148055,
-71.1041166403905 42.3146168544148,-71.1041258822717 42.3146748022936,
-71.1041375307579 42.3147318674446,-71.1041492906949 42.3147711126569,
-71.1041598612795 42.314808571739,-71.1042515013869 42.3151287620809,
-71.1041173835118 42.3150739481917,-71.1040809891419 42.3151344119048,
-71.1040438678912 42.3151191367447,-71.1040194562988 42.3151832057859,
-71.1038734225584 42.3151140942995,-71.1038446938243 42.3151006300338,
-71.1038315271889 42.315094347535,-71.1037393329282 42.315054824985,
-71.1035447555574 42.3152608696313,-71.1033436658644 42.3151648370544,
-71.1032580383161 42.3152269126061,-71.103223066939 42.3152517403219,
-71.1031880899493 42.3152774590236)),
((-71.1043632495873 42.315113108546,-71.1043583974082 42.3151211109857,
-71.1043443253471 42.3150676015829,-71.1043850704575 42.3150793250568,-71.1043632495873 ↩
    42.315113108546)))') AS geom;
```

Output

```
geometry_type   | srid
------------------+------
 ST_MultiPolygon   | 4269
(1 row)
```

Create a 3D CircularString without an SRID.

Code

```
SELECT ST_GeometryType(geom) AS geometry_type, ST_SRID(geom) AS srid
FROM ST_GeomFromEWKT('CIRCULARSTRING(220268 150415 1,220227 150505 2,220227 150406 3)') AS ↩
    geom;
```

Output

```
geometry_type    | srid
-------------------+------
 ST_CircularString  | 0
(1 row)
```

Create a PolyhedralSurface without an SRID.

Code

```
SELECT ST_GeometryType(geom) AS geometry_type, ST_SRID(geom) AS srid
FROM ST_GeomFromEWKT('POLYHEDRALSURFACE(
    ((0 0 0,0 0 1,0 1 1,0 1 0,0 0 0)),
    ((0 0 0,0 1 0,1 1 0,1 0 0,0 0 0)),
    ((0 0 0,1 0 0,1 0 1,0 0 1,0 0 0)),
    ((1 1 0,1 1 1,1 0 1,1 0 0,1 1 0)),
    ((0 1 0,0 1 1,1 1 1,1 1 0,0 1 0)),
    ((0 0 1,1 0 1,1 1 1,0 1 1,0 0 1))
)') AS geom;
```

Output

```
geometry_type       | srid
--------------------------+------
 ST_PolyhedralSurface     | 0
(1 row)
```

☒☒

ST_AsEWKT, ST_GeomFromText

7.8.1.7 ST_GeomFromMARC21

ST_GeomFromMARC21 — Takes MARC21/XML geographic data as input and returns a PostGIS geometry object.

☒☒

geometry **ST_GeomFromMARC21** (text marcxml);

☒☒

This function creates a PostGIS geometry from a MARC21/XML record, which can contain a `POINT` or a `POLYGON`. In case of multiple geographic data entries in the same MARC21/XML record, a `MULTIPOINT` or `MULTIPOLYGON` will be returned. If the record contains mixed geometry types, a `GEOMETRYCOLLECTION` will be returned. It returns NULL if the MARC21/XML record does not contain any geographic data (datafield:034).

LOC MARC21/XML versions supported:

- MARC21/XML 1.1

Availability: 3.3.0, requires libxml2 2.6+



□□
The MARC21/XML Coded Cartographic Mathematical Data currently does not provide any means to describe the Spatial Reference System of the encoded coordinates, so this function will always return a geometry with SRID 0.



□□
Returned POLYGON geometries will always be clockwise oriented.

□□

Converting MARC21/XML geographic data containing a single POINT encoded as hddd.dddddd

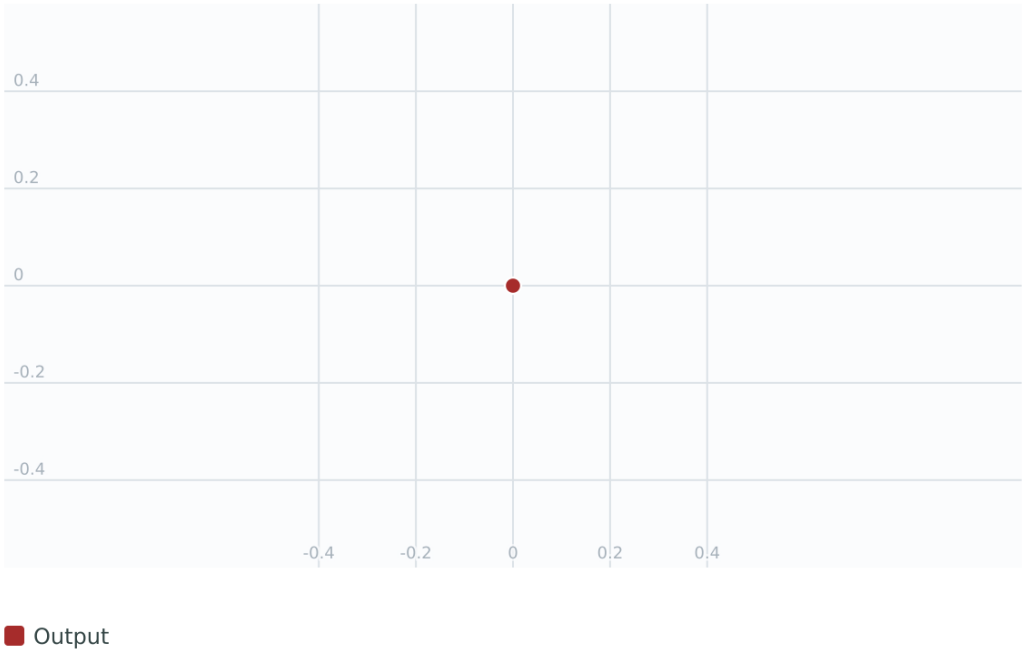
Code

```
SELECT ST_GeomFromMARC21(
'<record xmlns="http://www.loc.gov/MARC21/slim">
    <leader
>00000nz a2200000nc 4500</leader>
    <controlfield tag="001"
>040277569</controlfield>
    <datafield tag="034" ind1=" " ind2=" ">
        <subfield code="d"
>W004.500000</subfield>
        <subfield code="e"
>W004.500000</subfield>
        <subfield code="f"
>N054.250000</subfield>
        <subfield code="g"
>N054.250000</subfield>
    </datafield>
</record
>'
);
```

Output

```
POINT(-4.5 54.25)
```

Figure



Converting MARC21/XML geographic data containing a single `POLYGON` encoded as `hdddmmss`

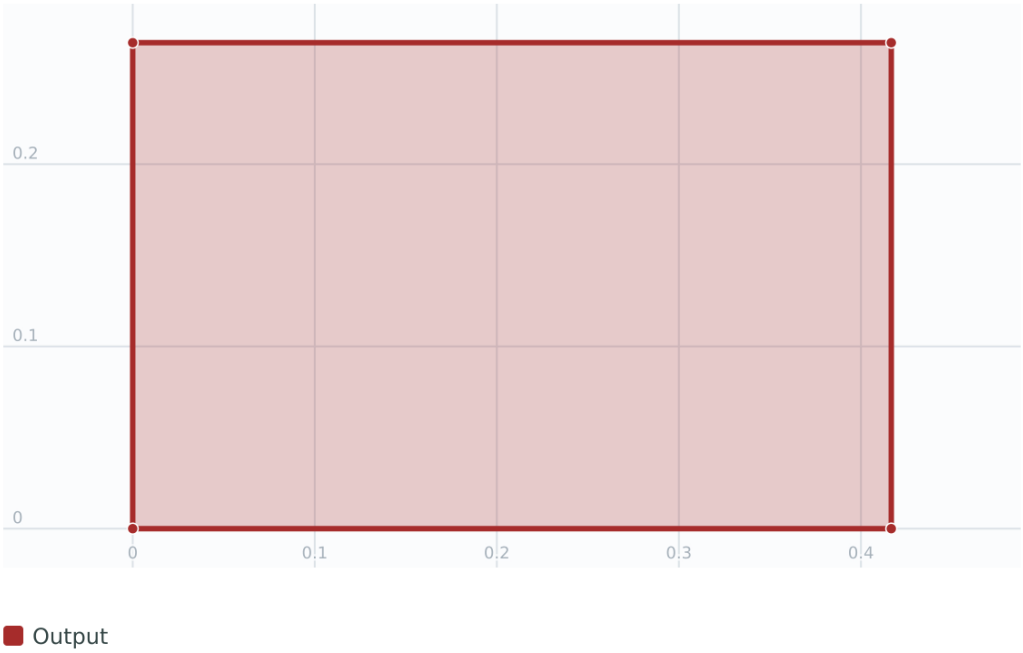
Code

```
SELECT ST_GeomFromMARC21(
'<record xmlns="http://www.loc.gov/MARC21/slim">
    <leader
>01062cem a2200241 a 4500</leader>
    <controlfield tag="001"
>   84696781 </controlfield>
    <datafield tag="034" ind1="1" ind2=" ">
        <subfield code="a"
>a</subfield>
        <subfield code="b"
>50000</subfield>
        <subfield code="d"
>E0130600</subfield>
        <subfield code="e"
>E0133100</subfield>
        <subfield code="f"
>N0523900</subfield>
        <subfield code="g"
>N0522300</subfield>
    </datafield>
</record
>'
);
```

Output

```
POLYGON((13.1 52.65,13.516666666666667 52.65,13.516666666666667 52.38333333333333,13.1 52.38333333333333,13.1 52.65))
```

Figure



Converting MARC21/XML geographic data containing a `POLYGON` and a `POINT`:

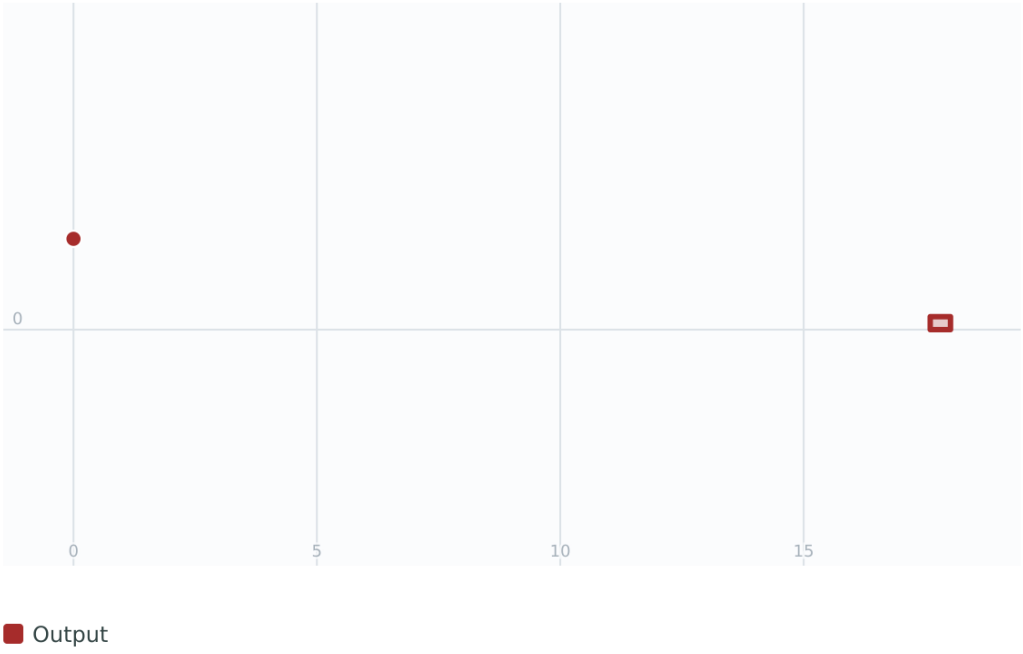
Code

```
SELECT ST_GeomFromMARC21(
'<record xmlns="http://www.loc.gov/MARC21/slim">
    <datafield tag="034" ind1="1" ind2=" ">
        <subfield code="a"
>a</subfield>
        <subfield code="b"
>50000</subfield>
        <subfield code="d"
>E0130600</subfield>
        <subfield code="e"
>E0133100</subfield>
        <subfield code="f"
>N0523900</subfield>
        <subfield code="g"
>N0522300</subfield>
    </datafield>
    <datafield tag="034" ind1=" " ind2=" ">
        <subfield code="d"
>W004.500000</subfield>
        <subfield code="e"
>W004.500000</subfield>
        <subfield code="f"
>N054.250000</subfield>
        <subfield code="g"
>N054.250000</subfield>
    </datafield>
</record
>'
);
```

Output

```
GEOMETRYCOLLECTION(POLYGON((13.1 52.65,13.516666666666667 52.65,13.516666666666667 52.38333333333333,13.1 ↩
    52.38333333333333,13.1 52.65)), POINT(-4.5 54.25))
```

Figure



☒☒

ST_AsMARC21

7.8.1.8 ST_GeometryFromText

ST_GeometryFromText — Returns a geometry from Well-Known Text (WKT). Alias for ST_GeomFromText.

☒☒

geometry **ST_GeometryFromText**(text WKT);
geometry **ST_GeometryFromText**(text WKT, integer srid);

☒☒

This function is an alias for ST_GeomFromText and provides the same signatures.
1.2.2 ☒☒☒☒☒☒☒☒☒☒☒.

This method implements the OGC Simple Features Implementation Specification for SQL 1.1.

This method implements the SQL/MM specification. SQL-MM 3: 5.1.40

☒☒

ST_GeomFromText

7.8.1.9 ST_GeomFromText

ST_GeomFromText — WKT □□□□□□□□□□ ST_Geometry □□□□□□□.

□□

geometry **ST_GeomFromText**(text WKT);
geometry **ST_GeomFromText**(text WKT, integer srid);

□□

OGC WKT(Well-Known Text) □□□□□□□ PostGIS ST_Geometry □□□□□□□.



□□
ST_GeomFromText □□□□□□ 2 □□□□, □□□□ SRID □□□□□□□□□□□□□□□□□□□□ □□□ (SRID=0) □□□□□□□□. □□□□ SRID □□□□□□□□□□□□□ SRID □□□□□□□ □□□□□□□□□□□□□□□□□□.


This method implements the OGC Simple Features Implementation Specification for SQL 1.1. □□ 3.2.6.2 - □□□□ SRID □□□□□□□ (conformance suite) □□□□□□□□□.


This method implements the SQL/MM specification. SQL-MM 3: 5.1.40


This method supports Circular Strings and Curves.



□□
While not OGC-compliant, ST_MakePoint is faster than ST_GeomFromText and ST_PointFromText. It is also easier to use for numeric coordinate values. ST_Point is another option similar in speed to ST_MakePoint and is OGC-compliant, but doesn't support anything but 2D points.

□□□□: PostGIS 2.0.0 □□□□□□□ ST_GeomFromText('GEOMETRYCOLLECTION(EMPTY)') □□ □□□□□□□□□. PostGIS 2.0.0 □□□□, SQL/MM □□□□□□□□□□□□□□□□□□□□□□□□□. □□ □ ST_GeomFromText('GEOMETRYCOLLECTION EMPTY') □□□□□□□□□.

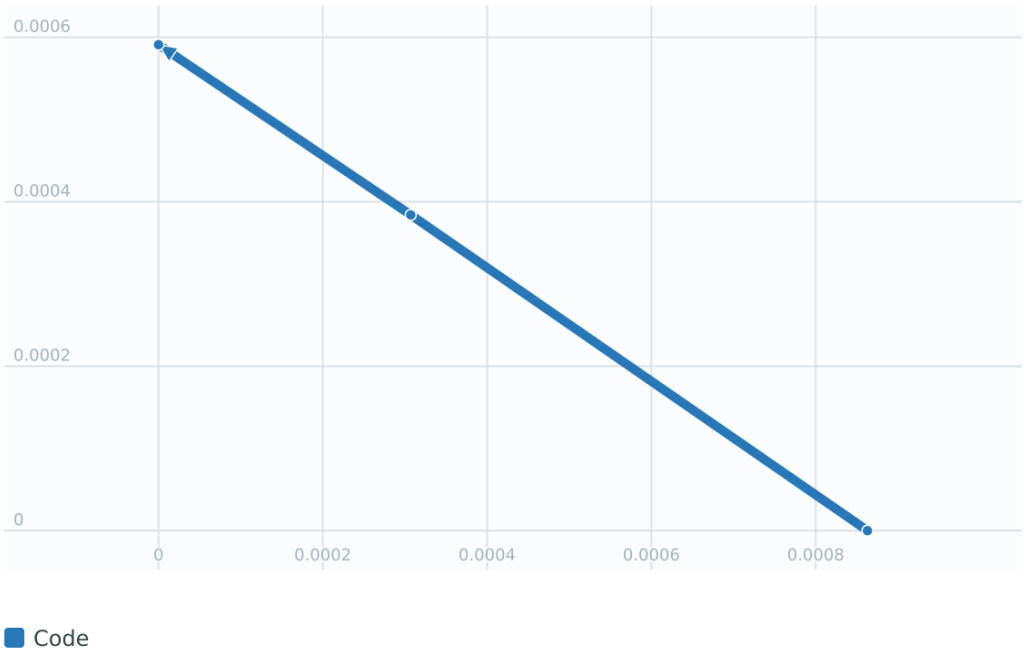
□□

Create a LineString with SRID 0.

Code

```
SELECT ST_SRID(ST_GeomFromText('LINESTRING(-71.160281 42.258729,-71.160837 ↩
    42.259113,-71.161144 42.25932)'));
```

Figure

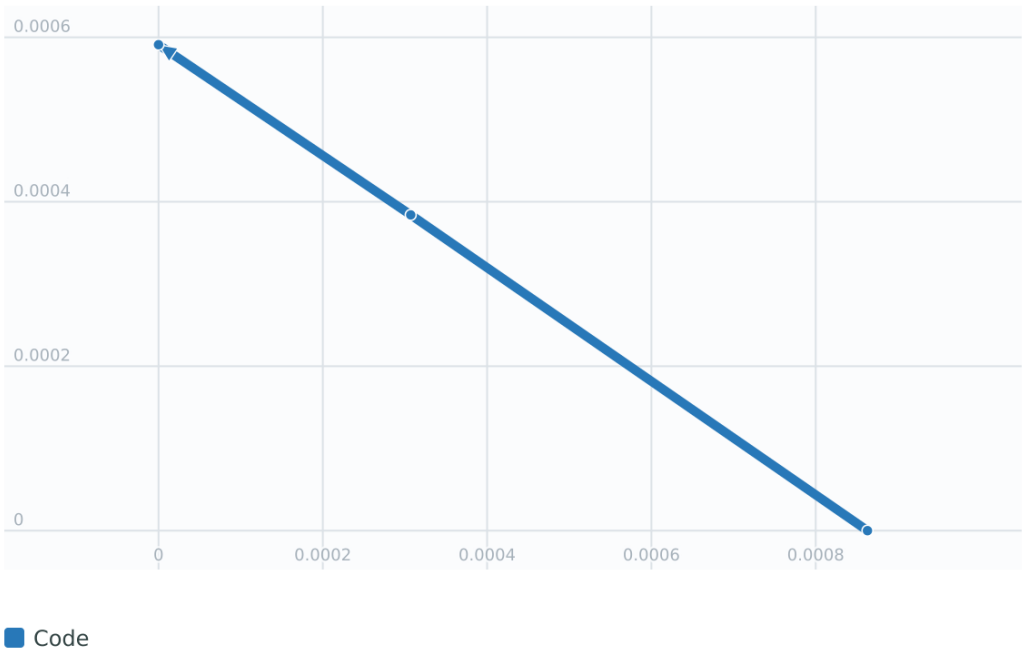


Create the same LineString with SRID 4269.

Code

```
SELECT ST_SRID(ST_GeomFromText('LINESTRING(-71.160281 42.258729,-71.160837 ↩
    42.259113,-71.161144 42.25932)', 4269));
```

Figure

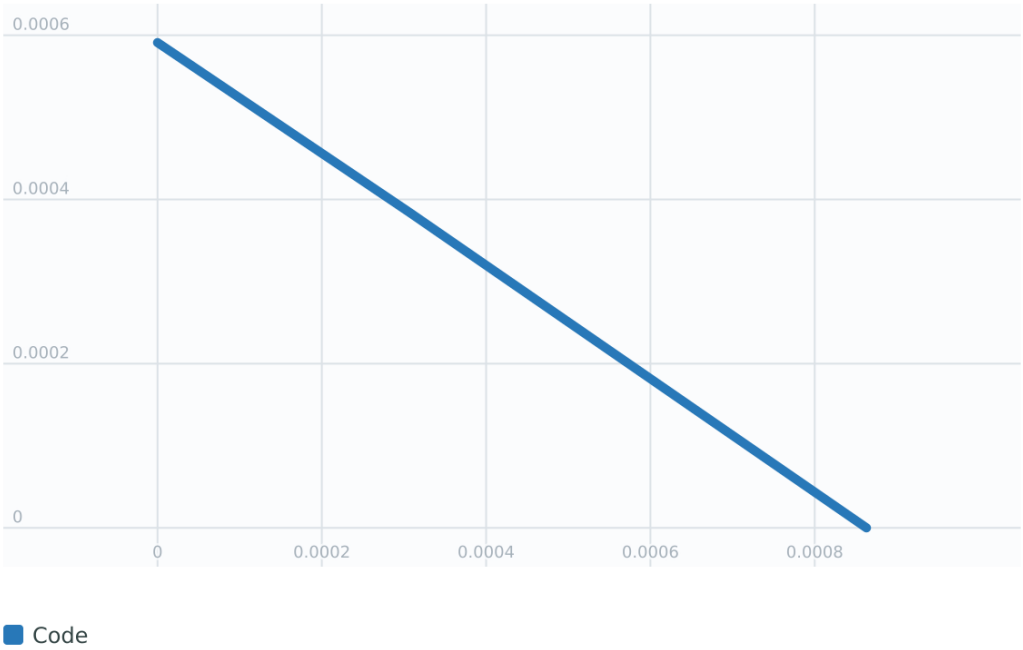


Create a MultiLineString.

Code

```
SELECT ST_GeometryType(ST_GeomFromText('MULTILINESTRING((-71.160281 42.258729,-71.160837 ↩
    42.259113,-71.161144 42.25932))'));
```

Figure



Create a Point.

Code

```
SELECT ST_GeometryType(ST_GeomFromText('POINT(-71.064544 42.28787)'));
```

Output

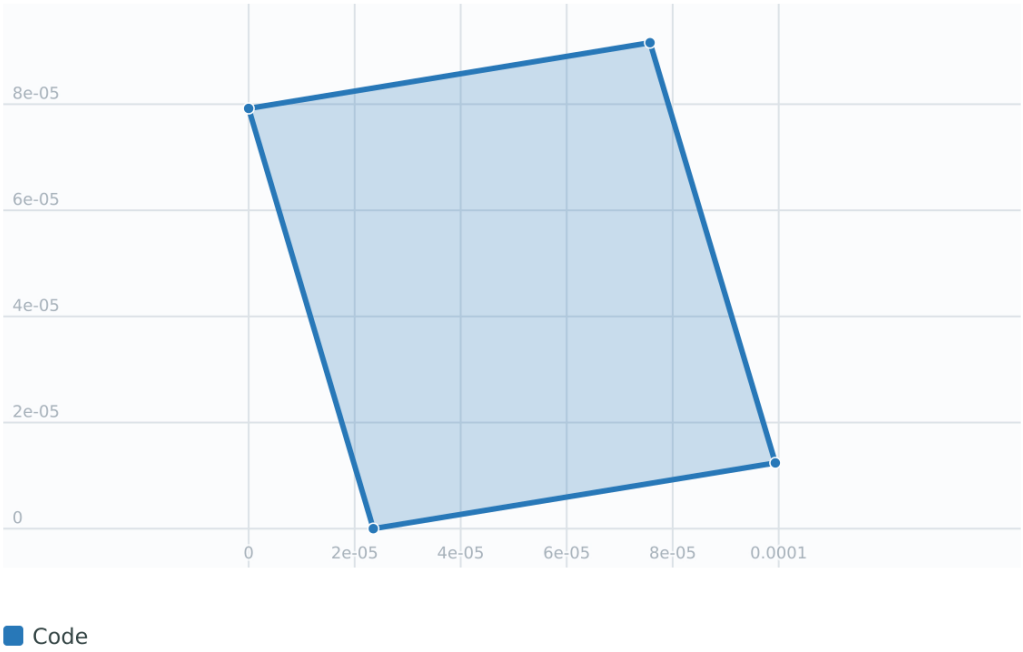
```
ST_Point
```

Create a Polygon.

Code

```
SELECT ST_GeometryType(ST_GeomFromText('POLYGON((-71.1776585052917 ↩
    42.3902909739571,-71.1776820268866 42.3903701743239,
-71.1776063012595 42.3903825660754,-71.1775826583081 42.3903033653531,-71.1776585052917 ↩
    42.3902909739571))'));
```

Figure

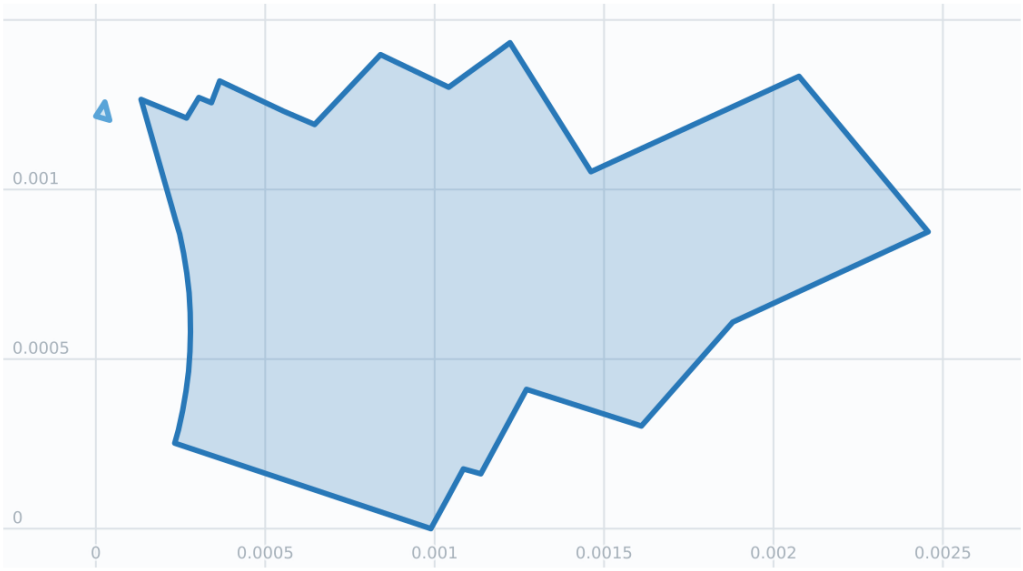


Create a MultiPolygon with SRID 4326.

Code

```
SELECT ST_GeometryType(ST_GeomFromText('MULTIPOLYGON(((-71.1031880899493 42.3152774590236,
-71.1031627617667 42.3152960829043,-71.102923838298 42.3149156848307,
-71.1023097974109 42.3151969047397,-71.1019285062273 42.3147384934248,
-71.102505233663 42.3144722937587,-71.10277487471 42.3141658254797,
-71.103113945163 42.3142739188902,-71.10324876416 42.31402489987,
-71.1033002961013 42.3140393340215,-71.1033488797549 42.3139495090772,
-71.103396240451 42.3138632439557,-71.1041521907712 42.3141153348029,
-71.1041411411543 42.3141545014533,-71.1041287795912 42.3142114839058,
-71.1041188134329 42.3142693656241,-71.1041112482575 42.3143272556118,
-71.1041072845732 42.3143851580048,-71.1041057218871 42.3144430686681,
-71.1041065602059 42.3145009876017,-71.1041097995362 42.3145589148055,
-71.1041166403905 42.3146168544148,-71.1041258822717 42.3146748022936,
-71.1041375307579 42.3147318674446,-71.1041492906949 42.3147711126569,
-71.1041598612795 42.314808571739,-71.1042515013869 42.3151287620809,
-71.1041173835118 42.3150739481917,-71.1040809891419 42.3151344119048,
-71.1040438678912 42.3151191367447,-71.1040194562988 42.3151832057859,
-71.1038734225584 42.3151140942995,-71.1038446938243 42.3151006300338,
-71.1038315271889 42.315094347535,-71.1037393329282 42.315054824985,
-71.1035447555574 42.3152608696313,-71.1033436658644 42.3151648370544,
-71.1032580383161 42.3152269126061,-71.103223066939 42.3152517403219,
-71.1031880899493 42.3152774590236)),
((-71.1043632495873 42.315113108546,-71.1043583974082 42.3151211109857,
-71.1043443253471 42.3150676015829,-71.1043850704575 42.3150793250568,-71.1043632495873 ↩
    42.315113108546)))', 4326));
```

Figure

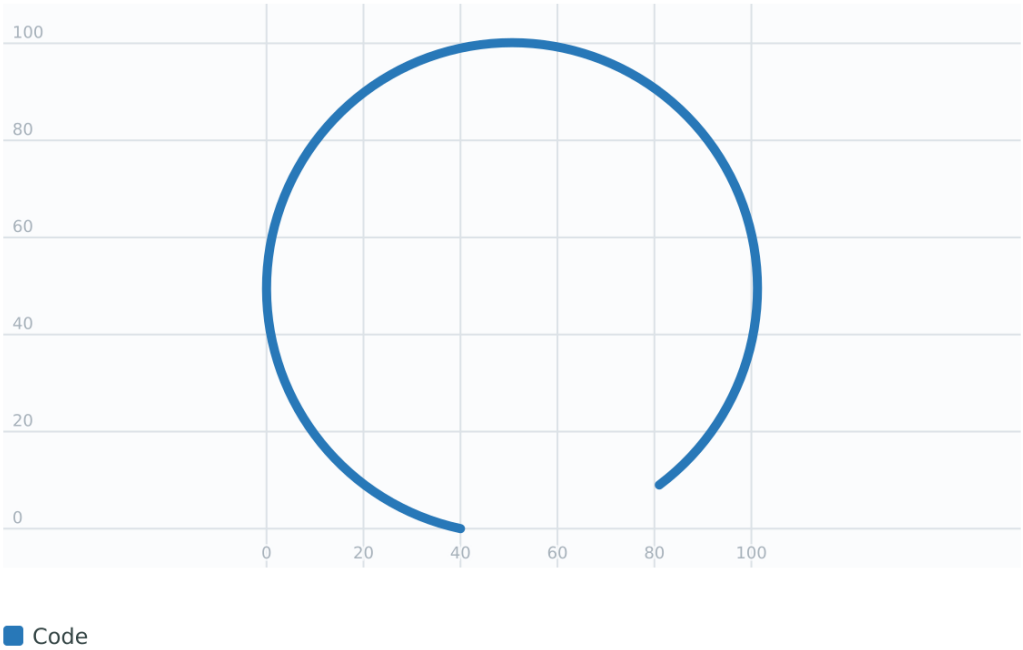


Create a CircularString.

Code

```
SELECT ST_GeometryType(ST_GeomFromText('CIRCULARSTRING(220268 150415,220227 150505,220227 ↩
    150406)'));
```

Figure



□□

ST_GeomFromEWKT, ST_GeomFromWKB, ST_SRID

7.8.1.10 ST_LineFromText

ST_LineFromText — □□□ SRID □□□ WKT □□□□□□□□□□□□□□□. SRID □□□□□□□□□, □□□□ 0 □□□□□.

□□

geometry **ST_LineFromText**(text WKT);
geometry **ST_LineFromText**(text WKT, integer srid);

□□

Makes a Geometry from WKT with the given SRID. If SRID is not given, it defaults to 0. If WKT passed in is not a LINESTRING, then null is returned.



□□
OGC □□ 3.2.6.2 - □□□□ SRID □□□□□□□ (conformance suite) □□□□□□□□□.



□□
□□□□□□□□□□□□□□□□□□□□□□□, □□ ST_GeomFromText □□□□□□□□□□□□□□□. □□□□ ST_GeomFromText □□□□□, □□□□□□□□□□□□□□□□□□□□□□□□□□.

 This method implements the OGC Simple Features Implementation Specification for SQL 1.1. s3.2.6.2

 This method implements the SQL/MM specification. SQL-MM 3: 7.2.8

□□

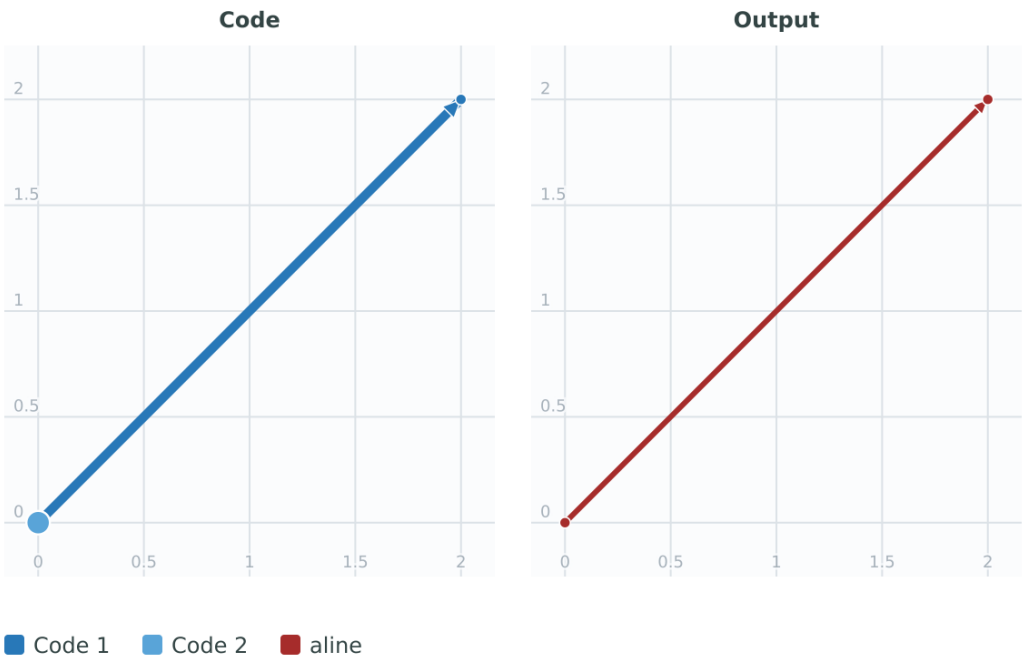
Code

```
SELECT ST_LineFromText('LINESTRING(1 2,3 4)') AS aline,
ST_LineFromText('POINT(1 2)') IS NULL AS null_return;
```

Output

```
aline                | null_return
-----------------------------------
LINESTRING(1 2,3 4)  | t
```

Figure



□□

ST_GeomFromText

7.8.1.11 ST_MLineFromText

ST_MLineFromText — WKT □□□□□□□□□□ ST_MultiLineString □□□□□□□.

□□

geometry **ST_MLineFromText**(text WKT, integer srid);
geometry **ST_MLineFromText**(text WKT);

□□

Makes a Geometry from Well-Known-Text (WKT) with the given SRID. If SRID is not given, it defaults to 0.

OGC □□ 3.2.6.2 - □□□□ SRID □□□□□□□ (conformance suite) □□□□□□□□□.

WKT □□□□□□□□□□□□□ null □□□□□□.

Note! □□
□□□ WKT □□□□□□□□□□□□□□□□, □□□□□□□□□□□□. □□□□□□□□□□□□□
□□□□□□□□ ST_GeomFromText □□□□□□.

This method implements the OGC Simple Features Implementation Specification for SQL 1.1. s3.2.6.2

This method implements the SQL/MM specification. SQL-MM 3: 9.4.4

□□

Code

```
SELECT ST_MLineFromText('MULTILINESTRING((1 2,3 4),(4 5,6 7))');
```

□□

ST_GeomFromText

7.8.1.12 ST_MPointFromText

ST_MPointFromText — Makes a Geometry from WKT with the given SRID. If SRID is not given, it defaults to 0.

□□

geometry **ST_MPointFromText**(text WKT, integer srid);
geometry **ST_MPointFromText**(text WKT);

□□

Makes a Geometry from WKT with the given SRID. If SRID is not given, it defaults to 0.

OGC □□ 3.2.6.2 - □□□□ SRID □□□□□□□ (conformance suite) □□□□□□□□.

WKT □□□□□□□□□□□ null □□□□□□.



□□
□□□ WKT □□□□□□□□□□□□□□□, □□□□□□□□□□□□. □□□□□□□□□□□□
□□□□□□□□ ST_GeomFromText □□□□□□.

This method implements the OGC Simple Features Implementation Specification for SQL 1.1. 3.2.6.2

This method implements the SQL/MM specification. SQL-MM 3: 9.2.4

□□

Code

```
SELECT ST_MPointFromText('MULTIPOINT((1 2),(3 4))');
SELECT ST_MPointFromText('MULTIPOINT((-70.9590 42.1180),(-70.9611 42.1223))', 4326);
```

□□

ST_GeomFromText

7.8.1.13 ST_MPolyFromText

ST_MPolyFromText — Makes a MultiPolygon Geometry from WKT with the given SRID. If SRID is not given, it defaults to 0.

□□

geometry **ST_MPolyFromText**(text WKT, integer srid);
geometry **ST_MPolyFromText**(text WKT);

□□

Makes a MultiPolygon from WKT with the given SRID. If SRID is not given, it defaults to 0.

OGC □□ 3.2.6.2 - □□□□ SRID □□□□□□□ (conformance suite) □□□□□□□□□.

WKT □□□□□□□□□□□□□□□□□□□□□.



□□
□□□ WKT □□□□□□□□□□□□□□□□□□□□, □□□□□□□□□□□□. □□□□□□□□□ □□□□□□□□□□□□ ST_GeomFromText □□□□□□.

 This method implements the OGC Simple Features Implementation Specification for SQL 1.1. s3.2.6.2

 This method implements the SQL/MM specification. SQL-MM 3: 9.6.4

□□

Create a 3D MultiPolygon with an interior ring and SRID 0.

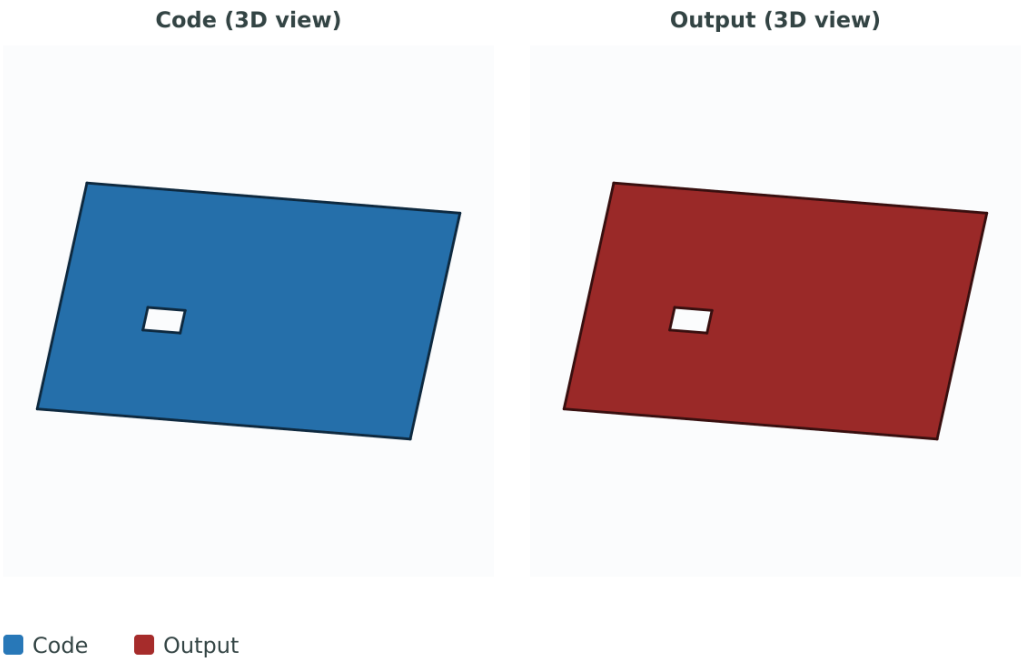
Code

```
SELECT ST_MPolyFromText(
    'MULTIPOLYGON(((0 0 1,20 0 1,20 20 1,0 20 1,0 0 1),
                   (5 5 3,5 7 3,7 7 3,7 5 3,5 5 3)))'
);
```

Output

```
MULTIPOLYGON(((0 0 1,20 0 1,20 20 1,0 20 1,0 0 1),(5 5 3,5 7 3,7 7 3,7 5 3,5 5 3)))
```

Figure

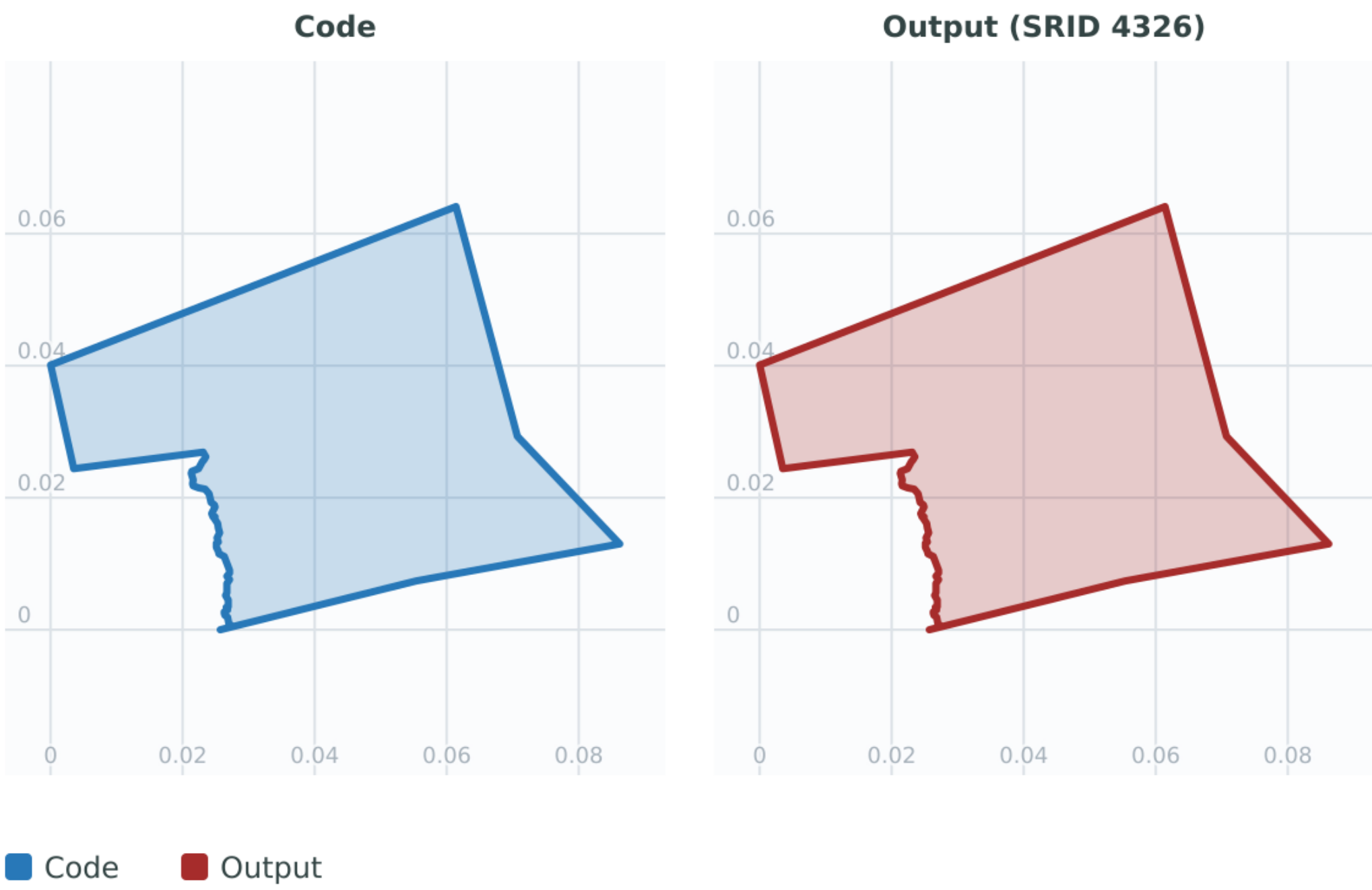


Create a MultiPolygon with SRID 4326.

Code

```
SELECT ST_MPolyFromText(
    'MULTIPOLYGON(((-70.916 42.1002,-70.9468 42.0946,-70.9765 42.0872,
      -70.9754 42.0875,-70.9749 42.0879,-70.9752 42.0881,-70.9754 42.0891,
      -70.9758 42.0894,-70.9759 42.0897,-70.9759 42.0899,-70.9754 42.0902,
      -70.9756 42.0906,-70.9753 42.0907,-70.9753 42.0917,-70.9757 42.0924,
      -70.9755 42.0928,-70.9755 42.0942,-70.9751 42.0948,-70.9755 42.0953,
      -70.9751 42.0958,-70.9751 42.0962,-70.9759 42.0983,-70.9767 42.0987,
      -70.9768 42.0991,-70.9771 42.0997,-70.9771 42.1003,-70.9768 42.1005,
      -70.977 42.1011,-70.9766 42.1019,-70.9768 42.1026,-70.9769 42.1033,
      -70.9775 42.1042,-70.9773 42.1043,-70.9776 42.1043,-70.9778 42.1048,
      -70.9773 42.1058,-70.9774 42.1061,-70.9779 42.1065,-70.9782 42.1078,
      -70.9788 42.1085,-70.9798 42.1087,-70.9806 42.109,-70.9807 42.1093,
      -70.9806 42.1099,-70.9809 42.1109,-70.9808 42.1112,-70.9798 42.1116,
      -70.9792 42.1127,-70.979 42.1129,-70.9787 42.1134,-70.979 42.1139,
      -70.9791 42.1141,-70.9987 42.1116,-71.0022 42.1273,-70.9408 42.1513,
      -70.9315 42.1165,-70.916 42.1002)))',
    4326
);
```

Figure



☒☒

ST_GeomFromText, ST_SRID

7.8.1.14 ST_PointFromText

ST_PointFromText — ☒☒☒ SRID ☒☒☒ WKT ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒. SRID ☒☒☒☒☒☒☒☒☒☒, ☒☒☒☒ 0 ☒☒☒☒.

☒☒

geometry **ST_PointFromText**(text WKT);
geometry **ST_PointFromText**(text WKT, integer srid);

☒☒

Constructs a PostGIS ST_Geometry point object from the OGC Well-Known text representation. If SRID is not given, it defaults to unknown (currently 0). If geometry is not a WKT point representation, returns null. If completely invalid WKT, then throws an error.

☒☒
ST_PointFromText ☒☒☒☒☒☒☒ 2 ☒☒☒☒☒, ☒☒☒☒☒ SRID ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒ SRID ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒ SRID ☒☒☒☒☒ ST_Geometry ☒☒☒☒☒☒. spatial_ref_sys ☒☒☒☒☒☒☒☒☒☒☒ SRID ☒☒☒☒☒.

> **⊠⊠**
> ⊠⊠⊠ WKT ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠ ST_GeomFromText ⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ OGC ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ST_MakePoint ⊠⊠ OGC ⊠⊠⊠⊠⊠⊠⊠⊠ ST_Point ⊠ ⊠⊠⊠⊠⊠⊠⊠⊠.

This method implements the OGC Simple Features Implementation Specification for SQL 1.1. ⊠ ⊠ 3.2.6.2 - ⊠⊠⊠⊠ SRID ⊠⊠⊠⊠⊠⊠⊠ (conformance suite) ⊠⊠⊠⊠⊠⊠⊠⊠⊠.

This method implements the SQL/MM specification. SQL-MM 3: 6.1.8

⊠⊠

Create a point with the default SRID 0.

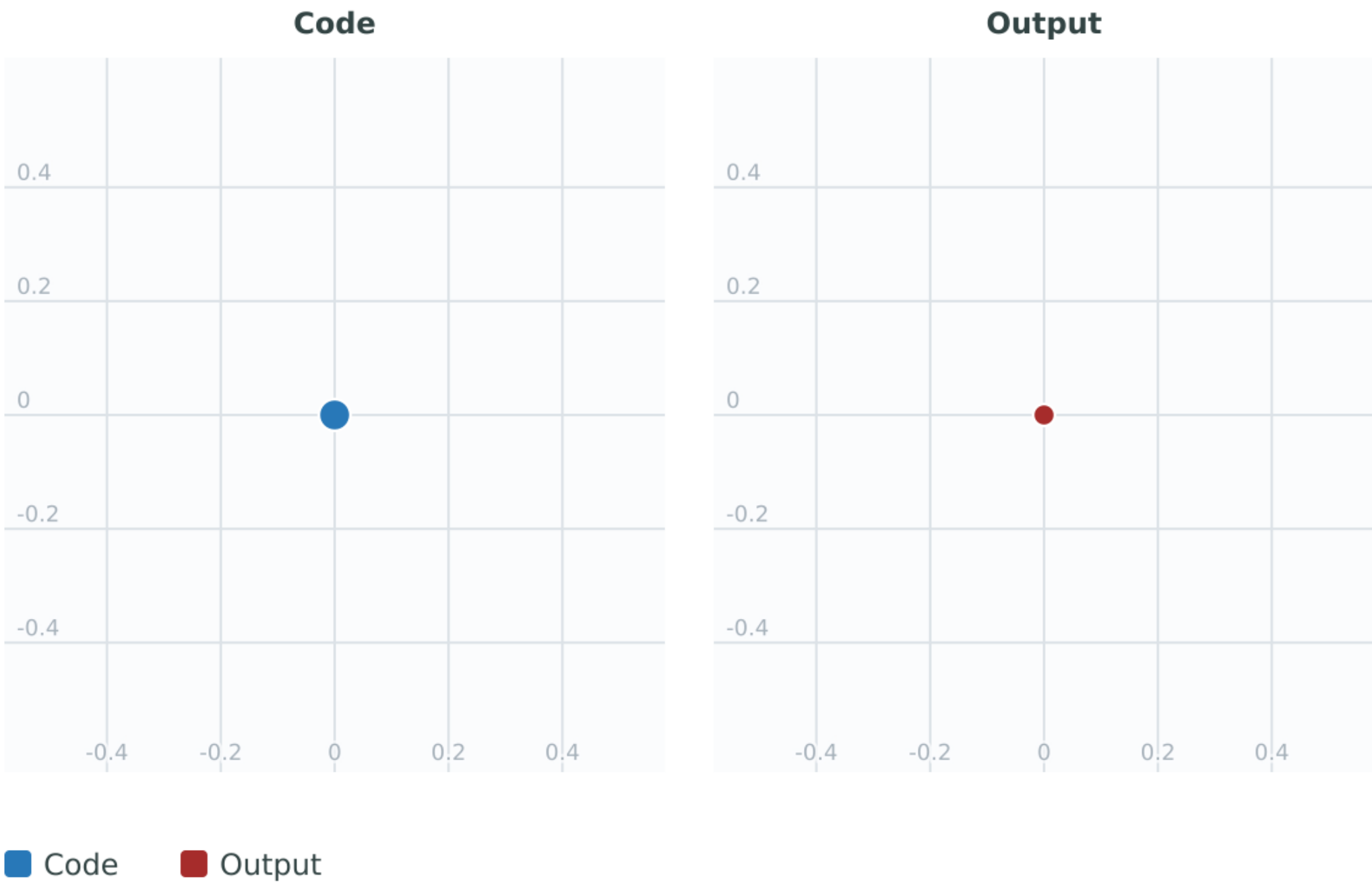
Code

```
SELECT ST_PointFromText('POINT(-71.064544 42.28787)');
```

Output

```
POINT(-71.064544 42.28787)
```

Figure



Create a point with SRID 4326.

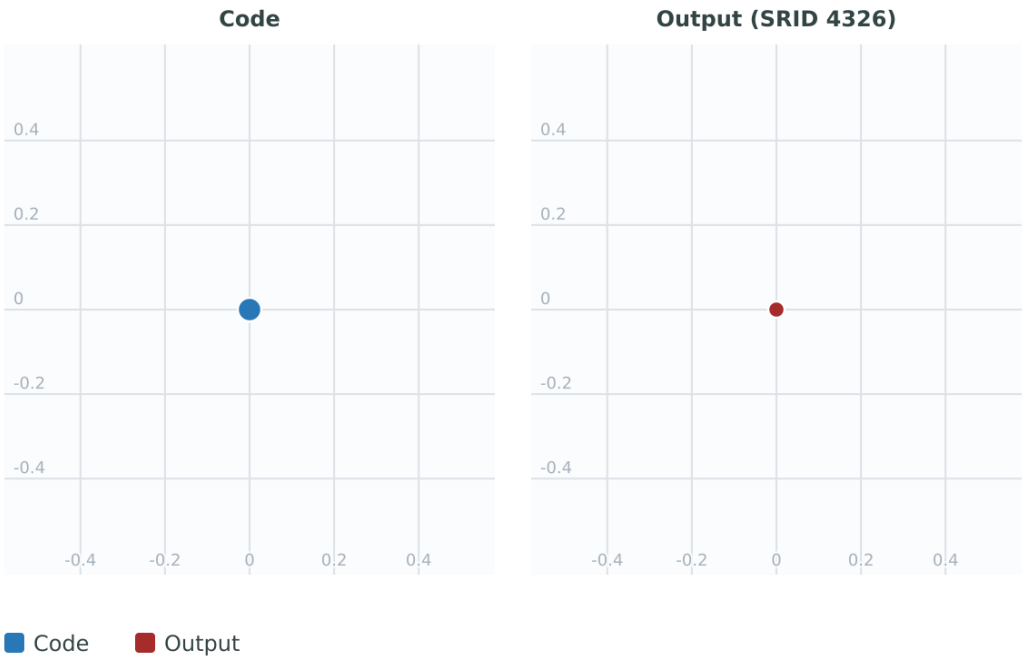
Code

```
SELECT ST_PointFromText('POINT(-71.064544 42.28787)', 4326);
```

Output

```
SRID=4326;POINT(-71.064544 42.28787)
```

Figure



□□

ST_GeomFromText, ST_MakePoint, ST_Point, ST_SRID

7.8.1.15 ST_PolygonFromText

ST_PolygonFromText — Makes a Geometry from WKT with the given SRID. If SRID is not given, it defaults to 0.

□□

geometry **ST_PolygonFromText**(text WKT);
geometry **ST_PolygonFromText**(text WKT, integer srid);

□□

Makes a Geometry from WKT with the given SRID. If SRID is not given, it defaults to 0. Returns null if WKT is not a polygon.

OGC □□ 3.2.6.2 - □□□□ SRID □□□□□□□ (conformance suite) □□□□□□□□□.



□□
□□□ WKT □□□□□□□□□□□□□□□□□, □□□□□□□□□□□□. □□□□□□□□□□□□
□□□□□□□□□ ST_GeomFromText □□□□□□.

This method implements the OGC Simple Features Implementation Specification for SQL 1.1. s3.2.6.2

This method implements the SQL/MM specification. SQL-MM 3: 8.3.6

□□

The result is returned as a native geometry. The documentation test runner compares that value directly with the readable EWKT shown below and uses the same geometry for the generated figure.

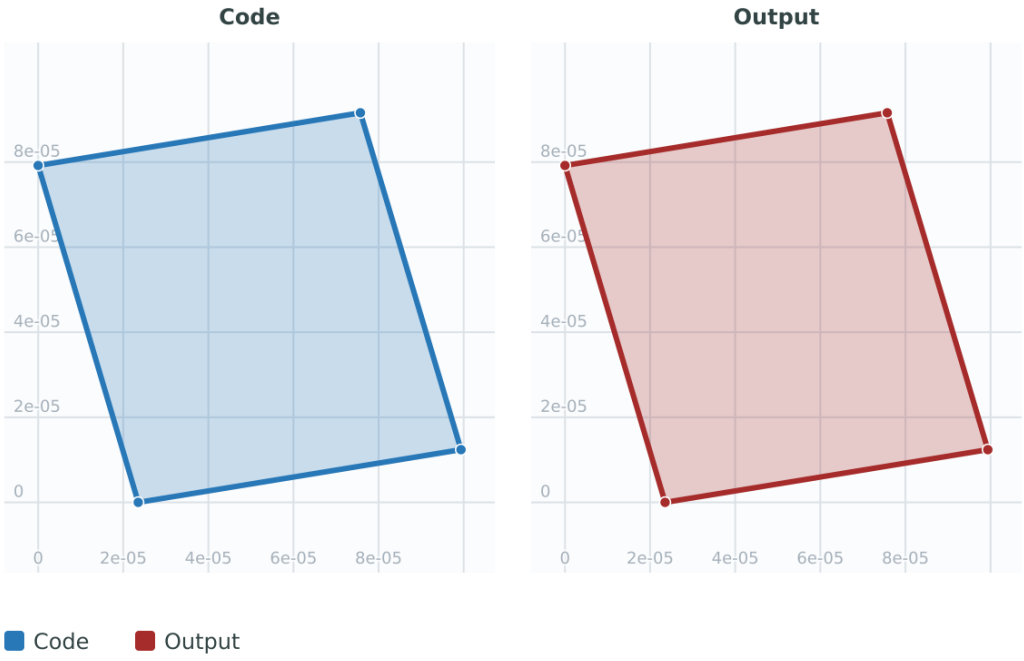
Code

```
SELECT ST_PolygonFromText('POLYGON((-71.1776585052917 42.3902909739571,-71.1776820268866 ↩
    42.3903701743239,
-71.1776063012595 42.3903825660754,-71.1775826583081 42.3903033653531,-71.1776585052917 ↩
    42.3902909739571))');
```

Output

```
POLYGON((-71.1776585052917 42.3902909739571,-71.1776820268866 42.3903701743239,-71.1776063012595 ↩
    42.3903825660754,-71.1775826583081 42.3903033653531,-71.1776585052917 42.3902909739571))
```

Figure



Code

```
SELECT ST_PolygonFromText('POINT(1 2)') IS NULL AS point_is_notpoly;
```

Output

```
t
```

□□

ST_GeomFromText

7.8.1.16 ST_WKTToSQL

ST_WKTToSQL — Returns a geometry from Well-Known Text (WKT). SQL/MM alias for ST_GeomFromText.

□□

geometry **ST_WKTToSQL**(text WKT);

□□

This function is the SQL/MM alias for ST_GeomFromText with no SRID argument.

This method implements the SQL/MM specification. SQL-MM 3: 5.1.34

□□

ST_GeomFromText

7.8.2 Well-Known Binary (WKB)

7.8.2.1 ST_GeogFromWKB

ST_GeogFromWKB — WKB □□□□□□□ EWKB(□□ WKB) □□□□□□□□□□□□□□□□□□.

□□

geography **ST_GeogFromWKB**(bytea wkb);

□□

ST_GeogFromWKB □□□□□□ WKB □□□□□ PostGIS □□ WKB □□□□□□□□□□□□□□□□□□□□ □□□□. □□□□ SQL □□□□□□□ (Geometry Factory) □□□□□□.

SRID □□□□□□□□□□, □□□□□ 4326(WGS84 □□□) □□□□.

This method supports Circular Strings and Curves.

□□

Although bytea rep contains single \, these need to be escaped when inserting into a table.

Code

```
SELECT ST_GeogFromWKB(E'\\001\\002\\000\\000\\000\\002\\000\\000\\000\\037\\205\\353Q ↩
    \\270~\\\\\\300\\323Mb\\020X\\231C@\\020X9\\264\\310~\\\\\\300)\\\\\\217\\302\\365\\230 ↩
    C@');
```

Output

```
LINESTRING(-113.98 39.198,-113.981 39.195)
```

Figure



□□

ST_GeogFromText, ST_AsBinary

7.8.2.2 ST_GeomFromEWKB

ST_GeomFromEWKB — EWKB(Extended Well-Known Binary) □□□□□□ ST_Geometry □□□□□□ □.

□□

geometry **ST_GeomFromEWKB**(bytea EWKB);

□□

Constructs a PostGIS geometry from an Extended Well-Known Binary (EWKB) representation.



□□
EWKB □□□ OGC □□□□□□□, SRID(□□□□□□□□□□) □□□□□ PostGIS □□□□□□ □.

□□□□: 2.0.0 □□□□□□□□□□ (polyhedral surface) □ TIN □□□□□□.

This function supports 3d and will not drop the z-index.

This method supports Circular Strings and Curves.

This function supports Polyhedral surfaces.

This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).

□□

This example constructs `LINESTRING(-71.160281 42.258729,-71.160837 42.259113,-71.161144 42.259` in NAD83 longitude/latitude (SRID 4269) from its EWKB representation.

PostgreSQL escape string syntax (`E'...'`) always interprets backslash escapes. Each backslash therefore has to be doubled so that the bytea input parser receives the octal escapes.

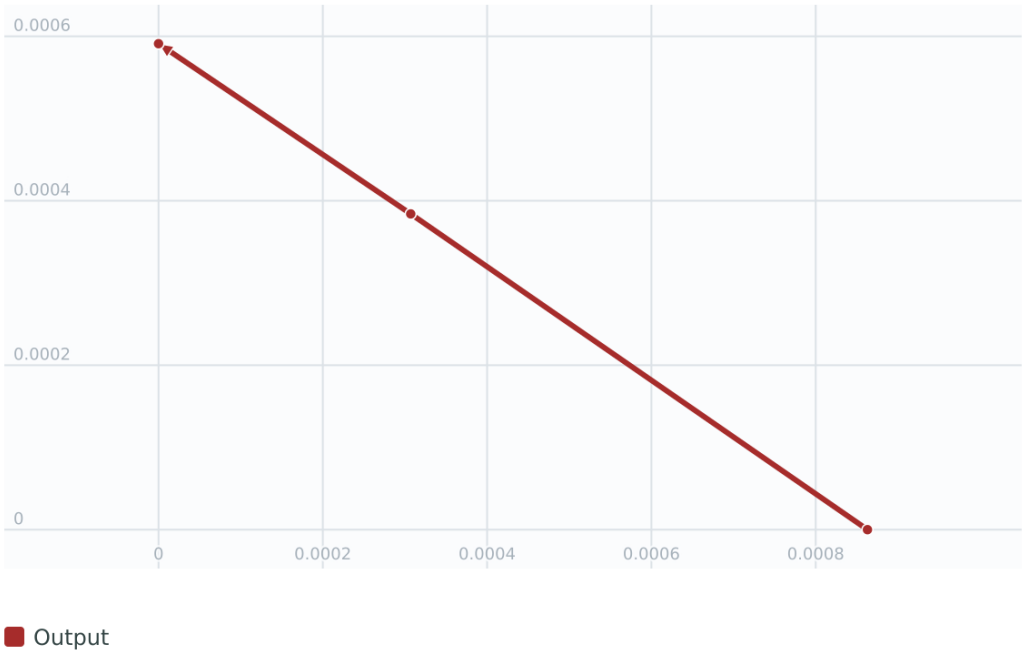
Code

```
SELECT ST_GeomFromEWKB((
    E'\\001\\002\\000\\000\\040\\255\\020\\000\\000\\003\\000\\000\\000\\344\\112\\075' ||
    E'\\013\\102\\312\\121\\300\\156\\303\\050\\010\\036\\041\\105\\100\\047\\277\\105' ||
    E'\\047\\113\\312\\121\\300\\366\\173\\142\\235\\052\\041\\105\\100\\225\\174\\354' ||
    E'\\056\\120\\312\\121\\300\\160\\231\\323\\145\\061\\041\\105\\100'
)::bytea);
```

Output

```
0102000020 ↩
    AD10000003000000E44A3D0B42CA51C06EC328081E21454027BF45274BCA51C0F67B629D2A214540957CEC2E50CA51C07099D36531214540
```

Figure



With `standard_conforming_strings` enabled (the default), ordinary string literals preserve backslashes. The same EWKB can therefore be written with a single backslash before each octal byte value.

Code

```
SELECT ST_GeomFromEWKB((
    '\001\002\000\000\040\255\020\000\000\003\000\000\000\344\112\075' ||
    '\013\102\312\121\300\156\303\050\010\036\041\105\100\047\277\105' ||
    '\047\113\312\121\300\366\173\142\235\052\041\105\100\225\174\354' ||
    '\056\120\312\121\300\160\231\323\145\061\041\105\100'
)::bytea);
```

Output

```
0102000020 ↩
    AD10000003000000E44A3D0B42CA51C06EC328081E21454027BF45274BCA51C0F67B629D2A214540957CEC2E50CA51C0
```

□□

ST_AsBinary, ST_AsEWKB, ST_GeomFromWKB

7.8.2.3 ST_GeomFromWKB

ST_GeomFromWKB — WKB(Well-Known Binary) □□□□□□□□□□ SRID □□□□□□□□□□□□□□ □□.

□□

geometry **ST_GeomFromWKB**(bytea geom);
geometry **ST_GeomFromWKB**(bytea geom, integer srid);

□□

ST_GeogFromWKB □□□□□□ WKB □□□□ SRID(□□□□□□□ ID) □□□□□□□□□□□□□□□□□□□□ □□□□. □□□□ SQL □□□□□□□ (Geometry Factory) □□□□□□. □□□□ ST_WKBToSQL □□□□ □□□□□□□□.

SRID □□□□□□□□□□, □□□□□□ 0(unkown) □□□□.

This method implements the OGC Simple Features Implementation Specification for SQL 1.1. s3.2.7.2 - □□□□ SRID □□□□□□□□ (conformance suite) □□□□□□□□□□.

This method implements the SQL/MM specification. SQL-MM 3: 5.1.41

This method supports Circular Strings and Curves.

□□

Although bytea rep contains single \, these need to be escaped when inserting into a table. Unless standard_conforming_strings is set to on.

Code

```
SELECT ST_GeomFromWKB(E'\\001\\002\\000\\000\\000\\002\\000\\000\\000\\037\\205\\353Q ↩
    \\270~\\\\\\300\\323Mb\\020X\\231C@\\020X9\\264\\310~\\\\\\300)\\\\\\217\\302\\365\\230 ↩
    C@', 4326);
```

Output

```
SRID=4326;LINESTRING(-113.98 39.198,-113.981 39.195)
```

Figure



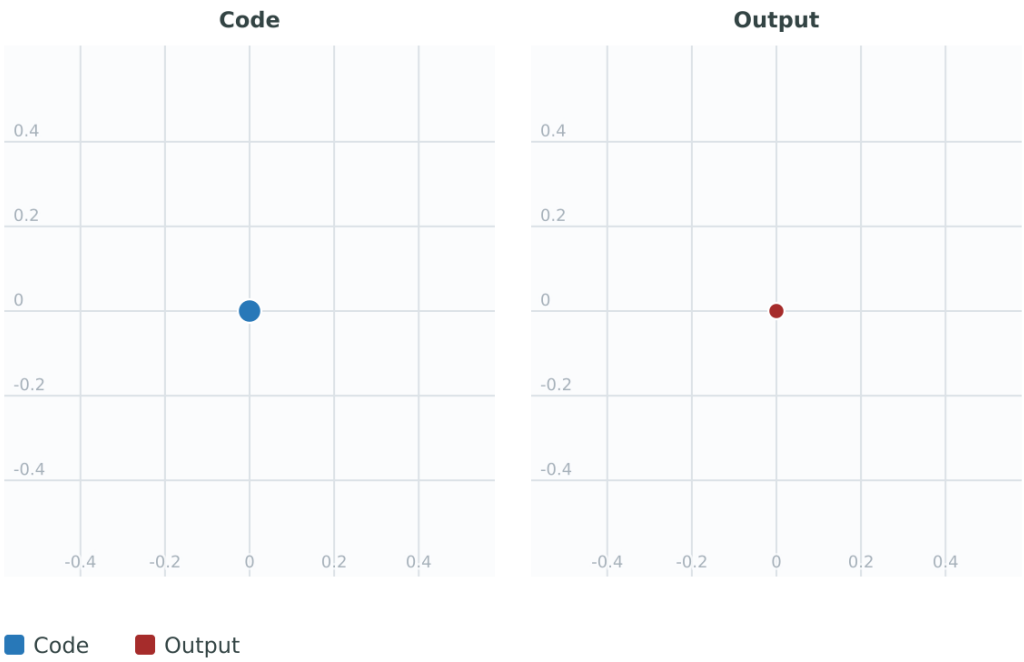
Code

```
SELECT ST_GeomFromWKB(ST_AsEWKB('POINT(2 5)'::geometry)
  );
```

Output

```
POINT(2 5)
```

Figure



The PostgreSQL `decode` function converts a hexadecimal WKB string to the bytea value accepted by `ST_GeomFromWKB`.

Code

```
SELECT ST_AsEWKT(ST_GeomFromWKB(
    decode('0101000000e5d022dbf93e2e40dbf97e6abc743540', 'hex'),
    4326
)) AS geom;
```

Output

```
SRID=4326;POINT(15.123 21.456)
```

☒☒

ST_WKBToSQL, ST_AsBinary, ST_AsEWKT, ST_GeomFromEWKB

7.8.2.4 ST_LineFromWKB

ST_LineFromWKB — Creates a `LINESTRING` from a WKB representation, optionally using a given SRID.

☒☒

geometry **ST_LineFromWKB**(bytea WKB);
geometry **ST_LineFromWKB**(bytea WKB, integer srid);

□□

The ST_LineFromWKB function takes a Well-Known Binary (WKB) representation of a geometry and an optional Spatial Reference System ID (SRID), and creates a LINESTRING geometry. It is a type-specific geometry factory in SQL.

If an SRID is not specified, it defaults to 0. NULL is returned if the input bytea does not represent a LINESTRING geometry.



□□
OGC □□ 3.2.6.2 - □□□□ SRID □□□□□□□□ (conformance suite) □□□□□□□□□.



□□
If you know all your geometries are LINESTRINGs, it is more efficient to use ST_GeomFromWKB directly. This function calls ST_GeomFromWKB and adds validation that the result is a LINESTRING.

This method implements the OGC Simple Features Implementation Specification for SQL 1.1. s3.2.6.2

This method implements the SQL/MM specification. SQL-MM 3: 7.2.9

□□

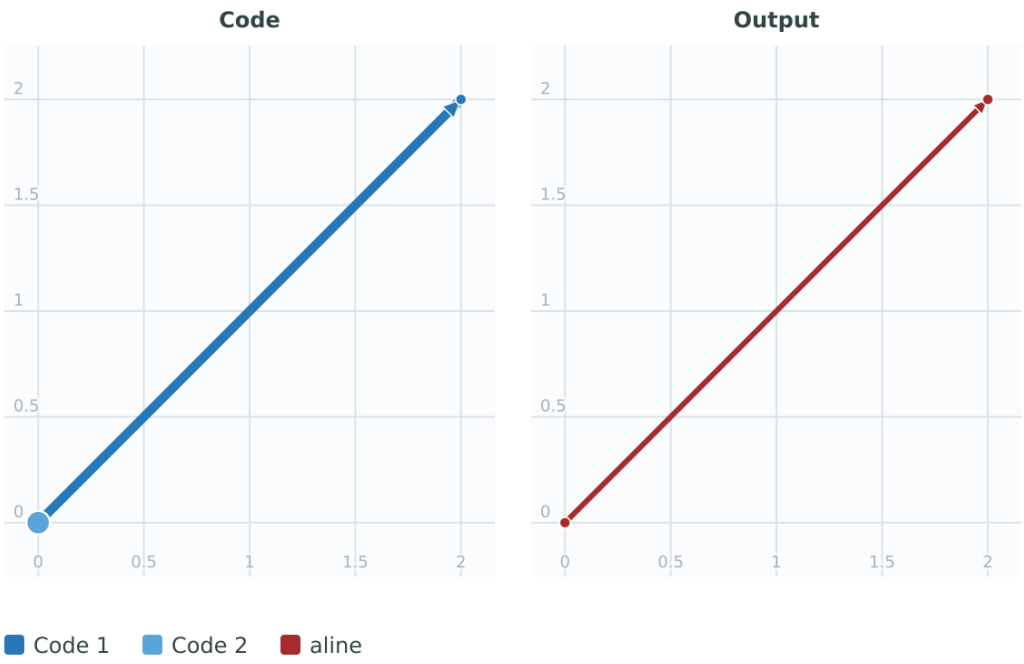
Code

```
SELECT ST_LineFromWKB(
    ST_AsBinary(ST_GeomFromText('LINESTRING(1 2,3 4)'))
) AS aline,
ST_LineFromWKB(
    ST_AsBinary(ST_GeomFromText('POINT(1 2)'))
) IS NULL AS null_return;
```

Output

```
aline                | null_return
-----------------------------------
LINESTRING(1 2,3 4)  | t
```

Figure



☒☒

ST_GeomFromWKB, ST_LinestringFromWKB

7.8.2.5 ST_LinestringFromWKB

ST_LinestringFromWKB — Creates a `LINESTRING` from a WKB representation, optionally using a given SRID.

☒☒

geometry **ST_LinestringFromWKB**(bytea WKB);
geometry **ST_LinestringFromWKB**(bytea WKB, integer srid);

☒☒

The `ST_LinestringFromWKB` function takes a Well-Known Binary (WKB) representation of a geometry and an optional Spatial Reference System ID (`SRID`), and creates a `LINESTRING` geometry. It is an alias for ST_LineFromWKB.

If an SRID is not specified, it defaults to 0. `NULL` is returned if the input `bytea` does not represent a `LINESTRING` geometry.



☒☒
OGC ☒☒ 3.2.6.2 - ☒☒☒☒ SRID ☒☒☒☒☒☒☒ (conformance suite) ☒☒☒☒☒☒☒☒☒.



☒☒

If you know all your geometries are LINESTRINGs, it is more efficient to use ST_GeomFromWKB directly. This function calls ST_GeomFromWKB and adds validation that the result is a LINESTRING.

 This method implements the OGC Simple Features Implementation Specification for SQL 1.1. s3.2.6.2

 This method implements the SQL/MM specification. SQL-MM 3: 7.2.9

☒☒

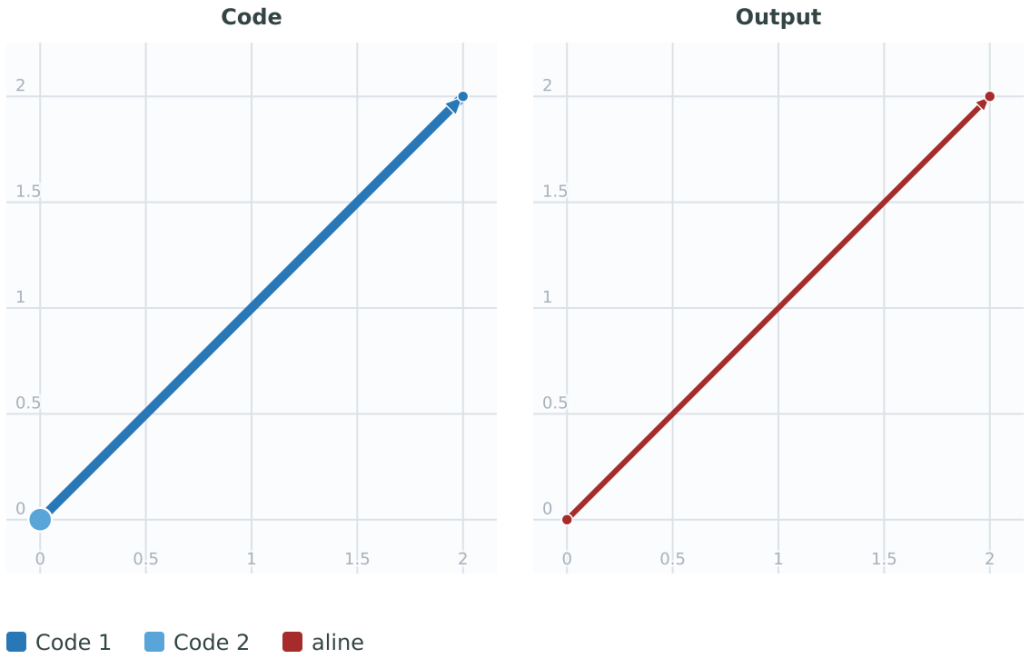
Code

```
SELECT ST_LinestringFromWKB(
    ST_AsBinary(ST_GeomFromText('LINESTRING(1 2,3 4)'))
) AS aline,
ST_LinestringFromWKB(
    ST_AsBinary(ST_GeomFromText('POINT(1 2)'))
) IS NULL AS null_return;
```

Output

```
aline                | null_return
-------------------------------------
LINESTRING(1 2,3 4)  | t
```

Figure



☒☒

ST_GeomFromWKB, ST_LineFromWKB

7.8.2.6 ST_PointFromWKB

ST_PointFromWKB — □□□ SRID □□□ WKB □□□□□□□□□□.

□□

geometry **ST_GeomFromWKB**(bytea geom);
geometry **ST_GeomFromWKB**(bytea geom, integer srid);

□□

ST_PointFromWKB □□□□□□ WKB □□□□ SRID(□□□□□□□ ID) □□□□□□□□□□□□□□□ - □□□, POINT □□ - □□□□□□. □□□□ SQL □□□□□□ (Geometry Factory) □□□□□□.

SRID □□□□□□□□□, □□□□ 0 □□□□. □□□ bytea □□□□□□□□□, NULL □□□□□□.

This method implements the OGC Simple Features Implementation Specification for SQL 1.1. s3.2.7.2

This method implements the SQL/MM specification. SQL-MM 3: 6.1.9

This function supports 3d and will not drop the z-index.

This method supports Circular Strings and Curves.

□□

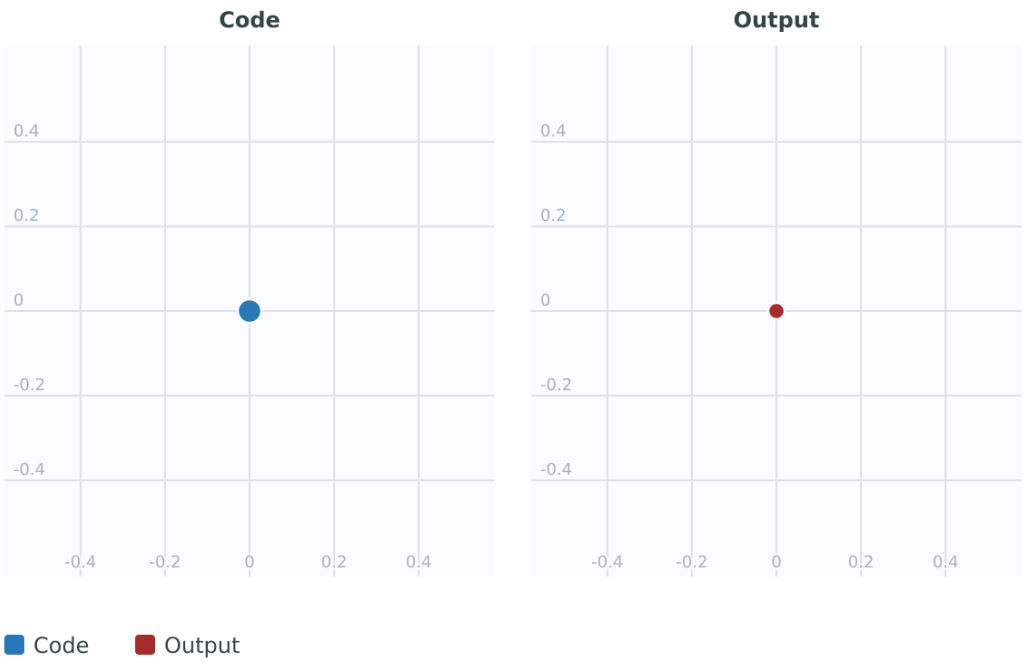
Code

```
SELECT ST_PointFromWKB(ST_AsEWKB('POINT(2 5)'::geometry));
```

Output

```
POINT(2 5)
```

Figure



Code

```
SELECT ST_PointFromWKB(ST_AsEWKB('LINESTRING(2 5,2 6)'::geometry));
```

Output

```
null
```

☒☒

ST_GeomFromWKB, ST_LineFromWKB

7.8.2.7 ST_WKBToSQL

ST_WKBToSQL — Returns a geometry from Well-Known Binary (WKB). SQL/MM alias for ST_GeomFromWKB without an SRID argument.

☒☒

geometry **ST_WKBToSQL**(bytea WKB);

☒☒

This function is the SQL/MM alias for ST_GeomFromWKB with no SRID argument.

This method implements the SQL/MM specification. SQL-MM 3: 5.1.36

☒☒

ST_GeomFromWKB

7.8.3 Other Formats

7.8.3.1 ST_Box2dFromGeoHash

ST_Box2dFromGeoHash — GeoHash □□□□□□□ BOX2D □□□□□□.

□□

box2d **ST_Box2dFromGeoHash**(text geohash, integer precision=full_precision_of_geohash);

□□

GeoHash □□□□□□□ BOX2D □□□□□□.

If no `precision` is specified ST_Box2dFromGeoHash returns a BOX2D based on full precision of the input GeoHash string.

`precision` □□□□□□, ST_Box2dFromGeoHash □ GeoHash □□□□□□□□□□□□□□□ BOX2D □□□□□□□. □□□□□□□□□□ BOX2D □□□□□□□□□□□□□□□□□□□□□□□□□□□.

2.1.0 □□□□□□□□□□□□.

□□

Code

```
SELECT ST_Box2dFromGeoHash('9qqj7nmxncgyy4d0dbxqz0');
```

Output

```
BOX(-115.172816 36.114646,-115.172816 36.114646)
```

Code

```
SELECT ST_Box2dFromGeoHash('9qqj7nmxncgyy4d0dbxqz0', 0);
```

Output

```
BOX(-180 -90,180 90)
```

Code

```
SELECT ST_Box2dFromGeoHash('9qqj7nmxncgyy4d0dbxqz0', 10);
```

Output

```
BOX(-115.17282128334 36.1146408319473,-115.172810554504 36.1146461963654)
```

□□

ST_GeoHash, ST_GeomFromGeoHash, ST_PointFromGeoHash

7.8.3.2 ST_GeomFromGeoHash

ST_GeomFromGeoHash — GeoHash □□□□□□□□□□□□□□□.

□□

geometry **ST_GeomFromGeoHash**(text geohash, integer precision=full_precision_of_geohash);

□□

GeoHash □□□□□□□□□□□□□□□□. □□□□□ GeoHash □□□□□□□□□□□□□□□□.

`precision` □□□□□□□□□, ST_GeomFromGeoHash □□□ GeoHash □□□□□□□□□□□□□□□□ □□□□□□□□□.

`precision` □□□□□□, ST_GeomFromGeoHash □ GeoHash □□□□□□□□□□□□□□□□□□□□□□□□ □□□□□.

2.1.0 □□□□□□□□□□□□.

□□

The cells are returned together so their relative sizes remain visible in a shared coordinate frame.

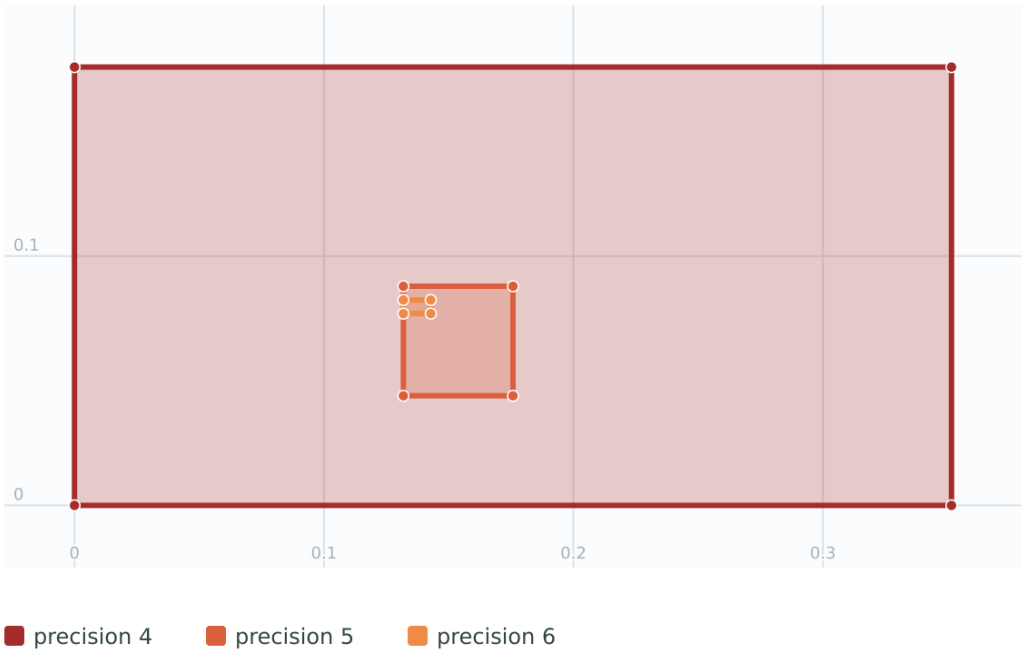
Code

```
SELECT ST_GeomFromGeoHash('9qqj7nmxncgyy4d0dbxqz0', 4) AS "precision 4", ST_GeomFromGeoHash ↩
    ('9qqj7nmxncgyy4d0dbxqz0', 5) AS "precision 5", ST_GeomFromGeoHash('9 ↩
    qqj7nmxncgyy4d0dbxqz0', 6) AS "precision 6";
```

Output

```
-[ RECORD 1 ]----------
precision 4 | POLYGON((-115.3125 36.03515625,-115.3125 36.2109375,-114.9609375 36.2109375,-114.9609375  ↩
     36.03515625,-115.3125 36.03515625))
precision 5 | POLYGON((-115.1806640625 36.0791015625,-115.1806640625 36.123046875,-115.13671875  ↩
     36.123046875,-115.13671875 36.0791015625,-115.1806640625 36.0791015625))
precision 6 | POLYGON((-115.1806640625 36.112060546875,-115.1806640625 36.1175537109375,-115.169677734375  ↩
     36.1175537109375,-115.169677734375 36.112060546875,-115.1806640625 36.112060546875))
```

Figure



☒☒

ST_GeoHash,ST_Box2dFromGeoHash, ST_PointFromGeoHash

7.8.3.3 ST_GeomFromGML

ST_GeomFromGML — ☒☒☒ GML ☒☒☒☒☒☒☒☒ PostGIS ☒☒☒☒☒☒☒☒☒☒☒.

☒☒

geometry **ST_GeomFromGML**(text geomgml);
geometry **ST_GeomFromGML**(text geomgml, integer srid);

☒☒

OGC GML ☒☒☒☒☒☒☒ PostGIS ST_Geometry ☒☒☒☒☒☒☒☒.

ST_GeomFromGML ☒ GML ☒☒☒☒ (geometry fragment) ☒☒☒☒☒☒☒☒☒☒. ☒☒☒ GML ☒☒☒☒ ☒☒☒☒☒☒☒☒☒☒☒☒☒☒.

☒☒☒☒ OGC GML ☒☒☒☒☒☒☒☒☒☒:

- GML 3.2.1 ☒☒☒☒☒☒
- GML 3.1.1 ☒☒☒☒☒☒☒☒ SF-2 (GML 3.1.0 ☒ 3.0.0 ☒☒☒☒)
- GML 2.1.2

OGC GML ☒☒: http://www.opengeospatial.org/standards/gml

1.5 ☒☒☒☒☒☒☒☒☒☒☒☒. LibXML2 1.6 ☒☒☒☒☒☒☒☒☒☒.

☒☒☒☒: 2.0.0 ☒☒☒☒☒☒☒☒☒ (polyhedral surface) ☒ TIN ☒☒☒☒☒☒.

☒☒☒☒: 2.0.0 ☒☒☒☒☒☒☒☒☒☒ SRID ☒☒☒☒☒☒☒☒☒☒☒.

Enhanced: 3.7.0 support for GML ArcString, curved Ring, and CurvePolygon geometries was introduced.

This function supports 3d and will not drop the z-index.

This function supports Polyhedral surfaces.

This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).

GML ☒☒☒☒☒☒ (☒☒☒☒☒☒☒☒☒☒☒ (MultiGeometry) ☒☒☒ 2D ☒ 3D ☒☒☒) ☒☒☒☒☒☒. PostGIS ☒☒☒☒☒☒☒☒☒☒☒, ☒☒☒ Z ☒☒☒☒☒☒☒☒ ST_GeomFromGML ☒☒☒☒☒☒ 2D ☒☒☒☒☒☒.

GML ☒☒☒☒☒☒☒☒☒☒☒☒☒ SRS ☒☒☒☒☒☒☒. PostGIS ☒☒☒☒☒☒☒☒☒☒☒, ☒☒☒ ST_GeomFromGML ☒☒☒☒☒☒☒☒☒ SRS ☒☒☒☒☒☒☒☒☒☒☒. GML ☒☒☒☒☒☒☒☒☒ srsName ☒☒☒☒☒☒☒, ☒☒☒ ☒☒☒☒☒.

ST_GeomFromGML ☒☒☒☒ GML ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒ ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒ GML ☒☒☒ XLink ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.

☒☒

A single geometry with srsName.

Code

```
        SELECT ST_AsEWKT(ST_GeomFromGML($$
            <gml:LineString xmlns:gml="http://www.opengis.net/gml"
                    srsName="EPSG:4269">
                <gml:coordinates>
            -71.16028,42.258729 -71.160837,42.259112 -71.161143,42.25932
                </gml:coordinates>
            </gml:LineString>
        $$));
```

Output

```
SRID=4269;LINESTRING(-71.16028 42.258729,-71.160837 42.259112,-71.161143 42.25932)
```

Figure



XLink usage.

Code

```
        SELECT ST_AsEWKT(ST_GeomFromGML($$
            <gml:LineString xmlns:gml="http://www.opengis.net/gml"
                    xmlns:xlink="http://www.w3.org/1999/xlink"
                    srsName="EPSG:4269">
        <gml:pointProperty>
            <gml:Point gml:id="p1"
><gml:pos
>-71.16028 42.258729</gml:pos
></gml:Point>
        </gml:pointProperty>
        <gml:pos
>-71.160837 42.259112</gml:pos>
        <gml:pos
>-71.161143 42.258729</gml:pos>
                <gml:pointProperty>
```

```
                    <gml:Point xlink:type="simple" xlink:href="#p1"/>
                </gml:pointProperty>
            </gml:LineString>
        $$));
```

Output

```
SRID=4269;LINESTRING(-71.16028 42.258729,-71.160837 42.259112,-71.161143 42.258729,-71.16028 42.258729)
```

Figure



Polyhedral Surface.

Code

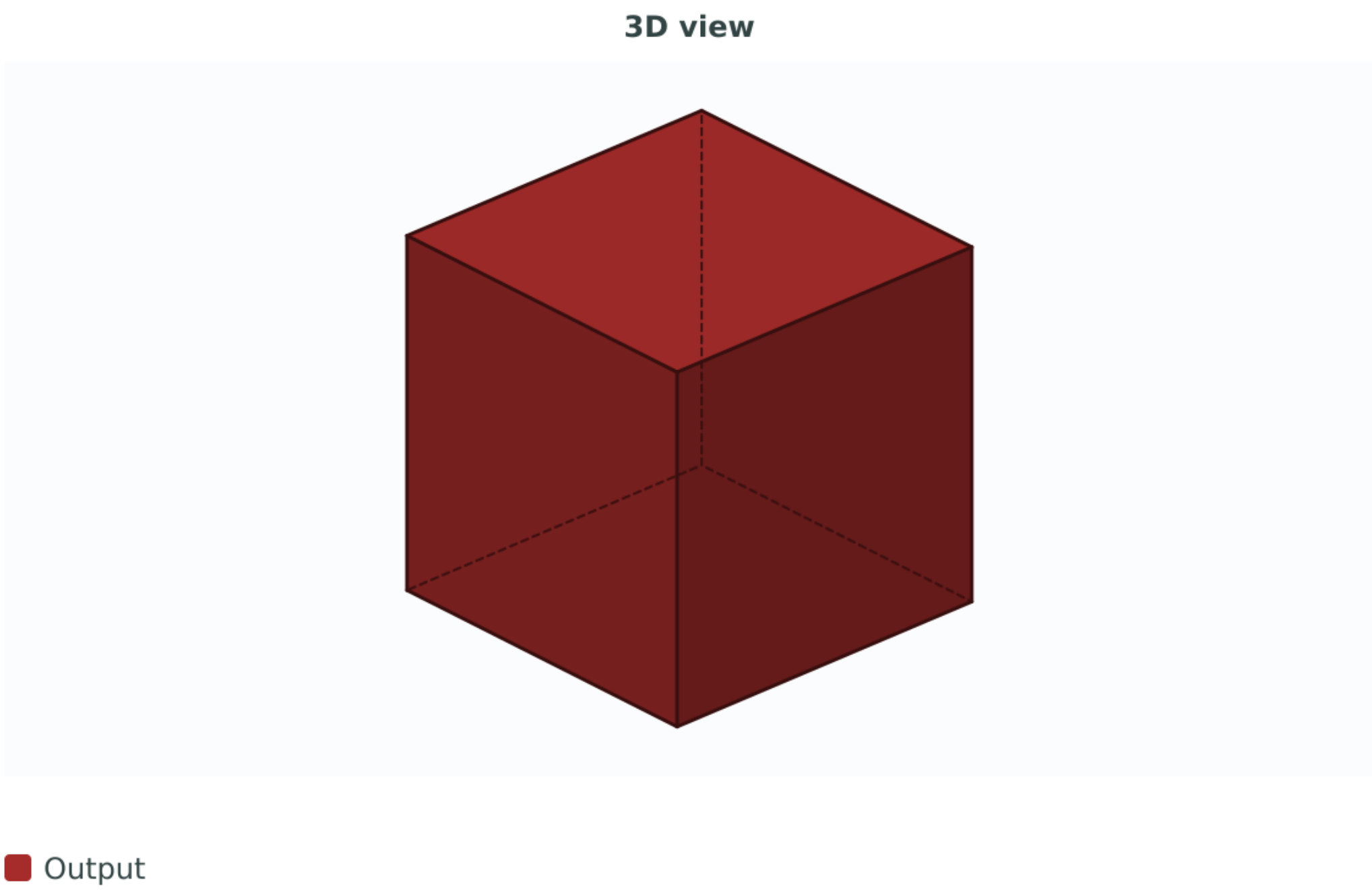
```
SELECT ST_AsEWKT(ST_GeomFromGML('
<gml:PolyhedralSurface xmlns:gml="http://www.opengis.net/gml">
<gml:polygonPatches>
  <gml:PolygonPatch>
    <gml:exterior>
      <gml:LinearRing
><gml:posList srsDimension="3"
>0 0 0 0 0 1 0 1 1 0 1 0 0 0 0</gml:posList
></gml:LinearRing>
    </gml:exterior>
  </gml:PolygonPatch>
  <gml:PolygonPatch>
    <gml:exterior>
    <gml:LinearRing
><gml:posList srsDimension="3"
>0 0 0 0 1 0 1 1 0 1 0 0 0 0 0</gml:posList
></gml:LinearRing>
    </gml:exterior>
  </gml:PolygonPatch>
  <gml:PolygonPatch>
    <gml:exterior>
```

```
    <gml:LinearRing
><gml:posList srsDimension="3"
>0 0 0 1 0 0 1 0 1 0 0 1 0 0 0</gml:posList
></gml:LinearRing>
    </gml:exterior>
  </gml:PolygonPatch>
  <gml:PolygonPatch>
    <gml:exterior>
    <gml:LinearRing
><gml:posList srsDimension="3"
>1 1 0 1 1 1 1 0 1 1 0 0 1 1 0</gml:posList
></gml:LinearRing>
    </gml:exterior>
  </gml:PolygonPatch>
  <gml:PolygonPatch>
    <gml:exterior>
    <gml:LinearRing
><gml:posList srsDimension="3"
>0 1 0 0 1 1 1 1 1 1 0 0 1 0</gml:posList
></gml:LinearRing>
    </gml:exterior>
  </gml:PolygonPatch>
  <gml:PolygonPatch>
    <gml:exterior>
    <gml:LinearRing
><gml:posList srsDimension="3"
>0 0 1 1 0 1 1 1 1 0 1 1 0 0 1</gml:posList
></gml:LinearRing>
    </gml:exterior>
  </gml:PolygonPatch>
</gml:polygonPatches>
</gml:PolyhedralSurface
>'));
```

Output

```
POLYHEDRALSURFACE(((0 0 0,0 0 1,0 1 1,0 1 0,0 0 0)),
((0 0 0,0 1 0,1 1 0,1 0 0,0 0 0)),
((0 0 0,1 0 0,1 0 1,0 0 1,0 0 0)),
((1 1 0,1 1 1,1 0 1,1 0 0,1 1 0)),
((0 1 0,0 1 1,1 1 1,1 1 0,0 1 0)),
((0 0 1,1 0 1,1 1 1,0 1 1,0 0 1)))
```

Figure



⊠⊠

2.2.3⊠, ST_AsGML, ST_GMLToSQL

7.8.3.4 ST_GeomFromGeoJSON

ST_GeomFromGeoJSON — GeoJSON ⊠⊠⊠⊠⊠⊠⊠⊠ PostGIS ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

geometry **ST_GeomFromGeoJSON**(text geomjson);
geometry **ST_GeomFromGeoJSON**(json geomjson);
geometry **ST_GeomFromGeoJSON**(jsonb geomjson);

⊠⊠

GeoJSON ⊠⊠⊠⊠⊠⊠⊠ PostGIS ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

ST_GeomFromGML ⊠ JSON ⊠⊠⊠⊠ (geometry fragment) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠ JSON ⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

Enhanced: 3.0.0 parsed geometry defaults to SRID=4326 if not specified otherwise.

Enhanced: 2.5.0 can now accept json and jsonb as inputs.

2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. JSON-C 0.9 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

> **⊠⊠**
> JSON-C ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. JSON-C ⊠⊠⊠⊠⊠⊠⊠, "--with-jsondir=/path/to/json-c" ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠ 2.2.3⊠⊠⊠⊠⊠⊠⊠⊠.

This function supports 3d and will not drop the z-index.

□□

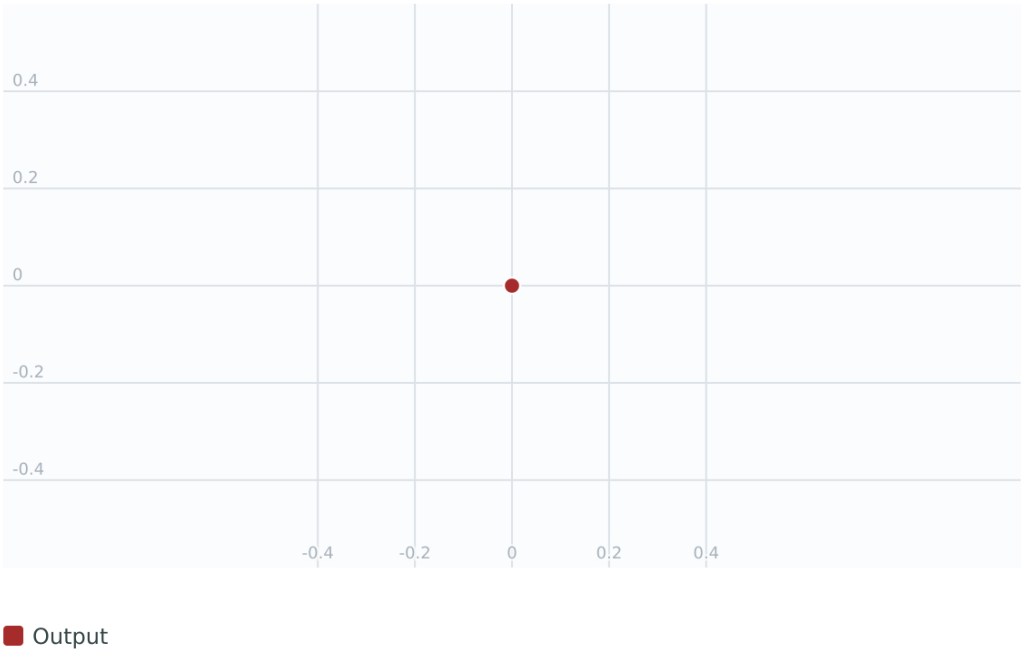
Code

```
SELECT ST_GeomFromGeoJSON('{"type":"Point","coordinates":[-48.23456,20.12345]}') AS wkt;
```

Output

```
POINT(-48.23456 20.12345)
```

Figure



This example creates a 3D LineString.

Code

```
SELECT ST_GeomFromGeoJSON('{"type":"LineString","coordinates":[[1,2,3],[4,5,6],[7,8,9]]}')
    AS wkt;
```

Output

```
LINESTRING Z (1 2 3,4 5 6,7 8 9)
```

Figure



□□

ST_AsText, ST_AsGeoJSON, 2.2.3□

7.8.3.5 ST_GeomFromKML

ST_GeomFromKML — □□□ KML □□□□□□□□ PostGIS □□□□□□□□□□.

□□

geometry **ST_GeomFromKML**(text geomkml);

□□

Constructs a PostGIS geometry from an OGC KML representation.

ST_GeomFromKML □ KML □□□□ (geometry fragment) □□□□□□□□□□. □□□ KML □□□□□ □□□□□□□□□□□□□□.

□□□□ OGC KML □□□□□□□□□□:

- KML 2.2.0 □□□□□□

OGC KML □□: http://www.opengeospatial.org/standards/kml

Availability: 1.5, requires libxml2 2.6+

This function supports 3d and will not drop the z-index.

□□
ST_GeomFromKML does not support SQL/MM curve geometries.

□□

A single LineString geometry.

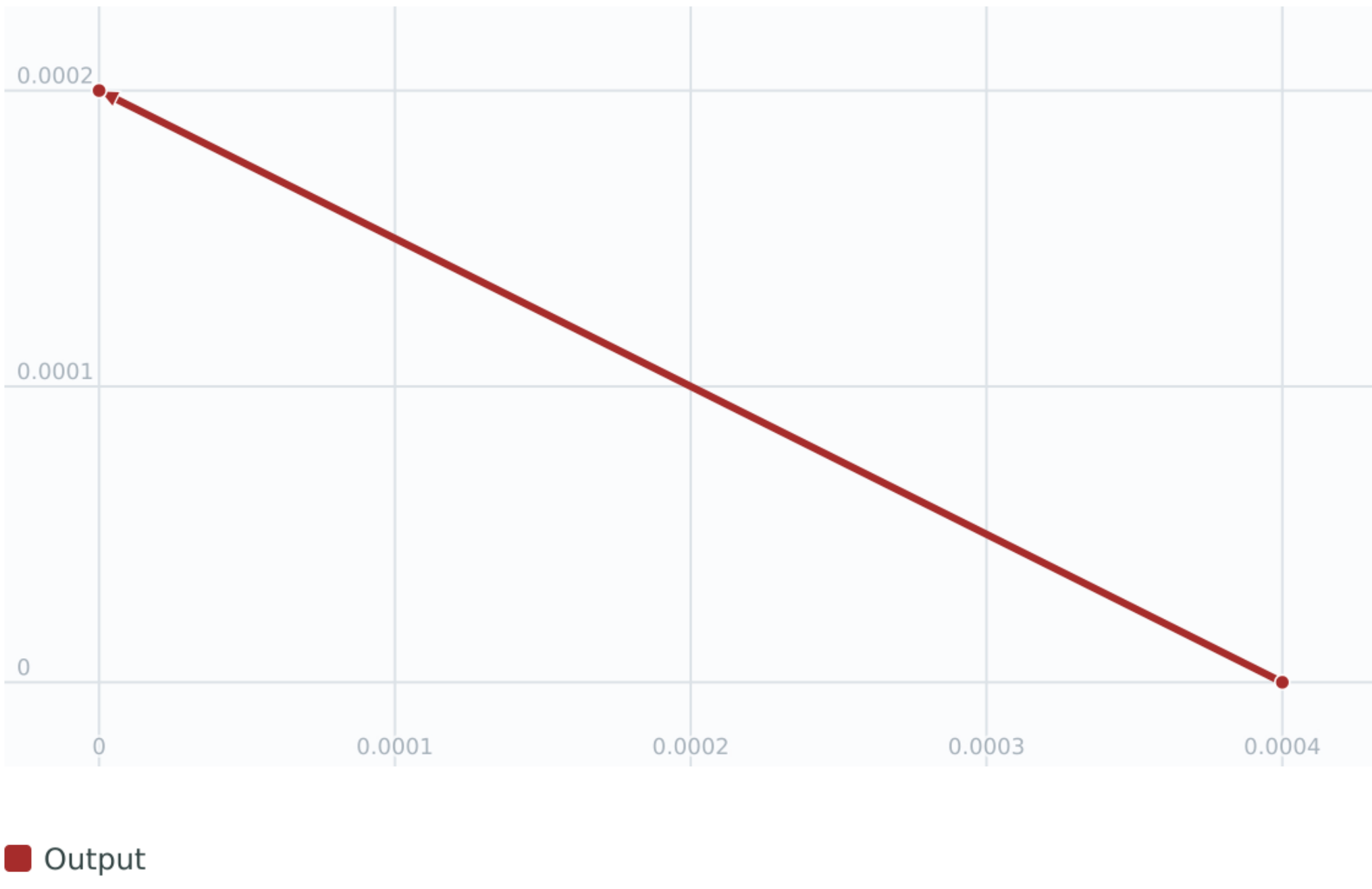
Code

```
        SELECT ST_AsEWKT(ST_GeomFromKML($$
            <LineString>
                <coordinates
>-71.1663, 42.2614
                    -71.1667, 42.2616</coordinates>
            </LineString>
        $$));
```

Output

```
SRID=4326;LINESTRING(-71.1663 42.2614,-71.1667 42.2616)
```

Figure



□□

2.2.3□, ST_AsKML

7.8.3.6 ST_GeomFromTWKB

ST_GeomFromTWKB — TWKB(”Tiny Well-Known Binary”) □□□□□□□□□□□□□□□□□□□□□□□□□□□.

□□

geometry **ST_GeomFromTWKB**(bytea twkb);

□□

ST_GeomFromTWKB □□□ TWKB("Tiny Well-Known Binary") □□□□□ (WKB) □□□□□□□□□□□□□□□□□□□□□□□□□□□□.

□□

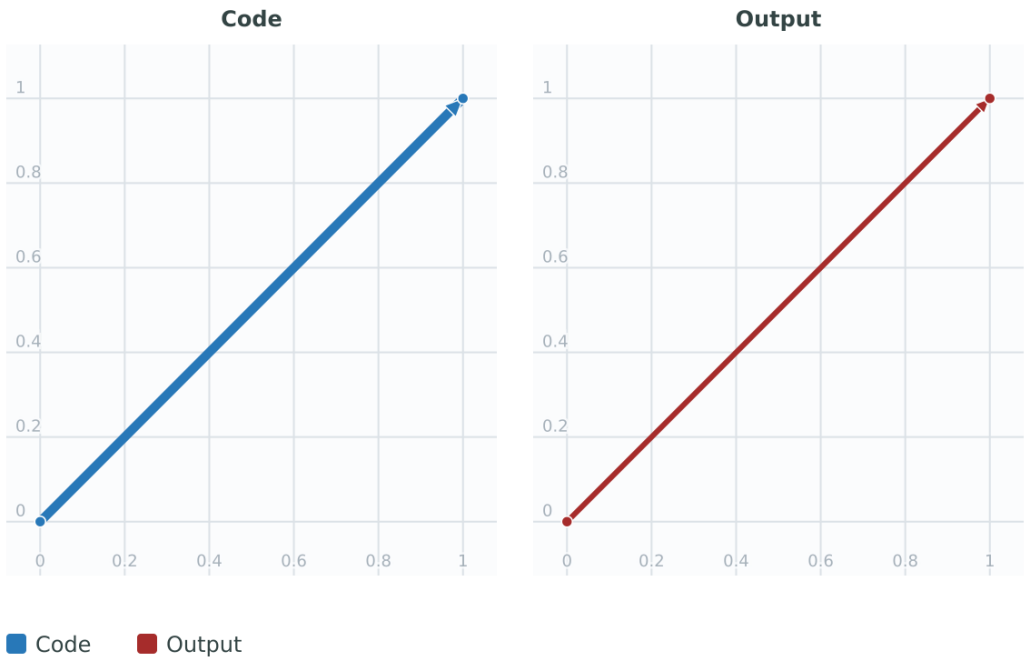
Code

```
SELECT ST_GeomFromTWKB(ST_AsTWKB('LINESTRING(126 34,127 35)'::geometry));
```

Output

```
LINESTRING(126 34, 127 35)
```

Figure



Code

```
SELECT ST_GeomFromTWKB(E'\\x620002f7f40dbce4040105');
```

Output

```
LINESTRING(-113.98 39.198,-113.981 39.195)
```

Figure



□□

ST_AsTWKB

7.8.3.7 ST_GMLToSQL

ST_GMLToSQL — Returns a geometry from GML. Alias for ST_GeomFromGML.

□□

geometry **ST_GMLToSQL**(text geomgml);
geometry **ST_GMLToSQL**(text geomgml, integer srid);

□□

This function is an alias for ST_GeomFromGML and provides the same signatures.

This method implements the SQL/MM specification. SQL-MM 3: 5.1.50.

1.5 □□□□□□□□□□□□□. LibXML2 1.6 □□□□□□□□□□□.

□□□□: 2.0.0 □□□□□□□□□□ (polyhedral surface) □ TIN □□□□□□.

□□□□: 2.0.0 □□□□□□□□□□□ SRID □□□□□□□□□□□.

Enhanced: 3.7.0 support for GML ArcString, curved Ring, and CurvePolygon geometries was introduced.

□□

2.2.3□, ST_GeomFromGML, ST_AsGML

7.8.3.8 ST_LineFromEncodedPolyline

ST_LineFromEncodedPolyline — □□□□□□□□ (polyline) □□□□□□□□□□□□□□□□.

□□

geometry **ST_LineFromEncodedPolyline**(text polyline, integer precision=5);

□□

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□.

Optional `precision` specifies how many decimal places will be preserved in Encoded Polyline. Value should be the same on encoding and decoding, or coordinates will be incorrect.

□□: http://developers.google.com/maps/documentation/utilities/polylinealgorithm

2.2.0 □□□□□□□□□□□□.

□□

This example creates a LineString from a polyline.

Code

```
SELECT ST_LineFromEncodedPolyline('_p~iF~ps|U_ulLnnqC_mqNvxq`@');
```

Output

```
SRID=4326;LINESTRING(-120.2 38.5,-120.95 40.7,-126.453 43.252)
```

Figure



This example selects a different precision than the one used for polyline encoding.

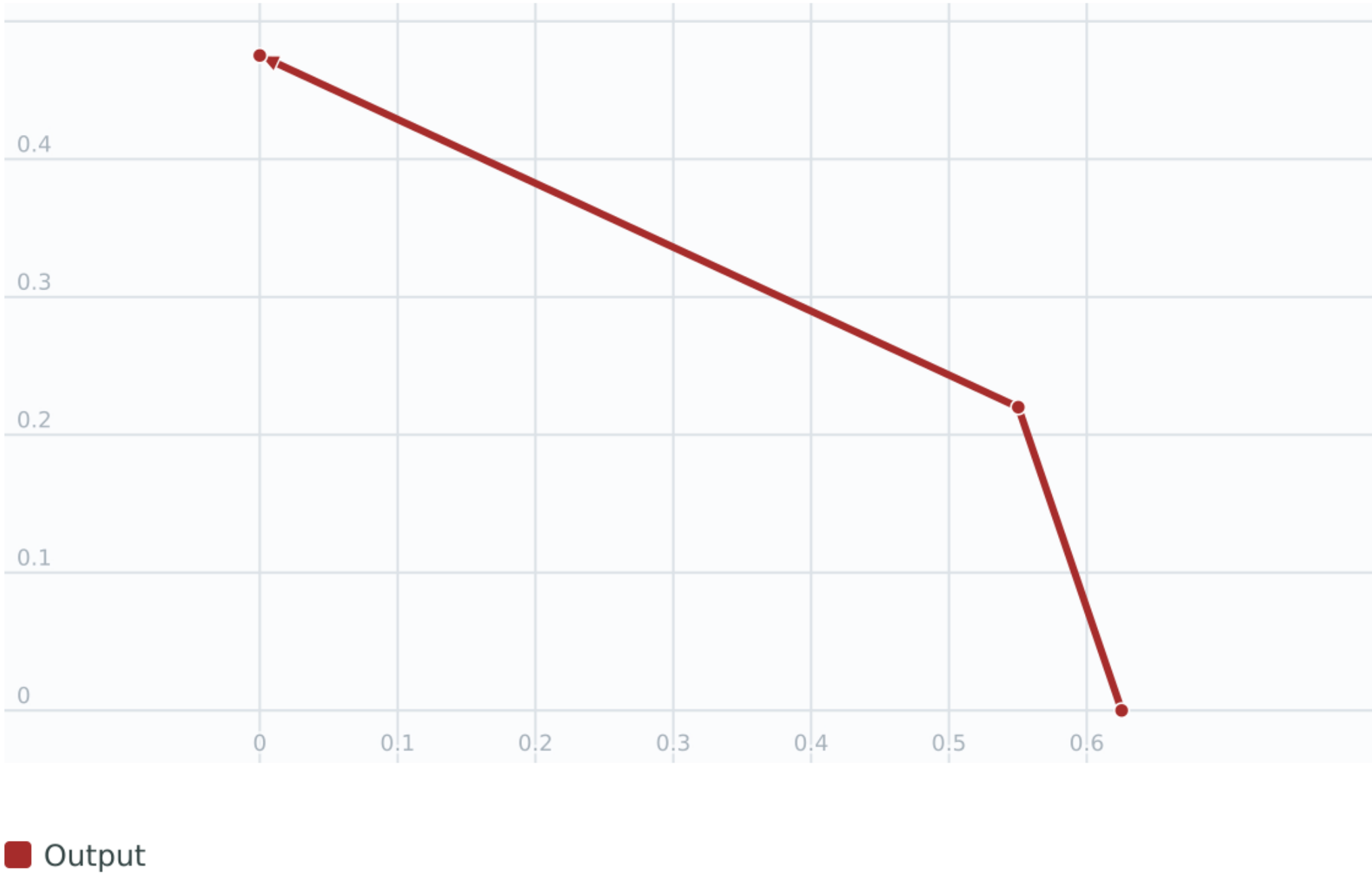
Code

```
SELECT ST_LineFromEncodedPolyline('_p~iF~ps|U_ulLnnqC_mqNvxq`@', 6);
```

Output

```
SRID=4326;LINESTRING(-12.02 3.85,-12.095 4.07,-12.6453 4.3252)
```

Figure



⊠⊠

ST_AsEncodedPolyline

7.8.3.9 ST_PointFromGeoHash

ST_PointFromGeoHash — GeoHash ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

point **ST_PointFromGeoHash**(text geohash, integer precision=full_precision_of_geohash);

⊠⊠

GeoHash ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠ GeoHash ⊠⊠⊠⊠⊠⊠⊠.

`precision` ⊠⊠⊠⊠⊠⊠⊠⊠⊠, ST_PointFromGeoHash ⊠⊠⊠ GeoHash ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠.

`precision` ⊠⊠⊠⊠⊠⊠, ST_PointFromGeoHash ⊠ GeoHash ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠.

2.1.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

□□

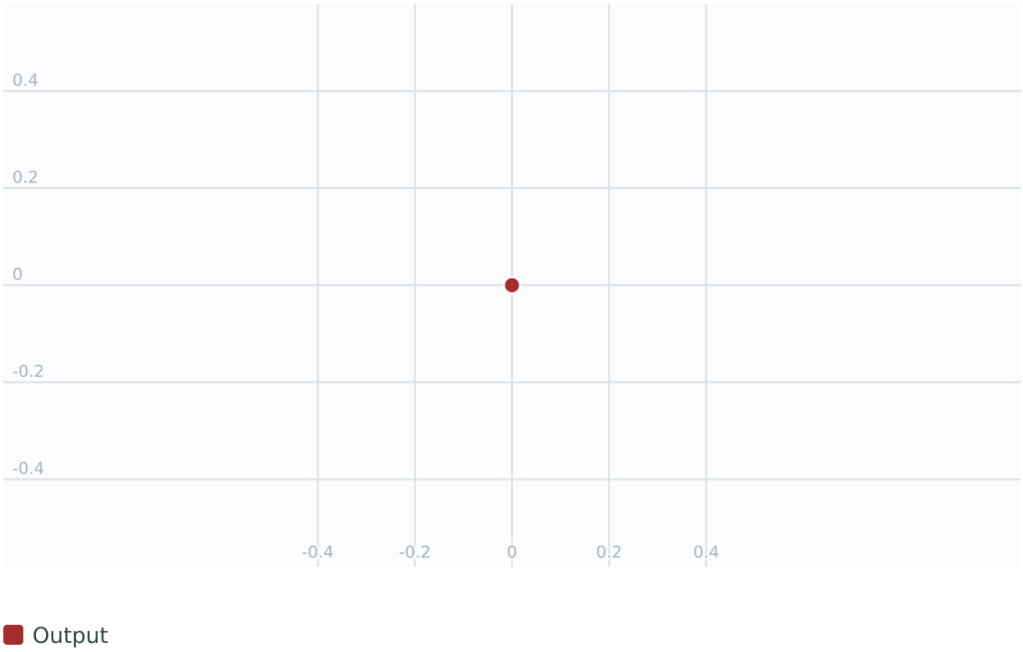
Code

```
SELECT ST_PointFromGeoHash('9qqj7nmxncgyy4d0dbxqz0');
```

Output

```
POINT(-115.172816 36.114646)
```

Figure



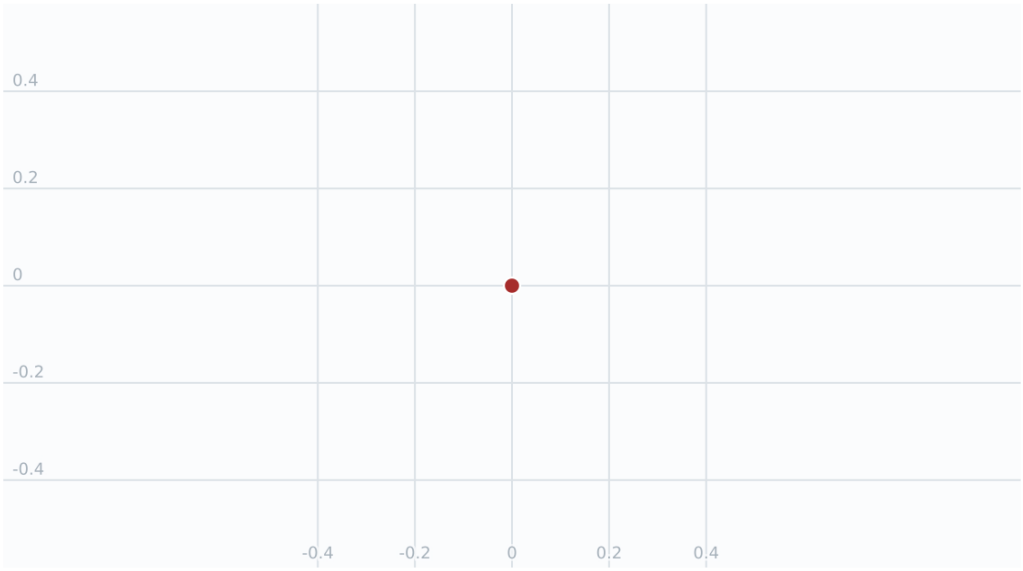
Code

```
SELECT ST_PointFromGeoHash('9qqj7nmxncgyy4d0dbxqz0', 4);
```

Output

```
POINT(-115.13671875 36.123046875)
```

Figure



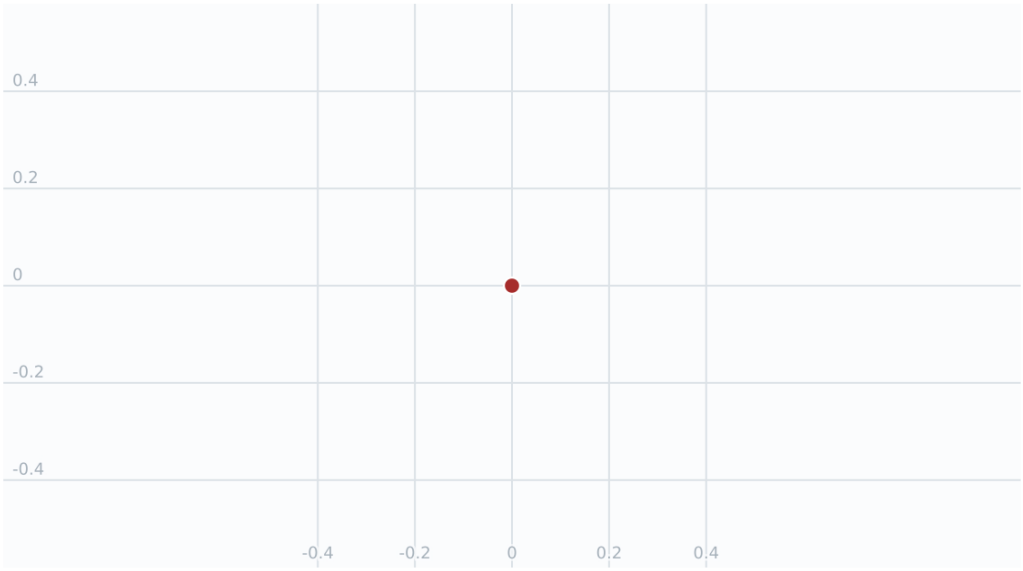
Code

```
SELECT ST_PointFromGeoHash('9qqj7nmxncgyy4d0dbxqz0', 10);
```

Output

```
POINT(-115.172815918922 36.1146435141563)
```

Figure



□□

ST_GeoHash, ST_Box2dFromGeoHash, ST_GeomFromGeoHash

7.8.3.10 ST_FromFlatGeobufToTable

ST_FromFlatGeobufToTable — Creates a table based on the structure of FlatGeobuf data.

□□

void **ST_FromFlatGeobufToTable**(text schemaname, text tablename, bytea FlatGeobuf input data);

□□

Creates a table based on the structure of FlatGeobuf data. (http://flatgeobuf.org).
`schema` Schema name.
`table` Table name.
`data` Input FlatGeobuf data.
Availability: 3.2.0

7.8.3.11 ST_FromFlatGeobuf

ST_FromFlatGeobuf — Reads FlatGeobuf data.

□□

setof anyelement **ST_FromFlatGeobuf**(anyelement Table reference, bytea FlatGeobuf input data);

□□

Reads FlatGeobuf data (http://flatgeobuf.org). NOTE: PostgreSQL bytea cannot exceed 1GB.
`tabletype` reference to a table type.
`data` input FlatGeobuf data.
Availability: 3.2.0

7.9 Geometry Output

7.9.1 Well-Known Text (WKT)

7.9.1.1 ST_AsEWKT

ST_AsEWKT — □□□ WKT(Well-Known Text) □□□□ SRID □□□□□□□□□□□□□□.

□□

text **ST_AsEWKT**(geometry g1);
text **ST_AsEWKT**(geometry g1, integer maxdecimaldigits=15);
text **ST_AsEWKT**(geography g1);
text **ST_AsEWKT**(geography g1, integer maxdecimaldigits=15);

□□

Returns the Well-Known Text representation of the geometry prefixed with the SRID. The optional *maxdecimaldigits* argument may be used to reduce the maximum number of decimal digits after floating point used in output (defaults to 15).

To perform the inverse conversion of EWKT representation to PostGIS geometry use ST_GeomFromEWKT.

□□
Using the *maxdecimaldigits* parameter can cause output geometry to become invalid. To avoid this use ST_ReducePrecision with a suitable gridsize first.

□□
The WKT spec does not include the SRID. To get the OGC WKT format use ST_AsText.

□□
WKT format does not maintain precision so to prevent floating truncation, use ST_AsBinary or ST_AsEWKB format for transport.

Enhanced: 3.1.0 support for optional precision parameter.

□□□□: 2.0.0 □□□□□□□□, □□□□□□, □□□□ TIN □□□□□□□□□□□□.

This function supports 3d and will not drop the z-index.

This method supports Circular Strings and Curves.

This function supports Polyhedral surfaces.

This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).

□□

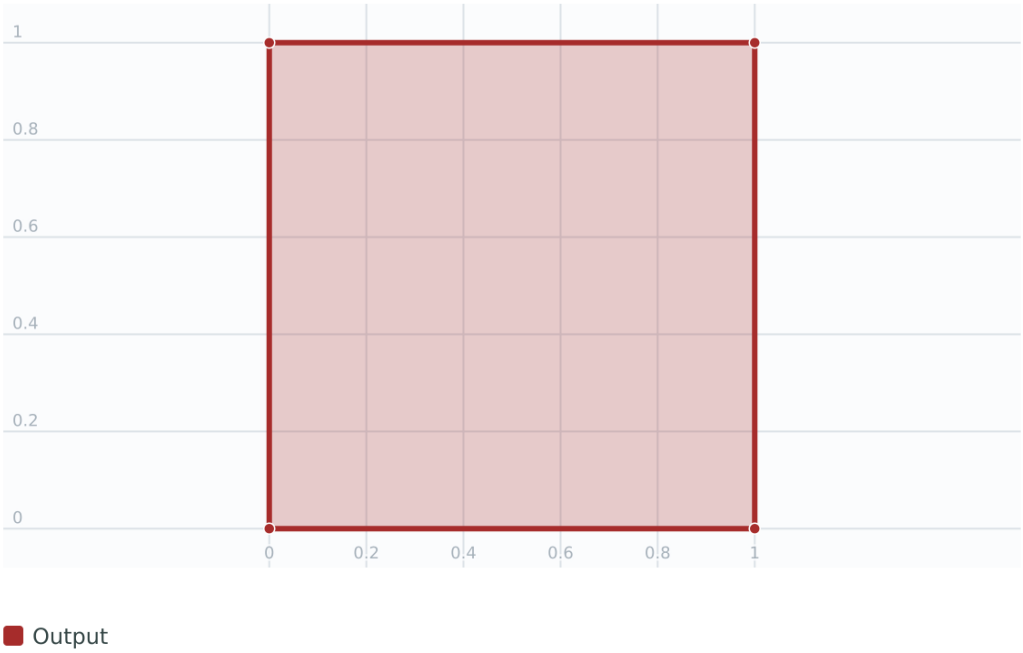
Code

```
SELECT ST_AsEWKT('0103000020 ↩
    e610000001000000050000000000000000000000000000000000000000000000000000000000000000000000000000000000f03f00000000
    '::geometry);
```

Output

```
SRID=4326;POLYGON((0 0,0 1,1 1,1 0,0 0))
```

Figure



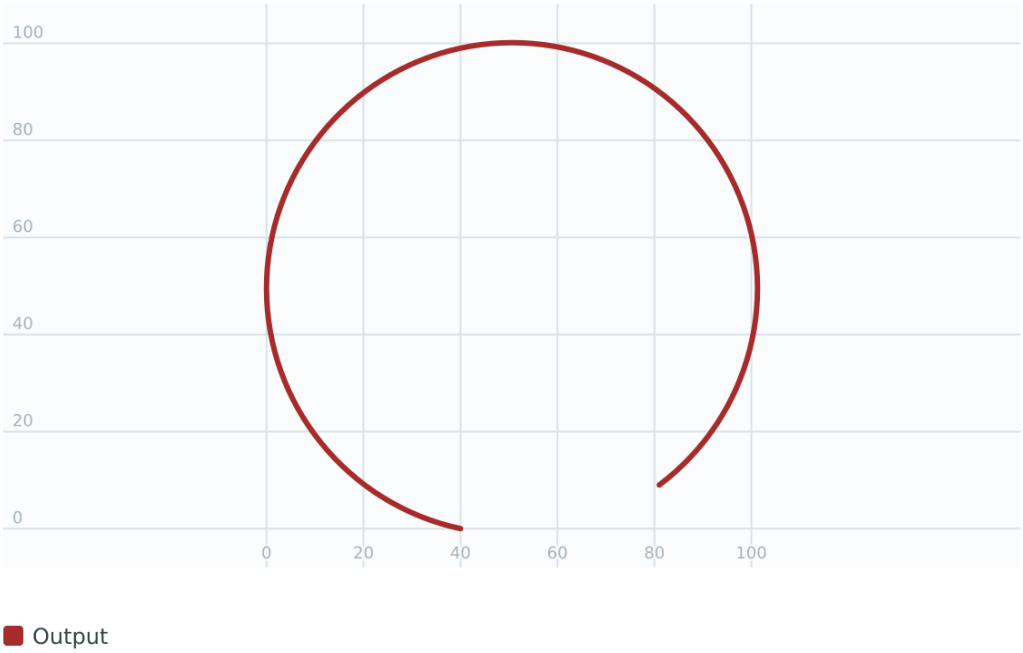
Code

```
SELECT ST_AsEWKT('0108000080030000000000000060 ↩
    E30A4100000000785C0241000000000000F03F0000000018E20A4100000000485F024100000000000000004000000000018
    ');
```

Output

```
CIRCULARSTRING(220268 150415 1,220227 150505 2,220227 150406 3)
```

Figure



☒☒

ST_AsBinary, ST_AsEWKB, ST_AsText, ST_GeomFromEWKT

7.9.1.2 ST_AsText

ST_AsText — ☒☒/☒☒☒☒ WKT(Well-Known Text) ☒☒☒☒ SRID ☒☒☒☒☒☒☒☒☒☒☒☒.

☒☒

text **ST_AsText**(geometry g1);
text **ST_AsText**(geometry g1, integer maxdecimaldigits = 15);
text **ST_AsText**(geography g1);
text **ST_AsText**(geography g1, integer maxdecimaldigits = 15);

☒☒

Returns the OGC Well-Known Text (WKT) representation of the geometry/geography. The optional *`maxdecimaldigits`* argument may be used to limit the number of digits after the decimal point in output ordinates (defaults to 15).

To perform the inverse conversion of WKT representation to PostGIS geometry use ST_GeomFromText.



☒☒
The standard OGC WKT representation does not include the SRID. To include the SRID as part of the output representation, use the non-standard PostGIS function ST_AsEWKT



□□
The textual representation of numbers in WKT may not maintain full floating-point precision. To ensure full accuracy for data storage or transport it is best to use Well-Known Binary (WKB) format (see ST_AsBinary and *maxdecimaldigits*).



□□
Using the *maxdecimaldigits* parameter can cause output geometry to become invalid. To avoid this use ST_ReducePrecision with a suitable gridsize first.

1.5.0 □□□□□□□□□□□□□□.

Enhanced: 2.5 - optional parameter precision introduced.

 This method implements the OGC Simple Features Implementation Specification for SQL 1.1. s2.1.1.1

 This method implements the SQL/MM specification. SQL-MM 3: 5.1.25

 This method supports Circular Strings and Curves.

□□

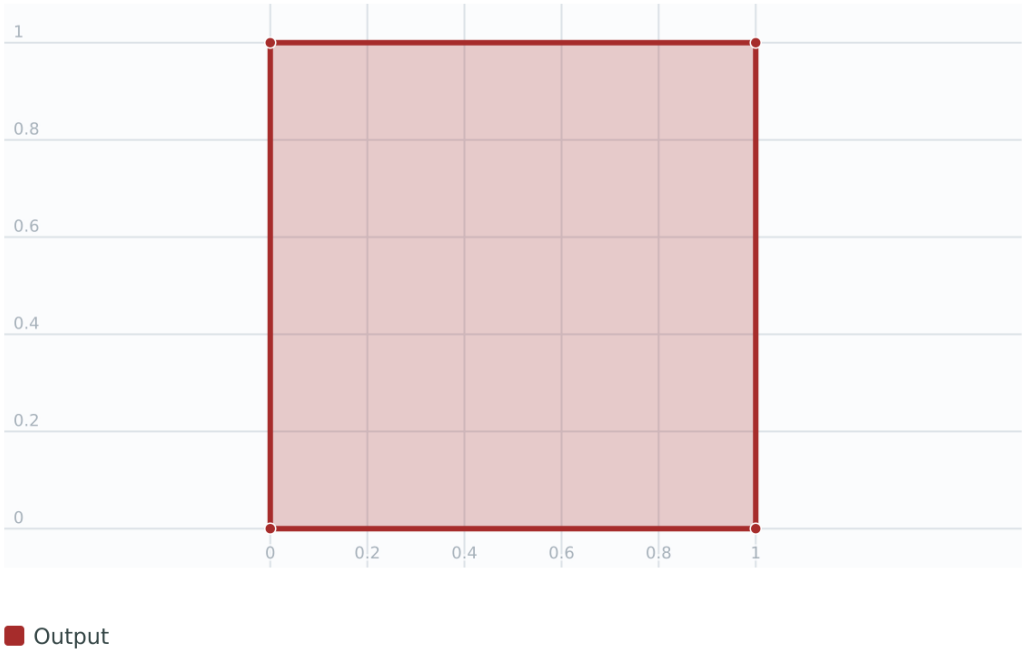
Code

```
SELECT ST_AsText(' ↩
    0103000000010000000500000000000000000000000000000000000000000000000000000000000000000000000000000000 ↩
    f03f000000000000f03f000000000000f03f000000000000f03f000000000000000000000000000000000000000000000000
    ');
```

Output

```
POLYGON((0 0,0 1,1 1,1 0,0 0))
```

Figure



Full precision output is the default.

Code

```
SELECT ST_AsText('POINT(111.1111111 1.1111111)'::geometry);
```

Output

```
POINT(111.1111111 1.1111111)
```

The *maxdecimaldigits* argument can be used to limit output precision.

Code

```
SELECT ST_AsText('POINT(111.1111111 1.1111111)'::geometry, 2);
```

Output

```
POINT(111.11 1.11)
```

☒☒

ST_AsBinary, ST_AsEWKB, ST_AsEWKT, ST_GeomFromText

7.9.2 Well-Known Binary (WKB)

7.9.2.1 ST_AsBinary

ST_AsBinary — Return the OGC/ISO Well-Known Binary (WKB) representation of the geometry/geography without SRID meta data.

□□

bytea **ST_AsBinary**(geometry g1);
bytea **ST_AsBinary**(geometry g1, text NDR_or_XDR);
bytea **ST_AsBinary**(geography g1);
bytea **ST_AsBinary**(geography g1, text NDR_or_XDR);

□□

Returns the OGC/ISO Well-Known Binary (WKB) representation of the geometry. The first function variant defaults to encoding using server machine endian. The second function variant takes a text argument specifying the endian encoding: either 'NDR' for little-endian; or 'XDR' for big-endian. Supplying unknown arguments will result in little-endian output.

WKB format is useful to read geometry data from the database and maintaining full numeric precision. This avoids the precision rounding that can happen with text formats such as WKT.

To perform the inverse conversion of WKB to PostGIS geometry use ST_GeomFromWKB.



□□
The OGC/ISO WKB format does not include the SRID. To get the EWKB format which does include the SRID use ST_AsEWKB



□□
PostgreSQL outputs bytea values in hex encoding by default. If your GUI tools require the old behavior, then SET `bytea_output='escape'` in your database.

□□□□: 2.0.0 □□□□□□□□□□, □□□□ TIN □□□□□□□□□□□□.

□□□□: 2.0.0 □□□□□□□□□□□□□□□□□□□□.

□□□□: 2.0.0 □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□.

1.5.0 □□□□□□□□□□□□□□.

Changed: 2.0.0 Inputs to this function can not be unknown -- must be geometry. Constructs such as `ST_AsBinary('POINT(1 2)')` are no longer valid and you will get an `n ST_AsBinary(unknown) is not unique error`. Code like that needs to be changed to `ST_AsBinary('POINT(1 2)'::geometry);`. If that is not possible, then install `legacy.sql`.

This method implements the OGC Simple Features Implementation Specification for SQL 1.1. s2.1.1.1

This method implements the SQL/MM specification. SQL-MM 3: 5.1.37

This method supports Circular Strings and Curves.

This function supports Polyhedral surfaces.

This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).

This function supports 3d and will not drop the z-index.

□□

Code

```
SELECT ST_AsBinary(ST_GeomFromText('POLYGON((0 0,0 1,1 1,1 0,0 0))', 4326));
```

Output

```
\x0103000000010000000500000000000000000000000000000000000000000000000000000000000000000000000000000000
000000f03f000000000000f03f000000000000f03f000000000000f03f00000000000000000000000000000000
00000000000000000000000000
```

Code

```
SELECT ST_AsBinary(ST_GeomFromText('POLYGON((0 0,0 1,1 1,1 0,0 0))', 4326), 'XDR');
```

Output

```
\x0000000003000000010000000500000000000000000000000000000000000000000000000000000000000000000003ff000
00000000003ff00000000000003ff00000000000003ff00000000000000000000000000000000000000000000000000
00000000000000000000000000
```

Use the PostgreSQL encode function to produce a hexadecimal WKB string without the \x prefix used for bytea display. Specifying NDR makes the byte order deterministic.

Code

```
SELECT encode(
    ST_AsBinary('POINT(15.123 21.456)'::geometry, 'NDR'),
    'hex'
) AS wkb_hex;
```

Output

```
0101000000e5d022dbf93e2e40dbf97e6abc743540
```

☒☒

ST_GeomFromWKB, ST_AsEWKB, ST_AsTWKB, ST_AsText,

7.9.2.2 ST_AsEWKB

ST_AsEWKB — Return the Extended Well-Known Binary (EWKB) representation of the geometry with SRID meta data.

☒☒

bytea **ST_AsEWKB**(geometry g1);
bytea **ST_AsEWKB**(geometry g1, text NDR_or_XDR);

☒☒

Returns the Extended Well-Known Binary (EWKB) representation of the geometry with SRID meta-data. The first function variant defaults to encoding using server machine endian. The second function variant takes a text argument specifying the endian encoding: either 'NDR' for little-endian; or 'XDR' for big-endian. Supplying unknown arguments will result in little-endian output.

WKB format is useful to read geometry data from the database and maintaining full numeric precision. This avoids the precision rounding that can happen with text formats such as WKT.

To perform the inverse conversion of EWKB to PostGIS geometry use ST_GeomFromEWKB.



□□
To get the OGC/ISO WKB format use ST_AsBinary. Note that OGC/ISO WKB format does not include the SRID.

□□□□: 2.0.0 □□□□□□□□□□, □□□□ TIN □□□□□□□□□□□□.


This function supports 3d and will not drop the z-index.


This method supports Circular Strings and Curves.


This function supports Polyhedral surfaces.


This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).

□□

Code

```
SELECT ST_AsEWKB(ST_GeomFromText('POLYGON((0 0,0 1,1 1,1 0,0 0))', 4326));
```

Output

```
\x0103000020e6100000010000000500000000000000000000000000000000000000000000000000000000000000
0000000000000f03f000000000000f03f000000000000f03f000000000000f03f00000000000000
00000000000000000000000000000000
```

Code

```
SELECT ST_AsEWKB(ST_GeomFromText('POLYGON((0 0,0 1,1 1,1 0,0 0))', 4326), 'XDR');
```

Output

```
\x002000000300000e6000000010000000500000000000000000000000000000000000000000000000000000000000000
003ff0000000000003ff0000000000003ff0000000000003ff0000000000000000000000000000
00000000000000000000000000000000
```

□□

ST_AsBinary, ST_GeomFromEWKB, ST_SRID

7.9.2.3 ST_AsHEXEWKB

ST_AsHEXEWKB — □□□□□□□□ (NDR) □□□□□□□ (XDR) □□□□□□□ HEXEWKB (□□□) □□□□□ □□□□□□.

□□

text **ST_AsHEXEWKB**(geometry g1, text NDRorXDR);
text **ST_AsHEXEWKB**(geometry g1);

⊠⊠

⊠⊠⊠⊠⊠⊠⊠ (NDR) ⊠⊠⊠⊠⊠⊠ (XDR) ⊠⊠⊠⊠⊠⊠ HEXEWKB (⊠⊠⊠) ⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠ NDR ⊠⊠⊠⊠.

1.2.2 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

✅ This function supports 3d and will not drop the z-index.

✅ This method supports Circular Strings and Curves.

⊠⊠

Code

```sql
SELECT ST_AsHEXEWKB(ST_GeomFromText('POLYGON((0 0,0 1,1 1,1 0,0 0))', 4326));
```

Output

```
0103000020E6100000010000000500
00000000000000000000000000000000
00000000000000000000000000000000F03F
000000000000F03F000000000000F03F000000000000F03
F000000000000000000000000000000000000000000000000
```

Which gives the same answer as:

Code

```sql
SELECT ST_GeomFromText('POLYGON((0 0,0 1,1 1,1 0,0 0))', 4326)::text;
```

7.9.3 Other Formats

7.9.3.1 ST_AsEncodedPolyline

ST_AsEncodedPolyline — ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

text **ST_AsEncodedPolyline**(geometry geom, integer precision=5);

⊠⊠

Returns the geometry in the Encoded Polyline Algorithm Format. The format stores latitude and longitude deltas as integers after multiplying coordinates by 10 raised to the requested `precision`.

The informal names *polyline5* and *polyline6* refer to the same encoding with precision 5 and 6. The precision must match when encoding and decoding; otherwise the coordinates are scaled incorrectly.

The input geometry must use SRID 4326, with longitude as X and latitude as Y. Use ST_Transform to transform geometries in other coordinate systems before encoding.

The OSRM API returns precision 5 for `geometries=polyline` and precision 6 for `geometries=polyline6`. Decode the resulting `route.geometry` with ST_LineFromEncodedPolyline using the matching precision.

2.2.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

Encode the same WGS 84 LineString using precision 5 and 6.

Code

```
WITH route AS (
  SELECT ST_GeomFromEWKT(
    'SRID=4326;LINESTRING(-120.2 38.5,-120.95 40.7,-126.453 43.252)'
  ) AS geom
)
SELECT ST_AsEncodedPolyline(geom, 5) AS polyline5,
       ST_AsEncodedPolyline(geom, 6) AS polyline6
FROM route;
```

Output

```
polyline5          |           polyline6
---------------------------+--------------------------------
 _p~iF~ps|U_ulLnnqC_mqNvxq`@ | _izlhA~rlgdF_{geC~ywl@_kwzCn`{nI
```

Decode an OSRM `route.geometry` returned with `geometries=polyline6`.

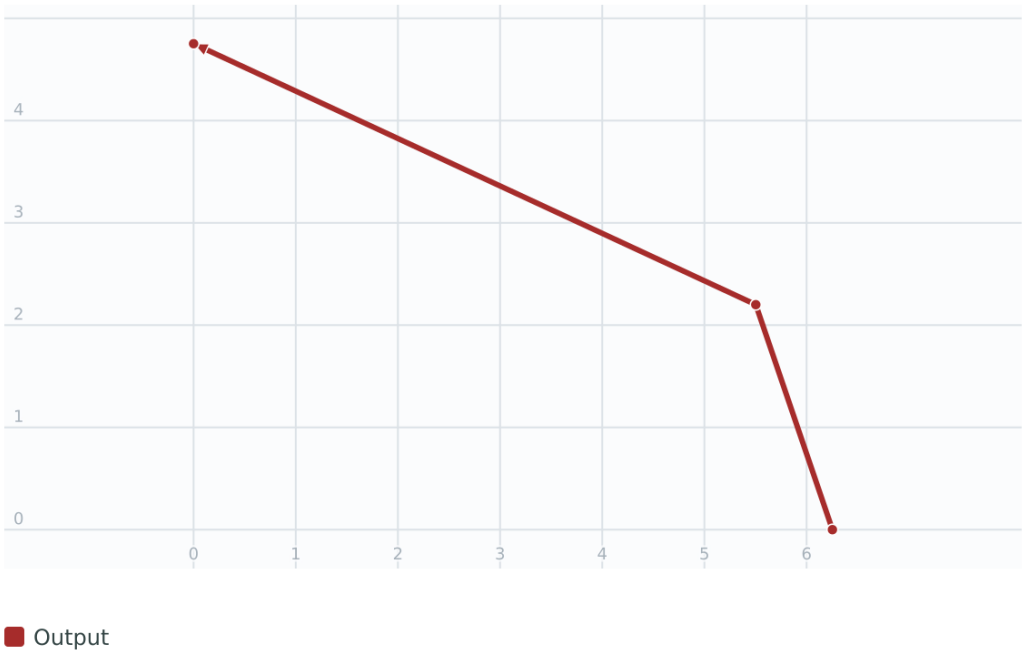
Code

```
SELECT ST_AsEWKT(ST_LineFromEncodedPolyline(
  '_izlhA~rlgdF_{geC~ywl@_kwzCn`{nI',
  6
)) AS route;
```

Output

```
SRID=4326;LINESTRING(-120.2 38.5,-120.95 40.7,-126.453 43.252)
```

Figure



The Google Maps JavaScript Geometry library decodes precision-5 strings. This browser example assumes that `map` is an initialized Google map.

Code

```
<script src="https://maps.googleapis.com/maps/api/js?key=YOUR_API_KEY&libraries=geometry"
></script>
<script>
  const encodedPath = "_p~iF~ps|U_ulLnnqC_mqNvxq`@";
  const flightPath = new google.maps.Polyline({
    path: google.maps.geometry.encoding.decodePath(encodedPath),
    map: map,
    strokeColor: "#0000CC",
    strokeOpacity: 1.0,
    strokeWeight: 4
  });
</script>
```

☒☒

ST_LineFromEncodedPolyline, ST_Transform

7.9.3.2 ST_AsFlatGeobuf

ST_AsFlatGeobuf — Return a FlatGeobuf representation of a set of rows.

☒☒

bytea **ST_AsFlatGeobuf**(anyelement set row);
bytea **ST_AsFlatGeobuf**(anyelement row, bool index);
bytea **ST_AsFlatGeobuf**(anyelement row, bool index, text geom_name);

☒☒

Return a FlatGeobuf representation (http://flatgeobuf.org) of a set of rows corresponding to a FeatureCollection. NOTE: PostgreSQL bytea cannot exceed 1GB.

`row` row data with at least a geometry column.

`index` toggle spatial index creation. Default is false.

`geom_name` is the name of the geometry column in the row data. If NULL it will default to the first found geometry column.

Availability: 3.2.0

7.9.3.3 ST_AsGeobuf

ST_AsGeobuf — Return a Geobuf representation of a set of rows.

☒☒

bytea **ST_AsGeobuf**(anyelement set row);
bytea **ST_AsGeobuf**(anyelement row, text geom_name);

□□

Return a Geobuf representation (https://github.com/mapbox/geobuf) of a set of rows corresponding to a FeatureCollection. Every input geometry is analyzed to determine maximum precision for optimal storage. Note that Geobuf in its current form cannot be streamed so the full output will be assembled in memory.

`row` row data with at least a geometry column.

`geom_name` is the name of the geometry column in the row data. If NULL it will default to the first found geometry column.

2.2.0 □□□□□□□□□□□□.

□□

Code

```
SELECT encode(ST_AsGeobuf(q, 'geom'), 'base64')
FROM (SELECT 'POLYGON((0 0,0 1,1 1,1 0,0 0))'::geometry AS geom) AS q;
```

Output

```
GAAiEAoOCgwIBBoIAAAAAgIAAAE=
```

7.9.3.4 ST_AsGeoJSON

ST_AsGeoJSON — Return a geometry or feature in GeoJSON format.

□□

text **ST_AsGeoJSON**(record feature, text geom_column="", integer maxdecimaldigits=9, boolean pretty_bool=false, text id_column="");
text **ST_AsGeoJSON**(geometry geom, integer maxdecimaldigits=9, integer options=8);
text **ST_AsGeoJSON**(geography geog, integer maxdecimaldigits=9, integer options=0);

□□

Returns a geometry as a GeoJSON "geometry" object, or a row as a GeoJSON "feature" object.

The resulting GeoJSON geometry and feature representations conform with the GeoJSON specifications RFC 7946, except when the parsed geometries are referenced with a CRS other than WGS84 longitude and latitude (EPSG:4326, urn:ogc:def:crs:OGC::CRS84); the GeoJSON geometry object will then have a short CRS SRID identifier attached by default. 2D and 3D Geometries are both supported. GeoJSON only supports SFS 1.1 geometry types (no curve support for example).

The `geom_column` parameter is used to distinguish between multiple geometry columns. If omitted, the first geometry column in the record will be determined. Conversely, passing the parameter will save column type lookups.

The `maxdecimaldigits` argument may be used to reduce the maximum number of decimal places used in output (defaults to 9). If you are using EPSG:4326 and are outputting the geometry only for display, `maxdecimaldigits=6` can be a good choice for many maps.



□□
Using the *maxdecimaldigits* parameter can cause output geometry to become invalid. To avoid this use ST_ReducePrecision with a suitable gridsize first.

The `options` argument can be used to add BBOX or CRS in GeoJSON output:

- 0: means no option
- 1: GeoJSON BBOX
- 2: GeoJSON Short CRS (□: EPSG:4326)
- 4: GeoJSON Long CRS (□: urn:ogc:def:crs:EPSG::4326)
- 8: GeoJSON Short CRS if not EPSG:4326 (default)

The `id_column` parameter is used to set the "id" member of the returned GeoJSON features. As per GeoJSON RFC, this SHOULD be used whenever a feature has a commonly used identifier, such as a primary key. When not specified, the produced features will not get an "id" member and any columns other than the geometry, including any potential keys, will just end up inside the feature's "properties" member.

When the input record includes duplicate column names, the resulting "properties" object will contain repeated keys. `ST_AsGeoJSON` raises a warning in that case because PostgreSQL jsonb keeps only the last value for duplicate keys.

The GeoJSON specification states that polygons are oriented using the Right-Hand Rule, and some clients require this orientation. This can be ensured by using ST_ForcePolygonCCW. The specification also requires that geometry be in the WGS84 coordinate system (SRID = 4326). If necessary geometry can be projected into WGS84 using ST_Transform: `ST_Transform(geom, 4326 )`.

GeoJSON can be tested and viewed online at geojson.io and geojsonlint.com. It is widely supported by web mapping frameworks:

- OpenLayers GeoJSON Example
- Leaflet GeoJSON Example
- Mapbox GL GeoJSON Example

1.3.4 □□□□□□□□□□□□.

1.5.0 □□□□□□□□□□□□.

□□□□: 2.0.0 □□□□□□□□□□ (default arg) □□□□□□□□ (named arg) □□□□□□.

Changed: 3.0.0 support records as input

Changed: 3.0.0 output SRID if not EPSG:4326.

Changed: 3.5.0 allow specifying the column containing the feature id

Changed: 3.7.0 added warning about duplicate keys

 This function supports 3d and will not drop the z-index.

□□

Generate a FeatureCollection:

Code

```
SELECT json_build_object(
    'type', 'FeatureCollection',
    'features', json_agg(ST_AsGeoJSON(t.*, id_column =
> 'id')::json)
    )
FROM ( VALUES (1, 'one', 'POINT(1 1)'::geometry),
              (2, 'two', 'POINT(2 2)'),
              (3, 'three', 'POINT(3 3)')
     ) as t(id, name, geom);
```

Output

```
{"type" : "FeatureCollection", "features" : [{"type": "Feature", "geometry": {"type":"Point","coordinates":[1,1]}, "id ↩
     ": 1, "properties": {"name": "one"}}, {"type": "Feature", "geometry": {"type":"Point","coordinates":[2,2]}, "id": ↩
      2, "properties": {"name": "two"}}, {"type": "Feature", "geometry": {"type":"Point","coordinates":[3,3]}, "id":  ↩
     3, "properties": {"name": "three"}}]}
```

Generate a Feature:

Code

```
SELECT ST_AsGeoJSON(t.*, id_column =
> 'id')
FROM (VALUES (1, 'one', 'POINT(1 1)'::geometry)) AS t(id, name, geom);
```

Output

```
{"type": "Feature", "geometry": {"type":"Point","coordinates":[1,1]}, "id": 1, "properties": {"name": "one"}}
```

Don't forget to transform your data to WGS84 longitude, latitude to conform with the GeoJSON specification:

Code

```
SELECT ST_AsGeoJSON(ST_Transform(geom, 4326)) FROM fe_edges LIMIT 1;
```

Output

```
{"type":"MultiLineString","coordinates":[[[-89.734634999999997,31.492072000000000],
[-89.734955999999997,31.492237999999997]]]}
```

3D geometries are supported:

Code

```
SELECT ST_AsGeoJSON('LINESTRING(1 2 3,4 5 6)');
```

Output

```
{"type":"LineString","coordinates":[[1,2,3],[4,5,6]]}
```

Options argument can be used to add BBOX and CRS in GeoJSON output:

Code

```
SELECT ST_AsGeoJSON(ST_SetSRID('POINT(1 1)'::geometry, 4326), 9, 4|1);
```

Output

```
{"type":"Point","crs":{"type":"name","properties":{"name":"urn:ogc:def:crs:EPSG::4326"}},"bbox ↩
    ":[1.000000000,1.000000000,1.000000000,1.000000000],"coordinates":[1,1]}
```

□□

ST_GeomFromGeoJSON, ST_ForcePolygonCCW, ST_Transform

7.9.3.5 ST_AsGML

ST_AsGML — □□□ GML 2 □□ GML 3 □□□□□□□□□□.

□□

text **ST_AsGML**(geometry geom, integer maxdecimaldigits=15, integer options=0);
text **ST_AsGML**(geography geog, integer maxdecimaldigits=15, integer options=0, text nprefix=null, text id=null);
text **ST_AsGML**(integer version, geometry geom, integer maxdecimaldigits=15, integer options=0, text nprefix=null, text id=null);
text **ST_AsGML**(integer version, geography geog, integer maxdecimaldigits=15, integer options=0, text nprefix=null, text id=null);

□□

Return the geometry as a Geography Markup Language (GML) element. The version parameter, if specified, may be either 2 or 3. If no version parameter is specified then the default is assumed to be 2. The `maxdecimaldigits` argument may be used to reduce the maximum number of decimal places used in output (defaults to 15).

□□
Using the *`maxdecimaldigits`* parameter can cause output geometry to become invalid. To avoid this use ST_ReducePrecision with a suitable gridsize first.

GML 2 □ 2.1.2 □□□, GML 3 □ 3.1.1 □□□□□□□□.

’ □□’ □□□□□□□□ (bitfield) □□□□□□. CRS □□□□□ GML □□□□□□□□, □□□□□□/□□□ □□□□□□□□□□□□.

- 0: GML Short CRS (□: EPSG:4326), □□□
- 1: GML Long CRS (□: urn:ogc:def:crs:EPSG::4326)
- 2: GML 3 □□□, □□□□□ srsDimension □□□□□□□□.
- 4: GML 3 □□□, □□□□□ <Curve> □□ <LineString> □□□□□□□□.
- 16: □□□□□□/□□ (□: srid=4326) □□□□□□□. □□□□□□□□□□□□□□□□□□□□□□□□. □□□□□□□□ (axis order) □□□□□□, GML 3.1.1 □□□□□□□□□□□□□□□□. □□□□□□□□□□□, □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□.
- 32: □□□□□□□□ (envelope) □□□□□□.

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (⊠⊠⊠⊠⊠⊠⊠) ⊠⊠⊠’ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠’ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠ NULL ⊠⊠⊠⊠⊠⊠⊠⊠⊠’gml’ ⊠⊠⊠⊠⊠⊠⊠⊠.

1.3.2 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

1.5.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠⊠⊠: 2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ GML 3 ⊠⊠⊠’4’ ⊠⊠⊠⊠⊠⊠⊠⊠. GML 3 ⊠⊠⊠⊠⊠⊠⊠⊠ TIN ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠’32’ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠⊠⊠: 2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (named arg) ⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠⊠⊠: 2.1.0 ⊠⊠⊠⊠ GML 3 ⊠⊠⊠ ID ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

> **⊠⊠**
> ST_AsGML ⊠⊠⊠ 3 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ TIN ⊠⊠⊠⊠⊠⊠⊠.

This method implements the SQL/MM specification. SQL-MM IEC 13249-3: 17.2

This function supports 3d and will not drop the z-index.

This function supports Polyhedral surfaces.

This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).

⊠⊠

Version 2.

Code

```
SELECT ST_AsGML(ST_GeomFromText('POLYGON((0 0,0 1,1 1,1 0,0 0))', 4326));
```

Output

```
<gml:Polygon srsName="EPSG:4326"
><gml:outerBoundaryIs
><gml:LinearRing
><gml:coordinates
>0,0 0,1 1,1 1,0 0,0</gml:coordinates
></gml:LinearRing
></gml:outerBoundaryIs
></gml:Polygon
>
```

Version 3.

Flip coordinates and output extended EPSG (16 | 1)

Code

```
SELECT ST_AsGML(3, ST_GeomFromText('POINT(5.234234233242 6.34534534534)', 4326), 5, 17);
```

Output

```
<gml:Point srsName="urn:ogc:def:crs:EPSG::4326"
><gml:pos srsDimension="2"
>6.34535 5.23423</gml:pos
></gml:Point
>
```

Output the envelope (32).

Code

```
SELECT ST_AsGML(3, ST_GeomFromText('LINESTRING(1 2, 3 4, 10 20)',4326), 5, 32);
```

Output

```
<gml:Envelope srsName="EPSG:4326" srsDimension="2"
><gml:lowerCorner
>1 2</gml:lowerCorner
><gml:upperCorner
>10 20</gml:upperCorner
></gml:Envelope
>
```

Output the envelope (32), reverse (lat lon instead of lon lat) (16), long srs (1) = 32 | 16 | 1 = 49.

Code

```
SELECT ST_AsGML(3, ST_GeomFromText('LINESTRING(1 2, 3 4, 10 20)',4326), 5, 49);
```

Output

```
<gml:Envelope srsName="urn:ogc:def:crs:EPSG::4326" srsDimension="2"
><gml:lowerCorner
>2 1</gml:lowerCorner
><gml:upperCorner
>20 10</gml:upperCorner
></gml:Envelope
>
```

This example uses a PolyhedralSurface.

Code

```
SELECT ST_AsGML(
3,
'POLYHEDRALSURFACE( ((0 0 0, 0 0 1, 0 1 1, 0 1 0, 0 0 0)),
((0 0 0, 0 1 0, 1 1 0, 1 0 0, 0 0 0)), ((0 0 0, 1 0 0, 1 0 1, 0 0 1, 0 0 0)),
((1 1 0, 1 1 1, 1 0 1, 1 0 0, 1 1 0)),
((0 1 0, 0 1 1, 1 1 1, 1 1 0, 0 1 0)), ((0 0 1, 1 0 1, 1 1 1, 0 1 1, 0 0 1)) )'::geometry);
```

Output

```
 <gml:PolyhedralSurface>
<gml:polygonPatches>
   <gml:PolygonPatch>
    <gml:exterior>
        <gml:LinearRing>
           <gml:posList srsDimension="3"
>0 0 0 0 0 1 0 1 1 0 1 0 0 0 0</gml:posList>
        </gml:LinearRing>
    </gml:exterior>
   </gml:PolygonPatch>
   <gml:PolygonPatch>
    <gml:exterior>
        <gml:LinearRing>
           <gml:posList srsDimension="3"
>0 0 0 0 1 0 1 1 0 1 0 0 0 0 0</gml:posList>
        </gml:LinearRing>
    </gml:exterior>
   </gml:PolygonPatch>
   <gml:PolygonPatch>
    <gml:exterior>
        <gml:LinearRing>
           <gml:posList srsDimension="3"
>0 0 0 1 0 0 1 0 1 0 0 1 0 0 0</gml:posList>
        </gml:LinearRing>
    </gml:exterior>
   </gml:PolygonPatch>
   <gml:PolygonPatch>
    <gml:exterior>
        <gml:LinearRing>
           <gml:posList srsDimension="3"
>1 1 0 1 1 1 1 0 1 1 0 0 1 1 0</gml:posList>
```

```
        </gml:LinearRing>
    </gml:exterior>
   </gml:PolygonPatch>
   <gml:PolygonPatch>
    <gml:exterior>
        <gml:LinearRing>
           <gml:posList srsDimension="3"
>0 1 0 0 1 1 1 1 1 1 1 0 0 1 0</gml:posList>
        </gml:LinearRing>
    </gml:exterior>
   </gml:PolygonPatch>
   <gml:PolygonPatch>
    <gml:exterior>
        <gml:LinearRing>
           <gml:posList srsDimension="3"
>0 0 1 1 0 1 1 1 1 0 1 1 0 0 1</gml:posList>
        </gml:LinearRing>
    </gml:exterior>
   </gml:PolygonPatch>
</gml:polygonPatches>
</gml:PolyhedralSurface
>
```

⊠⊠

ST_GeomFromGML

7.9.3.6 ST_AsKML

ST_AsKML — ⊠⊠⊠ GML 2 ⊠⊠ GML 3 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

text **ST_AsKML**(geometry geom, integer maxdecimaldigits=15, text nprefix=NULL);
text **ST_AsKML**(geography geog, integer maxdecimaldigits=15, text nprefix=NULL);

⊠⊠

⊠⊠⊠ KML(Keyhole Markup Language) ⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (⊠⊠⊠⊠ 15), ⊠⊠⊠⊠⊠⊠⊠⊠ 2 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠
Using the *maxdecimaldigits* parameter can cause output geometry to become invalid. To avoid this use ST_ReducePrecision with a suitable gridsize first.

1.2.2 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ 1.3.2 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠⊠⊠: 2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

Changed: 3.0.0 - Removed the ”versioned” variant signature

⊠⊠
PostGIS ⊠ Proj ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. Proj ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ PostGIS_Full_Version ⊠⊠⊠⊠⊠⊠⊠⊠.



□□
ST_AsKML □□□□ SRID □□□□□□□□□□□□□□□□□□□.

This function supports 3d and will not drop the z-index.

□□

Code

```
SELECT ST_AsKML(ST_GeomFromText('POLYGON((0 0,0 1,1 1,1 0,0 0))', 4326));
```

Output

```
<Polygon
><outerBoundaryIs
><LinearRing
><coordinates
>0,0 0,1 1,1 1,0 0,0</coordinates
></LinearRing
></outerBoundaryIs
></Polygon
>
```

This example outputs a 3D LineString.

Code

```
SELECT ST_AsKML('SRID=4326;LINESTRING(1 2 3,4 5 6)');
```

Output

```
<LineString
><coordinates
>1,2,3 4,5,6</coordinates
></LineString
>
```

□□

ST_AsSVG, ST_AsGML

7.9.3.7 ST_AsLatLonText

ST_AsLatLonText — □□□□□□□□□, □, □□□□□□□□□□.

□□

text **ST_AsLatLonText**(geometry pt, text format='');

⊠⊠

⊠⊠⊠⊠⊠, ⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

> **Note!** ⊠⊠
> ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠/⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠ X(⊠⊠) ⊠ Y(⊠⊠) ⊠⊠⊠” ⊠⊠” ⊠⊠ (⊠⊠⊠ -180 ⊠⊠⊠ 180 ⊠, ⊠⊠⊠ -90 ⊠⊠⊠ 90 ⊠) ⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠”D”, ⊠⊠⊠⊠⊠⊠”M”, ⊠⊠⊠⊠⊠⊠”S”, ⊠⊠⊠⊠⊠⊠⊠ (⊠⊠⊠⊠, cardinal direction) ⊠⊠⊠⊠⊠”C” ⊠⊠⊠. D, M, S ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (”SSS.SSSS” ⊠”1.0023” ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠).

M, S, C ⊠⊠⊠⊠⊠⊠⊠. ”C” ⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠”-” ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ”S” ⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ”M” ⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠⊠⊠⊠⊠⊠⊠ (⊠⊠⊠⊠⊠ 0 ⊠) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

2.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

⊠⊠⊠⊠

Code

```
SELECT (ST_AsLatLonText('POINT (-3.2342342 -2.32498)'));
```

Output

```
2°19'29.928"S 3°14'3.243"W
```

(⊠⊠⊠⊠⊠⊠⊠) ⊠⊠⊠⊠⊠⊠⊠⊠.

Code

```
SELECT (ST_AsLatLonText('POINT (-3.2342342 -2.32498)', 'D°M''S.SSS"C'));
```

Output

```
2°19'29.928"S 3°14'3.243"W
```

D, M, S, C ⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

Code

```
SELECT (ST_AsLatLonText('POINT (-3.2342342 -2.32498)', 'D degrees, M minutes, S seconds to ↩
    the C'));
```

Output

```
2 degrees, 19 minutes, 30 seconds to the S 3 degrees, 14 minutes, 3 seconds to the W
```

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

Code

```
SELECT (ST_AsLatLonText('POINT (-3.2342342 -2.32498)', 'D°M''S.SSS"'));
```

Output

```
-2°19'29.928" -3°14'3.243"
```

□□□□□□□□□□□□□□.

Code

```
SELECT (ST_AsLatLonText('POINT (-3.2342342 -2.32498)', 'D.DDDD degrees C'));
```

Output

```
2.3250 degrees S 3.2342 degrees W
```

To convert parsed degree, minute, and second components to decimal degrees, apply the cardinal-direction sign explicitly.

Code

```
WITH dms(label, degrees, minutes, seconds, hemisphere) AS (
  VALUES
    ('latitude', 43::numeric, 0::numeric, 50.60::numeric, 'S'),
    ('longitude', 147::numeric, 12::numeric, 18.20::numeric, 'E')
)
SELECT label,
       round(
         (CASE WHEN hemisphere IN ('S', 'W') THEN -1 ELSE 1 END)
         * (degrees + minutes / 60 + seconds / 3600),
         9
       ) AS decimal_degrees
FROM dms;
```

Output

```
   label   | decimal_degrees
-----------+-----------------
 latitude  |   -43.014055556
 longitude |   147.205055556
```

□□□□□□□□□□□□□□.

Code

```
SELECT (ST_AsLatLonText('POINT (-302.2342342 -792.32498)'));
```

Output

```
72°19'29.928"S 57°45'56.757"E
```

7.9.3.8 ST_AsMARC21

ST_AsMARC21 — Returns geometry as a MARC21/XML record with a geographic datafield (034).

□□

text **ST_AsMARC21** (geometry geom , text format='hdddmmss');

☒☒

This function returns a MARC21/XML record with Coded Cartographic Mathematical Data representing the bounding box of a given geometry. The `format` parameter allows to encode the coordinates in subfields `$d`,`$e`,`$f` and `$g` in all formats supported by the MARC21/XML standard. Valid formats are:

- cardinal direction, degrees, minutes and seconds (default): `hdddmmss`
- decimal degrees with cardinal direction: `hddd.dddddd`
- decimal degrees without cardinal direction: `ddd.dddddd`
- decimal minutes with cardinal direction: `hdddmm.mmmm`
- decimal minutes without cardinal direction: `dddmm.mmmm`
- decimal seconds with cardinal direction: `hdddmmss.sss`

The decimal sign may be also a comma, e.g. `hdddmm,mmmm`.

The precision of decimal formats can be limited by the number of characters after the decimal sign, e.g. `hdddmm.mm` for decimal minutes with a precision of two decimals.

This function ignores the Z and M dimensions.

LOC MARC21/XML versions supported:

- MARC21/XML 1.1

Availability: 3.3.0



☒☒

This function does not support non lon/lat geometries, as they are not supported by the MARC21/XML standard (Coded Cartographic Mathematical Data).



☒☒

The MARC21/XML Standard does not provide any means to annotate the spatial reference system for Coded Cartographic Mathematical Data, which means that this information will be lost after conversion to MARC21/XML.

☒☒

Converting a `POINT` to MARC21/XML formatted as hdddmmss (default)

Code

```
SELECT ST_AsMARC21('SRID=4326;POINT(-4.504289 54.253312)'::geometry);
```

Output

```
                                st_asmarc21
---------------------------------------------------
<record xmlns="http://www.loc.gov/MARC21/slim">
    <datafield tag="034" ind1="1" ind2=" ">
        <subfield code="a"
>a</subfield>
        <subfield code="d"
>W0043015</subfield>
        <subfield code="e"
```

```
>W0043015</subfield>
        <subfield code="f"
>N0541512</subfield>
        <subfield code="g"
>N0541512</subfield>
    </datafield>
</record>
```

Converting a POLYGON to MARC21/XML formatted in decimal degrees

Code

```
SELECT ST_AsMARC21('SRID=4326;POLYGON((-4.5792388916015625 ↩
    54.18172660239091,-4.56756591796875 54.196993557130355,-4.546623229980469 ↩
    54.18313300502024,-4.5792388916015625 54.18172660239091))'::geometry,'hddd.dddd');

<record xmlns="http://www.loc.gov/MARC21/slim">
    <datafield tag="034" ind1="1" ind2=" ">
        <subfield code="a"
>a</subfield>
        <subfield code="d"
>W004.5792</subfield>
        <subfield code="e"
>W004.5466</subfield>
        <subfield code="f"
>N054.1970</subfield>
        <subfield code="g"
>N054.1817</subfield>
    </datafield>
</record>
```

Converting a GEOMETRYCOLLECTION to MARC21/XML formatted in decimal minutes. The geometries order in the MARC21/XML output correspond to their order in the collection.

Code

```
SELECT ST_AsMARC21('SRID=4326;GEOMETRYCOLLECTION(POLYGON((13.1 52.65,13.516666666666667 ↩
    52.65,13.516666666666667 52.38333333333333,13.1 52.38333333333333,13.1 52.65)),POINT ↩
    (-4.5 54.25))'::geometry,'hdddmm.mmmm');
```

Output

```
                                st_asmarc21
---------------------------------------------------
<record xmlns="http://www.loc.gov/MARC21/slim">
    <datafield tag="034" ind1="1" ind2=" ">
        <subfield code="a"
>a</subfield>
        <subfield code="d"
>E01307.0000</subfield>
        <subfield code="e"
>E01331.0000</subfield>
        <subfield code="f"
>N05240.0000</subfield>
        <subfield code="g"
>N05224.0000</subfield>
    </datafield>
    <datafield tag="034" ind1="1" ind2=" ">
        <subfield code="a"
>a</subfield>
        <subfield code="d"
>W00430.0000</subfield>
        <subfield code="e"
>W00430.0000</subfield>
        <subfield code="f"
>N05415.0000</subfield>
        <subfield code="g"
```

```
>N05415.0000</subfield>
    </datafield>
</record>
```

□□

ST_GeomFromMARC21

7.9.3.9 ST_AsMVTGeom

ST_AsMVTGeom — Transforms a geometry into the coordinate space of a MVT tile.

□□

geometry **ST_AsMVTGeom**(geometry geom, box2d bounds, integer extent=4096, integer buffer=256, boolean clip_geom=true);

□□

Transforms a geometry into the coordinate space of a MVT (Mapbox Vector Tile) tile, clipping it to the tile bounds if required. The geometry must be in the coordinate system of the target map (using ST_Transform if needed). Commonly this is Web Mercator (SRID:3857).

The function attempts to preserve geometry validity, and corrects it if needed. This may cause the result geometry to collapse to a lower dimension.

The function performs geometry simplification appropriate to the integer tile coordinate space. After transforming to tile coordinates it snaps coordinates to the integer grid, removes repeated points, and removes points that lie on straight lines. It does not perform general-purpose, scale-dependent simplification such as ST_Simplify. To reduce detail before tile conversion, simplify the geometry in map coordinates before calling `ST_AsMVTGeom`.

The rectangular bounds of the tile in the target map coordinate space must be provided, so the geometry can be transformed, and clipped if required. The bounds can be generated using ST_TileEnvelope.

This function is used to convert geometry into the tile coordinate space required by ST_AsMVT.

`geom` is the geometry to transform, in the coordinate system of the target map.

`bounds` is the rectangular bounds of the tile in map coordinate space, with no buffer.

`extent` is the tile extent size in tile coordinate space as defined by the MVT specification. Defaults to 4096.

`buffer` is the buffer size in tile coordinate space for geometry clippig. Defaults to 256.

`clip_geom` is a boolean to control if geometries are clipped or encoded as-is. Defaults to true.

2.2.0 □□□□□□□□□□□□.



□□
From 3.0, Wagyu can be chosen at configure time to clip and validate MVT polygons. This library is faster and produces more correct results than the GEOS default, but it might drop small polygons.

□□

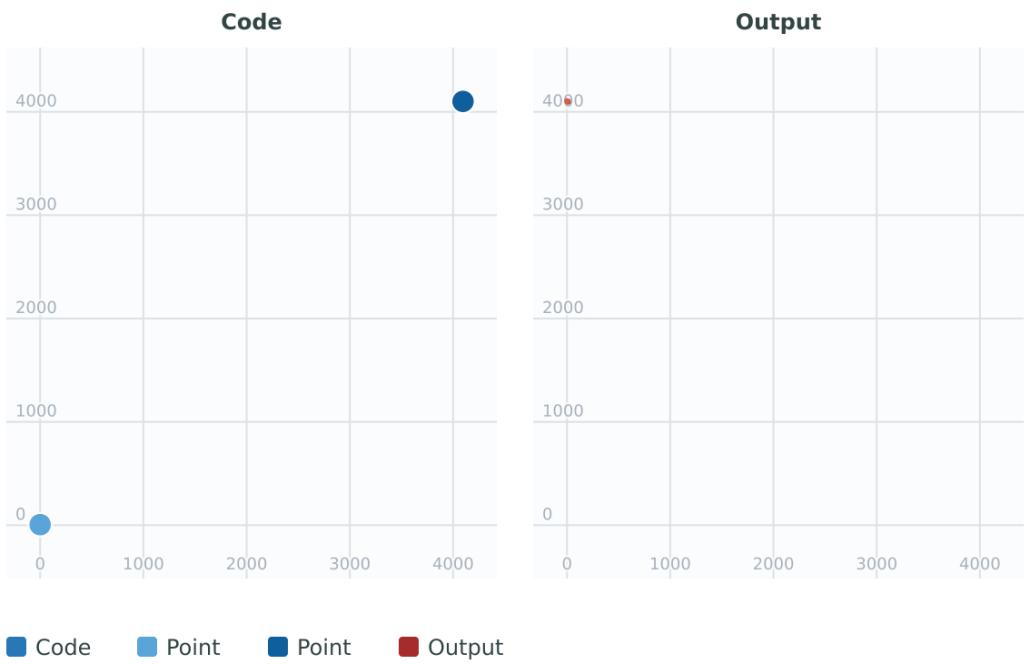
Code

```
SELECT ST_AsMVTGeom('POLYGON ((0 0,10 0,10 5,0 -5,0 0))',
    ST_MakeBox2D(ST_Point(0, 0), ST_Point(4096, 4096)),
    4096, 0, false);
```

Output

```
MULTIPOLYGON(((5 4096,10 4091,10 4096,5 4096)),((5 4096,0 4101,0 4096,5 4096)))
```

Figure



Canonical example for a Web Mercator tile using a computed tile bounds to query and clip geometry. This assumes the data.geom column has srid of 4326.

Code

```
SELECT ST_AsMVTGeom(ST_Transform(geom, 3857 ),
            ST_TileEnvelope(12, 513, 412),
            extent =
> 4096,
            buffer =
> 64) AS geom
  FROM data
  WHERE geom && ST_Transform(ST_TileEnvelope(12, 513, 412, margin =
> (64.0 / 4096)), 4326)
```

□□

ST_AsMVT, ST_TileEnvelope, PostGIS_Wagyu_Version

7.9.3.10 ST_AsMVT

ST_AsMVT — Aggregate function returning a MVT representation of a set of rows.

□□

bytea **ST_AsMVT**(anyelement set row);
bytea **ST_AsMVT**(anyelement row, text name);
bytea **ST_AsMVT**(anyelement row, text name, integer extent);
bytea **ST_AsMVT**(anyelement row, text name, integer extent, text geom_name);
bytea **ST_AsMVT**(anyelement row, text name, integer extent, text geom_name, text feature_id_name);

□□

An aggregate function which returns a binary Mapbox Vector Tile representation of a set of rows corresponding to a tile layer. The rows must contain a geometry column which will be encoded as a feature geometry. The geometry must be in tile coordinate space and valid as per the MVT specification. ST_AsMVTGeom can be used to transform geometry into tile coordinate space. Other row columns are encoded as feature attributes.

The Mapbox Vector Tile format can store features with varying sets of attributes. To use this capability supply a JSONB column in the row data containing Json objects one level deep. The keys and values in the JSONB values will be encoded as feature attributes.

Tiles with multiple layers can be created by concatenating multiple calls to this function using `||` or `STRING_AGG`.



□□
Do not call with a `GEOMETRYCOLLECTION` as an element in the row. However you can use ST_AsMVTGeom to prepare a geometry collection for inclusion.

`row` row data with at least a geometry column.

`name` is the name of the layer. Default is the string "default".

`extent` is the tile extent in screen space as defined by the specification. Default is 4096.

`geom_name` is the name of the geometry column in the row data. Default is the first geometry column. Note that PostgreSQL by default automatically folds unquoted identifiers to lower case, which means that unless the geometry column is quoted, e.g. `"MyMVTGeom"`, this parameter must be provided as lowercase.

`feature_id_name` is the name of the Feature ID column in the row data. If NULL or negative the Feature ID is not set. The first column matching name and valid type (smallint, integer, bigint) will be used as Feature ID, and any subsequent column will be added as a property. JSON properties are not supported.

Enhanced: 3.0 - added support for Feature ID.

Enhanced: 2.5.0 - added support parallel query.

2.2.0 □□□□□□□□□□□□.

□□

Code

```
WITH mvtgeom AS
(
  SELECT ST_AsMVTGeom(geom, ST_TileEnvelope(12, 513, 412), extent =
> 4096, buffer =
> 64) AS geom, name, description
```

```
  FROM points_of_interest
  WHERE geom && ST_TileEnvelope(12, 513, 412, margin =
> (64.0 / 4096))
)
SELECT ST_AsMVT(mvtgeom.*)
FROM mvtgeom;
```

⊠⊠

ST_AsMVTGeom, ST_TileEnvelope

7.9.3.11 ST_AsSVG

ST_AsSVG — Returns SVG path data for a geometry.

⊠⊠

text **ST_AsSVG**(geometry geom, integer rel=0, integer maxdecimaldigits=15);
text **ST_AsSVG**(geography geog, integer rel=0, integer maxdecimaldigits=15);

⊠⊠

⊠⊠⊠ SVG(Scalar Vector Graphics) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠ (relative move) ⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ 1 ⊠⊠⊠⊠⊠, ⊠⊠⊠⊠ (absolute move) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠ 0 ⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (⊠⊠⊠⊠ 15) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠. 'rel' ⊠⊠⊠⊠⊠ 0 ⊠⊠⊠⊠⊠⊠⊠⊠ cx/cy ⊠⊠⊠⊠⊠⊠, 'rel' ⊠⊠⊠⊠⊠ 1 ⊠⊠ x/y ⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (",") ⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (";") ⊠⊠⊠⊠⊠⊠⊠.

For working with PostGIS SVG graphics, checkout pg_svg library which provides plpgsql functions for working with outputs from ST_AsSVG.

Enhanced: 3.4.0 to support all curve types

⊠⊠⊠⊠: 2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (default arg) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (named arg) ⊠⊠⊠⊠⊠⊠.

1.2.2 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. 1.4.0 ⊠⊠⊠⊠ http://www.w3.org/TR/SVG/paths.html#PathDataBNF ⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ L ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

This method supports Circular Strings and Curves.

⊠⊠

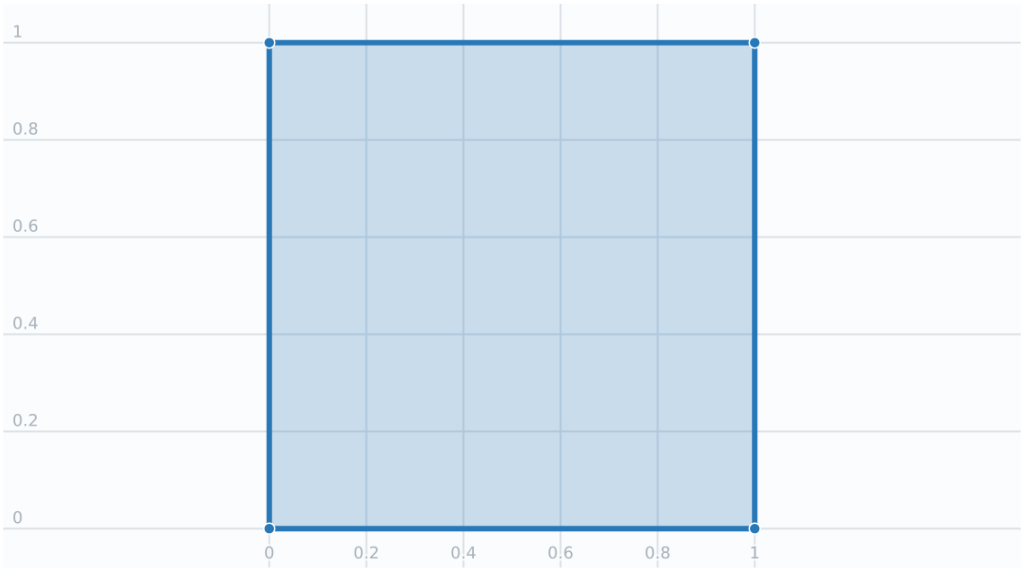
Code

```
SELECT ST_AsSVG('POLYGON((0 0,0 1,1 1,1 0,0 0))'::geometry);
```

Output

```
M 0 0 L 0 -1 1 -1 1 0 Z
```

Figure



Circular string

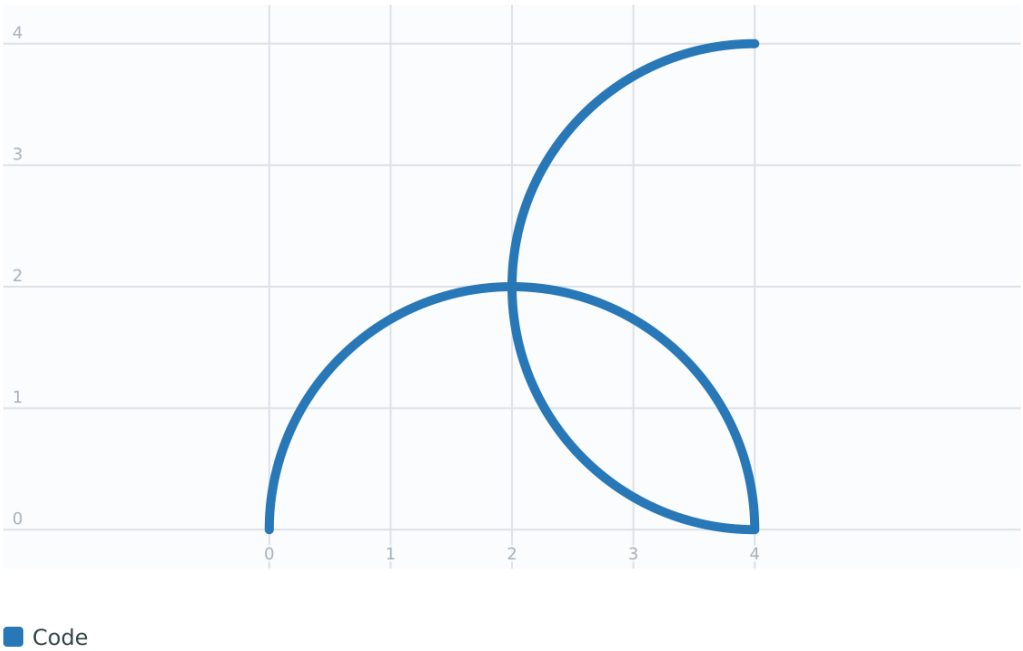
Code

```
SELECT ST_AsSVG('CIRCULARSTRING(-2 0,0 2,2 0,0 2,2 4)'::geometry);
```

Output

```
M -2 0 A 2 2 0 0 1 2 0 A 2 2 0 0 1 2 -4
```

Figure



Multi-curve

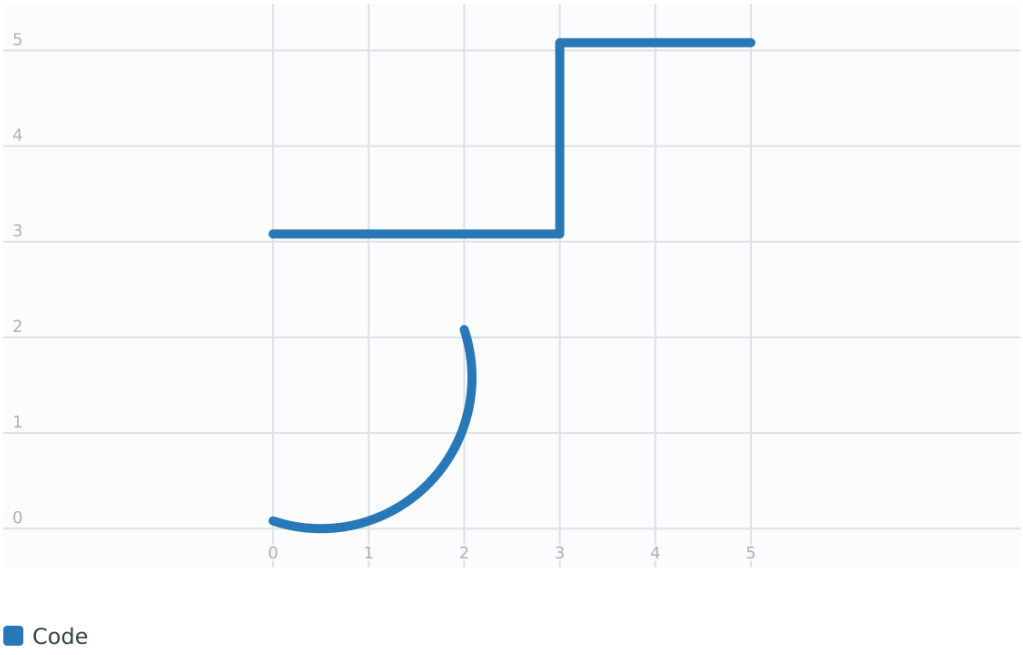
Code

```
SELECT ST_AsSVG('MULTICURVE((5 5,3 5,3 3,0 3),
CIRCULARSTRING(0 0,2 1,2 2))'::geometry, 0, 0);
```

Output

```
M 5 -5 L 3 -5 3 -3 0 -3 M 0 0 A 2 2 0 0 0 2 -2
```

Figure



Multi-surface

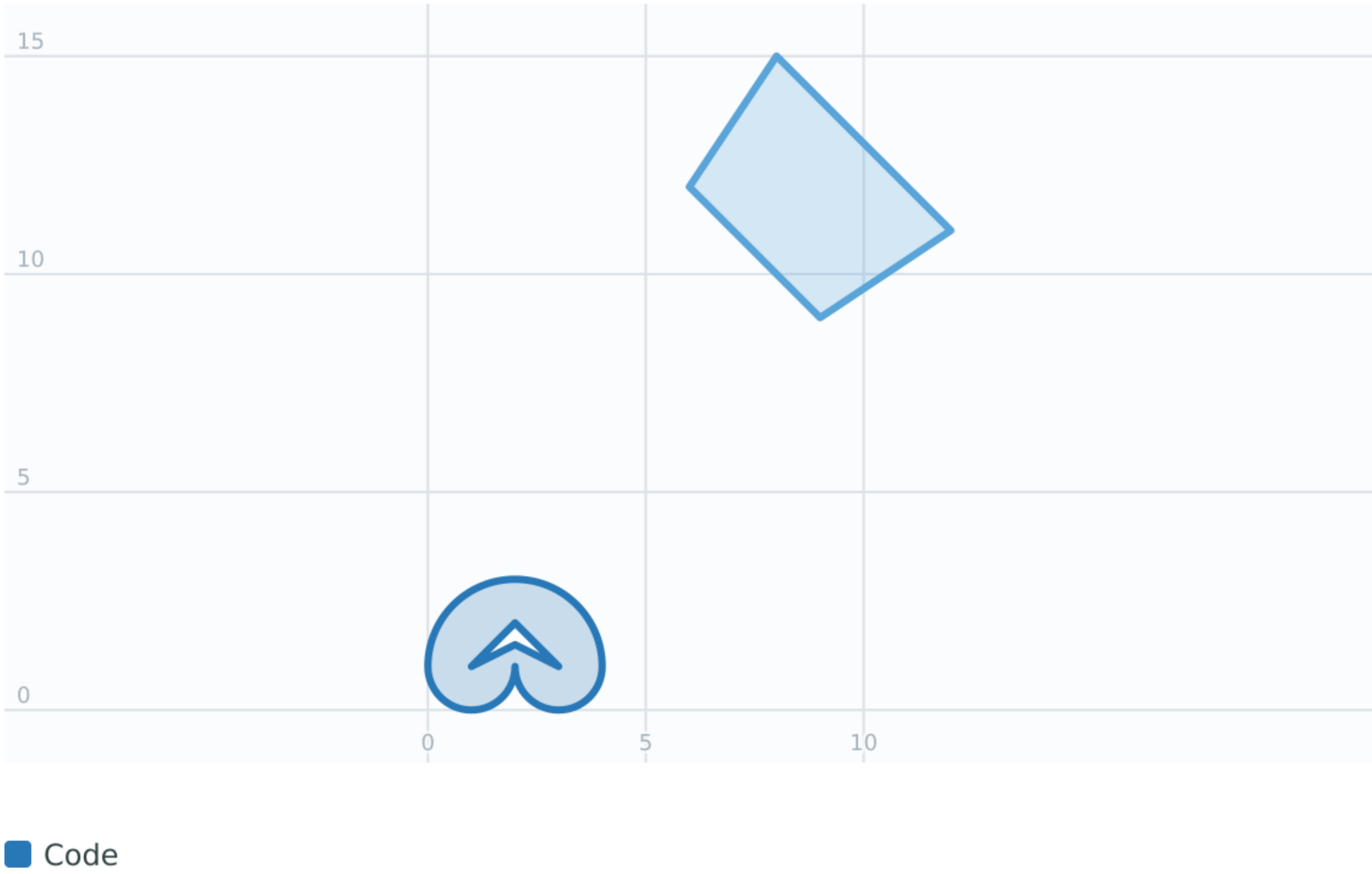
Code

```
SELECT ST_AsSVG('MULTISURFACE(
CURVEPOLYGON(CIRCULARSTRING(-2 0,-1 -1,0 0,1 -1,2 0,0 2,-2 0),
    (-1 0,0 0.5,1 0,0 1,-1 0)),
((7 8,10 10,6 14,4 11,7 8)))'::geometry, 0, 2);
```

Output

```
M -2 0 A 1 1 0 0 0 0 0 A 1 1 0 0 0 2 0 A 2 2 0 0 0 -2 0 Z
M -1 0 L 0 -0.5 1 0 0 -1 -1 0 Z
M 7 -8 L 10 -10 6 -14 4 -11 Z
```

Figure



7.9.3.12 ST_AsTWKB

ST_AsTWKB — □□□ TWKB(Tiny Well-Known Binary) □□□□□□□.

□□

bytea **ST_AsTWKB**(geometry geom, integer prec=0, integer prec_z=0, integer prec_m=0, boolean with_sizes=false, boolean with_boxes=false);
bytea **ST_AsTWKB**(geometry[] geom, bigint[] ids, integer prec=0, integer prec_z=0, integer prec_m=0, boolean with_sizes=false, boolean with_boxes=false);

□□

□□□ TWKB(Tiny Well-Known Binary) □□□□□□□□□. TWKB □□□□□□□□□□□□□□□□□□□ □□□□□□□□□ □□□.

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□. □□□□□□, □□□□□□□□□□□□□□□□□□□□□□□□□□□□□. □□□□□□□□□□□□□□□□□, □□□□□□□□□□. □□□□□, □□ 1 □□□□□□□□□□□□□□□□□□□□□□□□□□□□.

□□□. □□□□□□□□□□□□□□□□□. □□□. □□□□□□□□□□□□□□□□ (□□□□□□□□□□□□□□ TWKB □□□□□□□).

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ TWKB □□□□□□□□□□□□□□□. □□. array_agg □□□□□□□□□□□□□□□□□□□□. □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□.

> **Note!** □□
> https://github.com/TWKB/Specification □□□□□□□□□□□□□□□□□□, https://github.com/TWKB/twkb.js □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□.

Enhanced: 2.4.0 memory and speed improvements.

2.2.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

Code

```
SELECT ST_AsTWKB('LINESTRING(1 1,5 5)'::geometry);
```

Output

```
\x02000202020808
```

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ TWKB ⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠”array_agg()” ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ TWKB ⊠⊠⊠⊠⊠⊠⊠⊠⊠.

Code

```
SELECT ST_AsTWKB(array_agg(geom), array_agg(gid)) FROM mytable;
```

Output

```
\x040402020400000202
```

⊠⊠

ST_GeomFromTWKB, ST_AsBinary, ST_AsEWKB, ST_AsEWKT, ST_GeomFromText

7.9.3.13 ST_AsX3D

ST_AsX3D — ⊠⊠⊠ X3D XML ⊠⊠⊠⊠⊠⊠: ISO-IEC-19776-1.2-X3DEncodings-XML ⊠⊠⊠⊠⊠⊠.

⊠⊠

text **ST_AsX3D**(geometry g1, integer maxdecimaldigits=15, integer options=0);

⊠⊠

⊠⊠⊠ http://www.web3d.org/standards/number/19776-1 ⊠⊠⊠⊠⊠⊠ X3D XML ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. `maxdecimaldigits` (⊠⊠⊠) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ 15 ⊠⊠⊠⊠.

> **⊠⊠**
>
> X3D ⊠⊠⊠⊠⊠ PostGIS ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ PostGIS ⊠⊠⊠ X3D ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ X3D ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠ X3D ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (bug ticket) ⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠ PostGIS 2D/3D ⊠⊠⊠ X3D ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

The ’options’ argument is a bitfield that denotes whether to represent coordinates with the X3D GeoCoordinates geospatial node and whether to flip the x/y axis. By default, `ST_AsX3D` outputs in database form (long,lat or X,Y), but the X3D default of lat/lon, y/x may be preferred.

- 0: ☒☒☒☒☒☒☒☒☒☒☒☒ X/Y(☒☒☒☒☒☒/☒☒ = X,Y ☒☒☒☒☒☒☒☒☒☒☒☒☒☒), ☒☒☒☒, ☒ (非) ☒☒☒☒ (☒☒☒☒☒☒☒☒).
- 1: X ☒ Y ☒☒☒☒☒☒. ☒☒☒☒ (GeoCoordinate) ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒, ☒☒☒☒☒☒☒ ☒”latitude_first”(☒☒☒☒) ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.
- 2: ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒ WGS84 ☒☒☒ (SRID 4326) ☒☒☒☒☒, ☒☒☒☒☒☒☒☒ ☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒☒ X3D ☒☒ ☒☒☒☒☒☒☒. ☒ ☒☒☒☒☒☒☒☒ `GeoCoordinate geoSystem='"GD" "WE" "longitude_first"'` ☒☒☒☒. X3D ☒☒ ☒☒ `GeoCoordinate geoSystem='"GD" "WE" "latitude_first"'` ☒☒☒☒☒☒, `(2 + 1) = 3` ☒☒☒ ☒☒.

PostGIS ☒☒	**2D X3D ☒☒**	**3D X3D ☒☒**
LINESTRING	LineSet (coordinates are promoted to 3D)	LineSet
MULTILINESTRING	IndexedLineSet (coordinates are promoted to 3D)	IndexedLineSet
MULTIPOINT	Polypoint2D	PointSet
POINT	☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.	☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.
(MULTI) POLYGON, POLYHEDRALSURFACE	IndexedFaceSet (coordinates are promoted to 3D; inner rings currently output as another faceset)	IndexedFaceSet (☒☒☒☒☒☒ ☒☒☒☒☒☒☒ (faceset) ☒☒☒ ☒☒☒☒.)
TIN	TriangleSet2D (☒☒☒☒☒☒☒ ☒☒☒☒.)	IndexedTriangleSet
CIRCULARSTRING, COMPOUNDCURVE, CURVEPOLYGON, MULTICURVE, MULTISURFACE	linearized, then emitted with the linear geometry mapping	linearized, then emitted with the linear geometry mapping
NURBSCURVE	NurbsCurve with controlPoint, knot, weight, and order fields	NurbsCurve with controlPoint, knot, weight, and order fields

> ☒☒
> 2 ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒ ☒☒☒☒☒☒☒☒.

M coordinates are not represented in X3D output.

Lots of advancements happening in 3D space particularly with X3D Integration with HTML5

☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒ X3D ☒☒☒☒☒☒☒. ☒, ☒☒☒, ☒☒☒☒ Free Wrl ☒☒☒☒ ☒ http://freewrl.sourceforge.net/ ☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒ FreeWRL_Launcher ☒☒☒☒☒☒☒.

Also check out PostGIS minimalist X3D viewer that utilizes this function and x3dDom html/js open source toolkit.

2.0.0 ☒☒☒☒ ISO-IEC-19776-1.2-X3DEncodings-XML ☒☒☒☒☒☒☒☒☒.

☒☒☒☒: 2.2.0 ☒☒☒☒☒☒☒☒☒☒ (x/y, ☒☒/☒☒) ☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.

Enhanced: 3.7.0: Support for curve geometries. NURBSCURVE is emitted as X3D NurbsCurve; other curve types are linearized before output.

This function supports 3d and will not drop the z-index.

This function supports Polyhedral surfaces.

This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).

□□

Create a complete X3D document containing a cube that is viewable in FreeWRL and other X3D viewers.

Code

```
SELECT '<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE X3D PUBLIC "ISO//Web3D//DTD X3D 3.0//EN" "http://www.web3d.org/specifications/x3d ↩
    -3.0.dtd">
<X3D>
  <Scene>
    <Transform>
      <Shape>
       <Appearance>
            <Material emissiveColor=''0 0 1''/>
       </Appearance
> ' ||
       ST_AsX3D(ST_GeomFromEWKT('POLYHEDRALSURFACE( ((0 0 0, 0 0 1, 0 1 1, 0 1 0, 0 0 0)),
((0 0 0, 0 1 0, 1 1 0, 1 0 0, 0 0 0)), ((0 0 0, 1 0 0, 1 0 1, 0 0 1, 0 0 0)),
((1 1 0, 1 1 1, 1 0 1, 1 0 0, 1 1 0)),
((0 1 0, 0 1 1, 1 1 1, 1 1 0, 0 1 0)), ((0 0 1, 1 0 1, 1 1 1, 0 1 1, 0 0 1)) )')) ||
      '</Shape>
    </Transform>
  </Scene>
</X3D
>' As x3ddoc;
```

Output

```
    x3ddoc
    --------
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE X3D PUBLIC "ISO//Web3D//DTD X3D 3.0//EN" "http://www.web3d.org/specifications/x3d-3.0.dtd">
<X3D>
  <Scene>
    <Transform>
      <Shape>
       <Appearance>
            <Material emissiveColor='0 0 1'/>
       </Appearance>
       <IndexedFaceSet  coordIndex='0 1 2 3 -1 4 5 6 7 -1 8 9 10 11 -1 12 13 14 15 -1 16 17 18 19 -1 20 21 22 23'>
            <Coordinate point='0 0 0 0 0 1 0 1 1 0 1 0 0 0 0 0 1 0 1 1 0 1 0 0 0 0 0 1 0 0 1 0 1 0 0 1 1 1 0 1 1 1 1 0 ↩
     1 1 0 0 0 1 0 0 1 1 1 1 1 1 1 0 0 0 1 1 0 1 1 1 1 0 1 1' />
       </IndexedFaceSet>
      </Shape>
    </Transform>
  </Scene>
</X3D>
```

PostGIS buildings.

Copy and paste the output of this query to x3d scene viewer and click Show

Code

```
SELECT string_agg('<Shape
>' || ST_AsX3D(CG_Extrude(geom, 0,0, i*0.5)) ||
    '<Appearance>
          <Material diffuseColor="' || (0.01*i)::text || ' 0.8 0.2" specularColor="' ||  ↩
    (0.05*i)::text || ' 0 0.5"/>
        </Appearance>
```

```
    </Shape
>', '')
FROM ST_Subdivide(ST_Letters('PostGIS'),20) WITH ORDINALITY AS f(geom,i);
```

The following preview shows the extruded geometry generated from a simplified version of the same source lettering. It is a geometry preview, not X3D markup; use the preceding ST_AsX3D query output in an X3D viewer to inspect the native X3D scene.

Code

```
SELECT
    CG_Extrude(
        ST_SimplifyPreserveTopology(ST_Letters('PostGIS'), 2),
        0, 0, 10
    ) AS buildings;
```

Figure



An Octagon elevated 3 Units and decimal precision of 6.

Code

```
SELECT ST_AsX3D(ST_Translate(ST_Force3D(ST_Buffer(ST_Point(10,10),5, 'quad_segs=2')), 0,0,
    3),
  6) As x3dfrag;
```

Output

```
x3dfrag
--------
<IndexedFaceSet  convex='false' coordIndex='0 1 2 3 4 5 6 7'
><Coordinate point='15 10 3 13.535534 6.464466 3 10 5 3 6.464466 6.464466 3 5 10 3 6.464466 13.535534 3 10 15 3 ↩
     13.535534 13.535534 3 ' /></IndexedFaceSet>
```

TIN.

Code

```
SELECT ST_AsX3D(ST_GeomFromEWKT('TIN (((
                0 0 0,
                0 0 1,
                0 1 0,
                0 0 0
            )), ((
                0 0 0,
                0 1 0,
                1 1 0,
                0 0 0
            ))
            )')) As x3dfrag;
```

Output

```
    x3dfrag
    --------
<IndexedTriangleSet  index='0 1 2 3 4 5'
><Coordinate point='0 0 0 0 0 1 0 1 0 0 0 0 0 1 0 1 1 0'/></IndexedTriangleSet>
```

Closed multilinestring (the boundary of a polygon with holes).

Code

```
SELECT ST_AsX3D(ST_GeomFromEWKT('MULTILINESTRING((20 0 10,16 -12 10,0 -16 10,-12 -12 10,-20 ↩
     0 10,-12 16 10,0 24 10,16 16 10,20 0 10),
  (12 0 10,8 8 10,0 12 10,-8 8 10,-8 0 10,-8 -4 10,0 -8 10,8 -4 10,12 0 10))')
) As x3dfrag;
```

Output

```
    x3dfrag
    --------
<IndexedLineSet  coordIndex='0 1 2 3 4 5 6 7 0 -1 8 9 10 11 12 13 14 15 8'>
    <Coordinate point='20 0 10 16 -12 10 0 -16 10 -12 -12 10 -20 0 10 -12 16 10 0 24 10 16 16 10 12 0 10 8 8 10 0 12  ↩
     10 -8 8 10 -8 0 10 -8 -4 10 0 -8 10 8 -4 10 ' />
 </IndexedLineSet>
```

7.9.3.14 ST_GeoHash

ST_GeoHash — ☒☒☒ GeoHash ☒☒☒☒☒☒☒☒☒.

☒☒

text **ST_GeoHash**(geometry geom, integer maxchars=full_precision_of_point);

☒☒

Computes a GeoHash representation of a geometry. A GeoHash encodes a geographic Point into a text form that is sortable and searchable based on prefixing. A shorter GeoHash is a less precise representation of a point. It can be thought of as a box that contains the point.

Non-point geometry values with non-zero extent can also be mapped to GeoHash codes. The precision of the code depends on the geographic extent of the geometry.

If `maxchars` is not specified, the returned GeoHash code is for the smallest cell containing the input geometry. Points return a GeoHash with 20 characters of precision (about enough to hold the full double precision of the input). Other geometric types may return a GeoHash with less precision,

depending on the extent of the geometry. Larger geometries are represented with less precision, smaller ones with more precision. The box determined by the GeoHash code always contains the input feature.

If `maxchars` is specified the returned GeoHash code has at most that many characters. It maps to a (possibly) lower precision representation of the input geometry. For non-points, the starting point of the calculation is the center of the bounding box of the geometry.

1.4.0 □□□□□□□□□□□□.



□□
ST_GeoHash requires input geometry to be in geographic (lon/lat) coordinates.

 This method supports Circular Strings and Curves.

□□

Code

```
SELECT ST_GeoHash(ST_Point(-126, 48));
```

Output

```
c0w3hf1s70w3hf1s70w3
```

Code

```
SELECT ST_GeoHash(ST_Point(-126, 48), 5);
```

Output

```
c0w3h
```

This example shows that the GeoHash for a line containing a point is a prefix of the point code.

Code

```
SELECT ST_GeoHash('LINESTRING(-126 48,-126.1 48.1)'::geometry);
```

Output

```
c0w3
```

□□

ST_GeomFromGeoHash, ST_PointFromGeoHash, ST_Box2dFromGeoHash

7.10 □□□ (operator)

7.10.1 Bounding Box Operators

7.10.1.1 &&

&& — A □ 2D □□□□□ B □ 2D □□□□□□□□□□□ TRUE □□□□□□.

□□

boolean **&&**(geometry A , geometry B);
boolean **&&**(geography A , geography B);

□□

&& □□□□□□□ A □ 2D □□□□□□□□ B □ 2D □□□□□□□□□□□□ TRUE □□□□□□□.



□□
□□□□□□ (operand) □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□.

□□□□: 2.0.0 □□□□□□□□□□ (polyhedral surface) □□□□□□□.

1.5.0 □□□□□□□□□□□□□□□.

 This method supports Circular Strings and Curves.

 This function supports Polyhedral surfaces.

□□

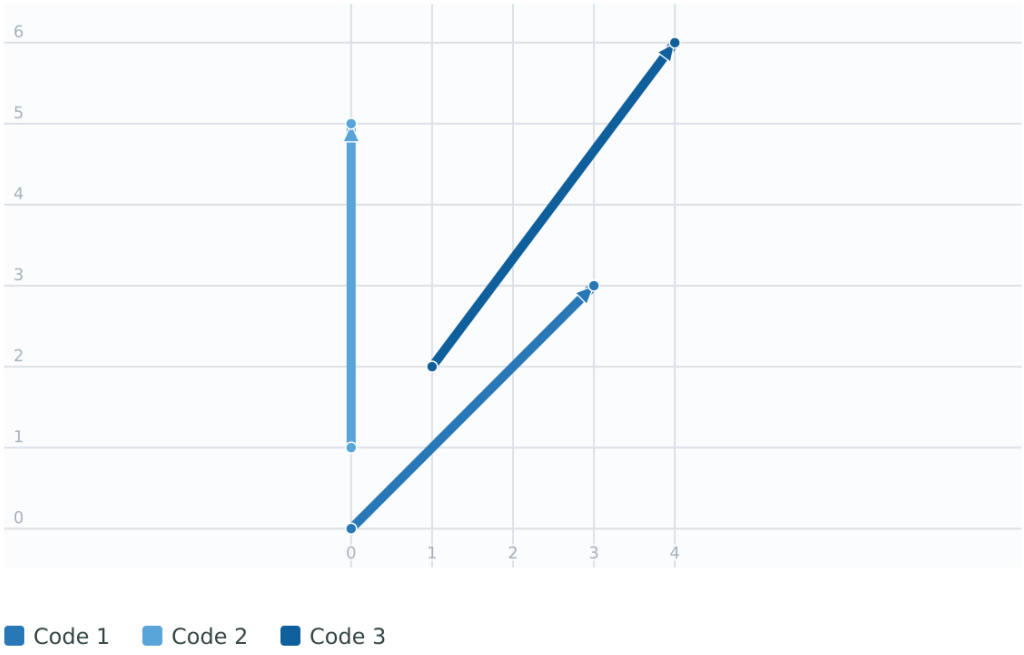
Code

```
SELECT tbl1.column1, tbl2.column1, tbl1.column2 && tbl2.column2 AS overlaps
FROM ( VALUES
    (1, 'LINESTRING(0 0, 3 3)'::geometry),
    (2, 'LINESTRING(0 1, 0 5)'::geometry)) AS tbl1,
( VALUES
    (3, 'LINESTRING(1 2, 4 6)'::geometry)) AS tbl2;
```

Output

```
 column1 | column1 | overlaps
---------+---------+----------
       1 |       3 | t
       2 |       3 | f
(2 rows)
```

Figure



☒☒

ST_Intersects, ST_Extent, |&>, &>, &<|, &<, ~, @

7.10.1.2 &&(geometry,box2df)

&&(geometry,box2df) — Returns TRUE if a geometry's (cached) 2D bounding box intersects a 2D float precision bounding box (BOX2DF).

☒☒

boolean **&&**(geometry A , box2df B);

☒☒

The && operator returns TRUE if the cached 2D bounding box of geometry A intersects the 2D bounding box B, using float precision. This means that if B is a (double precision) box2d, it will be internally converted to a float precision 2D bounding box (BOX2DF)



☒☒
This operand is intended to be used internally by BRIN indexes, more than by users.

Availability: 2.3.0 support for Block Range INdexes (BRIN) was introduced.


This method supports Circular Strings and Curves.


This function supports Polyhedral surfaces.

□□

Code

```
SELECT ST_Point(1,1) && ST_MakeBox2D(ST_Point(0,0), ST_Point(2,2)) AS overlaps;
```

Output

```
 overlaps
----------
 t
(1 row)
```

□□

&&(box2df,geometry), &&(box2df,box2df), ~(geometry,box2df), ~(box2df,geometry), ~(box2df,box2df), @(geometry,box2df), @(box2df,geometry), @(box2df,box2df)

7.10.1.3 &&(box2df,geometry)

&&(box2df,geometry) — Returns TRUE if a 2D float precision bounding box (BOX2DF) intersects a geometry's (cached) 2D bounding box.

□□

boolean **&&**(box2df A , geometry B);

□□

The && operator returns TRUE if the 2D bounding box A intersects the cached 2D bounding box of geometry B, using float precision. This means that if A is a (double precision) box2d, it will be internally converted to a float precision 2D bounding box (BOX2DF)



□□
This operand is intended to be used internally by BRIN indexes, more than by users.

Availability: 2.3.0 support for Block Range INdexes (BRIN) was introduced.

This method supports Circular Strings and Curves.

This function supports Polyhedral surfaces.

□□

Code

```
SELECT ST_MakeBox2D(ST_Point(0,0), ST_Point(2,2)) && ST_Point(1,1) AS overlaps;
```

Output

```
 overlaps
----------
 t
(1 row)
```

☒☒

&&(geometry,box2df), &&(box2df,box2df), ~(geometry,box2df), ~(box2df,geometry), ~(box2df,box2df), @(geometry,box2df), @(box2df,geometry), @(box2df,box2df)

7.10.1.4 &&(box2df,box2df)

&&(box2df,box2df) — Returns TRUE if two 2D float precision bounding boxes (BOX2DF) intersect each other.

☒☒

boolean **&&**(box2df A , box2df B);

☒☒

The && operator returns TRUE if two 2D bounding boxes A and B intersect each other, using float precision. This means that if A (or B) is a (double precision) box2d, it will be internally converted to a float precision 2D bounding box (BOX2DF)



☒☒
This operator is intended to be used internally by BRIN indexes, more than by users.

Availability: 2.3.0 support for Block Range INdexes (BRIN) was introduced.

This method supports Circular Strings and Curves.

This function supports Polyhedral surfaces.

☒☒

Code

```
SELECT ST_MakeBox2D(ST_Point(0,0), ST_Point(2,2)) && ST_MakeBox2D(ST_Point(1,1), ST_Point ↩
    (3,3)) AS overlaps;
```

Output

```
 overlaps
----------
 t
(1 row)
```

□□

&&(geometry,box2df), &&(box2df,geometry), ~(geometry,box2df), ~(box2df,geometry), ~(box2df,box2df), @(geometry,box2df), @(box2df,geometry), @(box2df,box2df)

7.10.1.5 &&&

&&& — A □ n □□□□□□□ B □ n □□□□□□□□□□□□□ TRUE □□□□□□.

□□

boolean **&&&**(geometry A , geometry B);

□□

&&& □□□□□□□ A □ n □□□□□□□□□ B □ n □□□□□□□□□□□□□ TRUE □□□□□□.



□□
□□□□□ (operand) □□□□□□□□□□□□□□□□□□□□□□□□□□□□□.

2.0.0 □□□□□□□□□□□□.

This method supports Circular Strings and Curves.

This function supports Polyhedral surfaces.

This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).

This function supports 3d and will not drop the z-index.

□□

3D LineStrings.

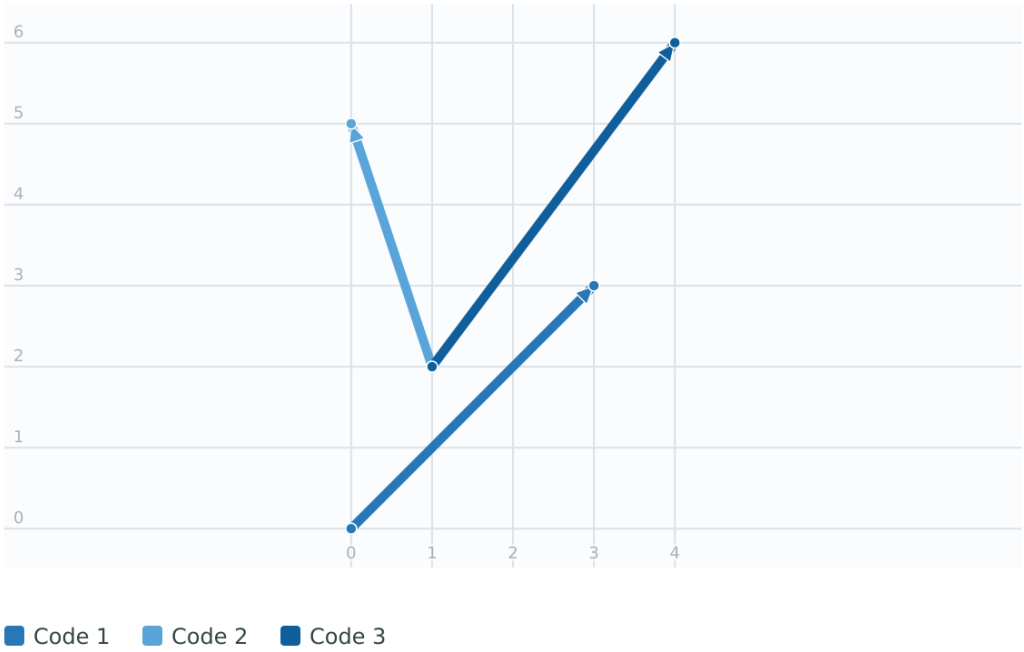
Code

```
SELECT tbl1.column1, tbl2.column1, tbl1.column2 &&& tbl2.column2 AS overlaps_3d,
                       tbl1.column2 && tbl2.column2 AS overlaps_2d
FROM ( VALUES
    (1, 'LINESTRING Z(0 0 1, 3 3 2)'::geometry),
    (2, 'LINESTRING Z(1 2 0, 0 5 -1)'::geometry)) AS tbl1,
( VALUES
    (3, 'LINESTRING Z(1 2 1, 4 6 1)'::geometry)) AS tbl2;
```

Output

```
 column1 | column1 | overlaps_3d | overlaps_2d
---------+---------+-------------+-------------
       1 |       3 | t           | t
       2 |       3 | f           | t
```

Figure



3M LineStrings.

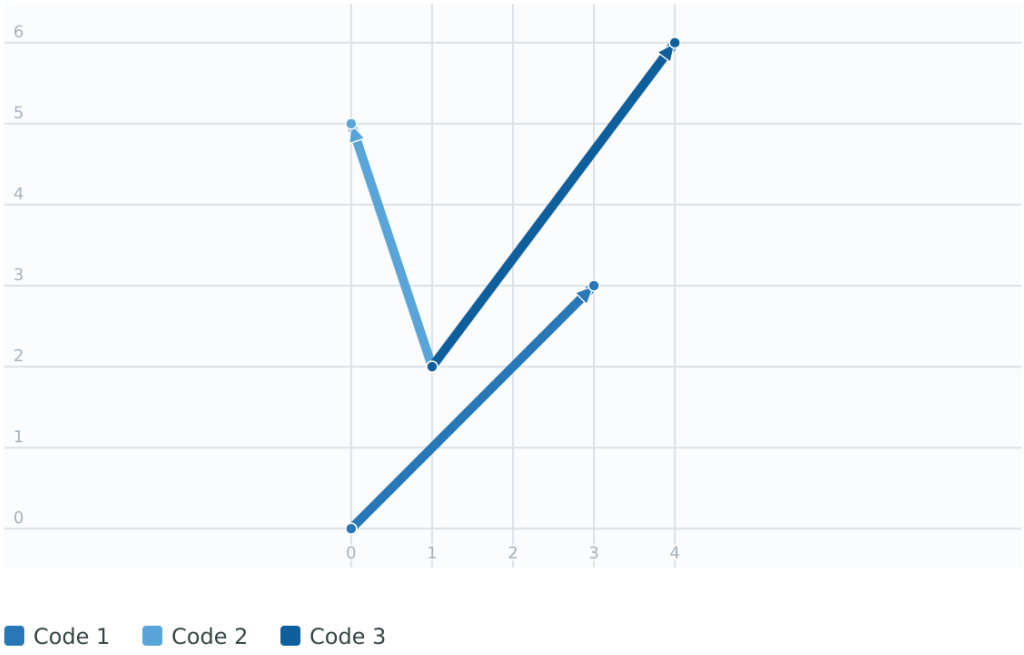
Code

```
SELECT tbl1.column1, tbl2.column1, tbl1.column2 &&& tbl2.column2 AS overlaps_3zm,
                       tbl1.column2 && tbl2.column2 AS overlaps_2d
FROM ( VALUES
    (1, 'LINESTRING M(0 0 1, 3 3 2)'::geometry),
    (2, 'LINESTRING M(1 2 0, 0 5 -1)'::geometry)) AS tbl1,
( VALUES
    (3, 'LINESTRING M(1 2 1, 4 6 1)'::geometry)) AS tbl2;
```

Output

```
 column1 | column1 | overlaps_3zm | overlaps_2d
---------+---------+--------------+-------------
       1 |       3 | t            | t
       2 |       3 | f            | t
```

Figure



□□

&&

7.10.1.6 &&&(geometry,gidx)

&&&(geometry,gidx) — Returns TRUE if a geometry's (cached) n-D bounding box intersects a n-D float precision bounding box (GIDX).

□□

boolean **&&&**(geometry A , gidx B);

□□

The &&& operator returns TRUE if the cached n-D bounding box of geometry A intersects the n-D bounding box B, using float precision. This means that if B is a (double precision) box3d, it will be internally converted to a float precision 3D bounding box (GIDX)



□□
This operator is intended to be used internally by BRIN indexes, more than by users.

Availability: 2.3.0 support for Block Range INdexes (BRIN) was introduced.

 This method supports Circular Strings and Curves.

This function supports Polyhedral surfaces.

This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).

This function supports 3d and will not drop the z-index.

□□

Code

```
SELECT ST_MakePoint(1,1,1) &&& ST_3DMakeBox(ST_MakePoint(0,0,0), ST_MakePoint(2,2,2)) AS ↩
    overlaps;
```

Output

```
 overlaps
----------
 t
(1 row)
```

□□

&&&(gidx,geometry), &&&(gidx,gidx)

7.10.1.7 &&&(gidx,geometry)

&&&(gidx,geometry) — Returns TRUE if a n-D float precision bounding box (GIDX) intersects a geometry's (cached) n-D bounding box.

□□

boolean **&&&**(gidx A , geometry B);

□□

The &&& operator returns TRUE if the n-D bounding box A intersects the cached n-D bounding box of geometry B, using float precision. This means that if A is a (double precision) box3d, it will be internally converted to a float precision 3D bounding box (GIDX)



□□
This operator is intended to be used internally by BRIN indexes, more than by users.

Availability: 2.3.0 support for Block Range INdexes (BRIN) was introduced.

This method supports Circular Strings and Curves.

This function supports Polyhedral surfaces.

This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).

This function supports 3d and will not drop the z-index.

□□

Code

```
SELECT ST_3DMakeBox(ST_MakePoint(0,0,0), ST_MakePoint(2,2,2)) &&& ST_MakePoint(1,1,1) AS
    overlaps;
```

Output

```
 overlaps
----------
 t
(1 row)
```

□□

&&&(geometry,gidx), &&&(gidx,gidx)

7.10.1.8 &&&(gidx,gidx)

&&&(gidx,gidx) — Returns TRUE if two n-D float precision bounding boxes (GIDX) intersect each other.

□□

boolean **&&&**(gidx A , gidx B);

□□

The &&& operator returns TRUE if two n-D bounding boxes A and B intersect each other, using float precision. This means that if A (or B) is a (double precision) box3d, it will be internally converted to a float precision 3D bounding box (GIDX)



□□
This operator is intended to be used internally by BRIN indexes, more than by users.

Availability: 2.3.0 support for Block Range INdexes (BRIN) was introduced.

This method supports Circular Strings and Curves.

This function supports Polyhedral surfaces.

This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).

This function supports 3d and will not drop the z-index.

□□

Code

```
SELECT ST_3DMakeBox(ST_MakePoint(0,0,0), ST_MakePoint(2,2,2)) &&& ST_3DMakeBox(ST_MakePoint ↩
    (1,1,1), ST_MakePoint(3,3,3)) AS overlaps;
```

Output

```
 overlaps
----------
 t
(1 row)
```

□□

&&&(geometry,gidx), &&&(gidx,geometry)

7.10.1.9 &<

&< — A □□□□□□ B □□□□□□□□□□□□□□□□□□□□ TRUE □□□□□□.

□□

boolean **&<**(geometry A , geometry B);

□□

&< □□□□□□ A □□□□□□□□ B □□□□□□□□□□□□□□□□□□□□, □□□□□□□□□□□□ B □ □□□□□□□□□□□□□□□□□□□□□□□□, TRUE □□□□□□.

Note! □□
□□□□□ (operand) □□□□□□□□□□□□□□□□□□□□□□□□□□□□.

□□

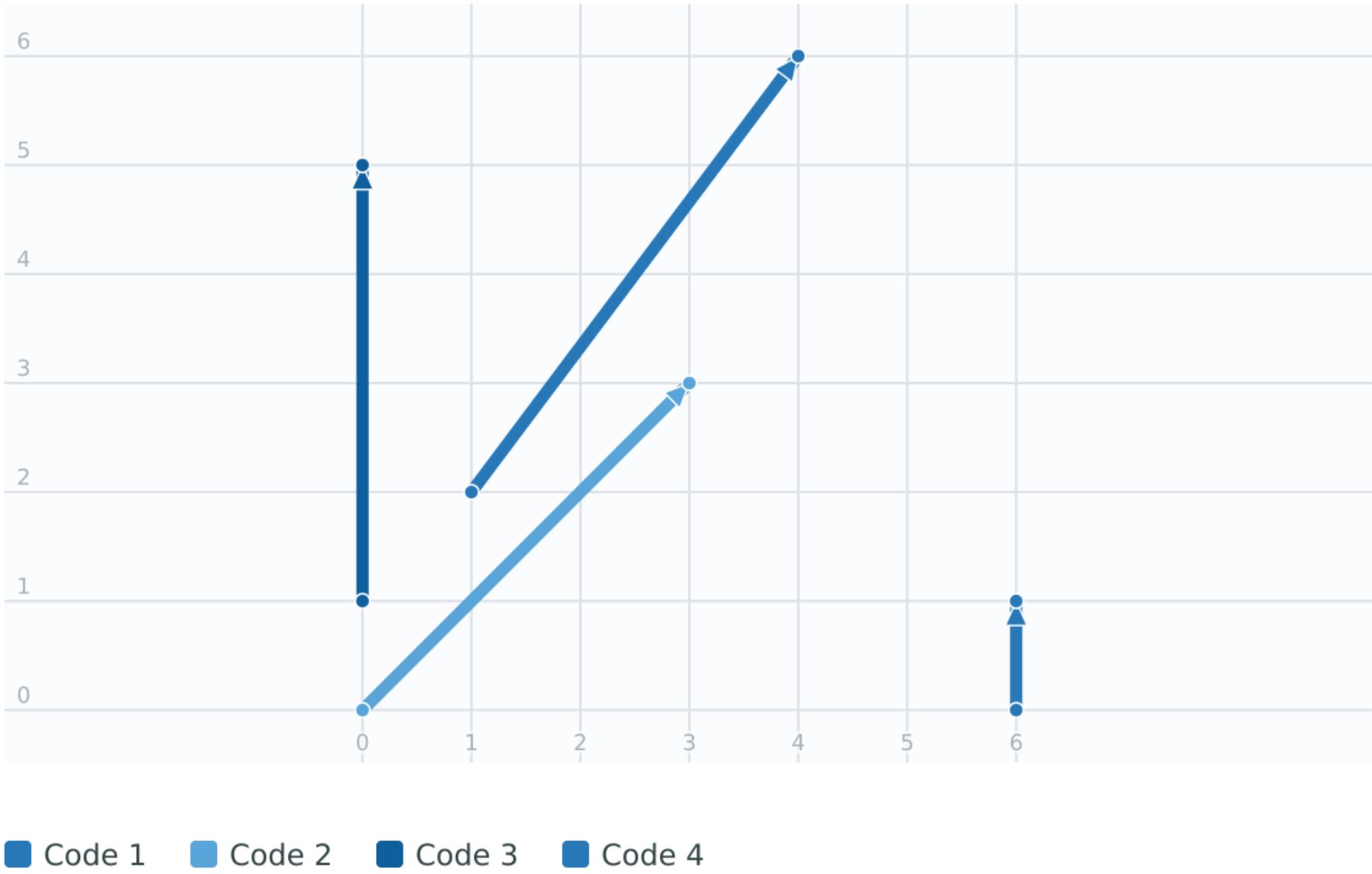
Code

```
SELECT tbl1.column1, tbl2.column1, tbl1.column2 &< tbl2.column2 AS overleft
FROM
  ( VALUES
    (1, 'LINESTRING(1 2, 4 6)'::geometry)) AS tbl1,
  ( VALUES
    (2, 'LINESTRING(0 0, 3 3)'::geometry),
    (3, 'LINESTRING(0 1, 0 5)'::geometry),
    (4, 'LINESTRING(6 0, 6 1)'::geometry)) AS tbl2;
```

Output

```
 column1 | column1 | overleft
---------+---------+----------
       1 |       2 | f
       1 |       3 | f
       1 |       4 | t
(3 rows)
```

Figure



☒☒

&&, |&>, &>, &<|

7.10.1.10 &<|

&<| — A ☒☒☒☒☒☒ B ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒ TRUE ☒☒☒☒☒☒.

☒☒

boolean **&<|**(geometry A , geometry B);

☒☒

&<| ☒☒☒☒☒☒ A ☒☒☒☒☒☒☒☒ B ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒, ☒☒☒☒☒☒☒☒☒☒☒☒ B ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒, TRUE ☒☒☒☒☒☒.

This method supports Circular Strings and Curves.

This function supports Polyhedral surfaces.

Note!

☒☒
☒☒☒☒☒ (operand) ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.

☒☒

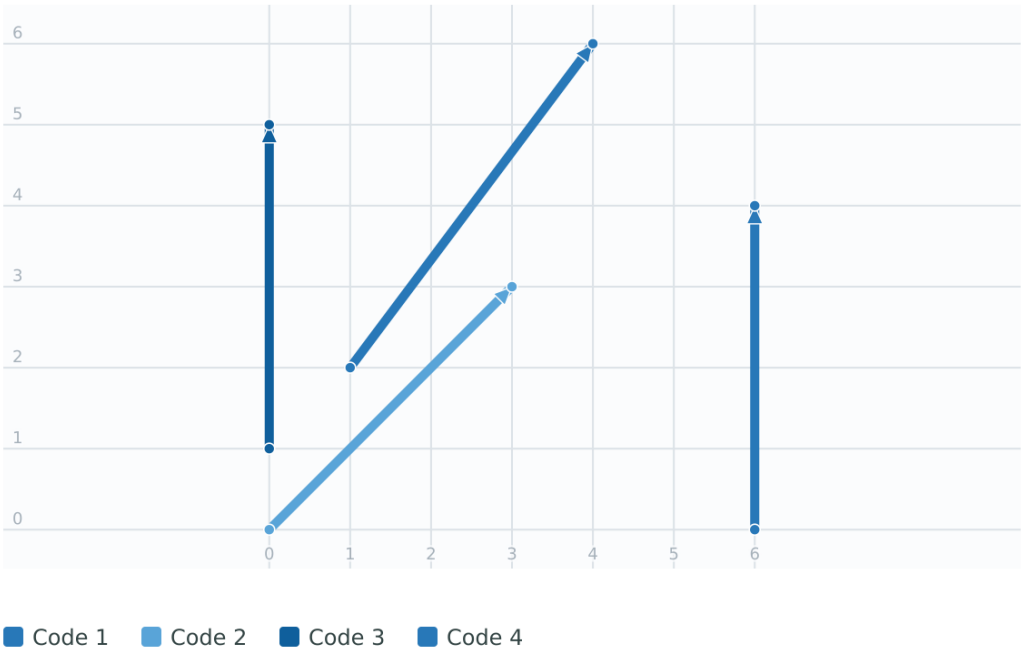
Code

```
SELECT tbl1.column1, tbl2.column1, tbl1.column2 &<| tbl2.column2 AS overbelow
FROM
  ( VALUES
    (1, 'LINESTRING(6 0, 6 4)'::geometry)) AS tbl1,
  ( VALUES
    (2, 'LINESTRING(0 0, 3 3)'::geometry),
    (3, 'LINESTRING(0 1, 0 5)'::geometry),
    (4, 'LINESTRING(1 2, 4 6)'::geometry)) AS tbl2;
```

Output

```
 column1 | column1 | overbelow
---------+---------+-----------
       1 |       2 | f
       1 |       3 | t
       1 |       4 | t
(3 rows)
```

Figure



□□

&&, |&>, &>, &<

7.10.1.11 &>

&> — A □□□□□□ B □□□□□□□□□□□□□□□□□□□□□ TRUE □□□□□□.

□□

boolean **&>**(geometry A , geometry B);

☒☒

&> ☒☒☒☒☒☒ A ☒☒☒☒☒☒☒☒ B ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒, ☒☒☒☒☒☒☒☒☒☒☒☒ B ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒, TRUE ☒☒☒☒☒☒.

> **Note!**
> ☒☒
> ☒☒☒☒☒ (operand) ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.

☒☒

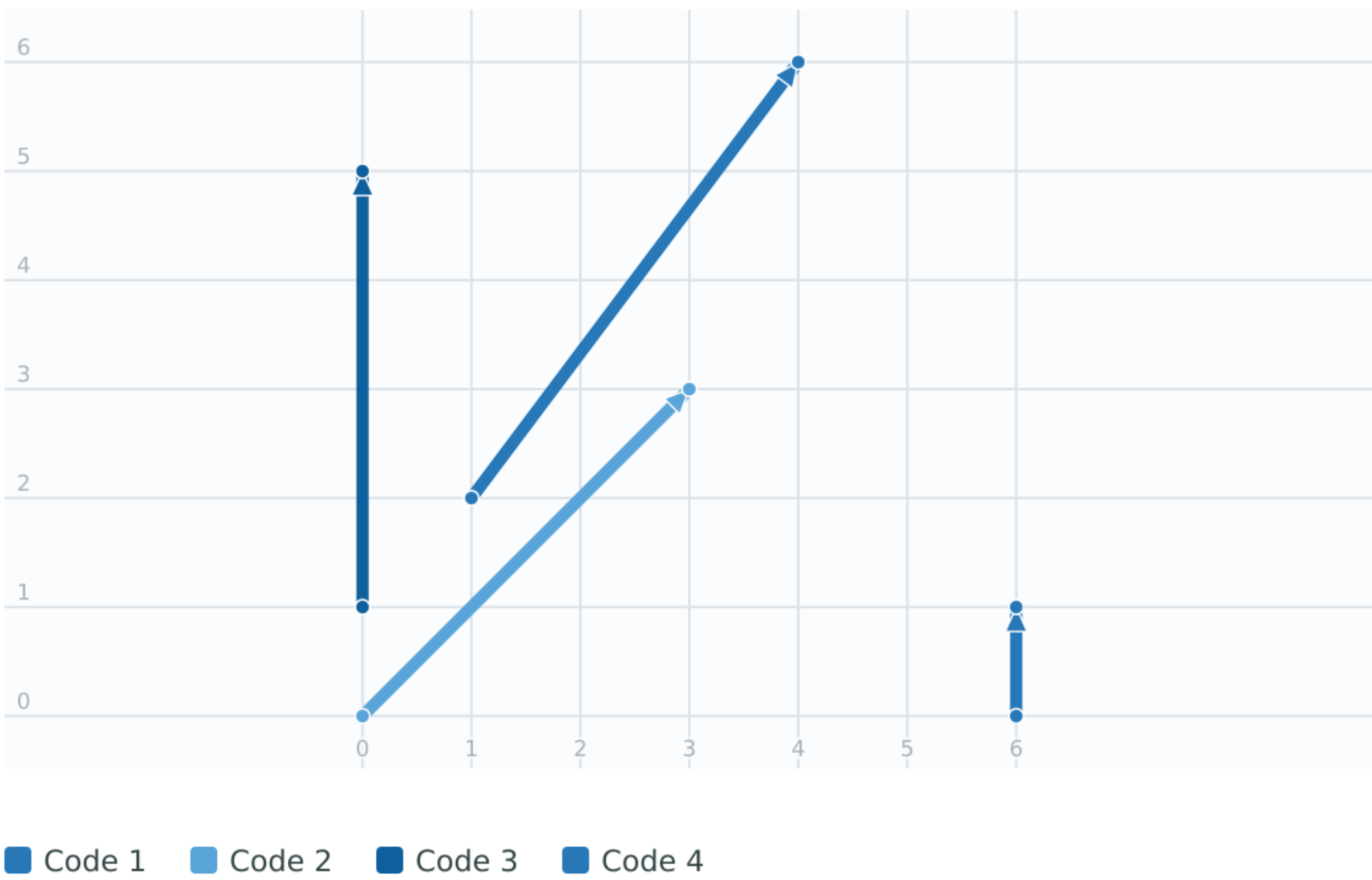
Code

```
SELECT tbl1.column1, tbl2.column1, tbl1.column2 &
> tbl2.column2 AS overright
FROM
  ( VALUES
    (1, 'LINESTRING(1 2,4 6)'::geometry)) AS tbl1,
  ( VALUES
    (2, 'LINESTRING(0 0,3 3)'::geometry),
    (3, 'LINESTRING(0 1,0 5)'::geometry),
    (4, 'LINESTRING(6 0,6 1)'::geometry)) AS tbl2;
```

Output

```
 column1 | column1 | overright
---------+---------+-----------
       1 |       2 | t
       1 |       3 | t
       1 |       4 | f
(3 rows)
```

Figure



⊠⊠

&&, |&>, &<|, &<

7.10.1.12 <<

<< — A ⊠⊠⊠⊠⊠⊠⊠⊠⊠ B ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ TRUE ⊠⊠⊠⊠⊠⊠.

⊠⊠

boolean **<<**(geometry A , geometry B);

⊠⊠

<< ⊠⊠⊠⊠⊠⊠ A ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ B ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ TRUE ⊠⊠⊠⊠⊠⊠.

> **Note!**
> ⊠⊠
> ⊠⊠⊠⊠⊠ (operand) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

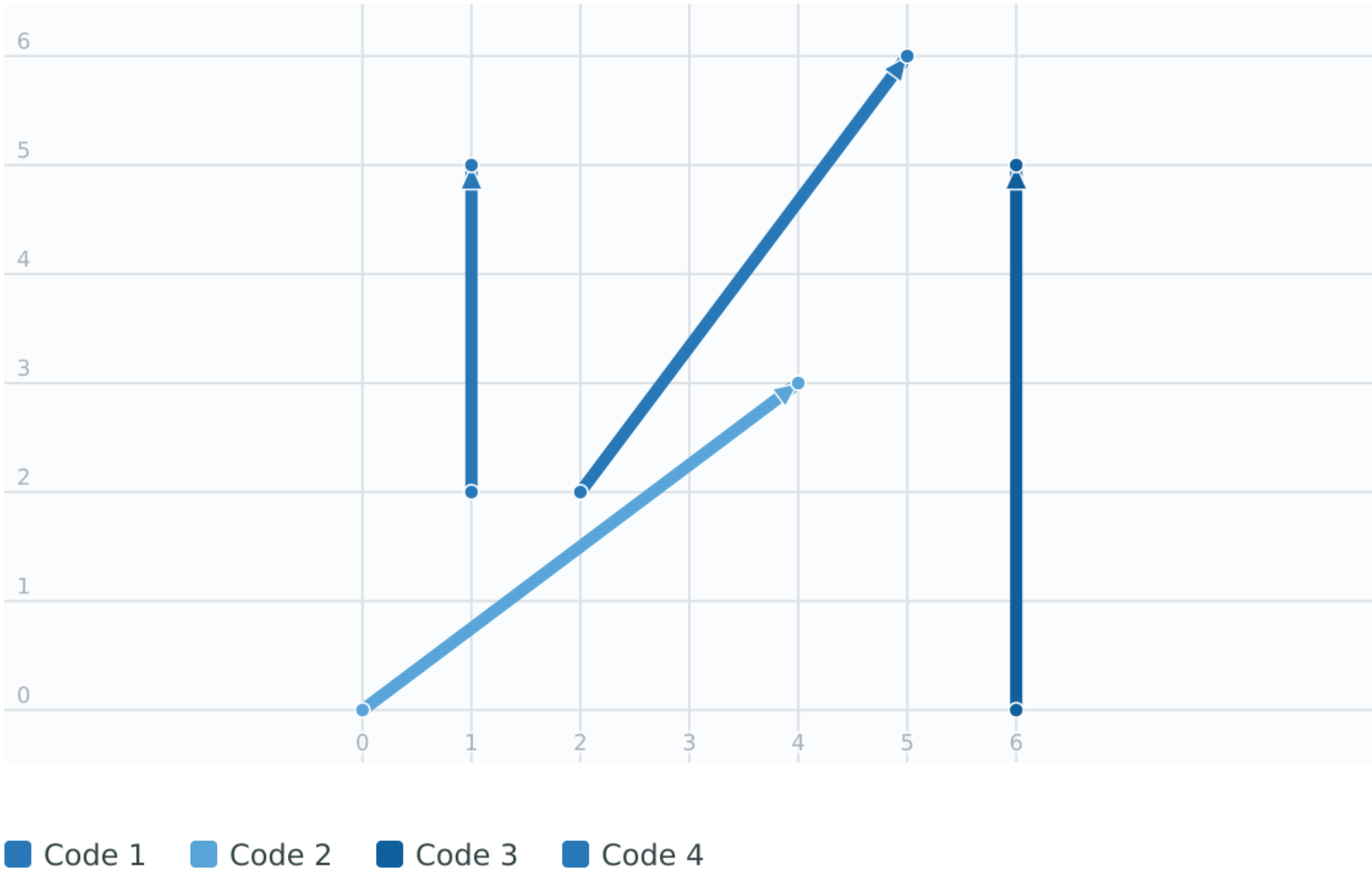
Code

```
SELECT tbl1.column1, tbl2.column1, tbl1.column2 << tbl2.column2 AS left
FROM
  ( VALUES
    (1, 'LINESTRING (1 2, 1 5)'::geometry)) AS tbl1,
  ( VALUES
    (2, 'LINESTRING (0 0, 4 3)'::geometry),
    (3, 'LINESTRING (6 0, 6 5)'::geometry),
    (4, 'LINESTRING (2 2, 5 6)'::geometry)) AS tbl2;
```

Output

```
 column1 | column1 | left
---------+---------+------
       1 |       2 | f
       1 |       3 | t
       1 |       4 | t
(3 rows)
```

Figure



⊠⊠

>>, |>>, <<|

7.10.1.13 <<|

<<| — A ⊠⊠⊠⊠⊠⊠⊠⊠⊠ B ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ TRUE ⊠⊠⊠⊠⊠⊠.

⊠⊠

boolean **<<|**(geometry A , geometry B);

⊠⊠

<<| ⊠⊠⊠⊠⊠⊠ A ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ B ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ TRUE ⊠⊠⊠⊠⊠⊠.

> ⊠⊠
> ⊠⊠⊠⊠⊠ (operand) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

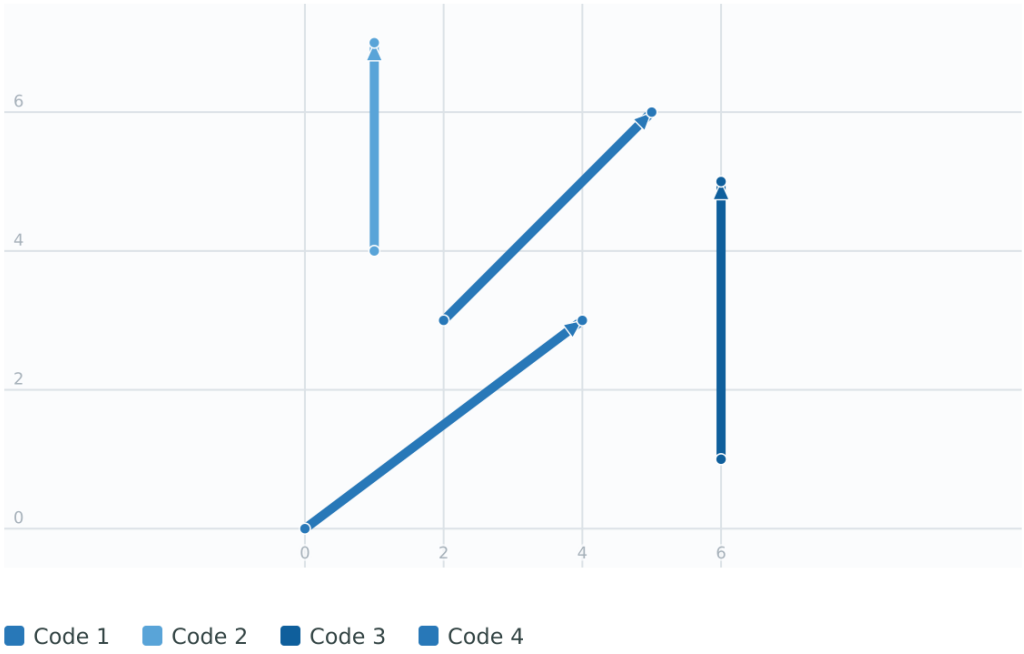
Code

```
SELECT tbl1.column1, tbl2.column1, tbl1.column2 <<| tbl2.column2 AS below
FROM
  ( VALUES
    (1, 'LINESTRING (0 0, 4 3)'::geometry)) AS tbl1,
  ( VALUES
    (2, 'LINESTRING (1 4, 1 7)'::geometry),
    (3, 'LINESTRING (6 1, 6 5)'::geometry),
    (4, 'LINESTRING (2 3, 5 6)'::geometry)) AS tbl2;
```

Output

```
 column1 | column1 | below
---------+---------+-------
       1 |       2 | t
       1 |       3 | f
       1 |       4 | f
(3 rows)
```

Figure



☒☒

<<, >>, |>>

7.10.1.14 =

= — Returns TRUE if the coordinates and coordinate order geometry/geography A are the same as the coordinates and coordinate order of geometry/geography B.

☒☒

boolean =(geometry A , geometry B);
boolean =(geography A , geography B);

□□

The = operator returns TRUE if the coordinates and coordinate order geometry/geography A are the same as the coordinates and coordinate order of geometry/geography B. PostgreSQL uses the =, <, and > operators defined for geometries to perform internal orderings and comparison of geometries (ie. in a GROUP BY or ORDER BY clause).



□□
Only geometry/geography that are exactly equal in all respects, with the same coordinates, in the same order, are considered equal by this operator. For "spatial equality", that ignores things like coordinate order, and can detect features that cover the same spatial area with different representations, use ST_OrderingEquals or ST_Equals



□□
This operand will NOT make use of any indexes that may be available on the geometries. For an index assisted exact equality test, combine = with &&.

Changed: 2.4.0, in prior versions this was bounding box equality not a geometric equality. If you need bounding box equality, use ~= instead.

This method supports Circular Strings and Curves.

This function supports Polyhedral surfaces.

□□

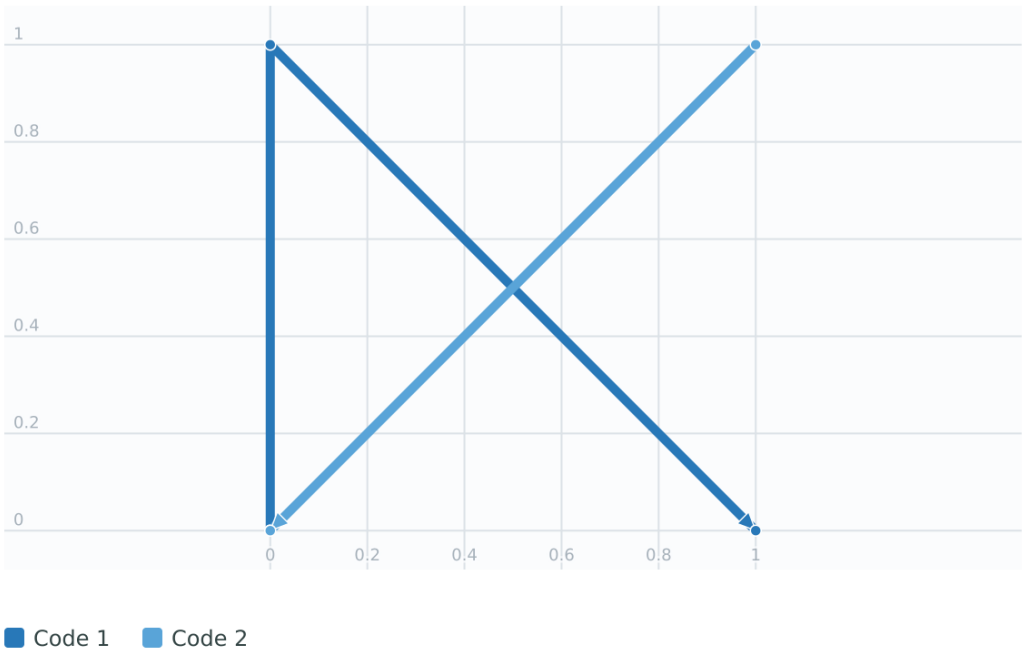
Code

```
SELECT 'LINESTRING(0 0,0 1,1 0)'::geometry = 'LINESTRING(1 1,0 0)'::geometry;
```

Output

```
 ?column?
----------
 f
(1 row)
```

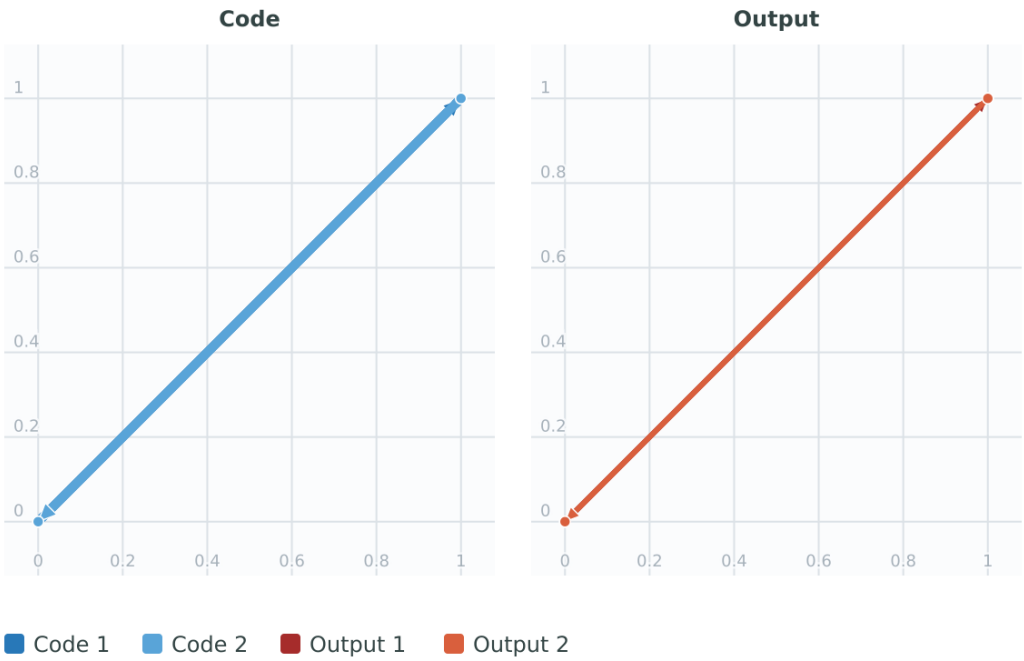
Figure



Code

```
SELECT column1
FROM ( VALUES
    ('LINESTRING(0 0, 1 1)'::geometry),
    ('LINESTRING(1 1, 0 0)'::geometry)) AS foo;
```

Figure

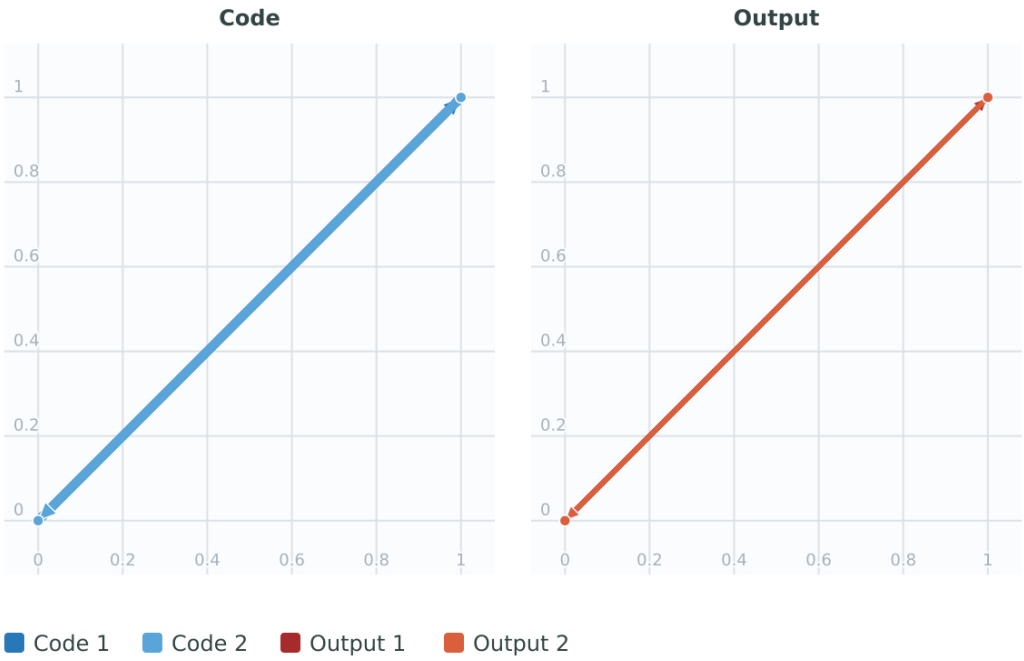


Note: the GROUP BY uses the ”=” to compare for geometry equivalency.

Code

```
SELECT column1
FROM ( VALUES
    ('LINESTRING(0 0, 1 1)'::geometry),
    ('LINESTRING(1 1, 0 0)'::geometry)) AS foo
GROUP BY column1;
```

Figure



In versions prior to 2.0, this used to return true.

Code

```
SELECT 'POINT(1707296.37 4820536.77)'::geometry =
   'POINT(1707296.27 4820536.87)'::geometry As pt_intersect;
```

Output

```
 pt_intersect
--------------
 f
(1 row)
```

☒☒

ST_Equals, ST_OrderingEquals, ~=

7.10.1.15 >>

>> — A ☒☒☒☒☒☒☒☒☒ B ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒ TRUE ☒☒☒☒☒☒.

☒☒

boolean **>>**(geometry A , geometry B);

□□

>> □□□□□□ A □□□□□□□□□□□□ B □□□□□□□□□□□□□□□□ TRUE □□□□□□.



□□
□□□□□ (operand) □□□□□□□□□□□□□□□□□□□□□□□□□□□□.

□□

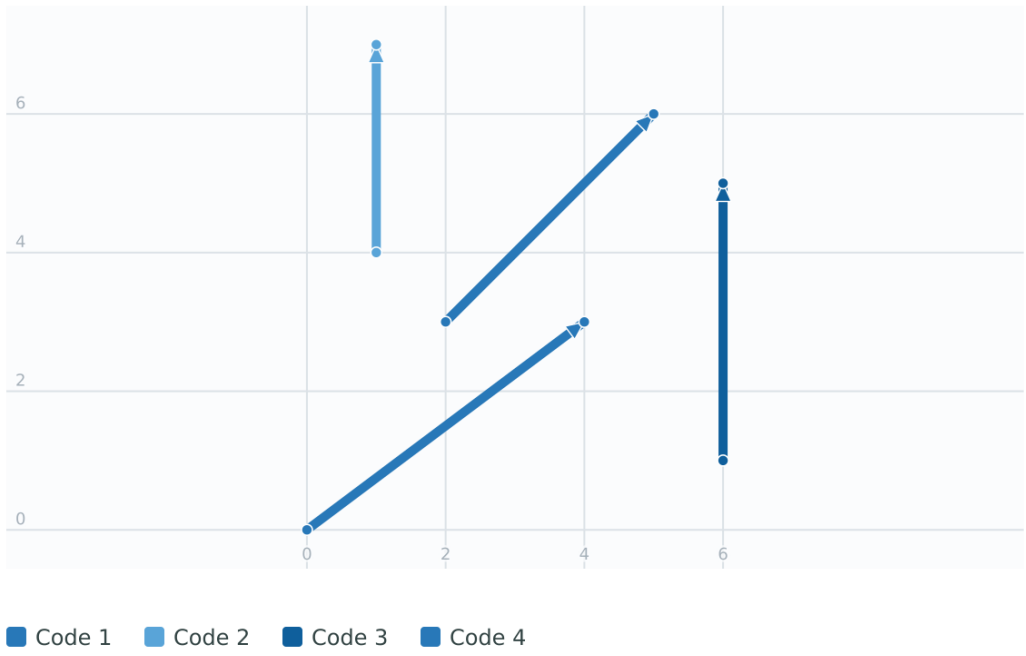
Code

```
SELECT tbl1.column1, tbl2.column1, tbl1.column2
>
> tbl2.column2 AS right
FROM
  ( VALUES
    (1, 'LINESTRING (2 3,5 6)'::geometry)) AS tbl1,
  ( VALUES
    (2, 'LINESTRING (1 4,1 7)'::geometry),
    (3, 'LINESTRING (6 1,6 5)'::geometry),
    (4, 'LINESTRING (0 0,4 3)'::geometry)) AS tbl2;
```

Output

```
 column1 | column1 | right
---------+---------+-------
       1 |       2 | t
       1 |       3 | f
       1 |       4 | f
(3 rows)
```

Figure



□□

<<, |>>, <<|

7.10.1.16 @

@ — B □□□□□□ A □□□□□□□□□□□□ TRUE □□□□□□.

□□

boolean @(geometry A , geometry B);

□□

@ □□□□□□ B □□□□□□□□ A □□□□□□□□□□□□□□ TRUE □□□□□□.

Note! □□
□□□□□ (operand) □□□□□□□□□□□□□□□□□□□□□□□□□□□□.

□□

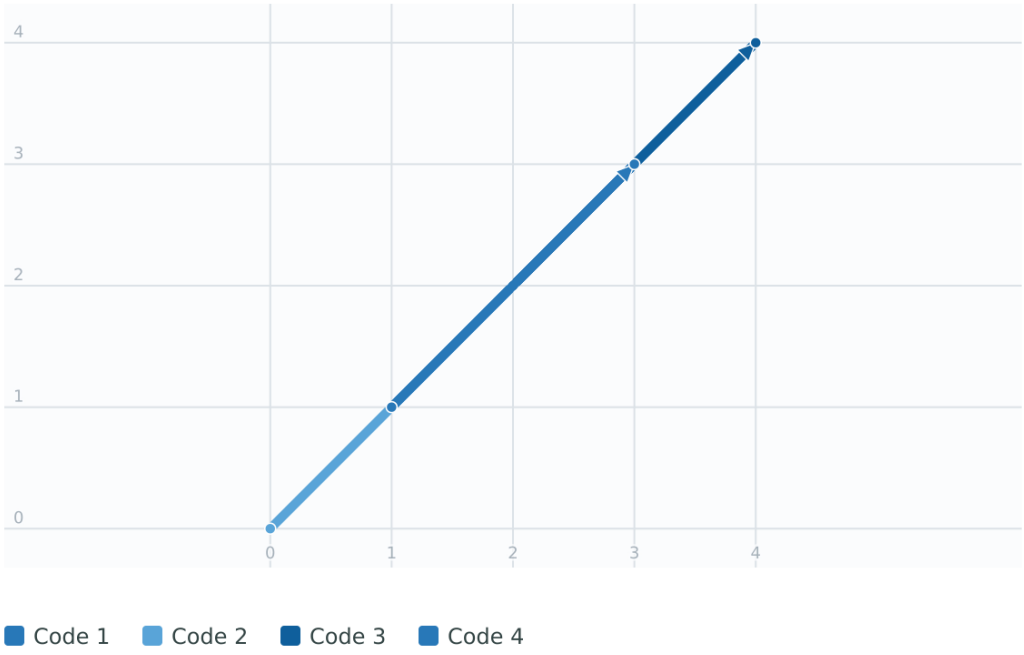
Code

```
SELECT tbl1.column1, tbl2.column1, tbl1.column2 @ tbl2.column2 AS contained
FROM
  ( VALUES
    (1, 'LINESTRING (1 1, 3 3)'::geometry)) AS tbl1,
  ( VALUES
    (2, 'LINESTRING (0 0, 4 4)'::geometry),
    (3, 'LINESTRING (2 2, 4 4)'::geometry),
    (4, 'LINESTRING (1 1, 3 3)'::geometry)) AS tbl2;
```

Output

```
 column1 | column1 | contained
---------+---------+-----------
       1 |       2 | t
       1 |       3 | f
       1 |       4 | t
(3 rows)
```

Figure



□□

~, &&

7.10.1.17 @(geometry,box2df)

@(geometry,box2df) — Returns TRUE if a geometry's 2D bounding box is contained into a 2D float precision bounding box (BOX2DF).

□□

boolean @(geometry A , box2df B);

□□

The @ operator returns TRUE if the A geometry's 2D bounding box is contained the 2D bounding box B, using float precision. This means that if B is a (double precision) box2d, it will be internally converted to a float precision 2D bounding box (BOX2DF)



□□
This operand is intended to be used internally by BRIN indexes, more than by users.

Availability: 2.3.0 support for Block Range INdexes (BRIN) was introduced.


This method supports Circular Strings and Curves.


This function supports Polyhedral surfaces.

□□

Code

```
SELECT ST_Buffer('POINT(2 2)'::geometry, 1) @ ST_MakeBox2D(ST_Point(0, 0), ST_Point(5, 5))  ↩
    AS is_contained;
```

Output

```
t
```

□□

&&(geometry,box2df), &&(box2df,geometry), &&(box2df,box2df), ~(geometry,box2df), ~(box2df,geometry)
~(box2df,box2df), @(box2df,geometry), @(box2df,box2df)

7.10.1.18 @(box2df,geometry)

@(box2df,geometry) — Returns TRUE if a 2D float precision bounding box (BOX2DF) is contained into a geometry's 2D bounding box.

□□

boolean @(box2df A , geometry B);

□□

The @ operator returns TRUE if the 2D bounding box A is contained into the B geometry's 2D bounding box, using float precision. This means that if B is a (double precision) box2d, it will be internally converted to a float precision 2D bounding box (BOX2DF)



□□
This operand is intended to be used internally by BRIN indexes, more than by users.

Availability: 2.3.0 support for Block Range INdexes (BRIN) was introduced.

This method supports Circular Strings and Curves.

This function supports Polyhedral surfaces.

□□

Code

```
SELECT ST_MakeBox2D(ST_Point(2, 2), ST_Point(3, 3)) @ ST_Buffer('POINT(1 1)'::geometry, 10) ↩
    AS is_contained;
```

Output

```
t
```

□□

&&(geometry,box2df), &&(box2df,geometry), &&(box2df,box2df), ~(geometry,box2df), ~(box2df,geometry)
~(box2df,box2df), @(geometry,box2df), @(box2df,box2df)

7.10.1.19 @(box2df,box2df)

@(box2df,box2df) — Returns TRUE if a 2D float precision bounding box (BOX2DF) is contained into another 2D float precision bounding box.

□□

boolean @(box2df A , box2df B);

□□

The @ operator returns TRUE if the 2D bounding box A is contained into the 2D bounding box B, using float precision. This means that if A (or B) is a (double precision) box2d, it will be internally converted to a float precision 2D bounding box (BOX2DF)



□□
This operand is intended to be used internally by BRIN indexes, more than by users.

Availability: 2.3.0 support for Block Range INdexes (BRIN) was introduced.

This method supports Circular Strings and Curves.

This function supports Polyhedral surfaces.

□□

Code

```
SELECT ST_MakeBox2D(ST_Point(2, 2), ST_Point(3, 3)) @ ST_MakeBox2D(ST_Point(0, 0), ST_Point ↩
    (5, 5)) AS is_contained;
```

Output

```
t
```

□□

&&(geometry,box2df), &&(box2df,geometry), &&(box2df,box2df), ~(geometry,box2df), ~(box2df,geometry)
~(box2df,box2df), @(geometry,box2df), @(box2df,geometry)

7.10.1.20 |&>

|&> — A ⊠⊠⊠⊠⊠⊠ B ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ TRUE ⊠⊠⊠⊠⊠⊠.

⊠⊠

boolean **|&>**(geometry A , geometry B);

⊠⊠

|&> ⊠⊠⊠⊠⊠⊠⊠ A ⊠⊠⊠⊠⊠⊠⊠⊠⊠ B ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ B ⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, TRUE ⊠⊠⊠⊠⊠⊠⊠.

> **⊠⊠**
> ⊠⊠⊠⊠⊠ (operand) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

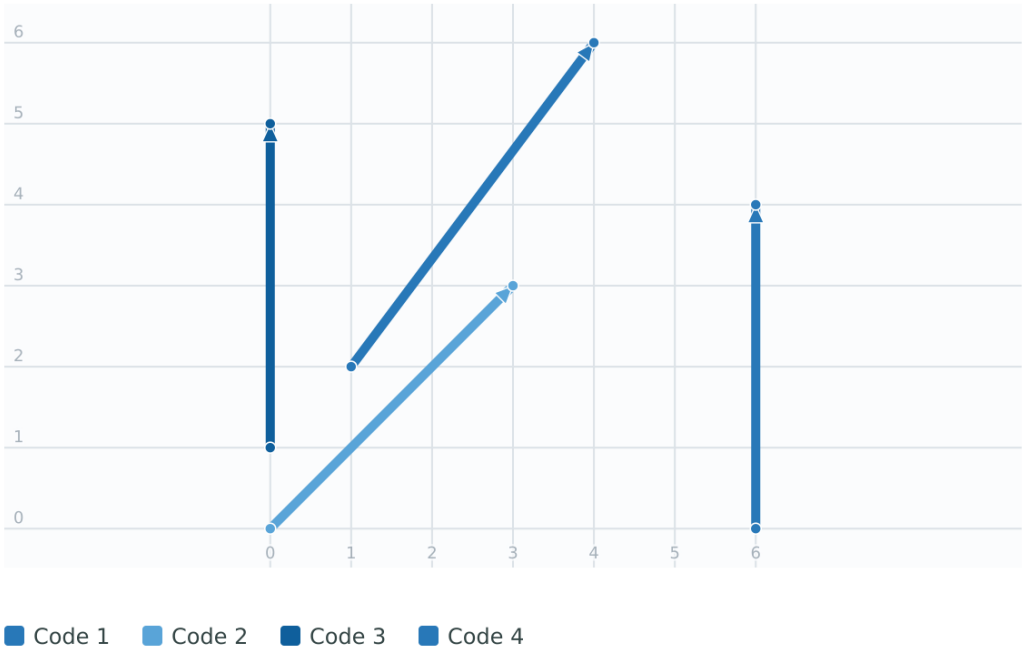
Code

```
SELECT tbl1.column1, tbl2.column1, tbl1.column2 |&
> tbl2.column2 AS overabove
FROM
  ( VALUES
    (1, 'LINESTRING(6 0,6 4)'::geometry)) AS tbl1,
  ( VALUES
    (2, 'LINESTRING(0 0,3 3)'::geometry),
    (3, 'LINESTRING(0 1,0 5)'::geometry),
    (4, 'LINESTRING(1 2,4 6)'::geometry)) AS tbl2;
```

Output

```
 column1 | column1 | overabove
---------+---------+-----------
       1 |       2 | t
       1 |       3 | f
       1 |       4 | f
(3 rows)
```

Figure



☒☒

&&, &>, &<|, &<

7.10.1.21 |>>

|>> — A ☒☒☒☒☒☒☒☒☒ B ☒☒☒☒☒☒☒☒☒☒☒☒☒ TRUE ☒☒☒☒☒☒.

☒☒

boolean **|>>**(geometry A , geometry B);

☒☒

The |>> operator returns TRUE if the bounding box of geometry A is strictly above the bounding box of geometry B.

Note! ☒☒
☒☒☒☒☒ (operand) ☒☒☒☒☒☒☒☒☒☒☒☒☒

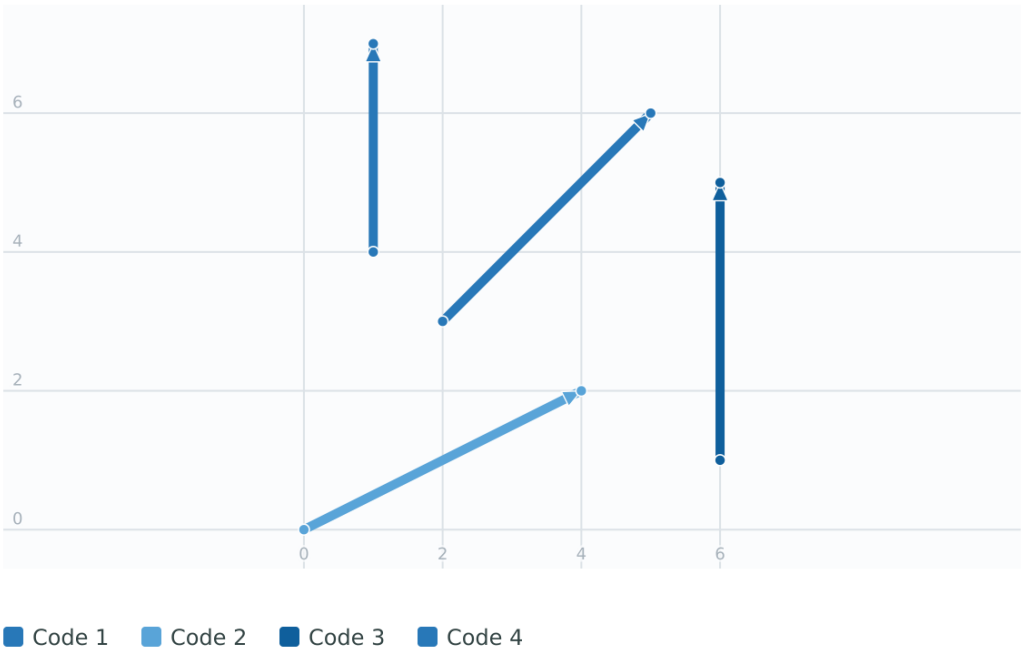
Code

```
SELECT tbl1.column1, tbl2.column1, tbl1.column2 |>> tbl2.column2 AS above
FROM
  ( VALUES
    (1, 'LINESTRING (1 4,1 7)'::geometry)) AS tbl1,
  ( VALUES
    (2, 'LINESTRING (0 0,4 2)'::geometry),
    (3, 'LINESTRING (6 1,6 5)'::geometry),
    (4, 'LINESTRING (2 3,5 6)'::geometry)) AS tbl2;
```

Output

```
column1 | column1 | above
---------+---------+-------
       1 |       2 | t
       1 |       3 | f
       1 |       4 | f
(3 rows)
```

Figure



☒☒

<<, >>, <<|

7.10.1.22 ~

~ — A ☒☒☒☒☒☒ B ☒☒☒☒☒☒☒☒☒☒☒☒ TRUE ☒☒☒☒☒☒.

☒☒

boolean ~(geometry A , geometry B);

□□

~ □□□□□□□ A □□□□□□□□□ B □□□□□□□□□□□□□□□ TRUE □□□□□□.

> **□□**
> □□□□□ (operand) □□□□□□□□□□□□□□□□□□□□□□□□□.

□□

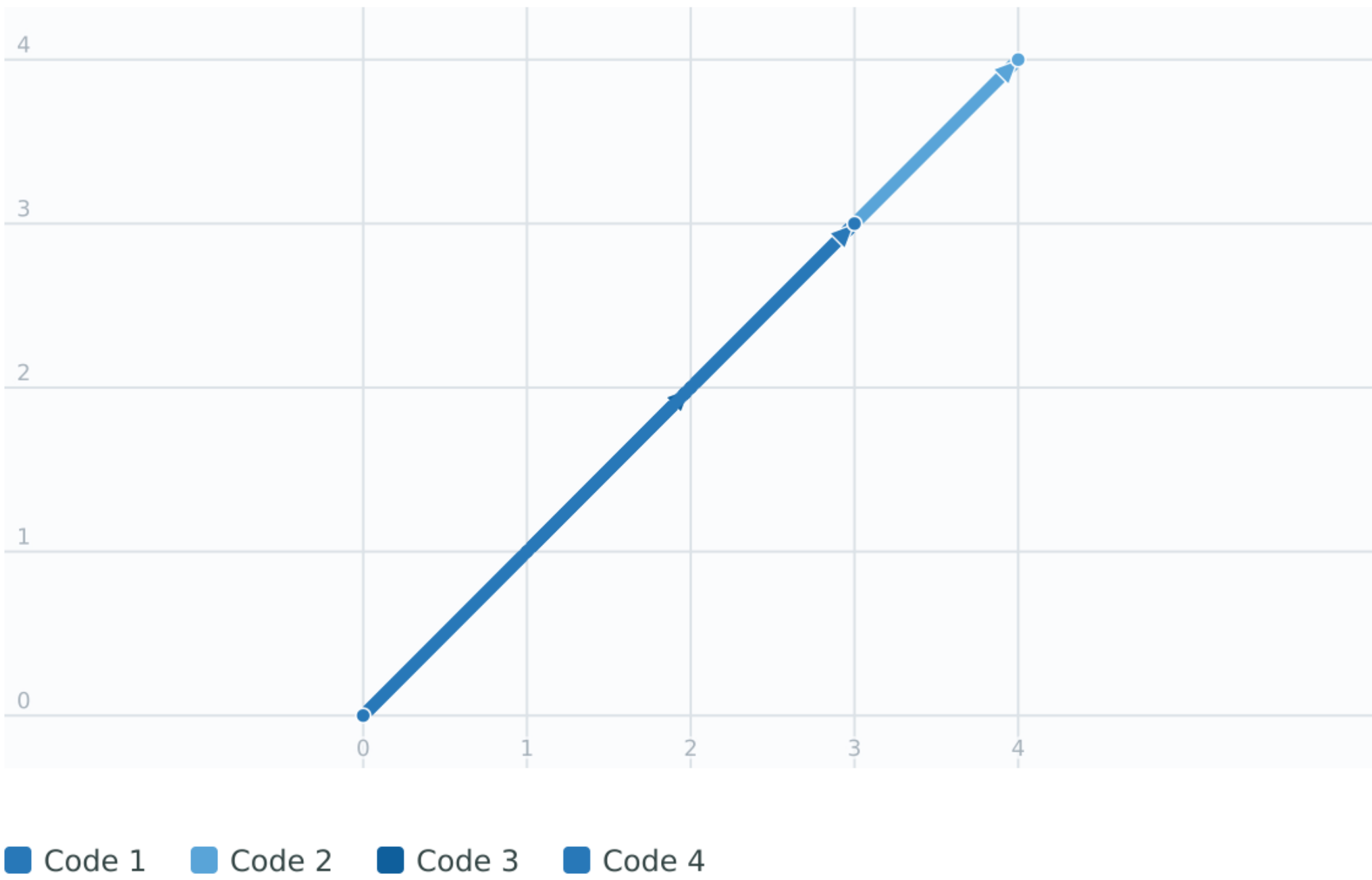
Code

```
SELECT tbl1.column1, tbl2.column1, tbl1.column2 ~ tbl2.column2 AS contains
FROM
  ( VALUES
    (1, 'LINESTRING (0 0,3 3)'::geometry)) AS tbl1,
  ( VALUES
    (2, 'LINESTRING (0 0,4 4)'::geometry),
    (3, 'LINESTRING (1 1,2 2)'::geometry),
    (4, 'LINESTRING (0 0,3 3)'::geometry)) AS tbl2;
```

Output

```
column1 | column1 | contains
---------+---------+----------
       1 |       2 | f
       1 |       3 | t
       1 |       4 | t
(3 rows)
```

Figure



□□

@, &&

7.10.1.23 ~(geometry,box2df)

~(geometry,box2df) — Returns TRUE if a geometry's 2D bonding box contains a 2D float precision bounding box (GIDX).

□□

boolean ~(geometry A , box2df B);

□□

The ~ operator returns TRUE if the 2D bounding box of a geometry A contains the 2D bounding box B, using float precision. This means that if B is a (double precision) box2d, it will be internally converted to a float precision 2D bounding box (BOX2DF)



□□
This operand is intended to be used internally by BRIN indexes, more than by users.

Availability: 2.3.0 support for Block Range INdexes (BRIN) was introduced.

This method supports Circular Strings and Curves.

This function supports Polyhedral surfaces.

□□

Code

```
SELECT ST_Buffer('POINT(1 1)'::geometry, 10) ~ ST_MakeBox2D(ST_Point(0, 0), ST_Point(2, 2)) ↩
    AS contains;
```

Output

```
t
```

□□

&&(geometry,box2df), &&(box2df,geometry), &&(box2df,box2df), ~(box2df,geometry), ~(box2df,box2df), @(geometry,box2df), @(box2df,geometry), @(box2df,box2df)

7.10.1.24 ~(box2df,geometry)

~(box2df,geometry) — Returns TRUE if a 2D float precision bounding box (BOX2DF) contains a geometry's 2D bonding box.

□□

boolean ~(box2df A , geometry B);

□□

The ~ operator returns TRUE if the 2D bounding box A contains the B geometry's bounding box, using float precision. This means that if A is a (double precision) box2d, it will be internally converted to a float precision 2D bounding box (BOX2DF)



□□
This operand is intended to be used internally by BRIN indexes, more than by users.

Availability: 2.3.0 support for Block Range INdexes (BRIN) was introduced.

This method supports Circular Strings and Curves.

This function supports Polyhedral surfaces.

□□

Code

```
SELECT ST_MakeBox2D(ST_Point(0, 0), ST_Point(5, 5)) ~ ST_Buffer('POINT(2 2)'::geometry, 1)  ↩
    AS contains;
```

Output

```
t
```

□□

&&(geometry,box2df), &&(box2df,geometry), &&(box2df,box2df), ~(geometry,box2df), ~(box2df,box2df), @(geometry,box2df), @(box2df,geometry), @(box2df,box2df)

7.10.1.25 ~(box2df,box2df)

~(box2df,box2df) — Returns TRUE if a 2D float precision bounding box (BOX2DF) contains another 2D float precision bounding box (BOX2DF).

□□

boolean ~(box2df A , box2df B);

□□

The ~ operator returns TRUE if the 2D bounding box A contains the 2D bounding box B, using float precision. This means that if A is a (double precision) box2d, it will be internally converted to a float precision 2D bounding box (BOX2DF)



□□
This operand is intended to be used internally by BRIN indexes, more than by users.

Availability: 2.3.0 support for Block Range INdexes (BRIN) was introduced.

This method supports Circular Strings and Curves.

This function supports Polyhedral surfaces.

□□

Code

```
SELECT ST_MakeBox2D(ST_Point(0, 0), ST_Point(5, 5)) ~ ST_MakeBox2D(ST_Point(2, 2), ST_Point ↩
    (3, 3)) AS contains;
```

Output

```
t
```

□□

&&(geometry,box2df), &&(box2df,geometry), &&(box2df,box2df), ~(geometry,box2df), ~(box2df,geometry)
@(geometry,box2df), @(box2df,geometry), @(box2df,box2df)

7.10.1.26 ~=

~= — A □□□□□□ B □□□□□□□□□□□ TRUE □□□□□□.

□□

boolean ~=(geometry A , geometry B);

□□

~= □□□□□□/□□□ A □□□□□□□□/□□□ B □□□□□□□□□□□ TRUE □□□□□□.



□□
□□□□□ (operand) □□□□□□□□□□□□□□□□□□□□□□□□□□□□□.

1.5.0 □□□□□□□□□□□□□□.

This function supports Polyhedral surfaces.

□□
This operator has changed behavior in PostGIS 1.5 from testing for actual geometric equality to only checking for bounding box equality. To complicate things it also depends on if you have done a hard or soft upgrade which behavior your database has. To find out which behavior your database has you can run the query below. To check for true equality use ST_OrderingEquals or ST_Equals.

□□

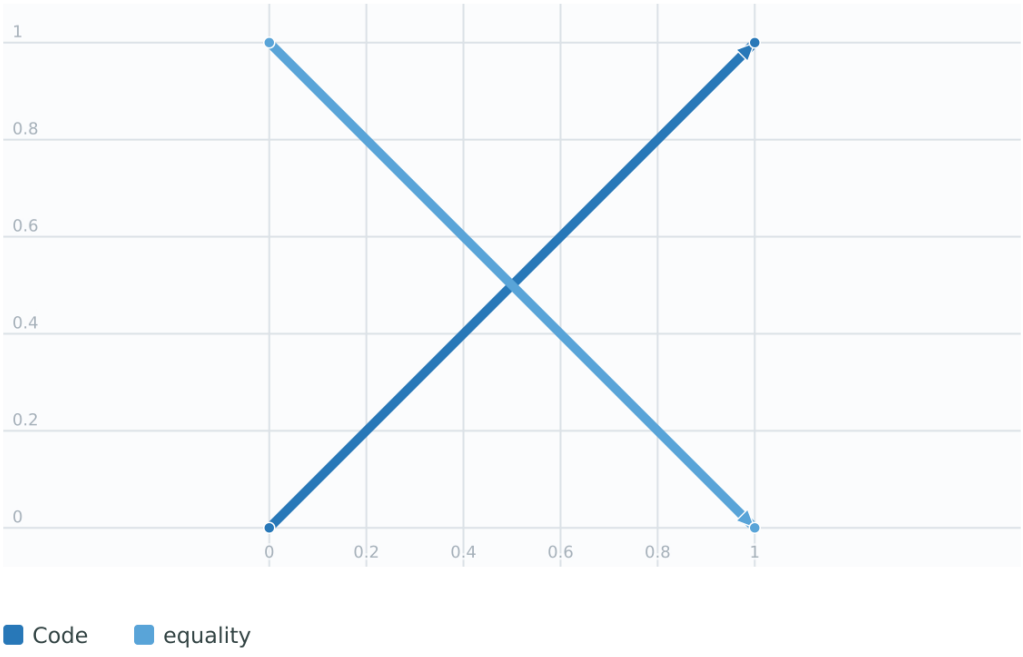
Code

```
SELECT 'LINESTRING(0 0,1 1)'::geometry ~= 'LINESTRING(0 1,1 0)'::geometry AS equality;
```

Output

```
t
```

Figure



□□

ST_Equals, ST_OrderingEquals, =

7.10.2 □□□ (operator)

7.10.2.1 <->

<-> — A □ B □□□ 2 □□□□□□□□□□.

□□

double precision **<->**(geometry A , geometry B);
double precision **<->**(geography A , geography B);

□□

The <-> operator returns the 2D distance between two geometries. Used in the "ORDER BY" clause provides index-assisted nearest-neighbor result sets with true KNN distance search for geometries, and distance on the sphere for geographies.

> **□□**
> □□□□□□ (operand) □□□□□□□□□□□□□□□□□□□ 2 □□ GiST □□□□□□□□□□□□□. □□□□ ORDER BY □□□.

> **□□**
> □□□□□□□□□, □□□□□ a.geom □□□□'SRID=3005;POINT(1011102 450541)'::geometry □□□, (□□□□□/CTE(common table expression) □□□□□□□) □□□□□□□□□□□□□□□□□□□□□.

□□□□□□□□□□□ OpenGeo workshop: Nearest-Neighbour Searching □□□□□□□.

Enhanced: 2.2.0 -- True KNN ("K nearest neighbor") behavior for geometry and geography. Note for geography KNN is based on sphere rather than spheroid.

Changed: 2.2.0 -- Old hybrid-syntax workarounds may be slower once true KNN is available. See examples below.

Availability: 2.0.0 -- Weak KNN provides nearest neighbors based on geometry centroid distances instead of true distances. Exact results for points, inexact for all other types.

□□

Code

```
SELECT ST_Distance(geom, 'SRID=3005;POINT(1011102 450541)'::geometry) as d, edabbr, vaabbr
FROM va2005
ORDER BY d limit 10;
```

Output

```
d         | edabbr | vaabbr
------------------+--------+--------
                0 | ALQ    | 128
 5541.57712511724 | ALQ    | 129A
 5579.67450712005 | ALQ    | 001
  6083.4207708641 | ALQ    | 131
  7691.2205404848 | ALQ    | 003
 7900.75451037313 | ALQ    | 122
 8694.20710669982 | ALQ    | 129B
 9564.24289057111 | ALQ    | 130
  12089.665931705 | ALQ    | 127
 18472.5531479404 | ALQ    | 002
(10 rows)
```

□□□ KNN □□□□□□□□□:

Code

```
SELECT ST_Distance(geom, 'SRID=3005;POINT(1011102 450541)'::geometry) as d,edabbr, vaabbr
FROM va2005
ORDER BY geom <-> 'SRID=3005;POINT(1011102 450541)'::geometry limit 10;
```

Output

```
        d         | edabbr | vaabbr
------------------+--------+--------
                0 | ALQ    | 128
 5541.57712511724 | ALQ    | 129A
 5579.67450712005 | ALQ    | 001
  6083.4207708641 | ALQ    | 131
  7691.2205404848 | ALQ    | 003
 7900.75451037313 | ALQ    | 122
 8694.20710669982 | ALQ    | 129B
 9564.24289057111 | ALQ    | 130
  12089.665931705 | ALQ    | 127
 18472.5531479404 | ALQ    | 002
(10 rows)
```

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠”EXPLAIN ANALYZE” ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

When a plan cannot produce exact distances directly, use a hybrid query to find the true nearest neighbors. First a CTE query using the index-assisted KNN, then an exact query to get correct ordering:

Code

```
WITH index_query AS (
  SELECT ST_Distance(geom, 'SRID=3005;POINT(1011102 450541)'::geometry) as d,edabbr, vaabbr
    FROM va2005
  ORDER BY geom <-> 'SRID=3005;POINT(1011102 450541)'::geometry LIMIT 100)
  SELECT *
    FROM index_query
  ORDER BY d limit 10;
```

Output

```
        d         | edabbr | vaabbr
------------------+--------+--------
                0 | ALQ    | 128
 5541.57712511724 | ALQ    | 129A
 5579.67450712005 | ALQ    | 001
  6083.4207708641 | ALQ    | 131
  7691.2205404848 | ALQ    | 003
 7900.75451037313 | ALQ    | 122
 8694.20710669982 | ALQ    | 129B
 9564.24289057111 | ALQ    | 130
  12089.665931705 | ALQ    | 127
 18472.5531479404 | ALQ    | 002
(10 rows)
```

⊠⊠

ST_DWithin, ST_Distance, <#>

7.10.2.2 |=|

|=| — A ⊠ B ⊠⊠⊠⊠⊠⊠⊠ (closest point of approach) ⊠⊠⊠⊠⊠ (trajectory) ⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

double precision **|=|**(geometry A , geometry B);

⊠⊠

The |=| operator returns the 3D distance between two trajectories (See ST_IsValidTrajectory). This is the same as ST_DistanceCPA but as an operator it can be used for doing nearest neighbor searches using an N-dimensional index.

> ⊠⊠
> ⊠⊠⊠⊠⊠ (operand) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ N ⊠⊠ GiST ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠ ORDER BY ⊠⊠⊠.

> ⊠⊠
> ⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠ a.geom ⊠⊠⊠⊠⊠’SRID=3005;LINESTRINGM(0 0 0,0 0 1)’::geometry ⊠⊠⊠, (⊠⊠⊠⊠⊠/CTE(common table expression) ⊠⊠⊠⊠⊠⊠⊠) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

Availability: 2.2.0.

⊠⊠

This example orders stored trajectories by their distance from a literal query trajectory.

Code

```
\set qt 'ST_AddMeasure(ST_MakeLine(ST_MakePointM(-350, 300, 0), ST_MakePointM(-410, 490, 0) ↩
    ), 10, 20)'

SELECT track_id, dist FROM (
  SELECT track_id,
    ST_DistanceCPA(tr, :qt) dist
  FROM trajectories
  ORDER BY tr |=| :qt
  LIMIT 5
) foo;
```

Output

```
track_id        dist
----------+-------------------
      395 | 0.576496831518066
      380 |  5.06797130410151
      390 |  7.72262293958322
      385 |   9.8004461358071
      405 |  10.9534397988433
(5 rows)
```

⊠⊠

ST_DistanceCPA, ST_ClosestPointOfApproach, ST_IsValidTrajectory

7.10.2.3 <#>

<#> — A ⊠ B ⊠⊠⊠⊠⊠⊠⊠⊠⊠ 2 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

double precision **<#>**(geometry A , geometry B);

⊠⊠

The <#> operator returns distance between two floating point bounding boxes, possibly reading them from a spatial index. Useful for doing nearest neighbor **approximate** distance ordering.

> ⊠⊠
> ⊠⊠⊠⊠⊠ (operand) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠ ORDER BY ⊠⊠⊠.

> ⊠⊠
> ⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠ g1.geom <#> ⊠⊠⊠⊠ ORDER BY (ST_GeomFromText('POINT(1 2)') <#> geom) ⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

Availability: 2.0.0.

⊠⊠

Code

```
SELECT *
FROM (
SELECT b.tlid, b.mtfcc,
    b.geom <#
> ST_GeomFromText('LINESTRING(746149 2948672,745954 2948576,
        745787 2948499,745740 2948468,745712 2948438,
        745690 2948384,745677 2948319)', 2249) As b_dist,
        ST_Distance(b.geom, ST_GeomFromText('LINESTRING(746149 2948672,745954 2948576,
        745787 2948499,745740 2948468,745712 2948438,
        745690 2948384,745677 2948319)', 2249)) As act_dist
    FROM bos_roads As b
    ORDER BY b_dist, b.tlid
    LIMIT 100) As foo
    ORDER BY act_dist, tlid LIMIT 10;
```

Output

```
   tlid    | mtfcc |      b_dist       |     act_dist
-----------+-------+-------------------+------------------
  85732027 | S1400 |                 0 |                0
  85732029 | S1400 |                 0 |                0
  85732031 | S1400 |                 0 |                0
  85734335 | S1400 |                 0 |                0
  85736037 | S1400 |                 0 |                0
```

```
 624683742 | S1400 |                0 | 128.528874268666
  85719343 | S1400 | 260.839270432962 | 260.839270432962
  85741826 | S1400 | 164.759294123275 | 260.839270432962
  85732032 | S1400 |           277.75 | 311.830282365264
  85735592 | S1400 |           222.25 | 311.830282365264
(10 rows)
```

□□

ST_DWithin, ST_Distance, <->

7.10.2.4 <<->>

<<->> — Returns the n-D distance between the A and B geometries or bounding boxes

□□

double precision **<<->>**(geometry A , geometry B);

□□

<<->> □□□□□□□□□□□□□□□□□□□ N □ (□□□□) □□□□□□□□. □□□ □□□□□□□□□ □□□□□□□□□□.

> **□□**
> □□□□□ (operand) □□□□□□□□□□□□□□□□□□□□□□ N □□ GiST □□□□□□□□□□□. □ □□□ ORDER BY □□ □□□.

> **□□**
> □□□□□□□□, □□□□ a.geom □□□□'SRID=3005;POINT(1011102 450541)'::geometry □ □□, (□□□□/CTE(common table expression) □□□□□□) □□□□□□□□□□□□□□□□□ □□.

Availability: 2.2.0.

□□

<->

7.11 Spatial Relationships

7.11.1 Topological Relationships

7.11.1.1 ST_3DIntersects

ST_3DIntersects — Tests if two geometries spatially intersect in 3D - only for points, linestrings, polygons, polyhedral surface (area)

□□

boolean **ST_3DIntersects**(geometry geomA , geometry geomB);

□□

Overlaps, Touches, Within all imply spatial intersection. If any of the aforementioned returns true, then the geometries also spatially intersect. Disjoint implies false for spatial intersection.



□□
This function automatically includes a bounding box comparison that makes use of any spatial indexes that are available on the geometries.



□□
Because of floating robustness failures, geometries don't always intersect as you'd expect them to after geometric processing. For example the closest point on a linestring to a geometry may not lie on the linestring. For these kind of issues where a distance of a centimeter you want to just consider as intersecting, use ST_3DDWithin.

Changed: 3.0.0 SFCGAL backend removed, GEOS backend supports TINs.

2.0.0 □□□□□□□□□□□□.

This function supports 3d and will not drop the z-index.

This function supports Polyhedral surfaces.

This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).

This method implements the SQL/MM specification. SQL-MM IEC 13249-3: 5.1

□□□□

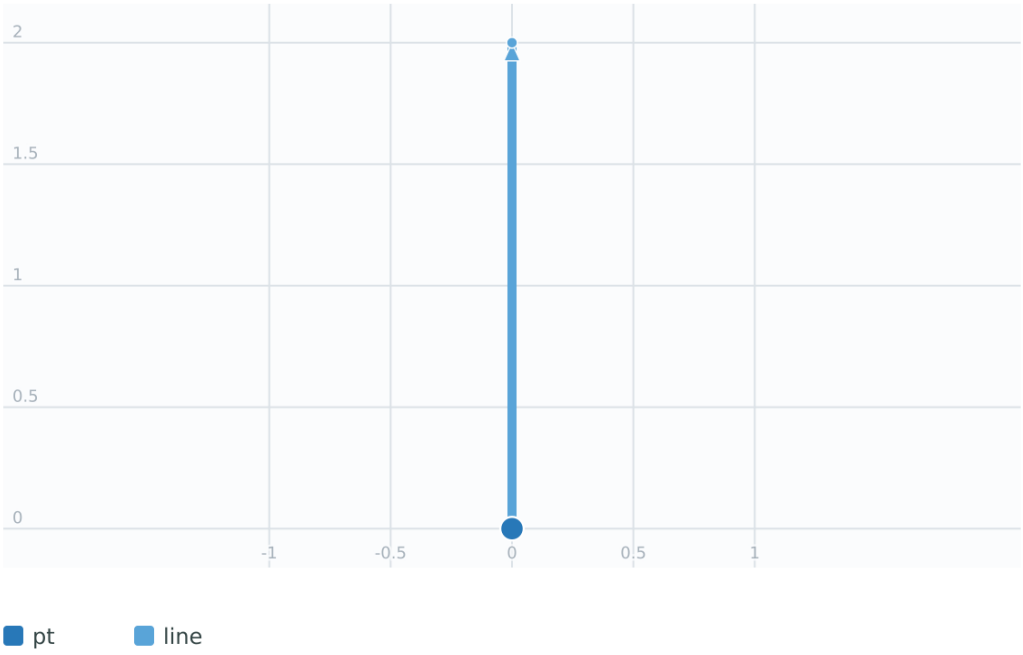
Code

```
SELECT ST_3DIntersects(pt, line), ST_Intersects(pt, line)
FROM (SELECT 'POINT(0 0 2)'::geometry As pt, 'LINESTRING (0 0 1,0 2 3)'::geometry As line)  ↩
    As foo;
```

Output

```
st_3dintersects | st_intersects
----------------+---------------
 f              | t
(1 row)
```

Figure



TIN Examples

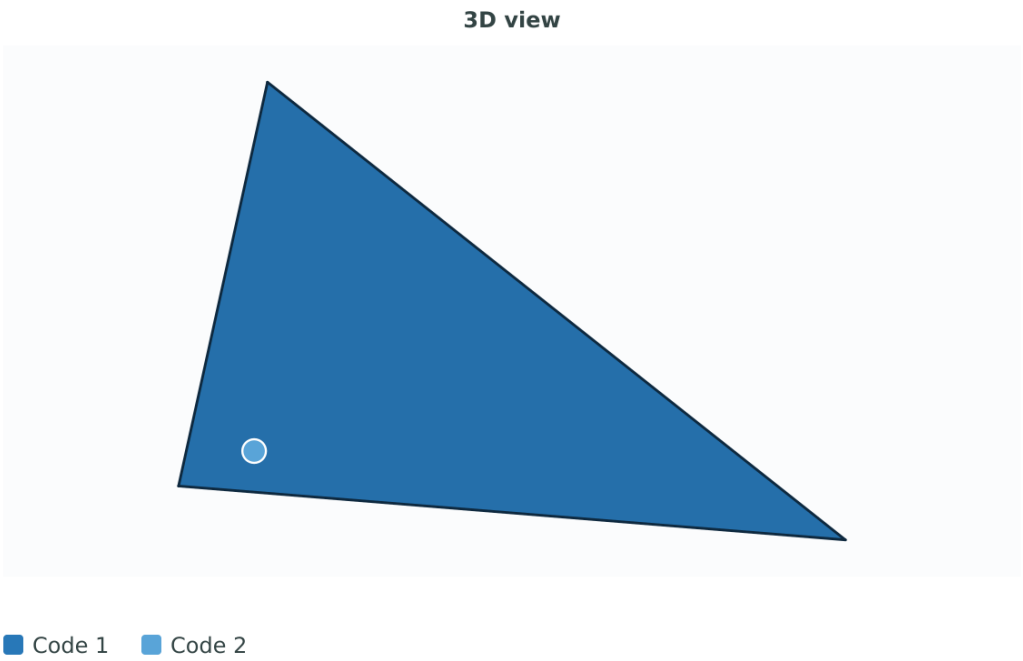
Code

```
SELECT ST_3DIntersects('TIN(((0 0 0,1 0 0,0 1 0,0 0 0)))'::geometry, 'POINT(.1 .1 0)':: ↩
    geometry);
```

Output

```
t
```

Figure



☒☒

ST_3DDWithin, ST_Intersects

7.11.1.2 ST_Contains

ST_Contains — Tests if every point of B lies in A, and their interiors have a point in common

☒☒

boolean **ST_Contains**(geometry geomA, geometry geomB);

☒☒

Returns TRUE if geometry A contains geometry B. A contains B if and only if all points of B lie inside (i.e. in the interior or boundary of) A (or equivalently, no points of B lie in the exterior of A), and the interiors of A and B have at least one point in common.

In mathematical terms: *ST_Contains(A, B) ⇔ (A ∩ B = B) ∧ (Int(A) ∩ Int(B) ≠ ∅)*

The contains relationship is reflexive: every geometry contains itself. (In contrast, in the ST_ContainsProperl predicate a geometry does *not* properly contain itself.) The relationship is antisymmetric: if `ST_Contains(A, = true` and `ST_Contains(B,A) = true`, then the two geometries must be topologically equal (`ST_Equals(A, = true`).

ST_Contains is the converse of ST_Within. So, `ST_Contains(A,B) = ST_Within(B,A)`.



□□
Because the interiors must have a common point, a subtlety of the definition is that polygons and lines do *not* contain lines and points lying fully in their boundary. For further details see Subtleties of OGC Covers, Contains, Within. The ST_Covers predicate provides a more inclusive relationship.



□□
This function automatically includes a bounding box comparison that makes use of any spatial indexes that are available on the geometries. To avoid index use, use the function `_ST_Contains`.

GEOS □□□□□

Enhanced: 2.3.0 Enhancement to PIP short-circuit extended to support MultiPoints with few points. Prior versions only supported point in polygon.

Enhanced: 3.0.0 enabled support for `GEOMETRYCOLLECTION`



□□
Do not use this function with invalid geometries. You will get unexpected results.

NOTE: this is the "allowable" version that returns a boolean, not an integer.

This method implements the OGC Simple Features Implementation Specification for SQL 1.1. s2.1.1.2 // s2.1.13.3 - same as within(geometry B, geometry A)

This method implements the SQL/MM specification. SQL-MM 3: 5.1.31

□□

`ST_Contains` returns `TRUE` in the following situations:

A LineString contains a MultiPoint whose points lie in its interior or boundary.

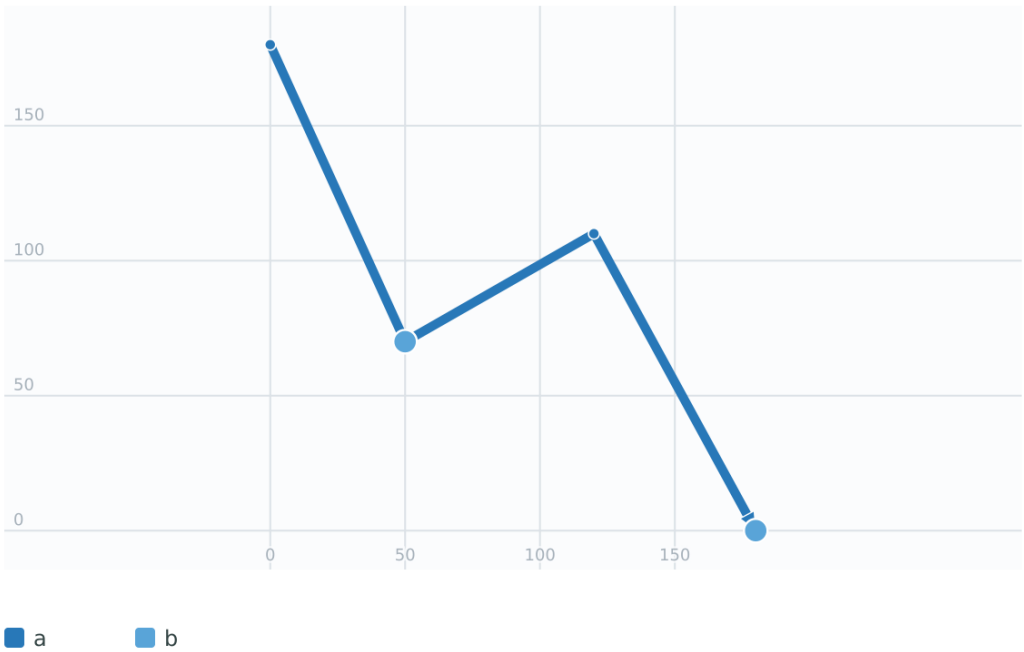
Code

```
SELECT ST_Contains(a, b)
FROM (SELECT
  'LINESTRING (10 190,60 80,130 120,190 10)'::geometry AS a,
  'MULTIPOINT (60 80,190 10)'::geometry AS b
) AS example;
```

Output

```
t
```

Figure



A Polygon contains a Point in its interior.

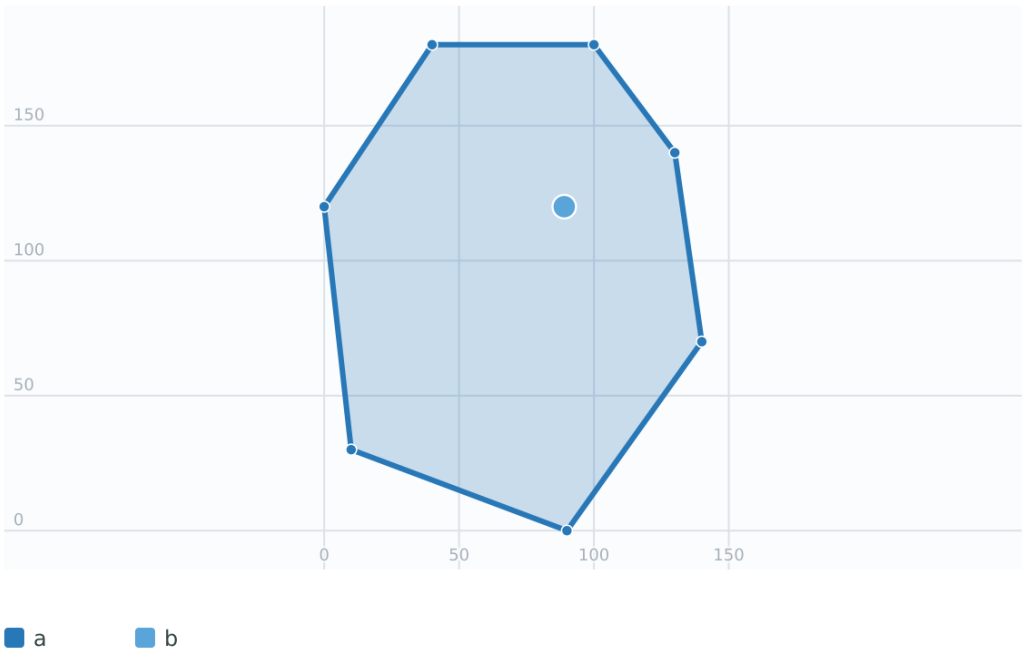
Code

```
SELECT ST_Contains(a, b)
FROM (SELECT
  'POLYGON ((10 130,50 190,110 190,140 150,150 80,100 10,20 40,10 130))'::geometry AS a,
  'POINT (99 130)'::geometry AS b
) AS example;
```

Output

```
t
```

Figure



A Polygon contains a LineString with interior points inside the Polygon.

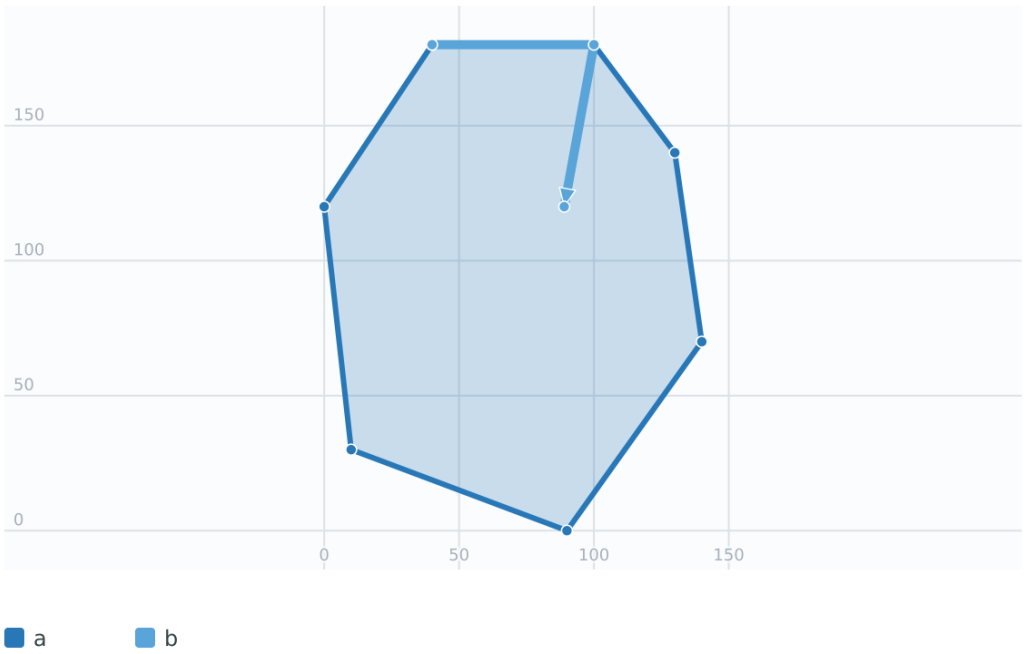
Code

```
SELECT ST_Contains(a, b)
FROM (SELECT
  'POLYGON ((10 130,50 190,110 190,140 150,150 80,100 10,20 40,10 130))'::geometry AS a,
  'LINESTRING (50 190,110 190,99 130)'::geometry AS b
) AS example;
```

Output

```
t
```

Figure



A Polygon contains another Polygon in its interior.

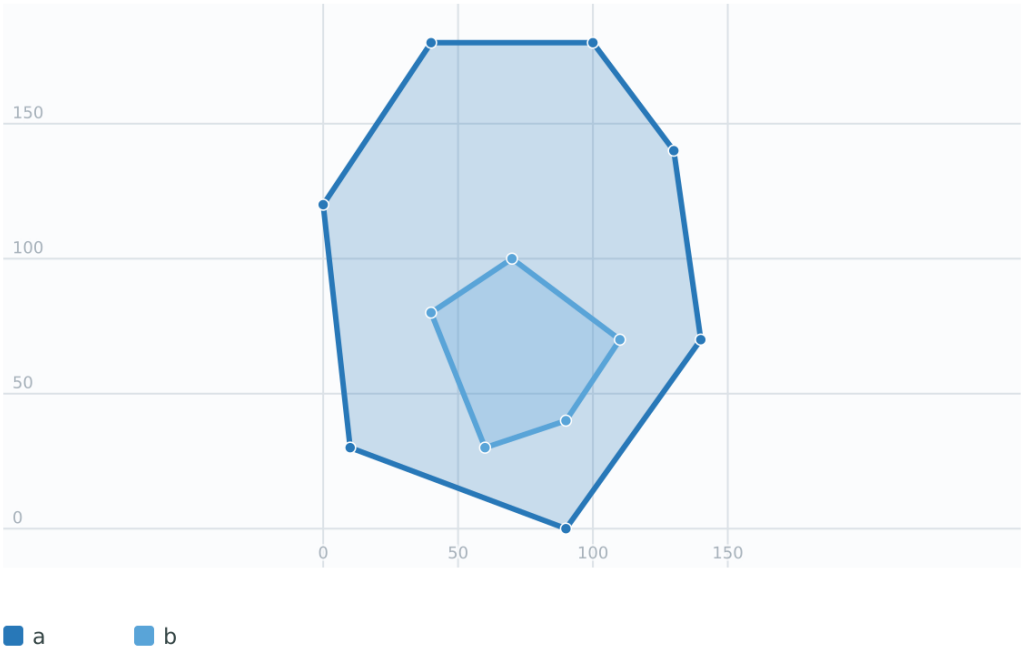
Code

```
SELECT ST_Contains(a, b)
FROM (SELECT
  'POLYGON ((10 130,50 190,110 190,140 150,150 80,100 10,20 40,10 130))'::geometry AS a,
  'POLYGON ((70 40,100 50,120 80,80 110,50 90,70 40))'::geometry AS b
) AS example;
```

Output

```
t
```

Figure



ST_Contains returns FALSE in the following situations:

A Polygon does not contain a MultiPoint when one point lies in a hole.

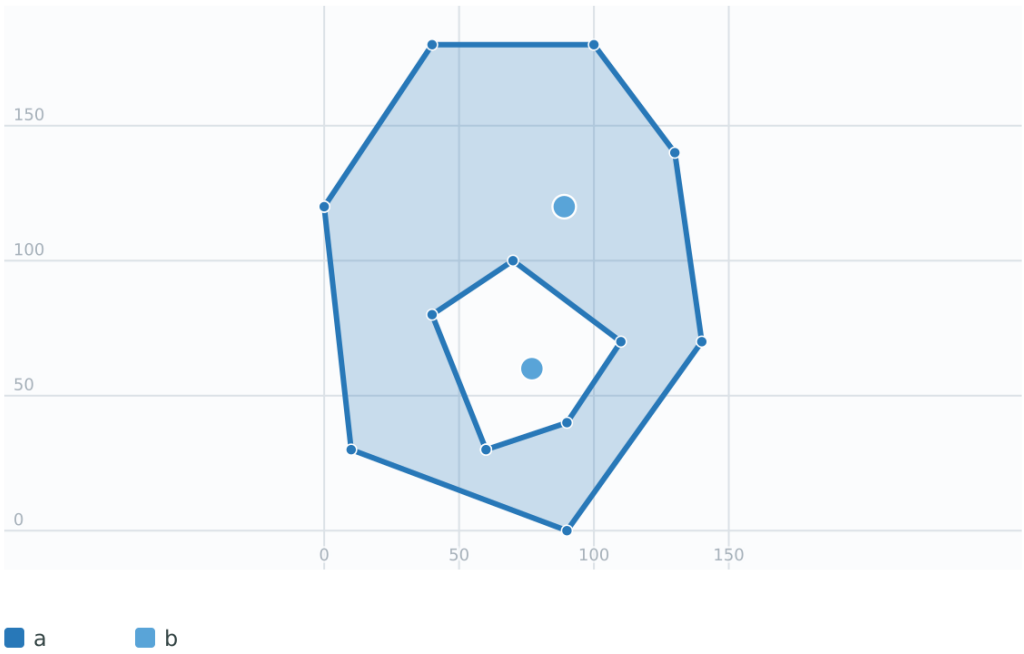
Code

```
SELECT ST_Contains(a, b)
FROM (SELECT
  'POLYGON ((10 130,50 190,110 190,140 150,150 80,100 10,20 40,10 130),(70 40,100 50,120 ↩
    80,80 110,50 90,70 40))'::geometry AS a,
  'MULTIPOINT (99 130,87 70)'::geometry AS b
) AS example;
```

Output

```
f
```

Figure



A Polygon does not contain a LineString extending outside it.

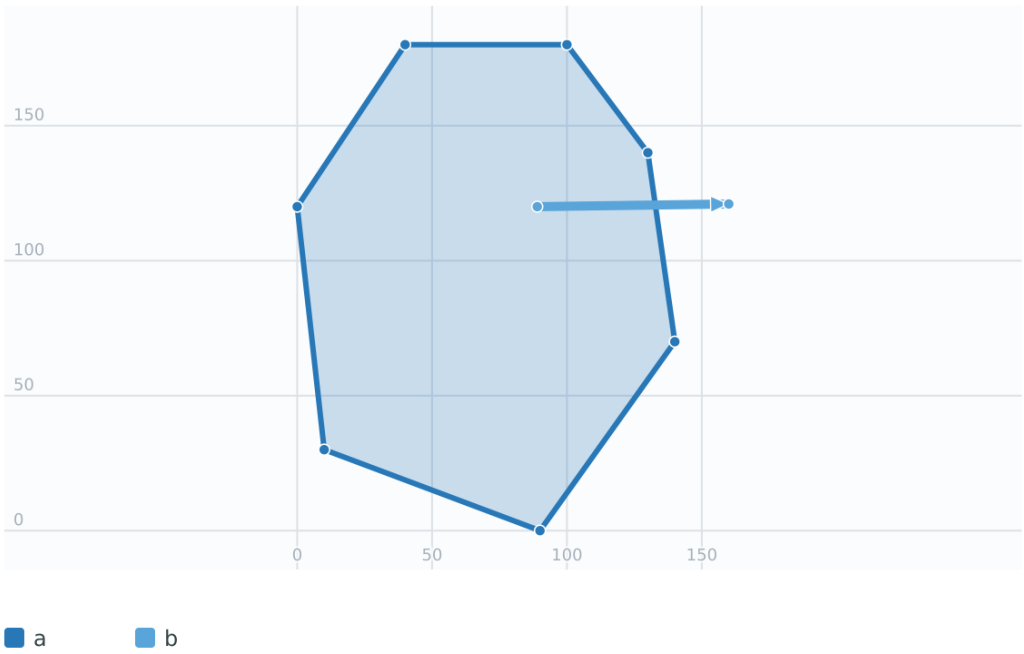
Code

```
SELECT ST_Contains(a, b)
FROM (SELECT
  'POLYGON ((10 130,50 190,110 190,140 150,150 80,100 10,20 40,10 130))'::geometry AS a,
  'LINESTRING (99 130,170 131)'::geometry AS b
) AS example;
```

Output

```
f
```

Figure



Due to the interior intersection condition `ST_Contains` returns `FALSE` in the following situations (whereas `ST_Covers` returns `TRUE`):

A LineString does not contain a Point lying only on its boundary.

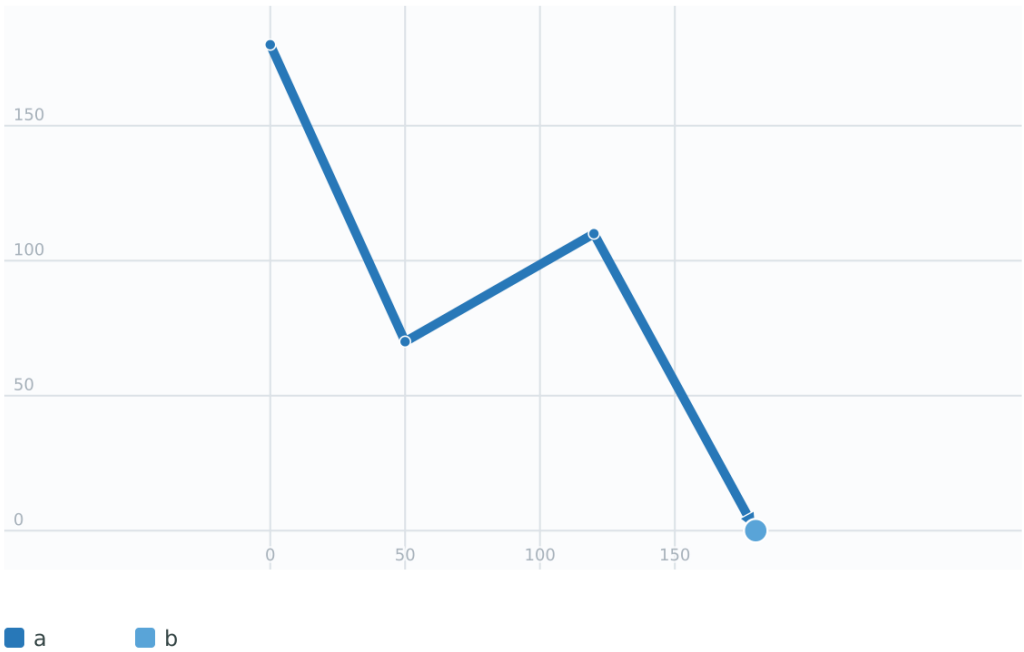
Code

```
SELECT ST_Contains(a, b), ST_Covers(a, b)
FROM (SELECT
  'LINESTRING (10 190,60 80,130 120,190 10)'::geometry AS a,
  'POINT (190 10)'::geometry AS b
) AS example;
```

Output

```
f | t
```

Figure



A Polygon does not contain a LineString lying entirely on its boundary.

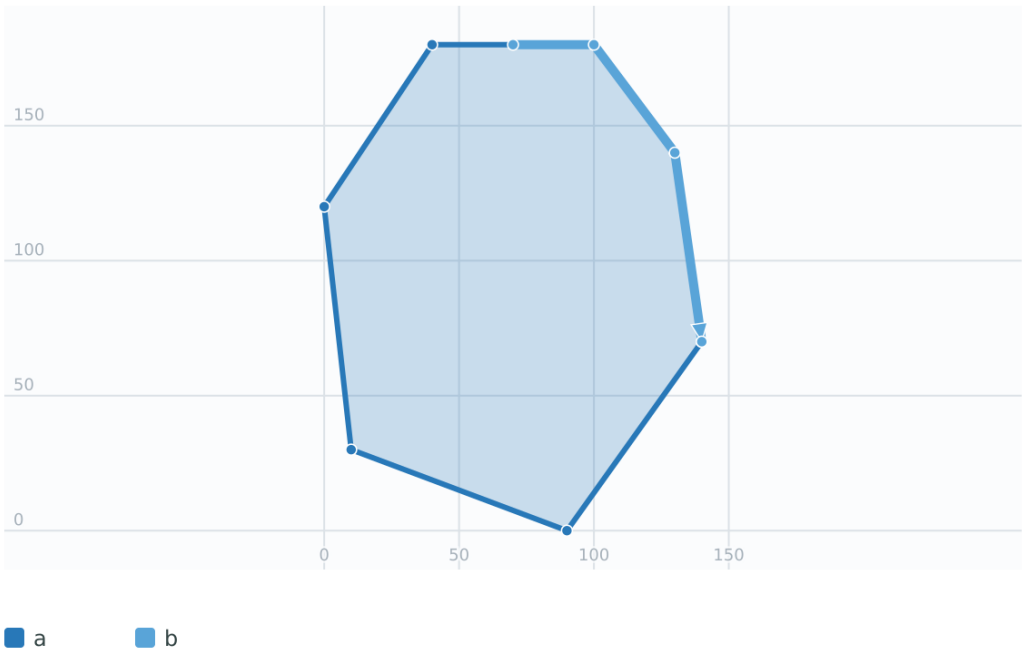
Code

```
SELECT ST_Contains(a, b), ST_Covers(a, b)
FROM (SELECT
  'POLYGON ((10 130,50 190,110 190,140 150,150 80,100 10,20 40,10 130))'::geometry AS a,
  'LINESTRING (80 190,110 190,140 150,150 80)'::geometry AS b
) AS example;
```

Output

```
f | t
```

Figure



This example tests a circle within another circle.

Code

```
WITH circles AS (
  SELECT ST_Buffer(center, 10) AS smallc,
         ST_Buffer(center, 20) AS bigc
  FROM (SELECT 'POINT(1 2)'::geometry AS center) AS p
)
SELECT
  ST_Contains(smallc, bigc) AS smallcontainsbig,
  ST_Contains(bigc, smallc) AS bigcontainssmall,
  ST_Contains(bigc, ST_Union(smallc, bigc)) AS bigcontainsunion,
  ST_Equals(bigc, ST_Union(smallc, bigc)) AS bigisunion,
  ST_Covers(bigc, ST_ExteriorRing(bigc)) AS bigcoversexterior,
  ST_Contains(bigc, ST_ExteriorRing(bigc)) AS bigcontainsexterior
FROM circles;
```

Output

```
-[ RECORD 1 ]-----
smallcontainsbig   | f
bigcontainssmall   | t
bigcontainsunion   | t
bigisunion         | t
bigcoversexterior  | t
bigcontainsexterior | f
```

Figure



This example demonstrates the difference between contains and contains-properly predicates.

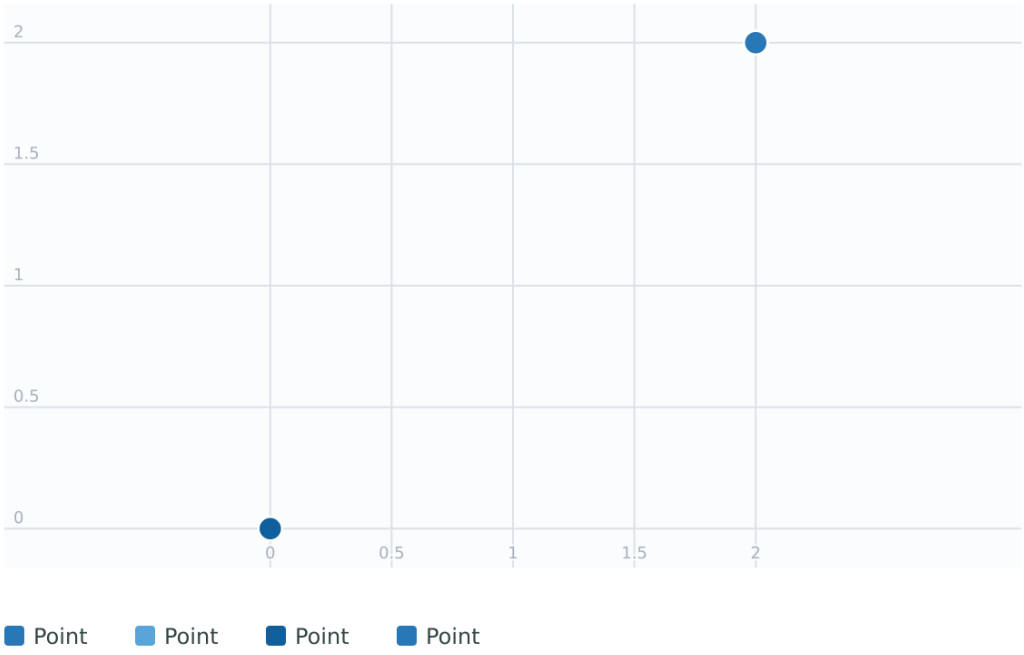
Code

```
SELECT
  ST_GeometryType(geom_a) AS geomtype,
  ST_Contains(geom_a, geom_a) AS acontainsa,
  ST_ContainsProperly(geom_a, geom_a) AS acontainspropa,
  ST_Contains(geom_a, ST_Boundary(geom_a)) AS acontainsba,
  ST_ContainsProperly(geom_a, ST_Boundary(geom_a)) AS acontainspropba
FROM (VALUES
    (ST_Buffer(ST_Point(1, 1), 5, 1)),
    (ST_MakeLine(ST_Point(1, 1), ST_Point(-1, -1))),
    (ST_Point(1, 1))
  ) AS foo(geom_a);
```

Output

```
geomtype    | acontainsa | acontainspropa | acontainsba | acontainspropba
------------+------------+----------------+-------------+-----------------
ST_Polygon    | t          | f              | f           | f
ST_LineString | t          | f              | f           | f
ST_Point      | t          | t              | f           | f
```

Figure



□□

ST_Boundary, ST_ContainsProperly, ST_Covers, ST_CoveredBy, ST_Equals, ST_Within

7.11.1.3 ST_ContainsProperly

ST_ContainsProperly — Tests if every point of B lies in the interior of A

□□

boolean **ST_ContainsProperly**(geometry geomA, geometry geomB);

□□

Returns `true` if every point of B lies in the interior of A (or equivalently, no point of B lies in the the boundary or exterior of A).

In mathematical terms: *ST_ContainsProperly(A, B) ⇔ Int(A)* □ *B = B*

A contains B properly if the DE-9IM Intersection Matrix for the two geometries matches [T**FF*FF*]

A does not properly contain itself, but does contain itself.

A use for this predicate is computing the intersections of a set of geometries with a large polygonal geometry. Since intersection is a fairly slow operation, it can be more efficient to use containsProperly to filter out test geometries which lie fully inside the area. In these cases the intersection is known a priori to be exactly the original test geometry.



□□
This function automatically includes a bounding box comparison that makes use of any spatial indexes that are available on the geometries. To avoid index use, use the function `_ST_ContainsProperly`.



☒☒
The advantage of this predicate over ST_Contains and ST_Intersects is that it can be computed more efficiently, with no need to compute topology at individual points.

GEOS ☒☒☒☒☒

1.4.0 ☒☒☒☒☒☒☒☒☒☒☒☒.

Enhanced: 3.0.0 enabled support for GEOMETRYCOLLECTION



☒☒
Do not use this function with invalid geometries. You will get unexpected results.

☒☒

This example tests a circle within another circle.

Code

```
WITH circles AS (
  SELECT ST_Buffer(center, 10) AS smallc,
         ST_Buffer(center, 20) AS bigc
  FROM (SELECT 'POINT(1 2)'::geometry AS center) AS p
)
SELECT
  ST_ContainsProperly(smallc, bigc) AS smallcontainspropbig,
  ST_ContainsProperly(bigc, smallc) AS bigcontainspropsmall,
  ST_ContainsProperly(bigc, ST_Union(smallc, bigc)) AS bigcontainspropunion,
  ST_Equals(bigc, ST_Union(smallc, bigc)) AS bigisunion,
  ST_Covers(bigc, ST_ExteriorRing(bigc)) AS bigcoversexterior,
  ST_ContainsProperly(bigc, ST_ExteriorRing(bigc)) AS bigcontainsexterior
FROM circles;
```

Output

```
-[ RECORD 1 ]-----
smallcontainspropbig   | f
bigcontainspropsmall   | t
bigcontainspropunion   | f
bigisunion             | t
bigcoversexterior      | t
bigcontainsexterior    | f
```

Figure



This example demonstrates the difference between contains and contains-properly predicates.

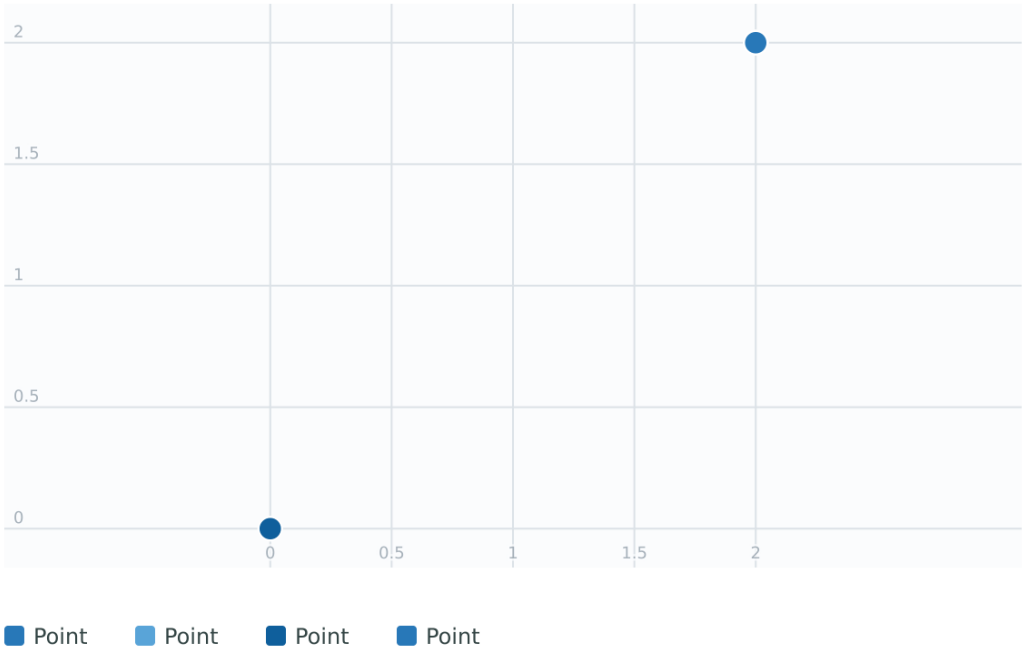
Code

```
WITH input(geom) AS (VALUES
  (ST_Buffer(ST_Point(1, 1), 5, 1)),
  (ST_MakeLine(ST_Point(1, 1), ST_Point(-1, -1))),
  (ST_Point(1, 1))
)
SELECT ST_GeometryType(geom) AS geomtype,
       ST_Contains(geom, geom) AS acontainsa,
       ST_ContainsProperly(geom, geom) AS acontainspropa,
       ST_Contains(geom, ST_Boundary(geom)) AS acontainsba,
       ST_ContainsProperly(geom, ST_Boundary(geom)) AS acontainspropba
FROM input;
```

Output

```
geomtype    | acontainsa | acontainspropa | acontainsba | acontainspropba
------------+------------+----------------+-------------+-----------------
ST_Polygon    | t          | f              | f           | f
ST_LineString | t          | f              | f           | f
ST_Point      | t          | t              | f           | f
```

Figure



□□

ST_GeometryType, ST_Boundary, ST_Contains, ST_Covers, ST_CoveredBy, ST_Equals, ST_Relate, ST_Withi

7.11.1.4 ST_CoveredBy

ST_CoveredBy — Tests if every point of A lies in B

□□

boolean **ST_CoveredBy**(geometry geomA, geometry geomB);
boolean **ST_CoveredBy**(geography geogA, geography geogB);

□□

Returns `true` if every point in Geometry/Geography A lies inside (i.e. intersects the interior or boundary of) Geometry/Geography B. Equivalently, tests that no point of A lies outside (in the exterior of) B.

In mathematical terms: *ST_CoveredBy(A, B)* ⇔ *A* □ *B* = *A*

ST_CoveredBy is the converse of ST_Covers. So, `ST_CoveredBy(A,B) = ST_Covers(B,A)`.

Generally this function should be used instead of ST_Within, since it has a simpler definition which does not have the quirk that ”boundaries are not within their geometry”.



□□
This function automatically includes a bounding box comparison that makes use of any spatial indexes that are available on the geometries. To avoid index use, use the function `_ST_CoveredBy`.

Enhanced: 3.0.0 enabled support for GEOMETRYCOLLECTION



□□
Do not use this function with invalid geometries. You will get unexpected results.

GEOS □□□□□

1.2.2 □□□□□□□□□□□□.

NOTE: this is the ”allowable” version that returns a boolean, not an integer.

Not an OGC standard, but Oracle has it too.

□□

This example tests a circle covered by another circle.

Code

```
WITH circles AS (
  SELECT ST_Buffer(center, 10) AS smallc,
         ST_Buffer(center, 20) AS bigc
  FROM (SELECT 'POINT(1 2)'::geometry AS center) AS p
)
SELECT
  ST_CoveredBy(smallc, smallc) AS smallinsmall,
  ST_CoveredBy(smallc, bigc) AS smallcoveredbybig,
  ST_CoveredBy(ST_ExteriorRing(bigc), bigc) AS exteriorcoveredbybig,
  ST_Within(ST_ExteriorRing(bigc), bigc) AS exteriorwithinbig
FROM circles;
```

Output

```
smallinsmall | smallcoveredbybig | exteriorcoveredbybig | exteriorwithinbig
-------------+-------------------+----------------------+------------------
 t           | t                 | t                    | f
(1 row)
```

Figure



☒☒

ST_Contains, ST_Covers, ST_ExteriorRing, ST_Within

7.11.1.5 ST_Covers

ST_Covers — Tests if every point of B lies in A

☒☒

boolean **ST_Covers**(geometry geomA, geometry geomB);
boolean **ST_Covers**(geography geogpolyA, geography geogpointB);

☒☒

Returns `true` if every point in Geometry/Geography B lies inside (i.e. intersects the interior or boundary of) Geometry/Geography A. Equivalently, tests that no point of B lies outside (in the exterior of) A.

In mathematical terms: *ST_Covers(A, B)* ⇔ *A* ▯ *B* = *B*

ST_Covers is the converse of ST_CoveredBy. So, `ST_Covers(A,B) = ST_CoveredBy(B,A)`.

Generally this function should be used instead of ST_Contains, since it has a simpler definition which does not have the quirk that ”geometries do not contain their boundary”.



☒☒
This function automatically includes a bounding box comparison that makes use of any spatial indexes that are available on the geometries. To avoid index use, use the function `_ST_Covers`.

Enhanced: 3.0.0 enabled support for GEOMETRYCOLLECTION



□□
Do not use this function with invalid geometries. You will get unexpected results.

GEOS □□□□□

Enhanced: 2.4.0 Support for polygon in polygon and line in polygon added for geography type

Enhanced: 2.3.0 Enhancement to PIP short-circuit for geometry extended to support MultiPoints with few points. Prior versions only supported point in polygon.

1.5.0 □□□□□□□□□□□□□.

1.2.2 □□□□□□□□□□□□.

NOTE: this is the ”allowable” version that returns a boolean, not an integer.

Not an OGC standard, but Oracle has it too.

□□

Geometry example

This example tests a circle covering another circle.

Code

```
WITH circles AS (
  SELECT ST_Buffer(center, 10) AS smallc,
         ST_Buffer(center, 20) AS bigc
  FROM (SELECT 'POINT(1 2)'::geometry AS center) AS p
)
SELECT
  ST_Covers(smallc, smallc) AS smallinsmall,
  ST_Covers(smallc, bigc) AS smallcoversbig,
  ST_Covers(bigc, ST_ExteriorRing(bigc)) AS bigcoversexterior,
  ST_Contains(bigc, ST_ExteriorRing(bigc)) AS bigcontainsexterior
FROM circles;
```

Output

```
smallinsmall | smallcoversbig | bigcoversexterior | bigcontainsexterior
-------------+----------------+-------------------+--------------------
 t           | f              | t                 | f
(1 row)
```

Figure



Geography example

This example compares a point with a 300-meter buffer against a point and against a 10-meter buffer around that point.

Code

```
SELECT ST_Covers(geog_poly, geog_pt) As poly_covers_pt,
  ST_Covers(ST_Buffer(geog_pt, 10), geog_pt) As buff_10m_covers_cent
  FROM (SELECT ST_Buffer('SRID=4326;POINT(-99.327 31.4821)'::geography, 300) As geog_poly,
        'SRID=4326;POINT(-99.33 31.483)'::geography As geog_pt ) As foo;
```

Output

```
poly_covers_pt | buff_10m_covers_cent
---------------+---------------------
 f             | t
```

□□

ST_Contains, ST_CoveredBy, ST_Within

7.11.1.6 ST_Crosses

ST_Crosses — Tests if two geometries have some, but not all, interior points in common

□□

boolean **ST_Crosses**(geometry g1, geometry g2);

□□

Compares two geometry objects and returns `true` if their intersection "spatially crosses"; that is, the geometries have some, but not all interior points in common. The intersection of the interiors of the geometries must be non-empty and must have dimension less than the maximum dimension of the two input geometries, and the intersection of the two geometries must not equal either geometry. Otherwise, it returns `false`. The crosses relation is symmetric and irreflexive.

In mathematical terms: *ST_Crosses(A, B) ⇔ (dim(Int(A) □ Int(B)) < max(dim(Int(A)), dim(Int(B)))) ∧ (A □ B ≠ A) ∧ (A □ B ≠ B)*

Geometries cross if their DE-9IM Intersection Matrix matches:

- `T*T******` for Point/Line, Point/Area, and Line/Area situations
- `T*****T**` for Line/Point, Area/Point, and Area/Line situations
- `0********` for Line/Line situations
- the result is `false` for Point/Point and Area/Area situations



□□
The OpenGIS Simple Features Specification defines this predicate only for Point/Line, Point/Area, Line/Line, and Line/Area situations. JTS / GEOS extends the definition to apply to Line/Point, Area/Point and Area/Line situations as well. This makes the relation symmetric.



□□
This function automatically includes a bounding box comparison that makes use of any spatial indexes that are available on the geometries.

Enhanced: 3.0.0 enabled support for `GEOMETRYCOLLECTION`

This method implements the OGC Simple Features Implementation Specification for SQL 1.1. s2.1.13.3

This method implements the SQL/MM specification. SQL-MM 3: 5.1.29

□□

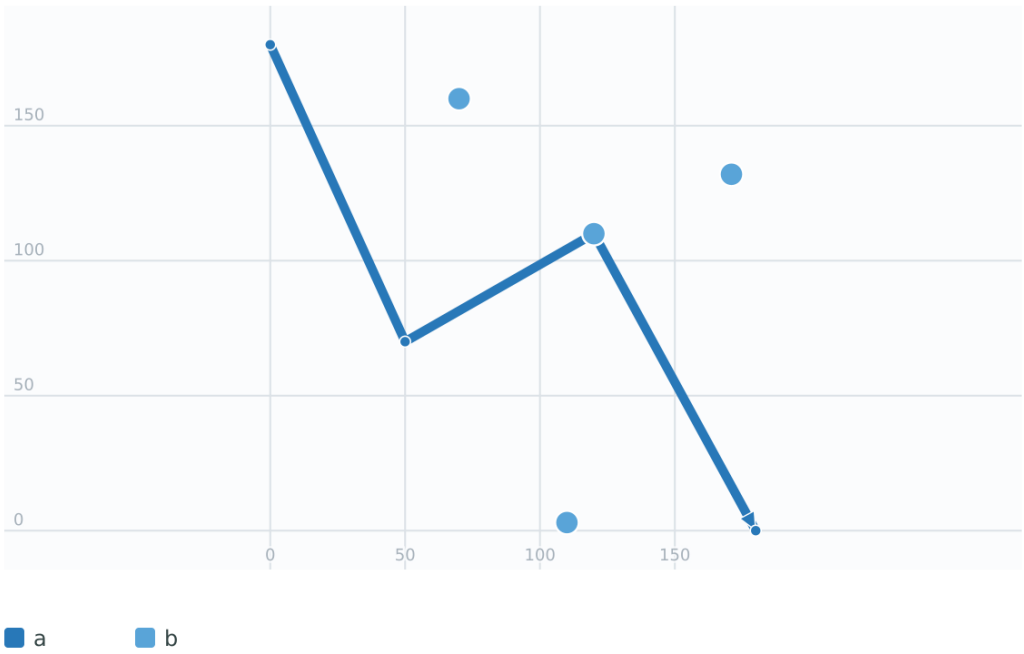
The following situations all return `true`.

A LineString crosses a MultiPoint when some, but not all, points lie on the line.

Code

```
SELECT ST_Crosses(a, b)
FROM (SELECT
  'LINESTRING (10 190,60 80,130 120,190 10)'::geometry AS a,
  'MULTIPOINT (80 170,120 13,130 120,181 142)'::geometry AS b
) AS example;
```

Figure

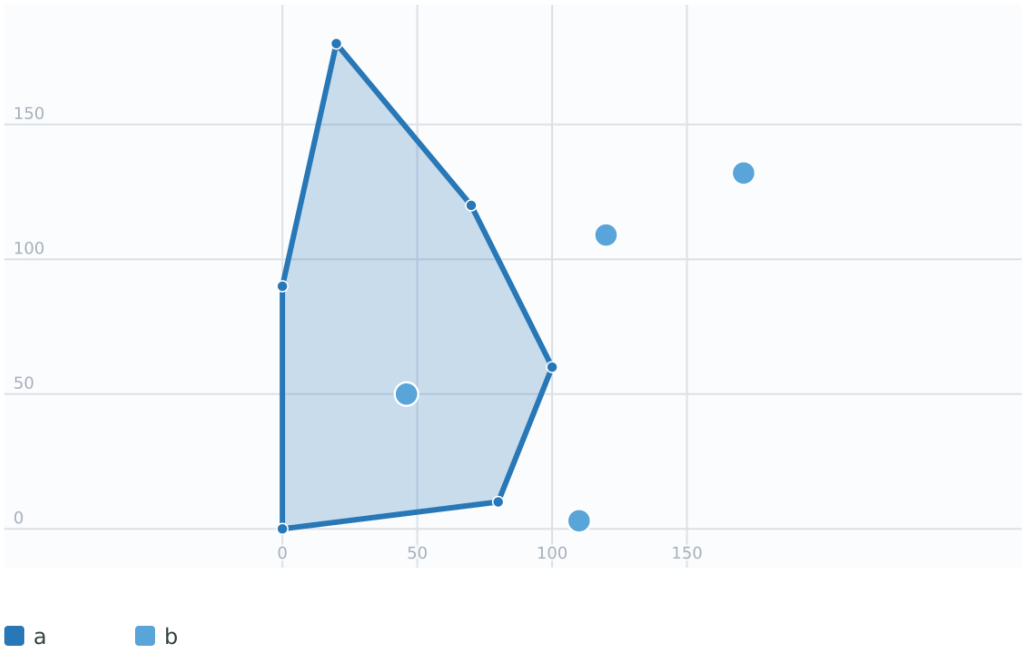


A Polygon crosses a MultiPoint when some, but not all, points lie in its interior.

Code

```
SELECT ST_Crosses(a, b)
FROM (SELECT
  'POLYGON ((30 190,10 100,10 10,90 20,110 70,80 130,30 190))'::geometry AS a,
  'MULTIPOINT (56 60,120 13,130 119,181 142)'::geometry AS b
) AS example;
```

Figure

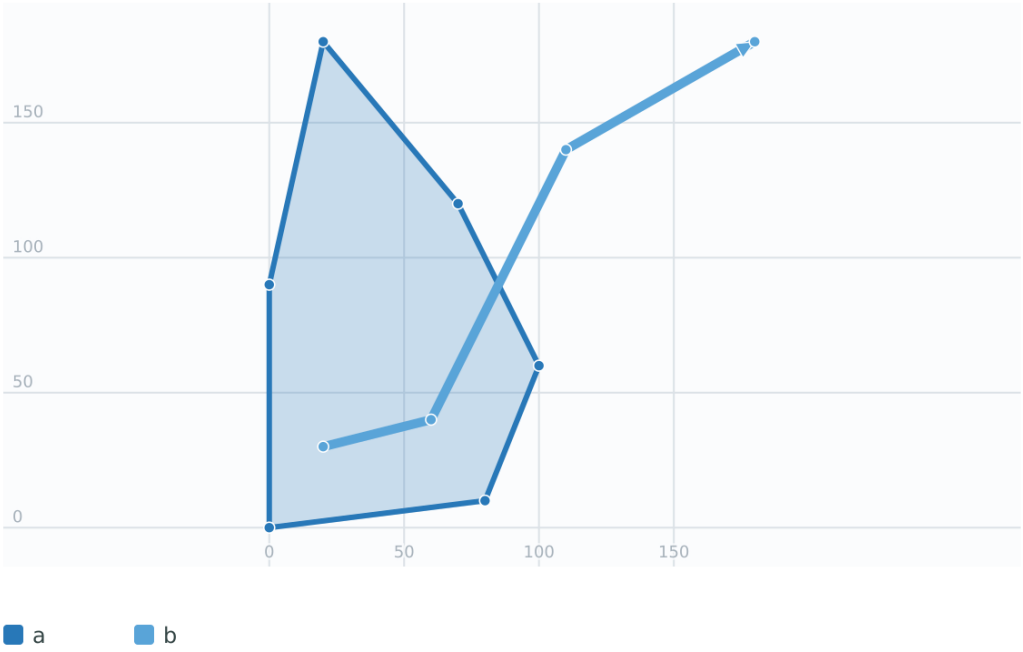


A LineString crosses a Polygon when part of the line lies inside and part outside.

Code

```
SELECT ST_Crosses(a, b)
FROM (SELECT
  'POLYGON ((30 190,10 100,10 10,90 20,110 70,80 130,30 190))'::geometry AS a,
  'LINESTRING (30 40,70 50,120 150,190 190)'::geometry AS b
) AS example;
```

Figure

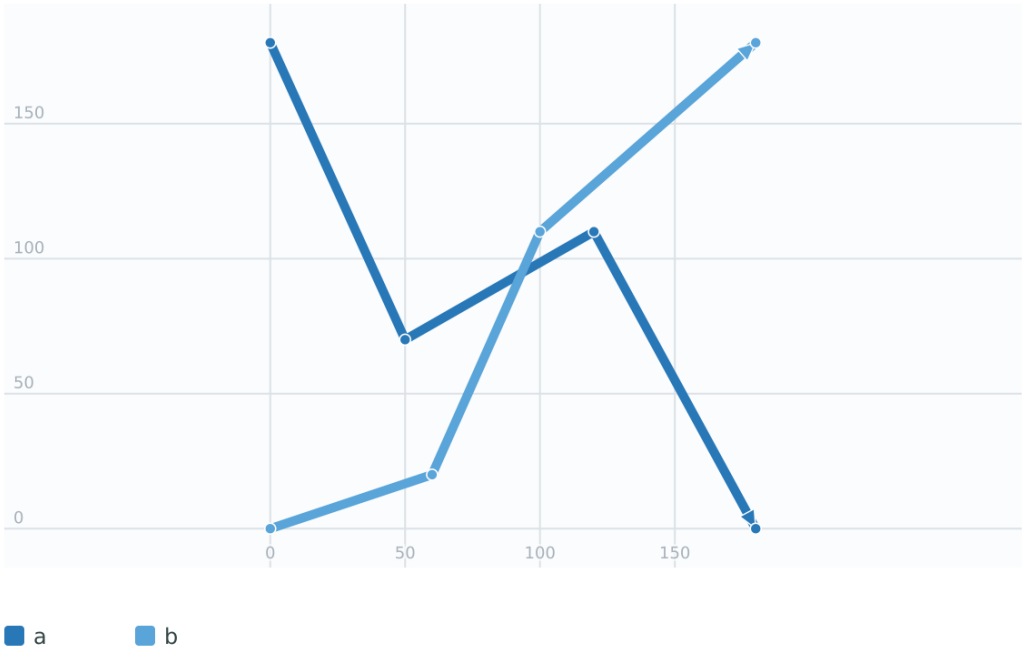


Two LineStrings cross when they share an interior point.

Code

```
SELECT ST_Crosses(a, b)
FROM (SELECT
  'LINESTRING (10 190,60 80,130 120,190 10)'::geometry AS a,
  'LINESTRING (10 10,70 30,110 120,190 190)'::geometry AS b
) AS example;
```

Figure



Consider a situation where a user has two tables: a table of roads and a table of highways.

Code

```
CREATE TABLE roads (
  id serial NOT NULL,
  geom geometry,
  CONSTRAINT roads_pkey PRIMARY KEY ( ↩
    road_id)
);
```

Code

```
CREATE TABLE highways (
  id serial NOT NULL,
  the_gem geometry,
  CONSTRAINT roads_pkey PRIMARY KEY ( ↩
    road_id)
);
```

To determine a list of roads that cross a highway, use a query similar to:

Code

```
SELECT roads.id
FROM roads, highways
WHERE ST_Crosses(roads.geom, highways.geom);
```

□□

ST_Contains, ST_Overlaps

7.11.1.7 ST_Disjoint

ST_Disjoint — Tests if two geometries have no points in common

□□

boolean **ST_Disjoint**(geometry A , geometry B);

□□

Returns `true` if two geometries are disjoint. Geometries are disjoint if they have no point in common.

If any other spatial relationship is true for a pair of geometries, they are not disjoint. Disjoint implies that ST_Intersects is false.

In mathematical terms: *ST_Disjoint(A, B)* ⇔ *A* ∩ *B* = ∅

Enhanced: 3.0.0 enabled support for `GEOMETRYCOLLECTION`

GEOS □□□□□



□□

This function call does not use indexes. A negated ST_Intersects predicate can be used as a more performant alternative that uses indexes: `ST_Disjoint(A,B) = NOT ST_Intersects(A,B)`



□□

NOTE: this is the "allowable" version that returns a boolean, not an integer.



This method implements the OGC Simple Features Implementation Specification for SQL 1.1. s2.1.1.2 //s2.1.13.3 - a.Relate(b, 'FF*FF****')



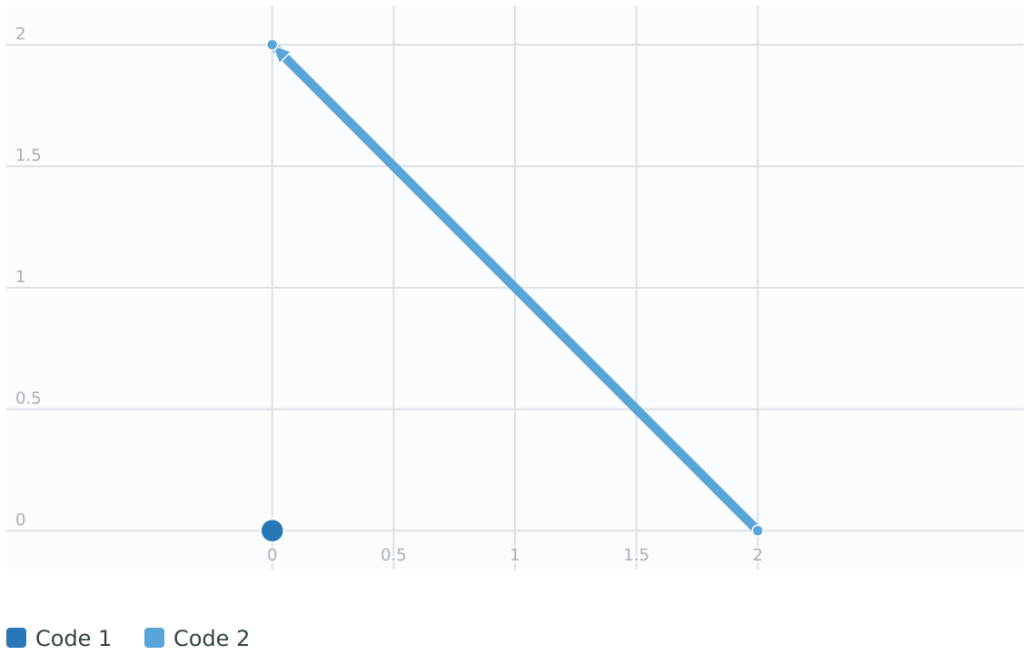
This method implements the SQL/MM specification. SQL-MM 3: 5.1.26

□□

Code

```
SELECT ST_Disjoint('POINT(0 0)'::geometry, 'LINESTRING ( 2 0,0 2 )'::geometry);
```

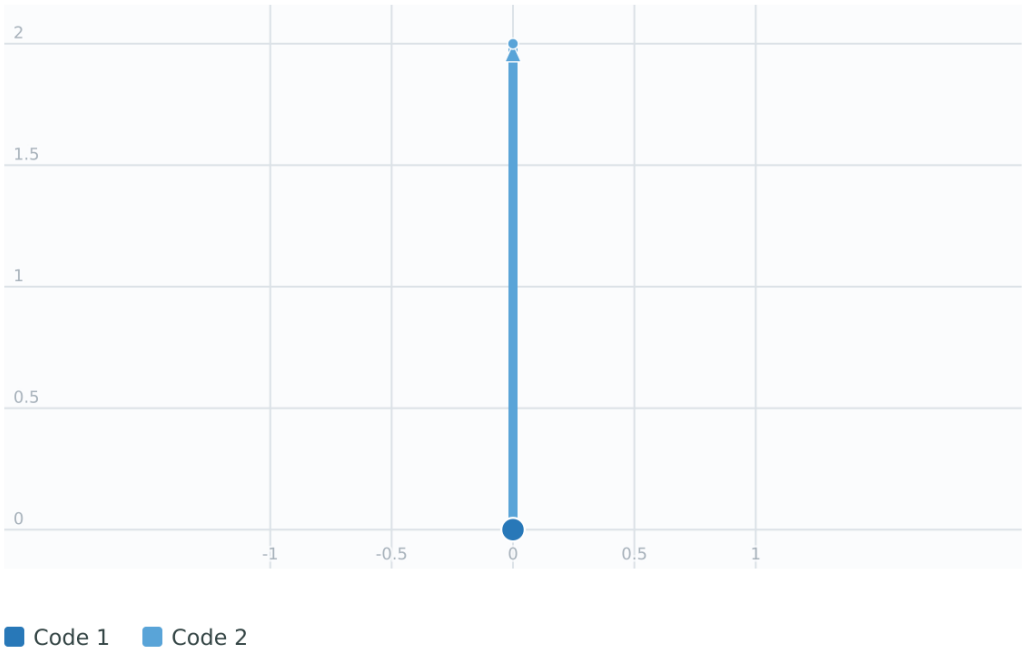
Figure



Code

```
SELECT ST_Disjoint('POINT(0 0)'::geometry, 'LINESTRING ( 0 0,0 2 )'::geometry);
```

Figure



☒☒

ST_Intersects

7.11.1.8 ST_Equals

ST_Equals — Tests if two geometries include the same set of points

☒☒

boolean **ST_Equals**(geometry A, geometry B);

☒☒

Returns `true` if the given geometries are ”topologically equal”. Use this for a ’better’ answer than ’=’. Topological equality means that the geometries have the same dimension, and their point-sets occupy the same space. This means that the order of vertices may be different in topologically equal geometries. To verify the order of points is consistent use ST_OrderingEquals (it must be noted ST_OrderingEquals is a little more stringent than simply verifying order of points are the same).

In mathematical terms: *ST_Equals(A, B) ⇔ A = B*

The following relation holds: *ST_Equals(A, B) ⇔ ST_Within(A,B) ∧ ST_Within(B,A)*

Enhanced: 3.0.0 enabled support for `GEOMETRYCOLLECTION`

This method implements the OGC Simple Features Implementation Specification for SQL 1.1. s2.1.1.2

This method implements the SQL/MM specification. SQL-MM 3: 5.1.24

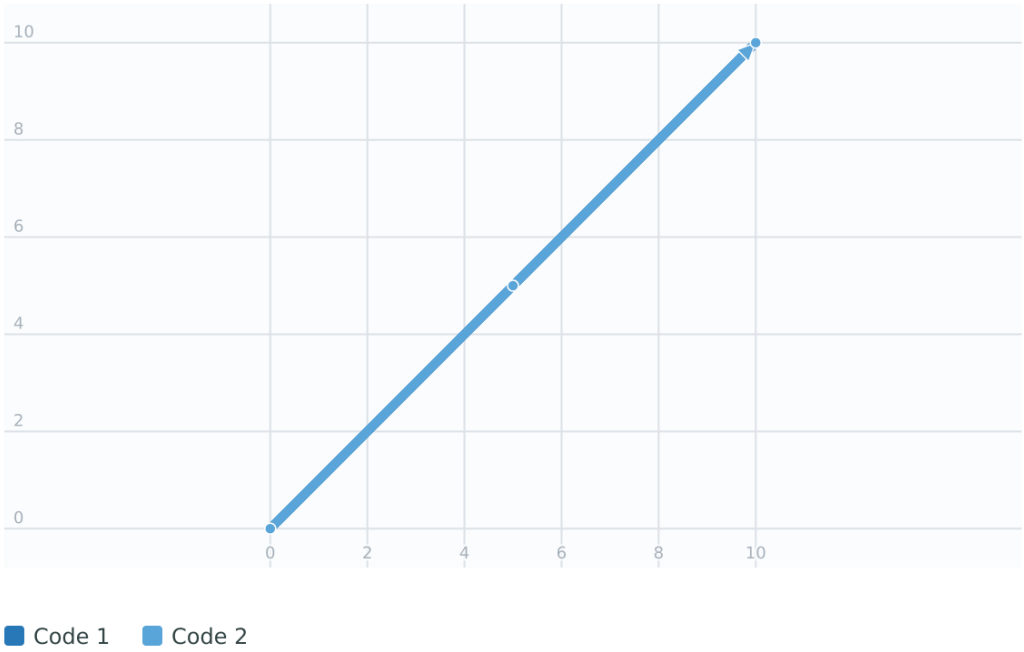
Changed: 2.2.0 Returns true even for invalid geometries if they are binary equal

☒☒

Code

```
SELECT ST_Equals('LINESTRING(0 0,10 10)',
'LINESTRING(0 0,5 5,10 10)');
```

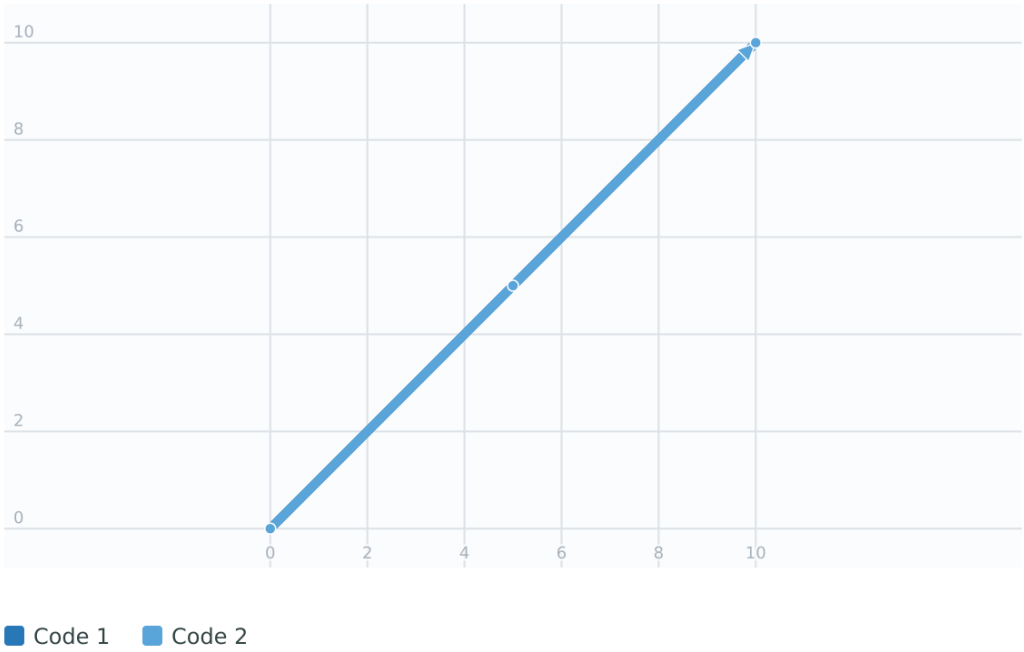
Figure



Code

```
SELECT ST_Equals(ST_Reverse('LINESTRING(0 0,10 10)'),
'LINESTRING(0 0,5 5,10 10)');
```

Figure



□□

ST_IsValid, ST_OrderingEquals, ST_Reverse, ST_Within

7.11.1.9 ST_Intersects

ST_Intersects — Tests if two geometries intersect (they have at least one point in common)

□□

boolean **ST_Intersects**(geometry geomA , geometry geomB);
boolean **ST_Intersects**(geography geogA , geography geogB);

□□

Returns `true` if two geometries intersect. Geometries intersect if they have any point in common.

For geography, a distance tolerance of 0.00001 meters is used (so points that are very close are considered to intersect).

In mathematical terms: *ST_Intersects(A, B)* ⇔ *A* ∩ *B* ≠ ∅

Geometries intersect if their DE-9IM Intersection Matrix matches one of:

- T********
- *T*******
- ***T*****
- ****T****

Spatial intersection is implied by all the other spatial relationship tests, except ST_Disjoint, which tests that geometries do NOT intersect.



□□
This function automatically includes a bounding box comparison that makes use of any spatial indexes that are available on the geometries.

Changed: 3.0.0 SFCGAL version removed and native support for 2D TINS added.

Enhanced: 2.5.0 Supports GEOMETRYCOLLECTION.

Enhanced: 2.3.0 Enhancement to PIP short-circuit extended to support MultiPoints with few points. Prior versions only supported point in polygon.

Performed by the GEOS module (for geometry), geography is native

Availability: 1.5 support for geography was introduced.



□□
For geography, this function has a distance tolerance of about 0.00001 meters and uses the sphere rather than spheroid calculation.



□□
NOTE: this is the ”allowable” version that returns a boolean, not an integer.

 This method implements the OGC Simple Features Implementation Specification for SQL 1.1. s2.1.1.2 //s2.1.13.3 - ST_Intersects(g1, g2) --> Not (ST_Disjoint(g1, g2))

 This method implements the SQL/MM specification. SQL-MM 3: 5.1.27

 This method supports Circular Strings and Curves.

 This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).

□□□□

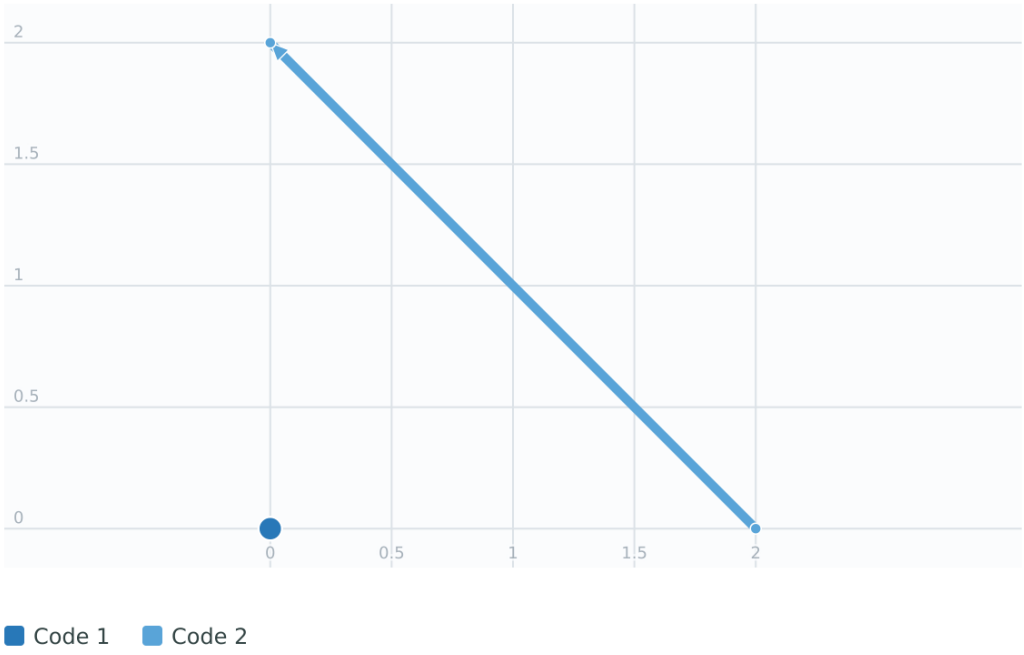
Code

```
SELECT ST_Intersects('POINT(0 0)'::geometry, 'LINESTRING ( 2 0,0 2 )'::geometry);
```

Output

```
f
```

Figure



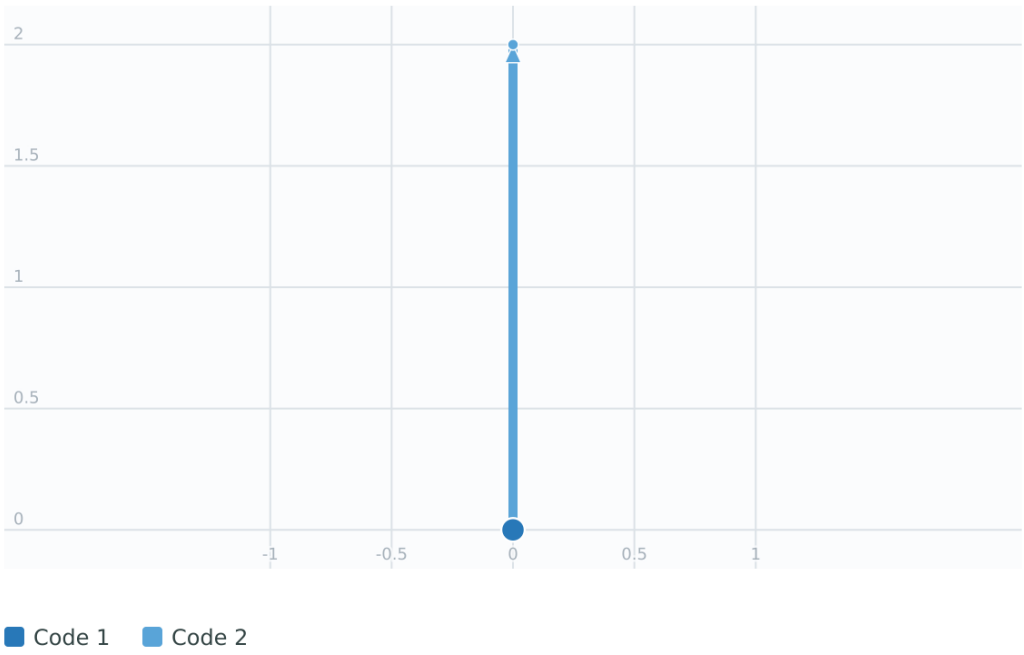
Code

```
SELECT ST_Intersects('POINT(0 0)'::geometry, 'LINESTRING ( 0 0,0 2 )'::geometry);
```

Output

```
t
```

Figure



This example looks up rows in a table; for faster lookup, the geometry column should have a GiST index.

Code

```
SELECT id, name FROM cities WHERE ST_Intersects(geom, 'SRID=4326;POLYGON((28 53,27.707 ↩
    52.293,27 52,26.293 52.293,26 53,26.293 53.707,27 54,27.707 53.707,28 53))');
```

Output

```
id | name
----+-------
  2 | Minsk
(1 row)
```

□□□□□

Code

```
SELECT ST_Intersects('SRID=4326;LINESTRING(-43.23456 72.4567,-43.23456 72.4568)'::geography ↩
    ,
'SRID=4326;POINT(-43.23456 72.4567772)'::geography
);
```

Output

```
t
```

□□

&&, ST_3DIntersects, ST_Disjoint

7.11.1.10 ST_LineCrossingDirection

ST_LineCrossingDirection — Returns a number indicating the crossing behavior of two LineStrings

□□

integer **ST_LineCrossingDirection**(geometry linestringA, geometry linestringB);

□□

Given two linestrings returns an integer between -3 and 3 indicating what kind of crossing behavior exists between them. 0 indicates no crossing. This is only supported for `LINESTRING`s.

The crossing number has the following meaning:

- 0: LINE NO CROSS
- -1: LINE CROSS LEFT
- 1: LINE CROSS RIGHT
- -2: LINE MULTICROSS END LEFT
- 2: LINE MULTICROSS END RIGHT
- -3: LINE MULTICROSS END SAME FIRST LEFT
- 3: LINE MULTICROSS END SAME FIRST RIGHT

Availability: 1.4

□□

Example: LINE CROSS LEFT and LINE CROSS RIGHT

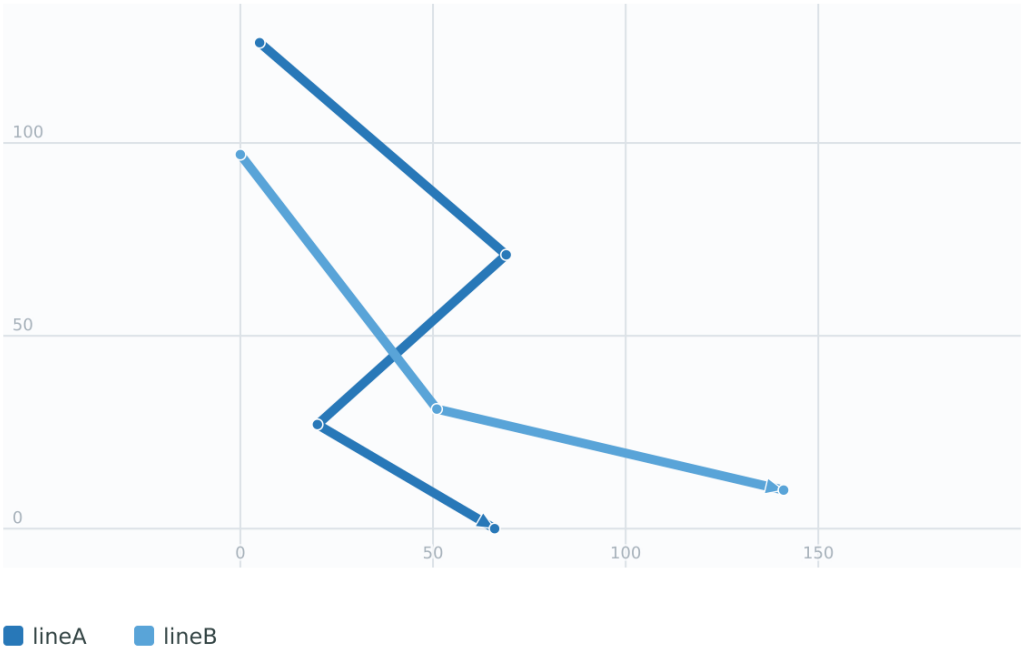
Code

```
SELECT ST_LineCrossingDirection(lineA, lineB) As A_cross_B,
       ST_LineCrossingDirection(lineB, lineA) As B_cross_A
FROM (SELECT
  'LINESTRING(25 169,89 114,40 70,86 43)'::geometry As lineA,
  'LINESTRING (20 140,71 74,161 53)'::geometry As lineB
  ) As foo;
```

Output

```
A_cross_B | B_cross_A
----------+----------
       -1 |         1
```

Figure



Example: LINE MULTICROSS END SAME FIRST LEFT and LINE MULTICROSS END SAME FIRST RIGHT

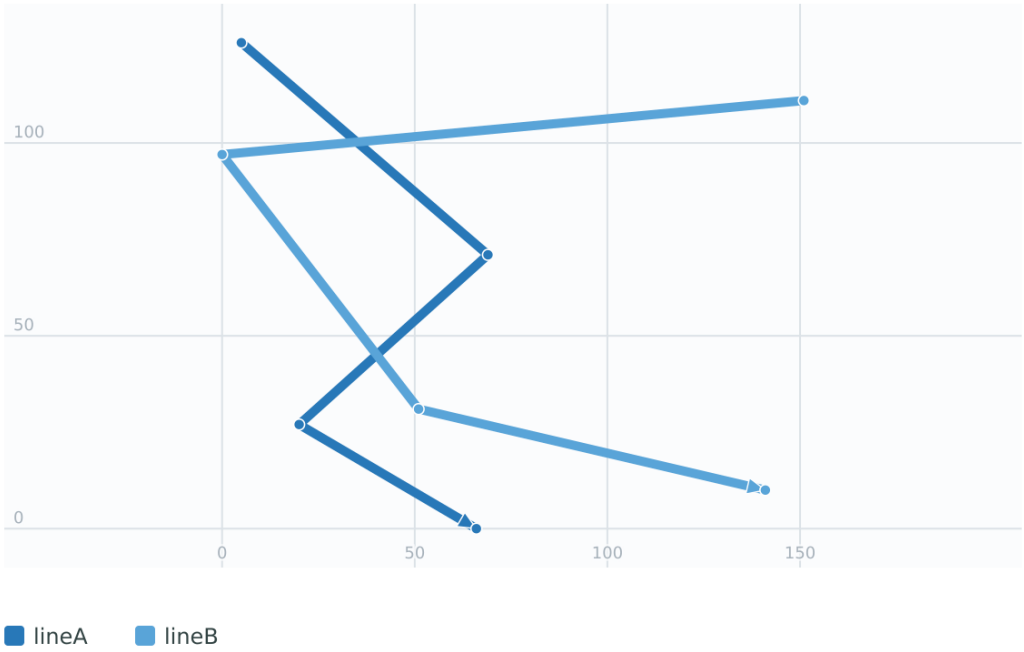
Code

```
SELECT ST_LineCrossingDirection(lineA, lineB) As A_cross_B,
       ST_LineCrossingDirection(lineB, lineA) As B_cross_A
FROM (SELECT
 'LINESTRING(25 169,89 114,40 70,86 43)'::geometry As lineA,
 'LINESTRING(171 154,20 140,71 74,161 53)'::geometry As lineB
  ) As foo;
```

Output

```
A_cross_B | B_cross_A
----------+----------
        3 |        -3
```

Figure



Example: LINE MULTICROSS END LEFT and LINE MULTICROSS END RIGHT

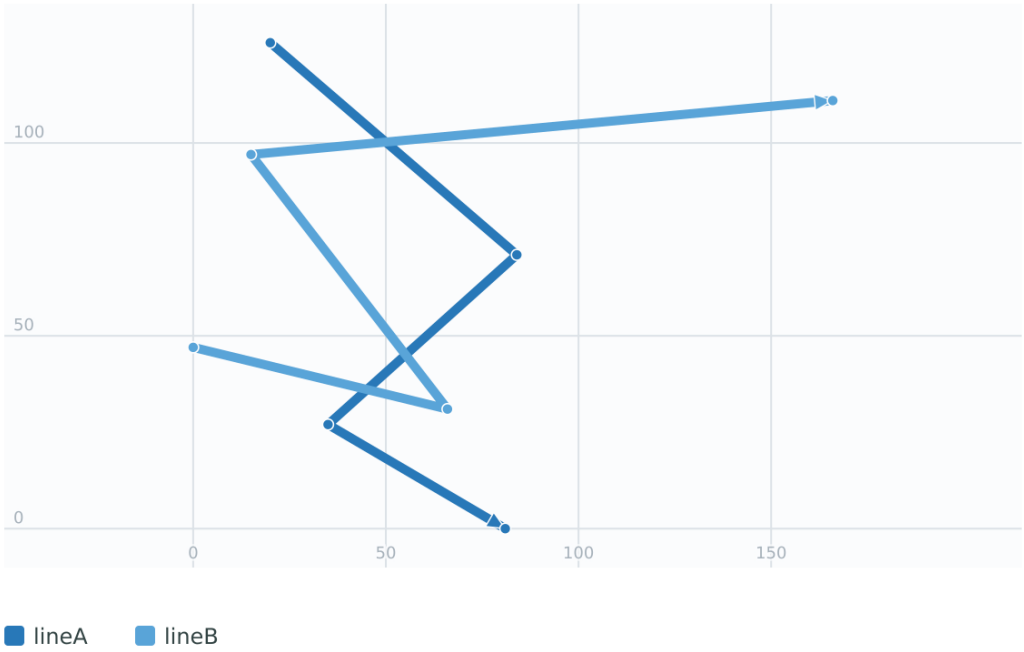
Code

```
SELECT ST_LineCrossingDirection(lineA, lineB) As A_cross_B,
       ST_LineCrossingDirection(lineB, lineA) As B_cross_A
FROM (SELECT
  'LINESTRING(25 169,89 114,40 70,86 43)'::geometry As lineA,
  'LINESTRING(5 90,71 74,20 140,171 154)'::geometry As lineB
  ) As foo;
```

Output

```
A_cross_B | B_cross_A
----------+----------
       -2 |         2
```

Figure



Example: Finds all streets that cross

Code

```
SELECT s1.gid, s2.gid, ST_LineCrossingDirection(s1.geom, s2.geom)
  FROM streets s1 CROSS JOIN streets s2
        ON (s1.gid != s2.gid AND s1.geom && s2.geom )
WHERE ST_LineCrossingDirection(s1.geom, s2.geom)
> 0;
```

□□

ST_Crosses

7.11.1.11 ST_OrderingEquals

ST_OrderingEquals — Tests if two geometries represent the same geometry and have points in the same directional order

□□

boolean **ST_OrderingEquals**(geometry A, geometry B);

□□

ST_OrderingEquals compares two geometries and returns t (TRUE) if the geometries are equal and the coordinates are in the same order; otherwise it returns f (FALSE).



□□
This function is implemented as per the ArcSDE SQL specification rather than SQL-MM. http://edndoc.esri.com/arcsde/9.1/sql_api/sqlapi3.htm#ST_OrderingEquals



This method implements the SQL/MM specification. SQL-MM 3: 5.1.43

□□

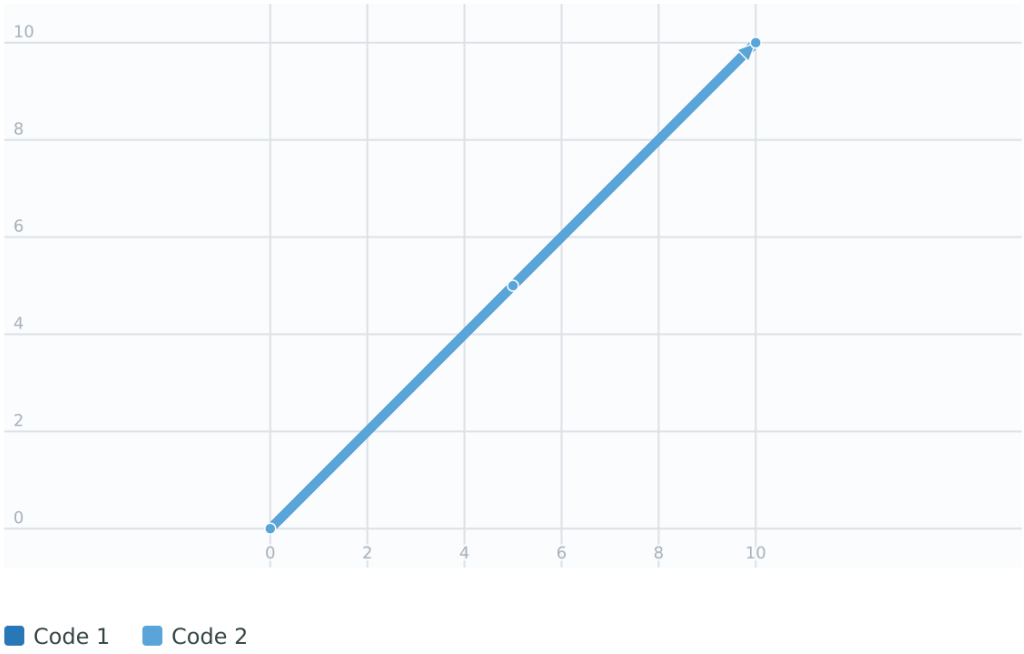
Code

```
SELECT ST_OrderingEquals('LINESTRING(0 0,10 10)',
'LINESTRING(0 0,5 5,10 10)');
```

Output

```
f
```

Figure



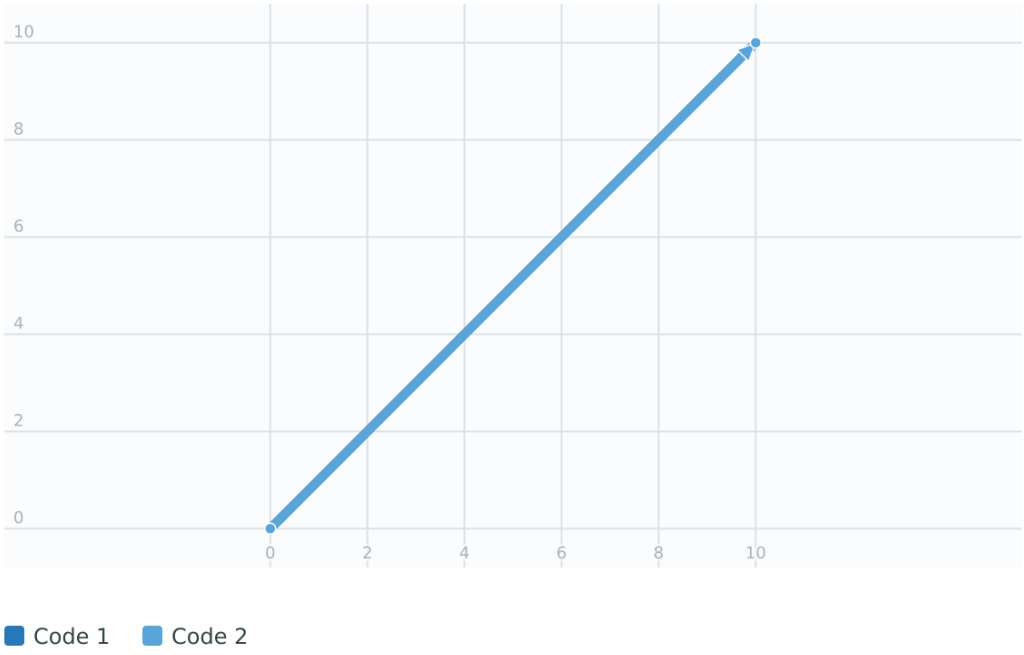
Code

```
SELECT ST_OrderingEquals('LINESTRING(0 0,10 10)',
'LINESTRING(0 0,10 10)');
```

Output

```
t
```

Figure



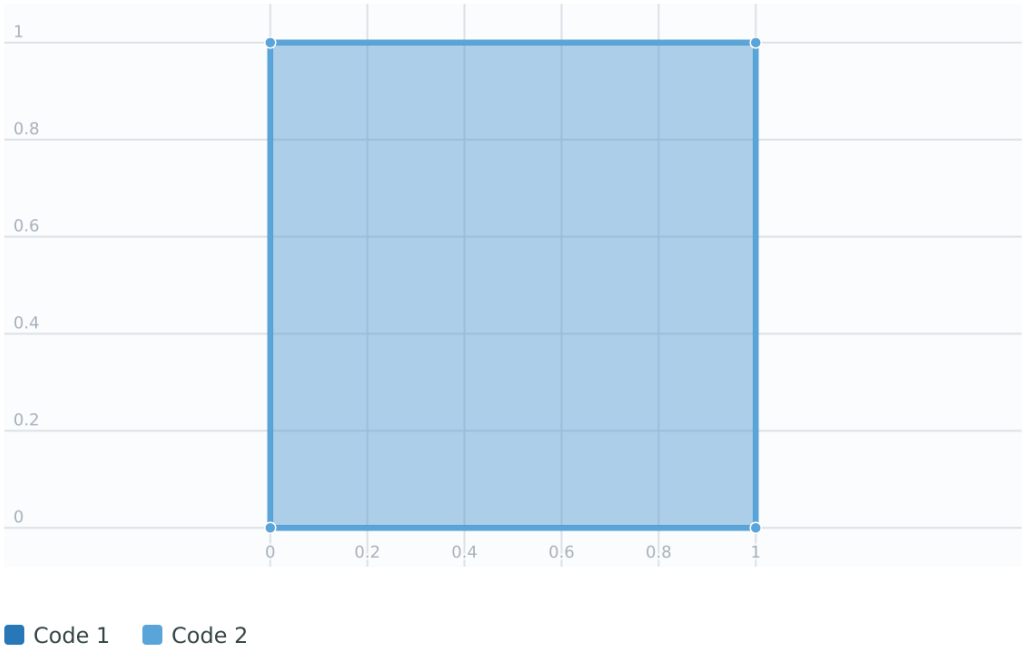
Code

```
SELECT ST_OrderingEquals('POLYGON((0 0,0 1,1 1,1 0,0 0))',
'POLYGON((0 0,1 0,1 1,0 1,0 0))');
```

Output

```
f
```

Figure



□□

&&, ST_Equals, ST_Reverse

7.11.1.12 ST_Overlaps

ST_Overlaps — Tests if two geometries have the same dimension and intersect, but each has at least one point not in the other

□□

boolean **ST_Overlaps**(geometry A, geometry B);

□□

Returns TRUE if geometry A and B "spatially overlap". Two geometries overlap if they have the same dimension, their interiors intersect in that dimension. and each has at least one point inside the other (or equivalently, neither one covers the other). The overlaps relation is symmetric and irreflexive.

In mathematical terms: *ST_Overlaps(A, B) ⇔ (dim(A) = dim(B) = dim(Int(A) □ Int(B))) ∧ (A □ B ≠ A) ∧ (A □ B ≠ B)*



□□
This function automatically includes a bounding box comparison that makes use of any spatial indexes that are available on the geometries. To avoid index use, use the function _ST_Overlaps.

GEOS □□□□□

Enhanced: 3.0.0 enabled support for GEOMETRYCOLLECTION

NOTE: this is the "allowable" version that returns a boolean, not an integer.

This method implements the OGC Simple Features Implementation Specification for SQL 1.1. s2.1.1.2 // s2.1.13.3

This method implements the SQL/MM specification. SQL-MM 3: 5.1.32

□□

ST_Overlaps returns TRUE in the following situations:

Two MultiPoints overlap when they share some, but not all, points.

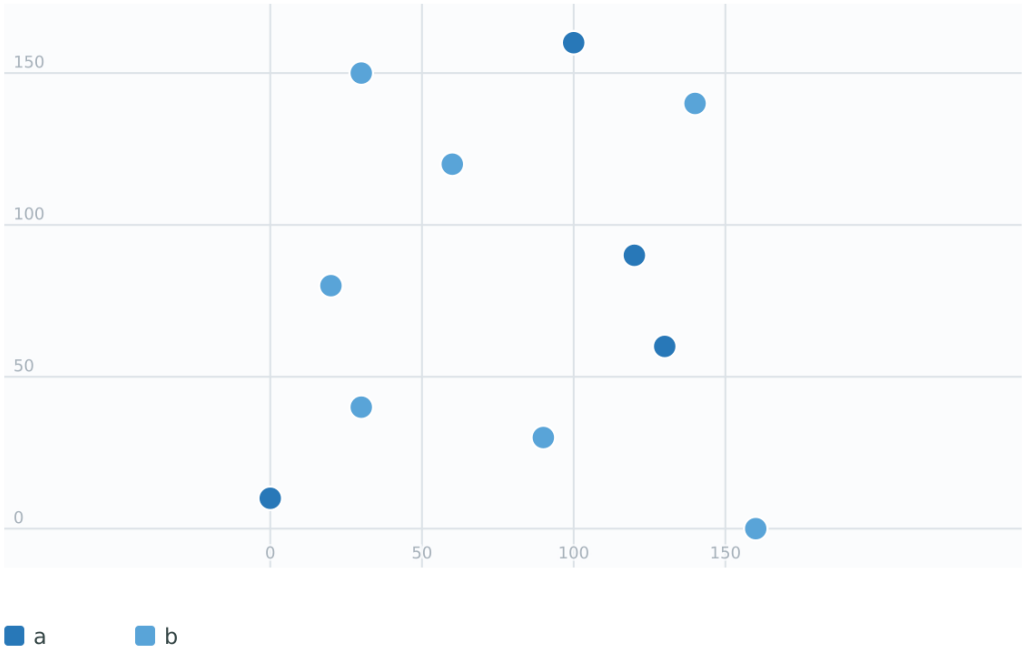
Code

```
SELECT ST_Overlaps(a, b)
FROM (SELECT
  'MULTIPOINT ((20 20),(40 90),(80 130),(140 100),(110 40),(120 170),(150 70))'::geometry ↩
    AS a,
  'MULTIPOINT ((180 10),(40 90),(80 130),(160 150),(110 40),(50 50),(50 160))'::geometry AS ↩
     b
) AS example;
```

Output

```
t
```

Figure



Two LineStrings overlap along part of their paths.

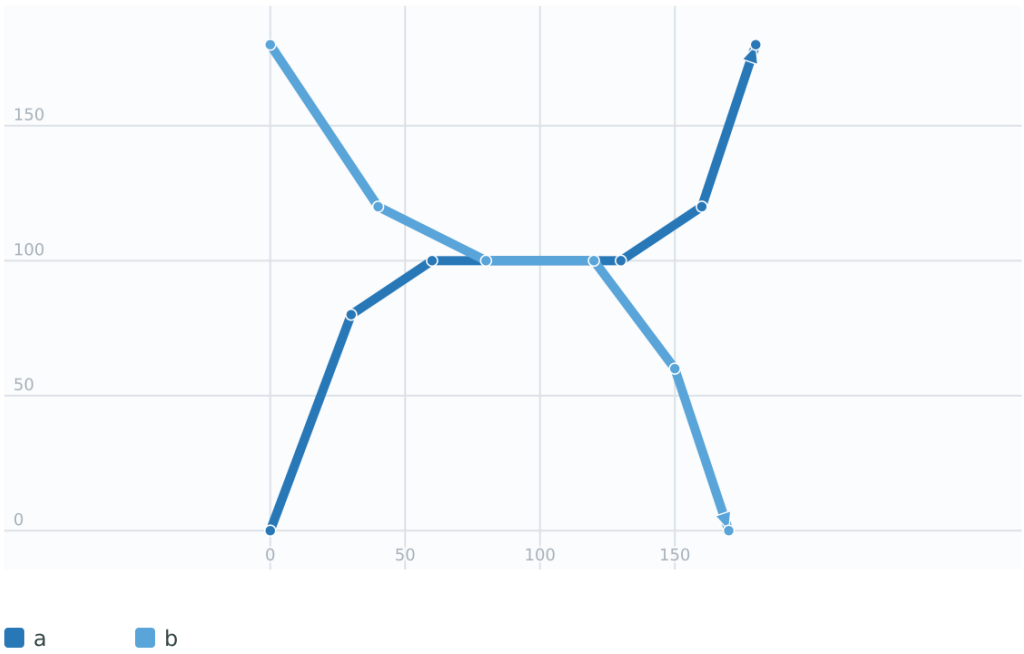
Code

```
SELECT ST_Overlaps(a, b)
FROM (SELECT
  'LINESTRING (10 10,40 90,70 110,140 110,170 130,190 190)'::geometry AS a,
  'LINESTRING (10 190,50 130,90 110,130 110,160 70,180 10)'::geometry AS b
) AS example;
```

Output

```
t
```

Figure



Two Polygons overlap where their interiors intersect and neither contains the other.

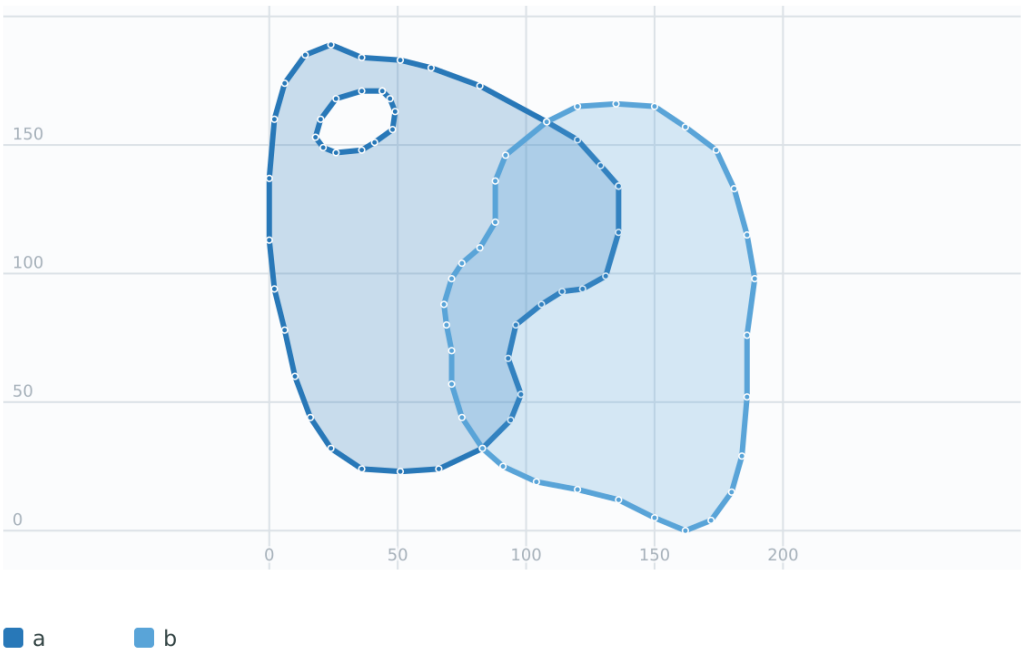
Code

```
SELECT ST_Overlaps(a, b)
FROM (SELECT
  'POLYGON ((140 140,140 122,135 105,126 100,118 99,110 94,100 86,97 73,102 59,98 49,87 ↩
    38,70 30,55 29,40 30,28 38,20 50,14 66,10 84,6 100,4 119,4 143,6 166,10 180,18 191,28 ↩
    195,40 190,55 189,67 186,86 179,112 165,124 158,133 148,140 140),(48 177,40 177,30 ↩
    174,24 166,22 159,25 155,30 153,40 154,45 157,52 162,53 169,51 174,48 177))'::geometry ↩
    AS a,
  'POLYGON ((75 63,79 50,87 38,95 31,108 25,124 22,140 18,154 11,166 6,176 10,184 21,188 ↩
    35,190 58,190 82,193 104,190 121,185 139,178 154,166 163,154 171,139 172,124 171,112 ↩
    165,96 152,92 142,92 126,86 116,79 110,75 104,72 94,73 86,75 76,75 63))'::geometry AS b
) AS example;
```

Output

```
t
```

Figure



A Point on a LineString is contained, but since it has lower dimension it does not overlap or cross.

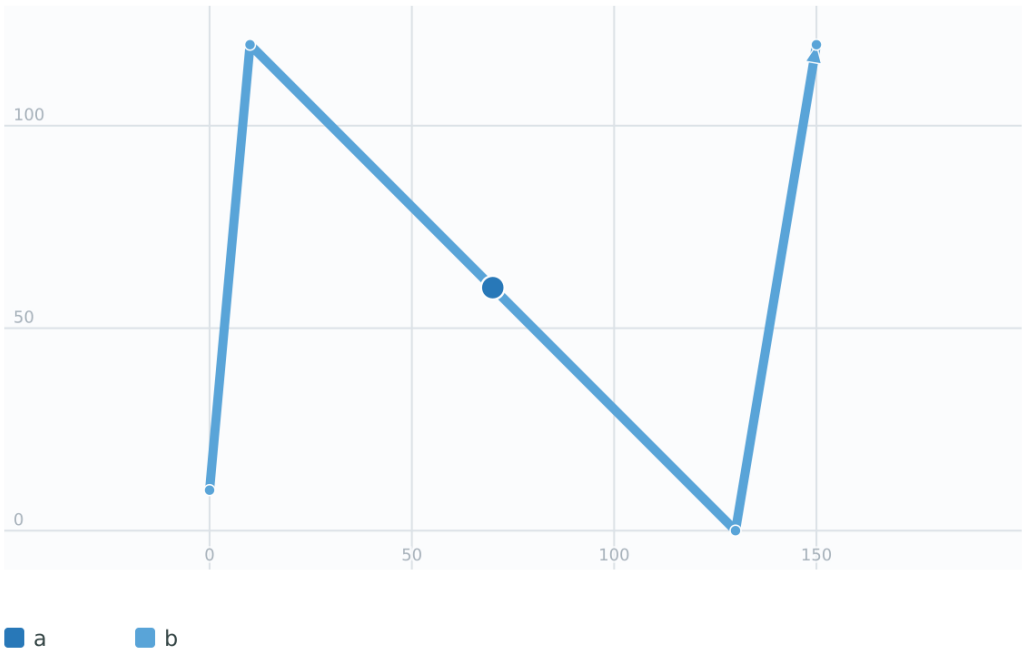
Code

```
SELECT ST_Overlaps(a, b) AS overlaps, ST_Crosses(a, b) AS crosses,
       ST_Intersects(a, b) AS intersects, ST_Contains(b, a) AS b_contains_a
FROM (SELECT 'POINT (100 100)'::geometry As a,
             'LINESTRING (30 50,40 160,160 40,180 160)'::geometry  AS b) AS t
```

Output

```
overlaps | crosses | intersects | b_contains_a
---------+---------+------------+-------------
f        | f       | t          | t
```

Figure



A LineString that partly covers a Polygon intersects and crosses, but does not overlap since it has different dimension.

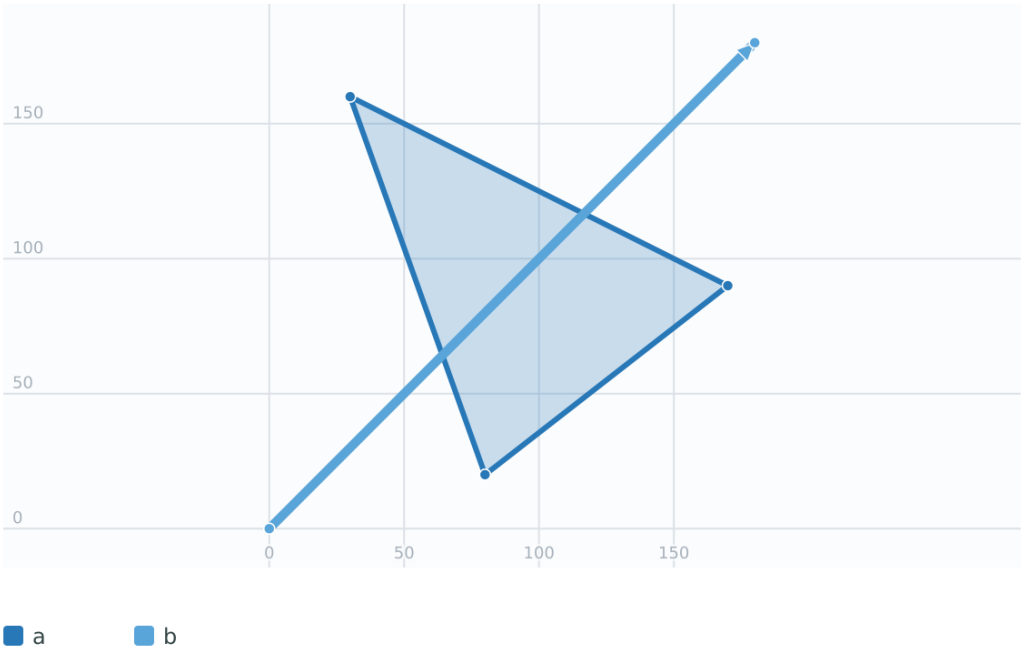
Code

```
SELECT ST_Overlaps(a, b) AS overlaps, ST_Crosses(a, b) AS crosses,
       ST_Intersects(a, b) AS intersects, ST_Contains(a, b) AS contains
FROM (SELECT 'POLYGON ((40 170,90 30,180 100,40 170))'::geometry AS a,
             'LINESTRING(10 10,190 190)'::geometry AS b) AS t;
```

Output

```
overlap | crosses | intersects | contains
---------+---------+------------+--------------
 f       | t       | t          | f
```

Figure



Two Polygons that intersect but with neither contained by the other overlap, but do not cross because their intersection has the same dimension.

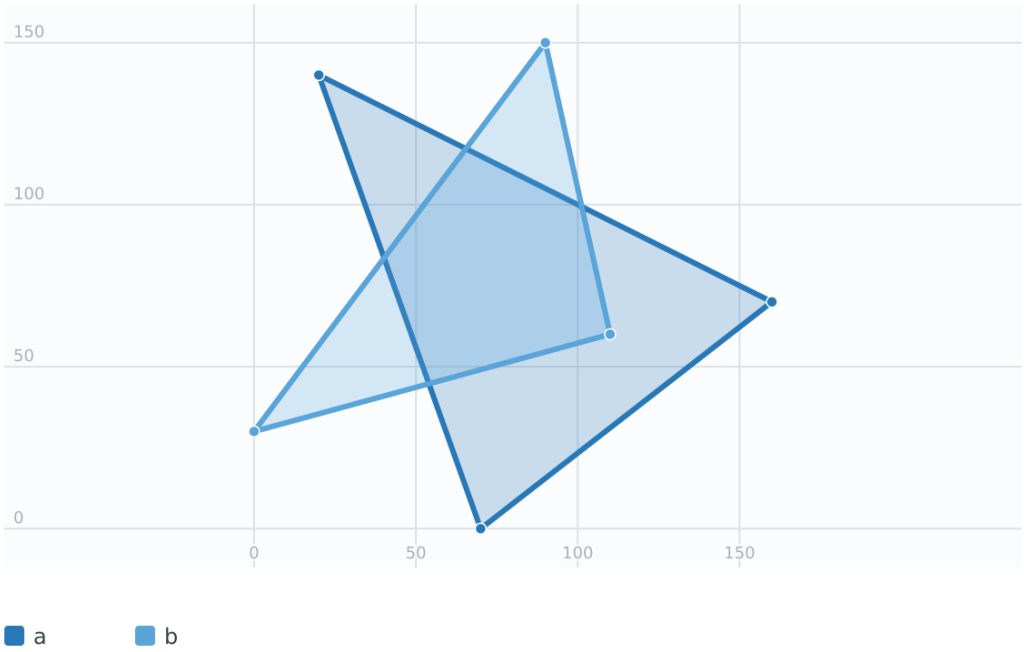
Code

```
SELECT ST_Overlaps(a, b) AS overlaps, ST_Crosses(a, b) AS crosses,
       ST_Intersects(a, b) AS intersects, ST_Contains(b, a) AS b_contains_a,
       ST_Dimension(a) AS dim_a, ST_Dimension(b) AS dim_b,
       ST_Dimension(ST_Intersection(a, b)) AS dim_int
FROM (SELECT 'POLYGON ((40 170,90 30,180 100,40 170))'::geometry AS a,
             'POLYGON ((110 180,20 60,130 90,110 180))'::geometry AS b) As t;
```

Output

```
overlaps | crosses | intersects | b_contains_a | dim_a | dim_b | dim_int
---------+---------+------------+--------------+-------+-------+----------
 t       | f       | t          | f            |     2 |     2 |       2
```

Figure



☒☒

ST_Contains, ST_Crosses, ST_Dimension, ST_Intersects

7.11.1.13 ST_Relate

ST_Relate — Tests if two geometries have a topological relationship matching an Intersection Matrix pattern, or computes their Intersection Matrix

☒☒

boolean **ST_Relate**(geometry geomA, geometry geomB, text intersectionMatrixPattern);
text **ST_Relate**(geometry geomA, geometry geomB);
text **ST_Relate**(geometry geomA, geometry geomB, integer boundaryNodeRule);

☒☒

These functions allow testing and evaluating the spatial (topological) relationship between two geometries, as defined by the Dimensionally Extended 9-Intersection Model (DE-9IM).

The DE-9IM is specified as a 9-element matrix indicating the dimension of the intersections between the Interior, Boundary and Exterior of two geometries. It is represented by a 9-character text string using the symbols 'F', '0', '1', '2' (e.g. `'FF1FF0102'`).

A specific kind of spatial relationship can be tested by matching the intersection matrix to an *intersection matrix pattern*. Patterns can include the additional symbols 'T' (meaning "intersection is non-empty") and '*' (meaning "any value"). Common spatial relationships are provided by the named functions ST_Contains, ST_ContainsProperly, ST_Covers, ST_CoveredBy, ST_Crosses, ST_Disjoint, ST_Equals, ST_Intersects, ST_Overlaps, ST_Touches, and ST_Within. Using an explicit pattern allows testing multiple conditions of intersects, crosses, etc in one step. It also allows testing spatial

relationships which do not have a named spatial relationship function. For example, the relationship "Interior-Intersects" has the DE-9IM pattern `T********`, which is not evaluated by any named predicate.

For more information refer to 5.1□.

Variant 1: Tests if two geometries are spatially related according to the given `intersectionMatrixPattern`



□□
Unlike most of the named spatial relationship predicates, this does NOT automatically include an index call. The reason is that some relationships are true for geometries which do NOT intersect (e.g. Disjoint). If you are using a relationship pattern that requires intersection, then include the && index call.



□□
It is better to use a named relationship function if available, since they automatically use a spatial index where one exists. Also, they may implement performance optimizations which are not available with full relate evaluation.

Variant 2: Returns the DE-9IM matrix string for the spatial relationship between the two input geometries. The matrix string can be tested for matching a DE-9IM pattern using ST_RelateMatch.

Variant 3: Like variant 2, but allows specifying a **Boundary Node Rule**. A boundary node rule allows finer control over whether the endpoints of MultiLineStrings are considered to lie in the DE-9IM Interior or Boundary. The `boundaryNodeRule` values are:

- 1: **OGC-Mod2** - line endpoints are in the Boundary if they occur an odd number of times. This is the rule defined by the OGC SFS standard, and is the default for `ST_Relate`.
- 2: **Endpoint** - all endpoints are in the Boundary.
- 3: **MultivalentEndpoint** - endpoints are in the Boundary if they occur more than once. In other words, the boundary is all the "attached" or "inner" endpoints (but not the "unattached/outer" ones).
- 4: **MonovalentEndpoint** - endpoints are in the Boundary if they occur only once. In other words, the boundary is all the "unattached" or "outer" endpoints.

This function is not in the OGC spec, but is implied. see s2.1.13.2

 This method implements the OGC Simple Features Implementation Specification for SQL 1.1. s2.1.1.2 // s2.1.13.3

 This method implements the SQL/MM specification. SQL-MM 3: 5.1.25

GEOS □□□□□

Enhanced: 2.0.0 - added support for specifying boundary node rule.

Enhanced: 3.0.0 enabled support for `GEOMETRYCOLLECTION`

□□

Using the boolean-valued function to test spatial relationships.

Code

```
SELECT ST_Relate('POINT(1 2)', ST_Buffer('POINT(1 2)', 2), '0FFFFF212');
```

Output

```
t
```

Code

```
SELECT ST_Relate('POINT(1 2)', ST_Buffer('POINT(1 2)', 2), '*FF*FF212');
```

Output

```
t
```

Testing a custom spatial relationship pattern as a query condition, with && to enable using a spatial index.

Find compounds that properly intersect (not just touch) a poly (Interior Intersects)

Code

```
SELECT c.* , p.name As poly_name
    FROM polys AS p
    INNER JOIN compounds As c
          ON c.geom && p.geom
             AND ST_Relate(p.geom, c.geom, 'T********');
```

Computing the intersection matrix for spatial relationships.

Code

```
SELECT ST_Relate('POINT(1 2)',
ST_Buffer('POINT(1 2)', 2));
```

Output

```
0FFFFF212
```

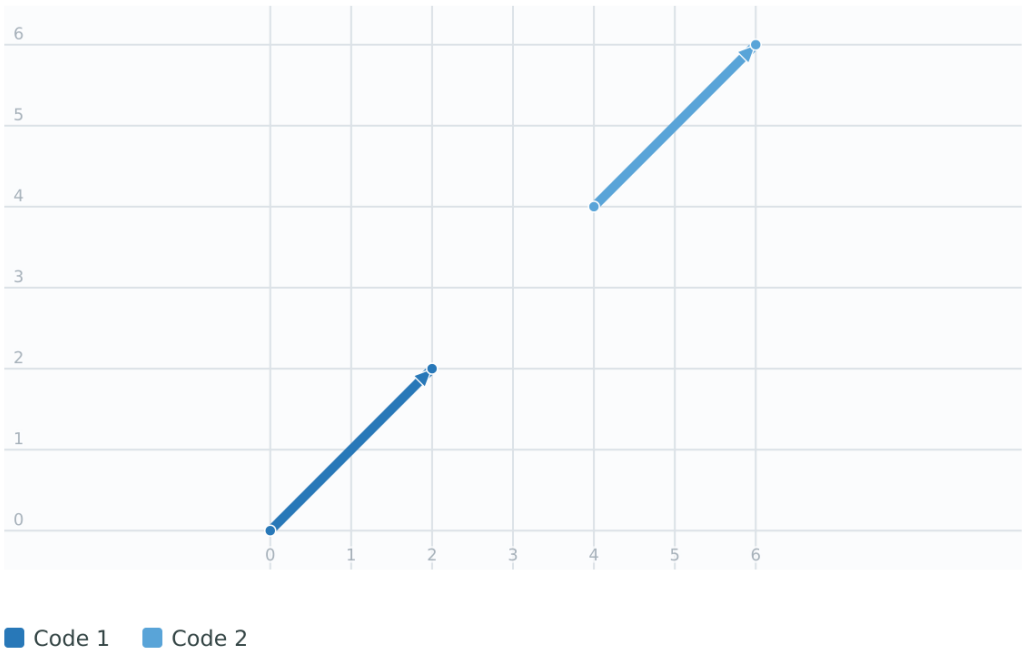
Code

```
SELECT ST_Relate('LINESTRING(1 2,3 4)',
'LINESTRING(5 6,7 8)' );
```

Output

```
FF1FF0102
```

Figure



Using different Boundary Node Rules to compute the spatial relationship between a LineString and a MultiLineString with a duplicate endpoint `(3 3)`:

- Using the **OGC-Mod2** rule (1) the duplicate endpoint is in the **interior** of the MultiLineString, so the DE-9IM matrix entry [aB:bI] is `0` and [aB:bB] is `F`.
- Using the **Endpoint** rule (2) the duplicate endpoint is in the **boundary** of the MultiLineString, so the DE-9IM matrix entry [aB:bI] is `F` and [aB:bB] is `0`.

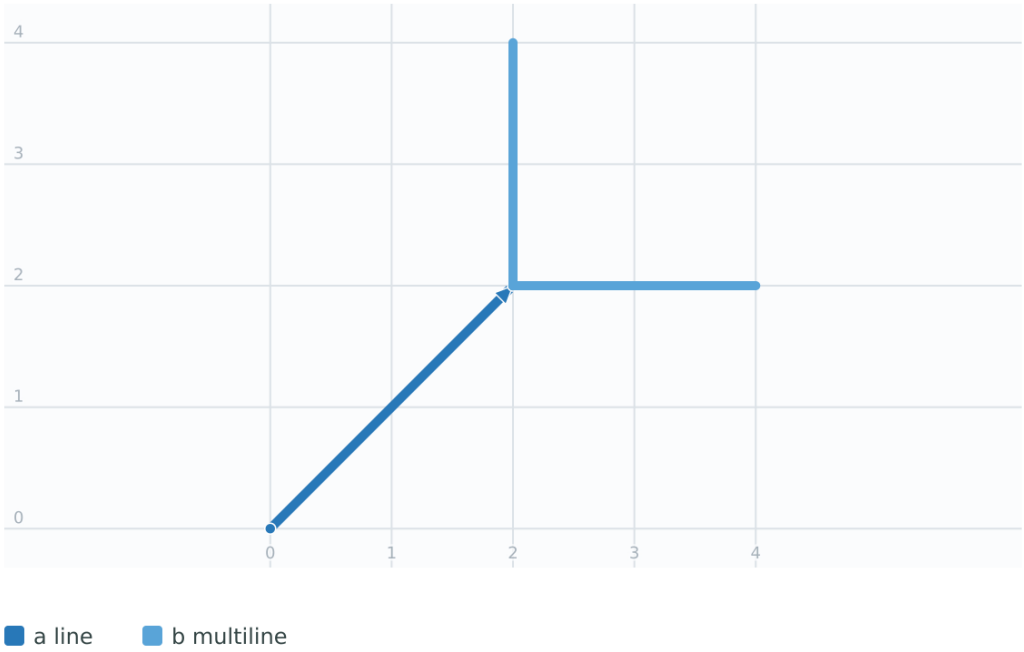
Code

```
WITH data AS (SELECT
  'LINESTRING(1 1,3 3)'::geometry AS a_line,
  'MULTILINESTRING((3 3,3 5),(3 3,5 3))'::geometry AS b_multiline
)
SELECT ST_Relate(a_line, b_multiline, 1) AS bnr_mod2,
       ST_Relate(a_line, b_multiline, 2) AS bnr_endpoint
    FROM data;
```

Output

```
bnr_mod2  | bnr_endpoint
----------+-------------
 FF10F0102 | FF1F00102
```

Figure



☒☒

5.1☒, ST_RelateMatch, ST_Contains, ST_ContainsProperly, ST_Covers, ST_CoveredBy, ST_Crosses, ST_Disjoint, ST_Equals, ST_Intersects, ST_Overlaps, ST_Touches, ST_Within

7.11.1.14 ST_RelateMatch

ST_RelateMatch — Tests if a DE-9IM Intersection Matrix matches an Intersection Matrix pattern

☒☒

boolean **ST_RelateMatch**(text intersectionMatrix, text intersectionMatrixPattern);

☒☒

Tests if a Dimensionally Extended 9-Intersection Model (DE-9IM) `intersectionMatrix` value satisfies an `intersectionMatrixPattern`. Intersection matrix values can be computed by ST_Relate.

For more information refer to 5.1☒.

GEOS ☒☒☒☒☒

2.0.0 ☒☒☒☒☒☒☒☒☒☒☒☒.

☒☒

Code

```
SELECT ST_RelateMatch('101202FFF', 'TTTTTTFFF') ;
```

Output

```
t
```

Patterns for common spatial relationships matched against intersection matrix values, for a line in various positions relative to a polygon

Code

```
SELECT pat.name AS relationship, pat.val AS pattern,
       mat.name AS position, mat.val AS matrix,
       ST_RelateMatch(mat.val, pat.val) AS match
    FROM (VALUES ( 'Equality', 'T1FF1FFF1' ),
                 ( 'Overlaps', 'T*T***T**' ),
                 ( 'Within', 'T*F**F***' ),
                 ( 'Disjoint', 'FF*FF****' )) AS pat(name, val)
    CROSS JOIN
        (VALUES  ('non-intersecting', 'FF1FF0212'),
                 ('overlapping', '1010F0212'),
                 ('inside', '1FF0FF212')) AS mat(name, val);
```

Output

```
relationship |  pattern  |     position     |  matrix   | match
-------------+-----------+------------------+-----------+-------
 Equality    | T1FF1FFF1 | non-intersecting | FF1FF0212 | f
 Equality    | T1FF1FFF1 | overlapping      | 1010F0212 | f
 Equality    | T1FF1FFF1 | inside           | 1FF0FF212 | f
 Overlaps    | T*T***T** | non-intersecting | FF1FF0212 | f
 Overlaps    | T*T***T** | overlapping      | 1010F0212 | t
 Overlaps    | T*T***T** | inside           | 1FF0FF212 | f
 Within      | T*F**F*** | non-intersecting | FF1FF0212 | f
 Within      | T*F**F*** | overlapping      | 1010F0212 | f
 Within      | T*F**F*** | inside           | 1FF0FF212 | t
 Disjoint    | FF*FF**** | non-intersecting | FF1FF0212 | t
 Disjoint    | FF*FF**** | overlapping      | 1010F0212 | f
 Disjoint    | FF*FF**** | inside           | 1FF0FF212 | f
```

☒☒

5.1☒, ST_Relate

7.11.1.15 ST_Touches

ST_Touches — Tests if two geometries have at least one point in common, but their interiors do not intersect

☒☒

boolean **ST_Touches**(geometry A, geometry B);

□□

Returns TRUE if A and B intersect, but their interiors do not intersect. Equivalently, A and B have at least one point in common, and the common points lie in at least one boundary. For Point/Point inputs the relationship is always FALSE, since points do not have a boundary.

In mathematical terms: *ST_Touches(A, B) ⇔ (Int(A) □ Int(B) = □) ∧ (A □ B ≠ □)*

This relationship holds if the DE-9IM Intersection Matrix for the two geometries matches one of:

- FT*******
- F**T*****
- F***T****



□□
This function automatically includes a bounding box comparison that makes use of any spatial indexes that are available on the geometries. To avoid using an index, use _ST_Touches instead.

Enhanced: 3.0.0 enabled support for GEOMETRYCOLLECTION

 This method implements the OGC Simple Features Implementation Specification for SQL 1.1. s2.1.1.2 // s2.1.13.3

 This method implements the SQL/MM specification. SQL-MM 3: 5.1.28

□□

The ST_Touches predicate returns TRUE in the following examples.

Two Polygons touching along a boundary segment.

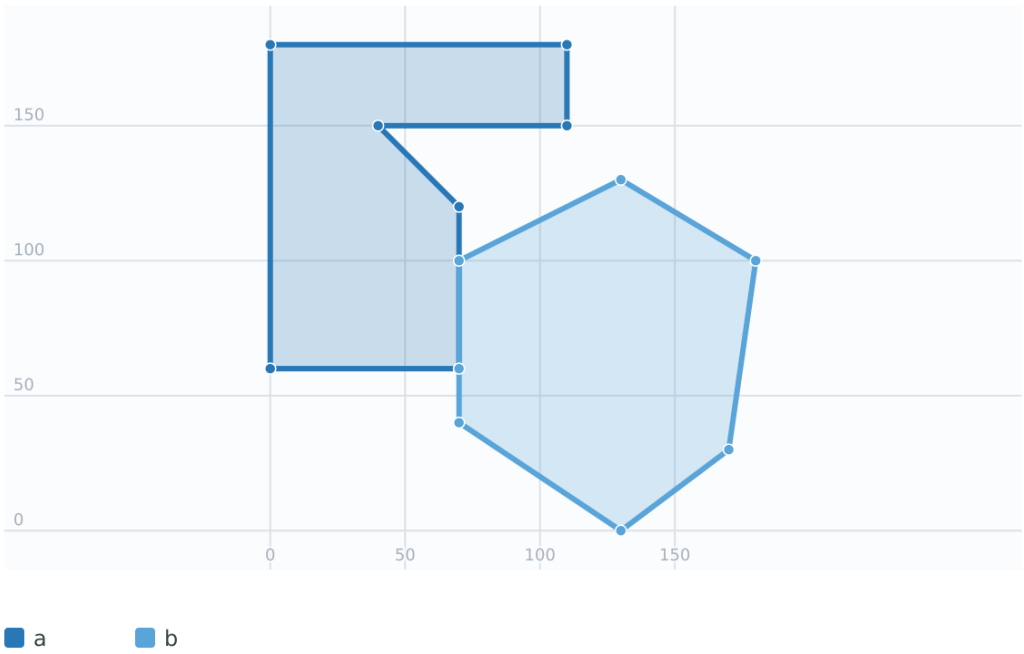
Code

```
SELECT ST_Touches(a, b)
FROM (SELECT
  'POLYGON ((10 190,10 70,80 70,80 130,50 160,120 160,120 190,10 190))'::geometry AS a,
  'POLYGON ((80 110,140 140,190 110,180 40,140 10,80 50,80 70,80 110))'::geometry AS b
) AS example;
```

Output

```
t
```

Figure



Two Polygons touching at a boundary point.

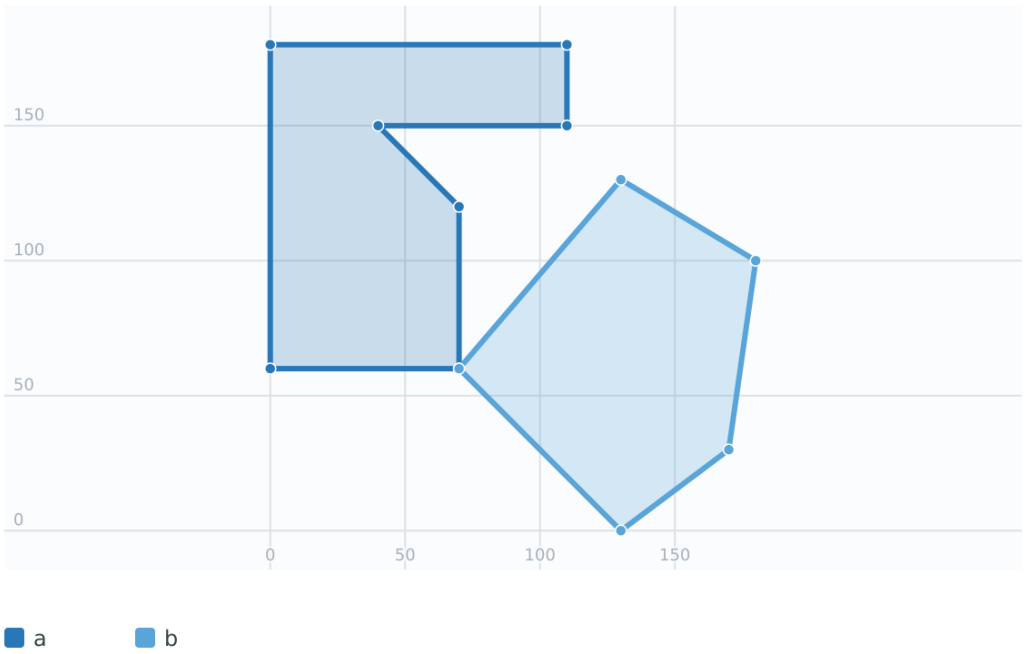
Code

```
SELECT ST_Touches(a, b)
FROM (SELECT
  'POLYGON ((10 190,10 70,80 70,80 130,50 160,120 160,120 190,10 190))'::geometry AS a,
  'POLYGON ((140 140,190 110,180 40,140 10,80 70,140 140))'::geometry AS b
) AS example;
```

Output

```
t
```

Figure



A Polygon and a LineString touching at the line endpoint.

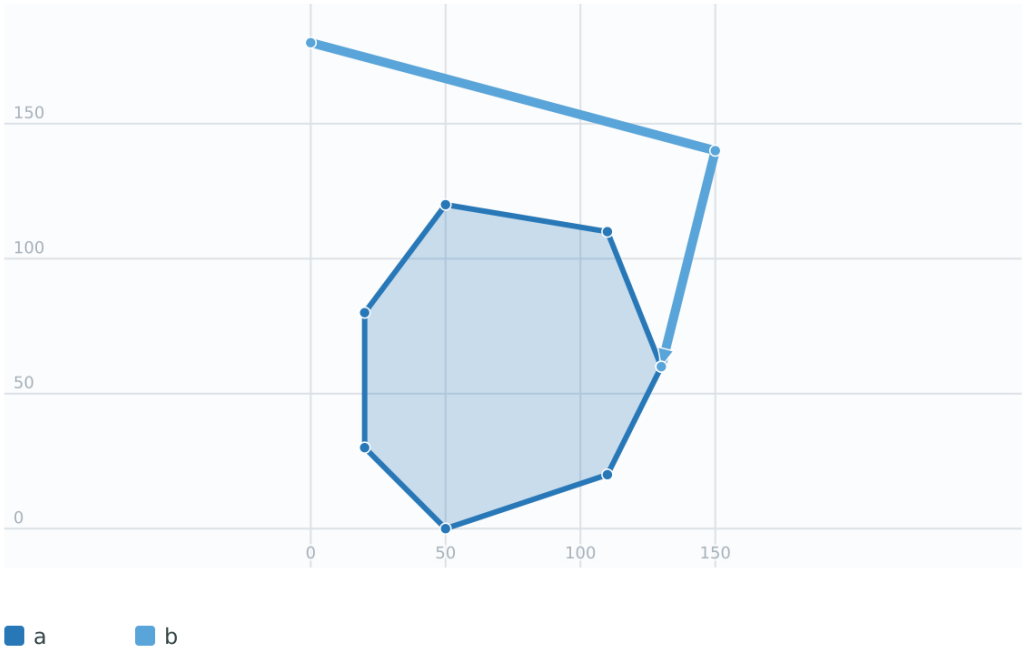
Code

```
SELECT ST_Touches(a, b)
FROM (SELECT
  'POLYGON ((80 10,50 40,50 90,80 130,140 120,160 70,140 30,80 10))'::geometry AS a,
  'LINESTRING (30 190,180 150,160 70)'::geometry AS b
) AS example;
```

Output

```
t
```

Figure



Two LineStrings touching at their endpoints.

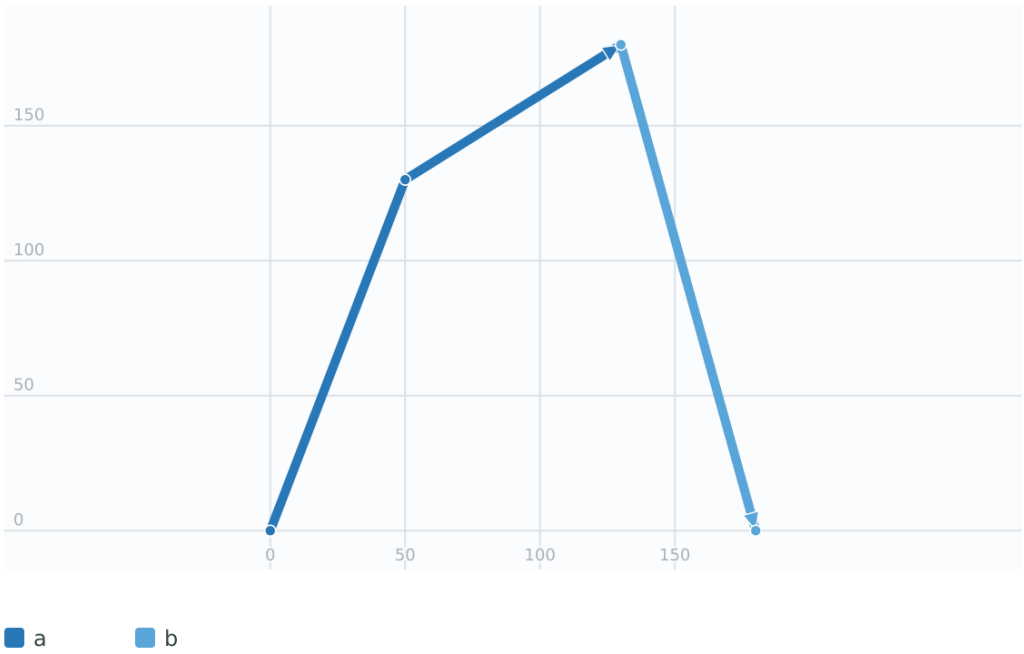
Code

```
SELECT ST_Touches(a, b)
FROM (SELECT
  'LINESTRING (10 10,60 140,140 190)'::geometry AS a,
  'LINESTRING (140 190,190 10)'::geometry AS b
) AS example;
```

Output

```
t
```

Figure



Two LineStrings touching where an endpoint meets the other line.

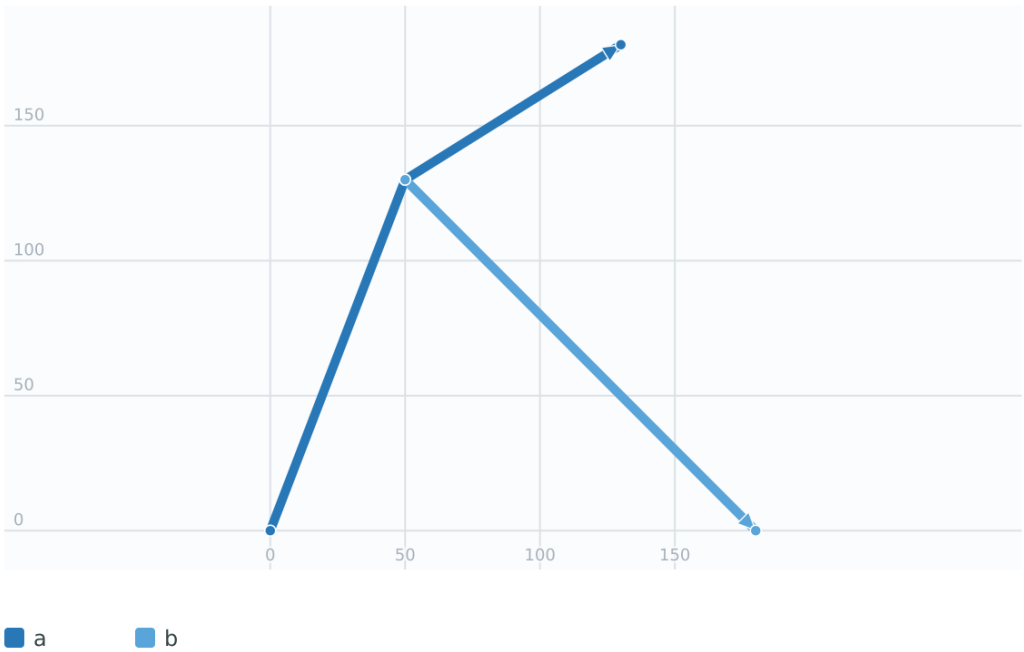
Code

```
SELECT ST_Touches(a, b)
FROM (SELECT
  'LINESTRING (10 10,60 140,140 190)'::geometry AS a,
  'LINESTRING (60 140,190 10)'::geometry AS b
) AS example;
```

Output

```
t
```

Figure



A Point touching a Polygon boundary.

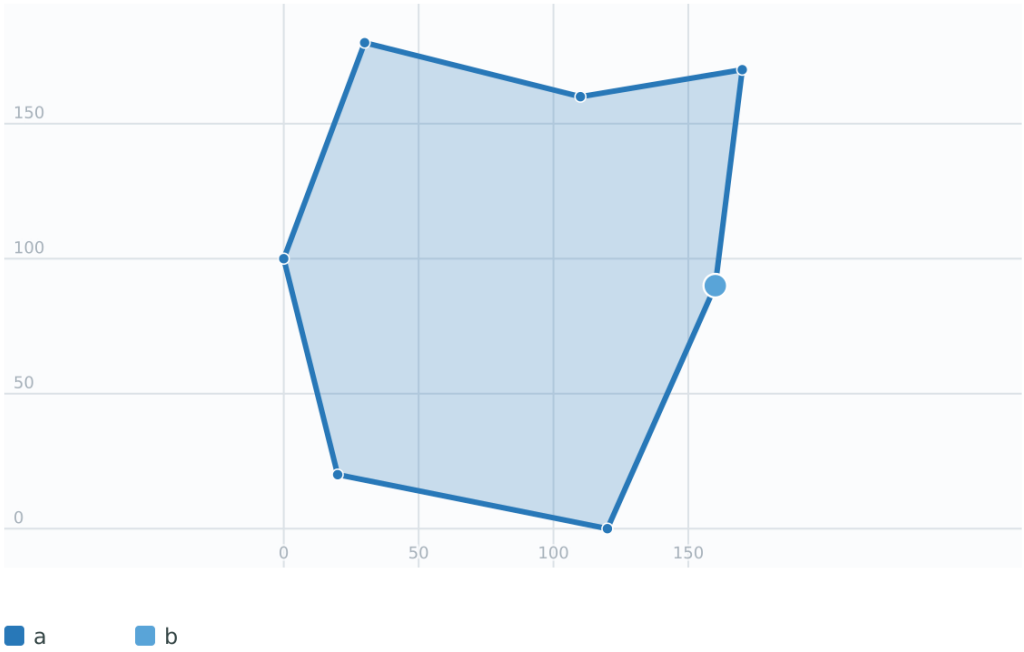
Code

```
SELECT ST_Touches(a, b)
FROM (SELECT
  'POLYGON ((30 30,10 110,40 190,120 170,180 180,170 100,130 10,30 30))'::geometry AS a,
  'POINT (170 100)'::geometry AS b
) AS example;
```

Output

```
t
```

Figure



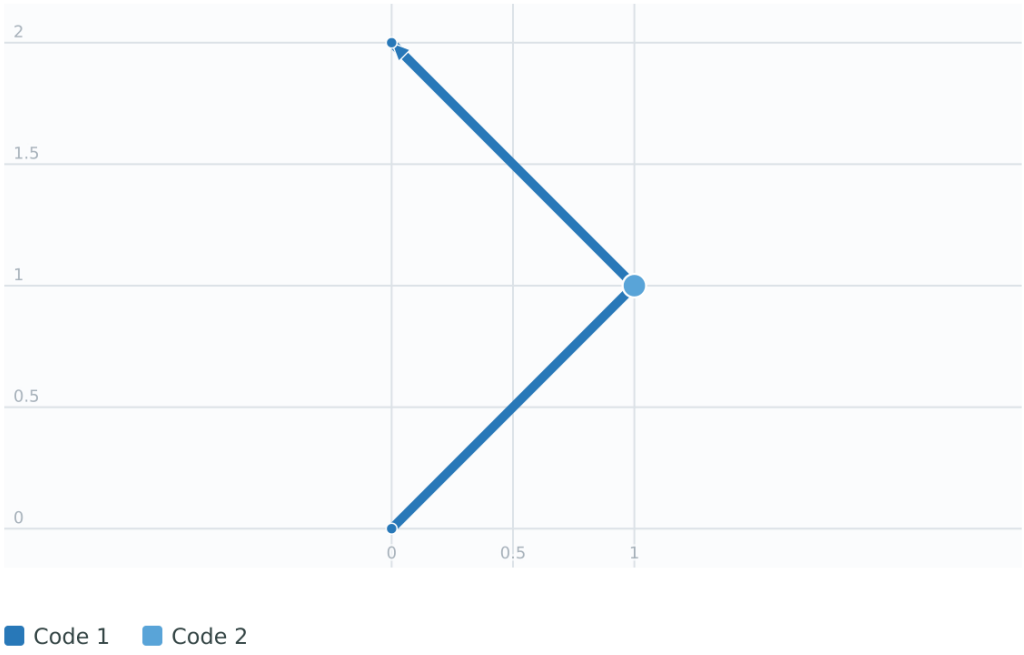
Code

```
SELECT ST_Touches('LINESTRING(0 0,1 1,0 2)'::geometry, 'POINT(1 1)'::geometry);
```

Output

```
f
```

Figure



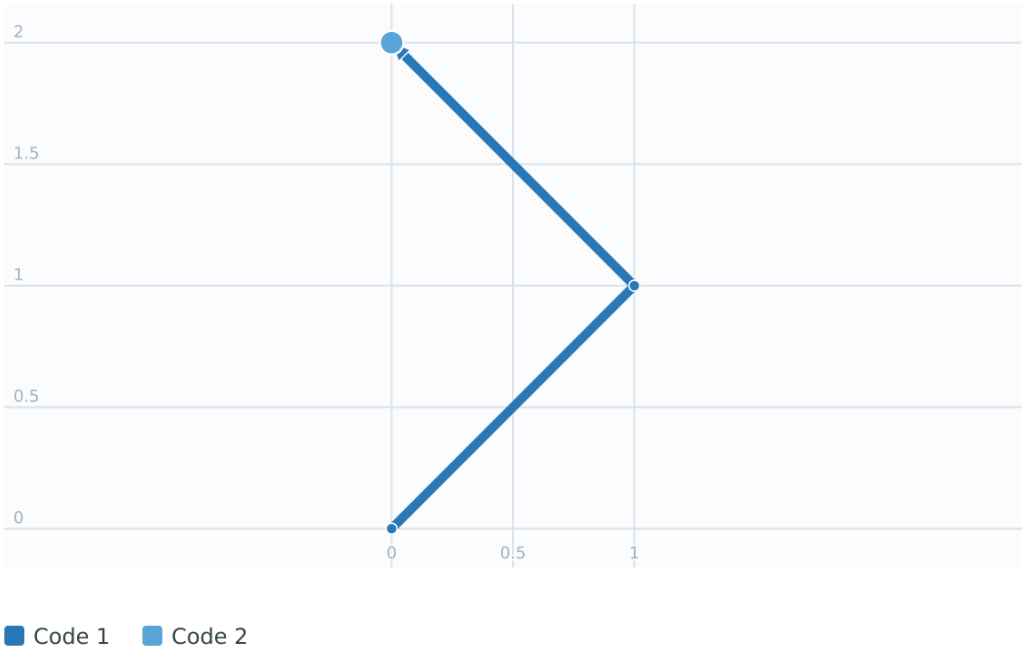
Code

```
SELECT ST_Touches('LINESTRING(0 0,1 1,0 2)'::geometry, 'POINT(0 2)'::geometry);
```

Output

```
t
```

Figure



7.11.1.16 ST_Within

ST_Within — Tests if every point of A lies in B, and their interiors have a point in common

□□

boolean **ST_Within**(geometry A, geometry B);

□□

Returns TRUE if geometry A is within geometry B. A is within B if and only if all points of A lie inside (i.e. in the interior or boundary of) B (or equivalently, no points of A lie in the exterior of B), and the interiors of A and B have at least one point in common.

For this function to make sense, the source geometries must both be of the same coordinate projection, having the same SRID.

In mathematical terms: *ST_Within(A, B) ⇔ (A □ B = A) ∧ (Int(A) □ Int(B) ≠ □)*

The within relation is reflexive: every geometry is within itself. The relation is antisymmetric: if `ST_Within(A,B) = true` and `ST_Within(B,A) = true`, then the two geometries must be topologically equal (`ST_Equals(A,B) = true`).

ST_Within is the converse of ST_Contains. So, `ST_Within(A,B) = ST_Contains(B,A)`.



□□
Because the interiors must have a common point, a subtlety of the definition is that lines and points lying fully in the boundary of polygons or lines are *not* within the geometry. For further details see Subtleties of OGC Covers, Contains, Within. The ST_CoveredBy predicate provides a more inclusive relationship.



□□
This function automatically includes a bounding box comparison that makes use of any spatial indexes that are available on the geometries. To avoid index use, use the function `_ST_Within`.

GEOS □□□□□

Enhanced: 2.3.0 Enhancement to PIP short-circuit for geometry extended to support MultiPoints with few points. Prior versions only supported point in polygon.

Enhanced: 3.0.0 enabled support for `GEOMETRYCOLLECTION`



□□
Do not use this function with invalid geometries. You will get unexpected results.

NOTE: this is the "allowable" version that returns a boolean, not an integer.



This method implements the OGC Simple Features Implementation Specification for SQL 1.1. s2.1.1.2 // s2.1.13.3 - a.Relate(b, 'T*F**F***')



This method implements the SQL/MM specification. SQL-MM 3: 5.1.30

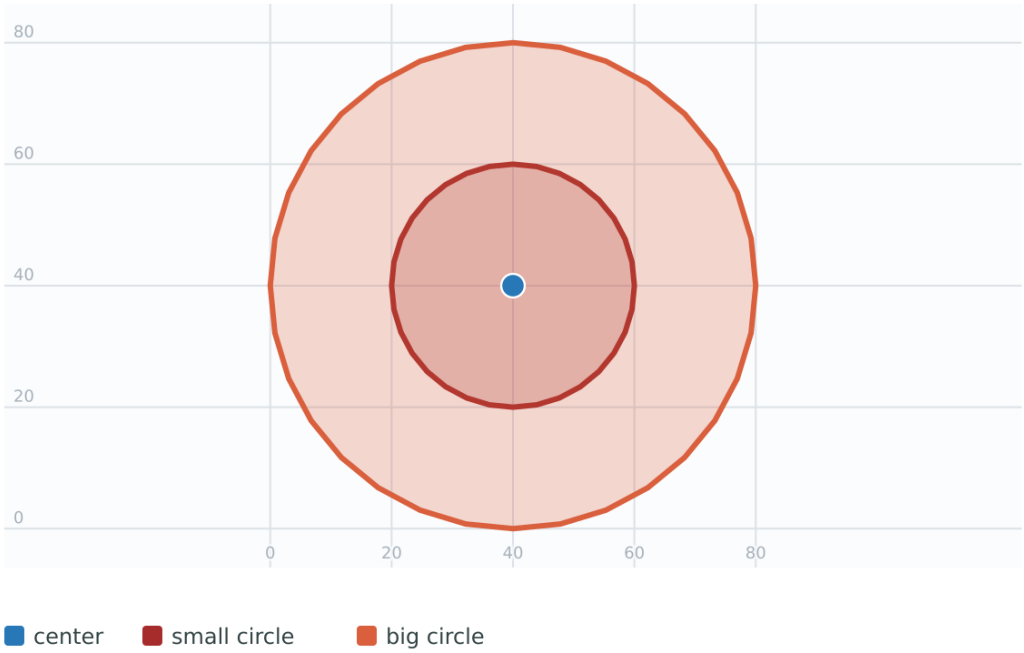
□□

This example defines two concentric circles once in a CTE so the same inputs can be rendered and then reused in related predicate checks.

Code

```
WITH circles AS (
  SELECT ST_Buffer(center, 20) AS smallc,
         ST_Buffer(center, 40) AS bigc
  FROM (SELECT 'POINT(50 50)'::geometry AS center) AS p
)
SELECT smallc AS small_circle,
       bigc AS big_circle
FROM circles;
```

Figure



Using the same circles, `ST_Within` is reflexive on each circle, false from the larger circle into the smaller one, and true for `ST_Union` only when the union does not extend beyond the larger circle.

Code

```
WITH circles AS (
  SELECT ST_Buffer(center, 20) AS smallc,
         ST_Buffer(center, 40) AS bigc
  FROM (SELECT 'POINT(50 50)'::geometry AS center) AS p
)
SELECT
  ST_Within(smallc, smallc) AS smallinsmall,
  ST_Within(smallc, bigc) AS smallinbig,
  ST_Within(bigc, smallc) AS biginsmall,
  ST_Within(ST_Union(smallc, bigc), bigc) AS unioninbig,
  ST_Within(bigc, ST_Union(smallc, bigc)) AS biginunion,
  ST_Equals(bigc, ST_Union(smallc, bigc)) AS bigisunion
FROM circles;
```

Output

```
t | t | f | t | t | t
```

Figure



□□

ST_Contains, ST_CoveredBy, ST_Equals, ST_IsValid

7.11.2 Distance Relationships

7.11.2.1 ST_3DDWithin

ST_3DDWithin — Tests if two 3D geometries are within a given 3D distance

□□

boolean **ST_3DDWithin**(geometry g1, geometry g2, double precision distance_of_srid);

□□

Returns true if the 3D distance between two geometry values is no larger than distance `distance_of_srid`. The distance is specified in units defined by the spatial reference system of the geometries. For this function to make sense the source geometries must be in the same coordinate system (have the same SRID).



□□
This function automatically includes a bounding box comparison that makes use of any spatial indexes that are available on the geometries.

This function supports 3d and will not drop the z-index.

This function supports Polyhedral surfaces.

This method implements the SQL/MM specification. SQL-MM ?

2.0.0 □□□□□□□□□□□□.

□□

Compare the same transformed point and line in both 3D and 2D. The 3D test fails because the Z difference matters even though the 2D footprints are close enough.

Code

```
WITH sample AS (
  SELECT ST_Transform('SRID=4326;POINT(-72.1235 42.3521 4)'::geometry, 2163) AS input_point ↩
    ,
         ST_Transform(
           'SRID=4326;LINESTRING(-72.1260 42.45 15,-72.123 42.1546 20)'::geometry,
           2163
         ) AS input_line
)
SELECT ST_3DDWithin(input_point, input_line, 126.8) AS within_dist_3d,
       ST_DWithin(input_point, input_line, 126.8) AS within_dist_2d,
       input_point AS input_point,
       input_line AS input_line
FROM sample;
```

Output

```
-[ RECORD 1 ↩
    ]--+----------------------------------------------------------------------------------------------------------------------------------------------------------------

within_dist_3d | f
within_dist_2d | t
input_point    | POINT Z (-11774128.9 5237160.2 4)
input_line     | LINESTRING Z (-11774339.5 5243556.8 15,-11773706.7 5215256.3 20)
```

□□

ST_3DDFullyWithin, ST_DWithin, ST_DFullyWithin, ST_3DDistance, ST_Distance, ST_3DMaxDistance, ST_Transform

7.11.2.2 ST_3DDFullyWithin

ST_3DDFullyWithin — Tests if two 3D geometries are entirely within a given 3D distance

□□

boolean **ST_3DDFullyWithin**(geometry g1, geometry g2, double precision distance);

□□

Returns true if the 3D geometries are fully within the specified distance of one another. The distance is specified in units defined by the spatial reference system of the geometries. For this function to make sense, the source geometries must both be of the same coordinate projection, having the same SRID.



□□
This function automatically includes a bounding box comparison that makes use of any spatial indexes that are available on the geometries.

2.0.0 □□□□□□□□□□□□□.

This function supports 3d and will not drop the z-index.

This function supports Polyhedral surfaces.

□□

This example compares distance-within predicates in 2D and 3D. The point is distance-within the line in 3D, but the full 3D line is not within the same distance of the point.

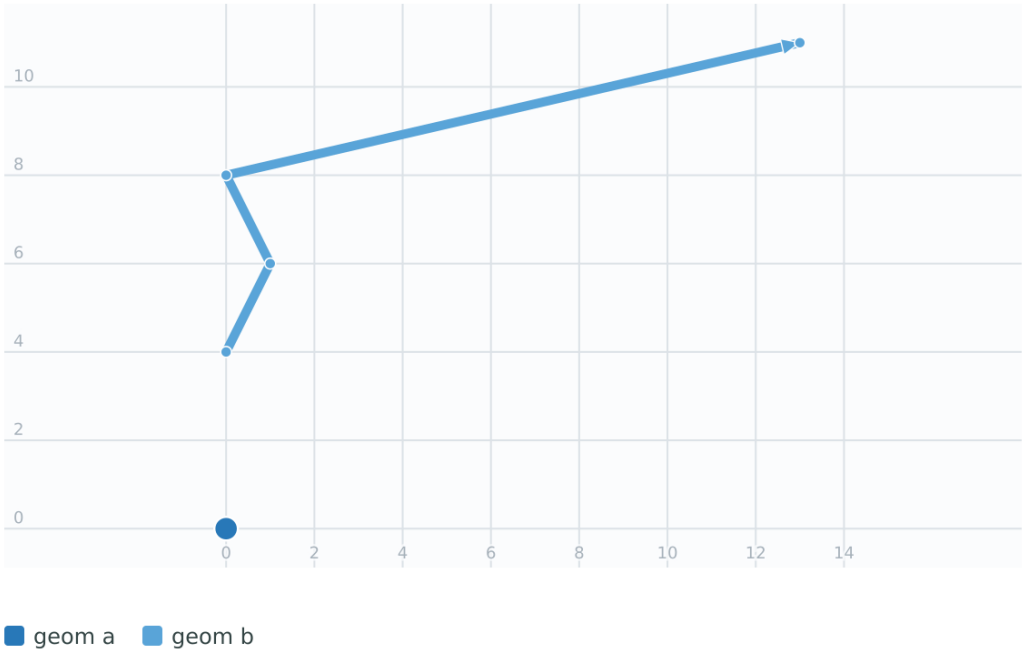
Code

```
WITH data AS (
  SELECT
    'POINT(1 1 2)'::geometry AS geom_a,
    'LINESTRING(1 5 2,2 7 20,1 9 100,14 12 3)'::geometry AS geom_b
)
SELECT
  ST_3DDFullyWithin(geom_a, geom_b, 10) AS d3dfullywithin10,
  ST_3DDWithin(geom_a, geom_b, 10) AS d3dwithin10,
  ST_DFullyWithin(geom_a, geom_b, 20) AS d2dfullywithin20,
  ST_3DDFullyWithin(geom_a, geom_b, 20) AS d3dfullywithin20
FROM data;
```

Output

```
d3dfullywithin10 | d3dwithin10 | d2dfullywithin20 | d3dfullywithin20
-----------------+-------------+------------------+-----------------
 f               | t           | t                | f
```

Figure



□□

ST_3DDWithin, ST_DWithin, ST_DFullyWithin, ST_3DMaxDistance

7.11.2.3 ST_DFullyWithin

ST_DFullyWithin — Tests if a geometry is entirely inside a distance of another

□□

boolean **ST_DFullyWithin**(geometry g1, geometry g2, double precision distance);

□□

Returns true if `g2` is entirely within `distance` of `g1`. Visually, the condition is true if `g2` is contained within a `distance` buffer of `g1`. The distance is specified in units defined by the spatial reference system of the geometries.



□□
This function automatically includes a bounding box comparison that makes use of any spatial indexes that are available on the geometries.

1.5.0 □□□□□□□□□□□□.

Changed: 3.5.0 : the logic behind the function now uses a test of containment within a buffer, rather than the ST_MaxDistance algorithm. Results will differ from prior versions, but should be closer to user expectations.

□□

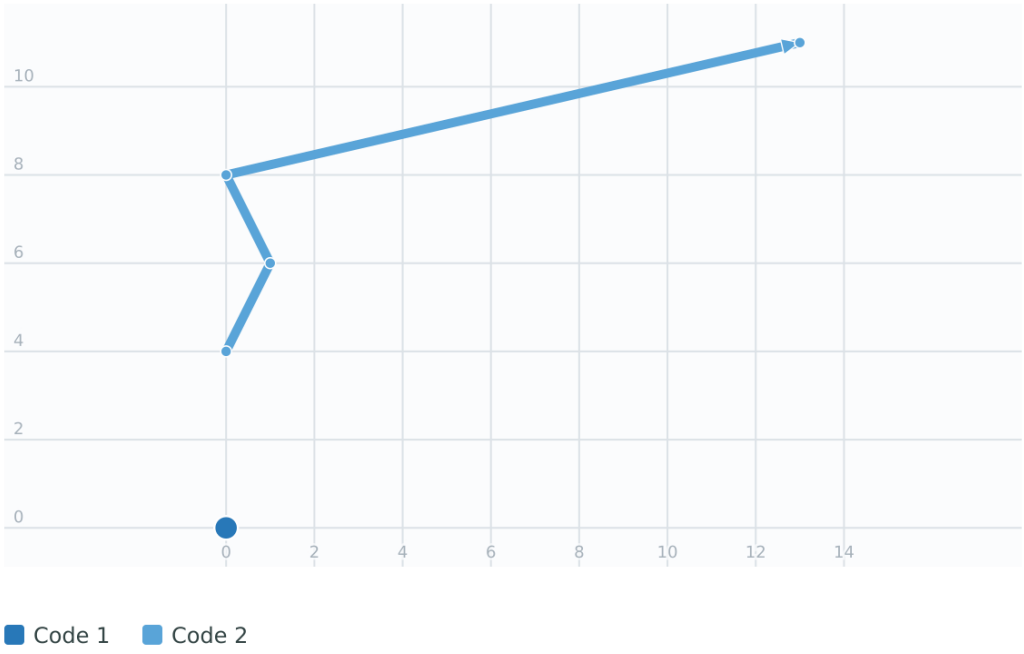
Code

```
SELECT
    ST_DFullyWithin(geom_a, geom_b, 10) AS DFullyWithin10,
    ST_DWithin(geom_a, geom_b, 10) AS DWithin10,
    ST_DFullyWithin(geom_a, geom_b, 20) AS DFullyWithin20
FROM (VALUES
    ('POINT(1 1)'::geometry, 'LINESTRING(1 5,2 7,1 9,14 12)'::geometry)
    ) AS v(geom_a, geom_b)
```

Output

```
dfullywithin10 | dwithin10 | dfullywithin20
---------------+-----------+----------------
 f             | t         | t
```

Figure



□□

ST_MaxDistance, ST_DWithin, ST_3DDWithin, ST_3DDFullyWithin

7.11.2.4 ST_DWithin

ST_DWithin — Tests if two geometries are within a given distance

□□

boolean **ST_DWithin**(geometry g1, geometry g2, double precision distance_of_srid);
boolean **ST_DWithin**(geography gg1, geography gg2, double precision distance_meters, boolean use_spheroid = true);

□□

Returns true if the geometries are within a given distance

For geometry: The distance is specified in units defined by the spatial reference system of the geometries. For this function to make sense, the source geometries must be in the same coordinate system (have the same SRID).

For geography: units are in meters and distance measurement defaults to `use_spheroid = true`. For faster evaluation use `use_spheroid = false` to measure on the sphere.



□□
Use ST_3DDWithin for 3D geometries.



□□
This function call includes a bounding box comparison that makes use of any indexes that are available on the geometries.



This method implements the OGC Simple Features Implementation Specification for SQL 1.1.

Availability: 1.5.0 support for geography was introduced

Enhanced: 2.1.0 improved speed for geography. See Making Geography faster for details.

Enhanced: 2.1.0 support for curved geometries was introduced.

Prior to 1.3, ST_Expand was commonly used in conjunction with && and ST_Distance to test for distance, and in pre-1.3.4 this function used that logic. From 1.3.4, ST_DWithin uses a faster short-circuit distance function.

□□

This self-contained geometry example keeps the compared shapes visible while also returning the boolean results.

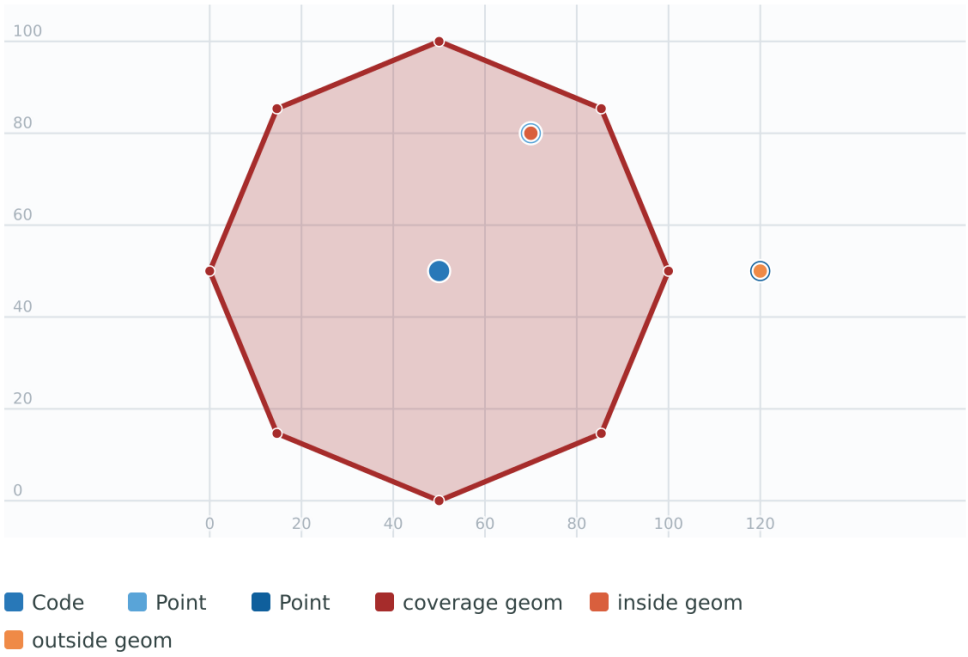
Code

```
WITH sample AS (
  SELECT ST_Buffer('POINT(100 90)'::geometry, 50, 'quad_segs=2') AS coverage,
         ST_Point(120, 120) AS inside_pt,
         ST_Point(170, 90) AS outside_pt
)
SELECT ST_DWithin(coverage, inside_pt, 0) AS inside_now,
       ST_DWithin(coverage, outside_pt, 10) AS outside_within_10,
       coverage AS coverage_geom,
       inside_pt AS inside_geom,
       outside_pt AS outside_geom
FROM sample;
```

Output

```
-[ RECORD 1 ]-----
inside_now        | t
outside_within_10 | f
coverage_geom     | POLYGON((150 90,135 55,100 40,65 55,50 90,65 125,100 140,135 125,150 90))
inside_geom       | POINT(120 120)
outside_geom      | POINT(170 90)
```

Figure



Find the nearest hospital to each school. That is within 3000 units of the school. We do an ST_DWithin search to utilize indexes to limit our search list. That the non-indexable ST_Distance needs to process. If the units of the spatial reference is meters then units would be meters.

Code

```
SELECT DISTINCT ON (s.gid) s.gid, s.school_name, s.geom, h.hospital_name
  FROM schools s
    LEFT JOIN hospitals h ON ST_DWithin(s.geom, h.geom, 3000)
  ORDER BY s.gid, ST_Distance(s.geom, h.geom);
```

The schools with no close hospitals. Find all schools with no hospital within 3000 units away from the school. Units are in units of spatial reference (for example meters, feet, or degrees).

Code

```
SELECT s.gid, s.school_name
  FROM schools s
    LEFT JOIN hospitals h ON ST_DWithin(s.geom, h.geom, 3000)
  WHERE h.gid IS NULL;
```

Find locations more than 3000 units away from their nearest hospital. Use `NOT EXISTS` so that a location is returned only when no nearby hospital exists.

Code

```
SELECT l.location_name
  FROM locations l
  WHERE NOT EXISTS (
    SELECT 1
    FROM hospitals h
    WHERE ST_DWithin(l.geom, h.geom, 3000)
  );
```

Find broadcasting towers that a receiver with limited range can receive. Data is geometry in Spherical Mercator (SRID=3857), ranges are approximate.

Code

```
CREATE INDEX ON broadcasting_towers USING gist (geom);
CREATE INDEX ON broadcasting_towers USING gist (ST_Expand(geom, sending_range));
```

Query towers that a 4-kilometer receiver in Minsk Hackerspace can get. Note: two conditions are required because the shorter `LEAST(b.sending_range, 4000)` will not use an index.

Code

```
SELECT b.tower_id, b.geom
  FROM broadcasting_towers b
  WHERE ST_DWithin(b.geom, 'SRID=3857;POINT(3072163.4 7159374.1)', 4000)
    AND ST_DWithin(b.geom, 'SRID=3857;POINT(3072163.4 7159374.1)', b.sending_range);
```

□□

ST_Distance, ST_3DDWithin

7.11.2.5 ST_PointInsideCircle

ST_PointInsideCircle — Tests if a point geometry is inside a circle defined by a center and radius

□□

boolean **ST_PointInsideCircle**(geometry a_point, float center_x, float center_y, float radius);

□□

Returns true if the geometry is a point and is inside the circle with center `center_x`,`center_y` and radius `radius`.



□□
Does not use spatial indexes. Use ST_DWithin instead.

Availability: 1.2

Changed: 2.2.0 In prior versions this was called ST_Point_Inside_Circle

□□

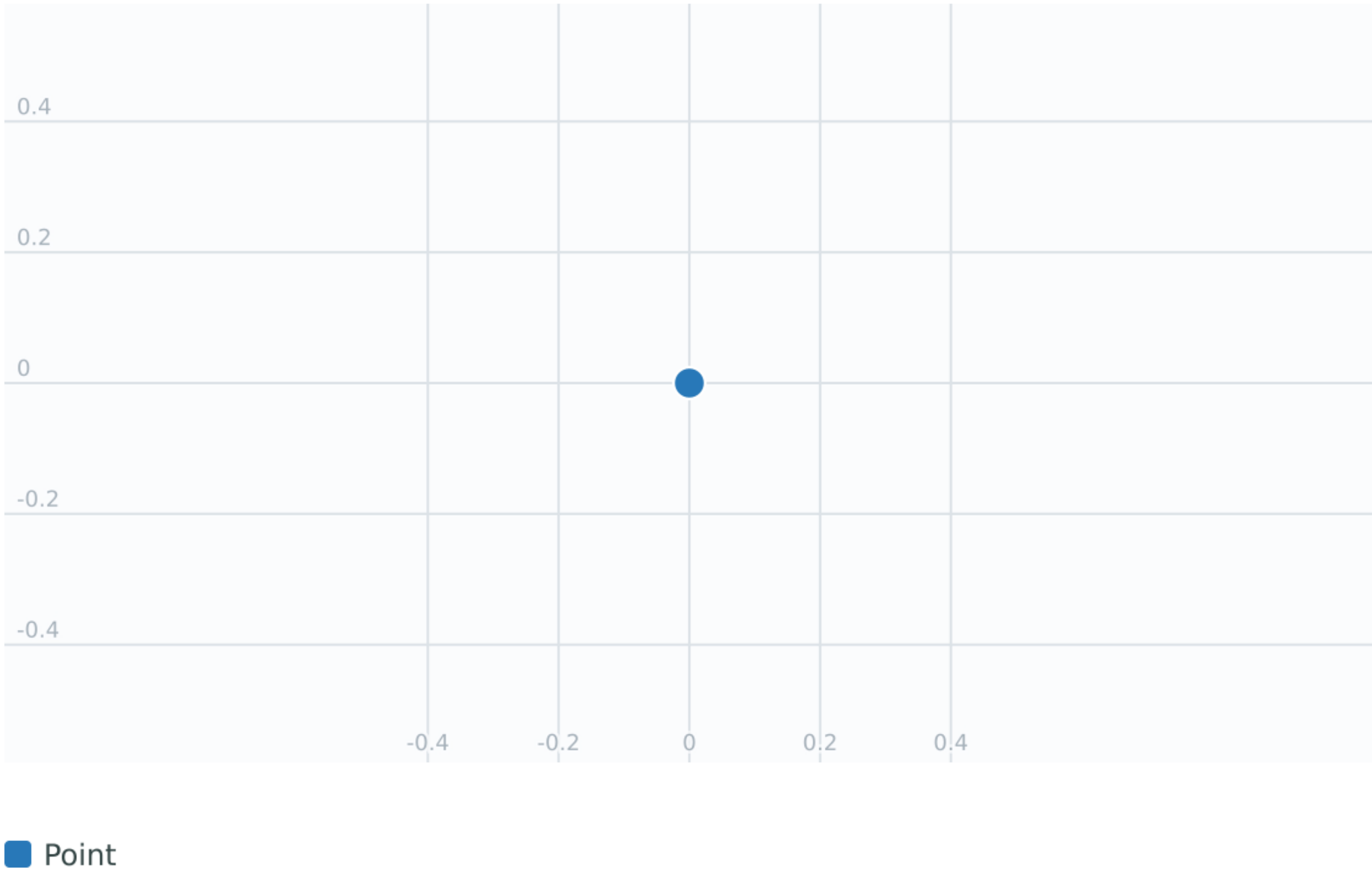
Code

```
SELECT ST_PointInsideCircle(ST_Point(1, 2), 0.5, 2, 3);
```

Output

```
t
```

Figure



⊠⊠

ST_DWithin

7.12 Measurement Functions

7.12.1 ST_Area

ST_Area — ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

float **ST_Area**(geometry g1);
float **ST_Area**(geography geog, boolean use_spheroid = true);

⊠⊠

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ - ST_Surface ⊠⊠ ST_MultiSurface ⊠⊠⊠⊠⊠⊠⊠⊠⊠ - ⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠, SRID ⊠⊠⊠⊠⊠⊠⊠⊠ 2 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠ (curved surface) ⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ST_Area(geog,false) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠⊠⊠: 2.0.0 ⊠⊠⊠⊠ 2 ⊠⊠⊠⊠⊠⊠⊠⊠ (polyhedral surface) ⊠⊠⊠⊠⊠⊠⊠.

⊠⊠⊠⊠: 2.2.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ GeographicLib ⊠⊠⊠⊠⊠⊠⊠. ⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ Proj 4.9.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

Changed: 3.0.0 - does not depend on SFCGAL anymore.

This method implements the OGC Simple Features Implementation Specification for SQL 1.1.

This method implements the SQL/MM specification. SQL-MM 3: 8.1.2, 9.5.3

This function supports Polyhedral surfaces.

> **Note!**
> ⊠⊠
> ⊠⊠⊠⊠⊠⊠⊠⊠, (2.5 ⊠⊠⊠⊠⊠⊠) 2 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. 2.5 ⊠⊠⊠⊠⊠ 0 ⊠⊠⊠ (non-zero) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, XY ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (plot) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. EPSG:2249 ⊠⊠.

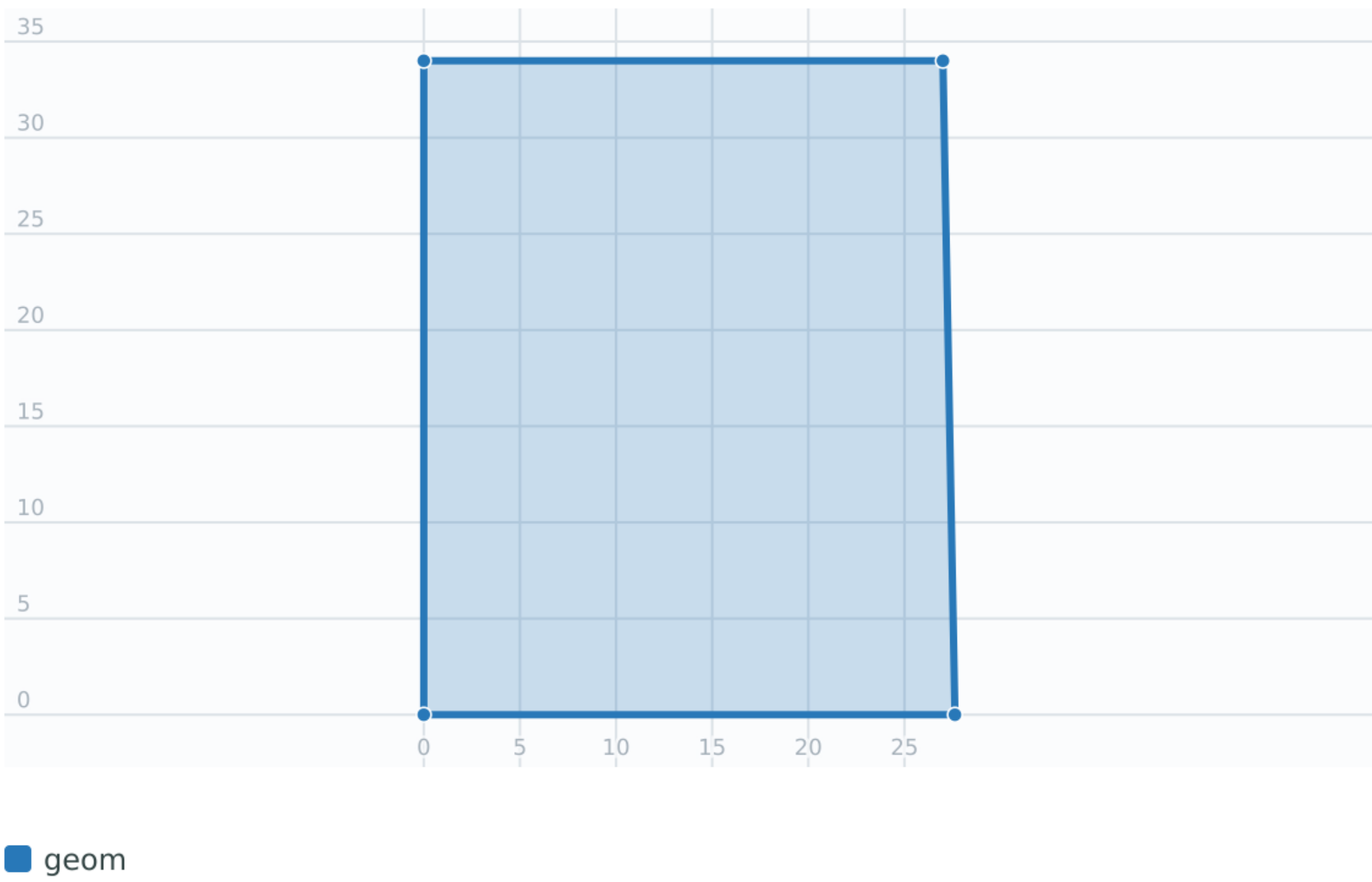
Code

```
SELECT ST_Area(geom) AS sqft,
       ST_Area(geom) * 0.3048 ^ 2 AS sqm
FROM (
  SELECT 'SRID=2249;POLYGON((743238 2967416,743238 2967450,
          743265 2967450,743265.625 2967416,743238 2967416))'::geometry AS geom
) AS subquery;
```

Output

```
sqft   |     sqm
--------+-------------
928.625 | 86.27208552
```

Figure



Return area square feet and transform to Massachusetts state plane meters (EPSG:26986) to get square meters. Note this is in square feet because 2249 is Massachusetts State Plane Feet and transformed area is in square meters since EPSG:26986 is state plane Massachusetts meters.

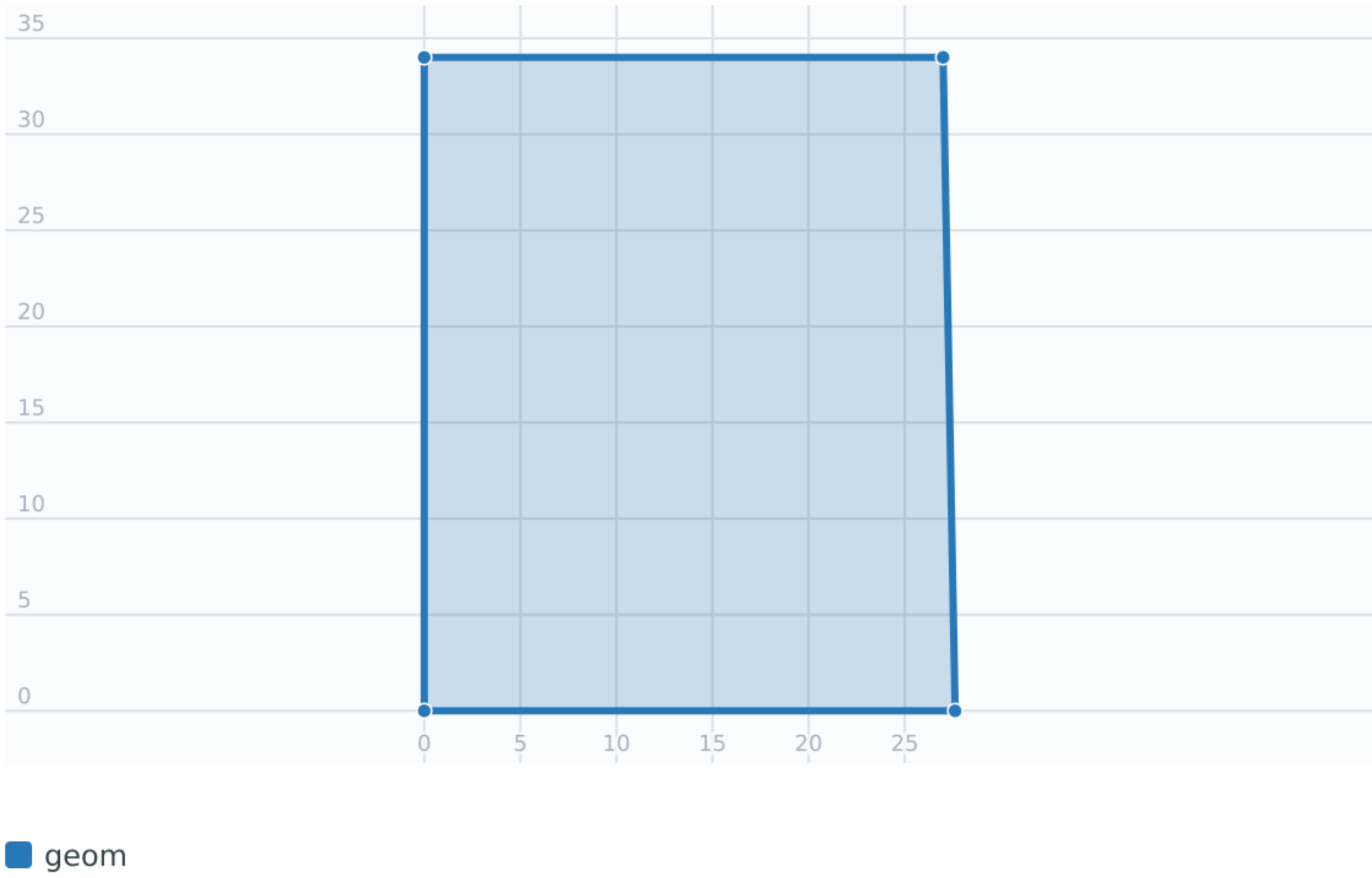
Code

```
SELECT ST_Area(geom) AS sqft,
       round(ST_Area(ST_Transform(geom, 26986))::numeric, 6) AS sqm
FROM (
  SELECT 'SRID=2249;POLYGON((743238 2967416,743238 2967450,
          743265 2967450,743265.625 2967416,743238 2967416))'::geometry AS geom
) AS subquery;
```

Output

```
sqft   |    sqm
--------+-----------
928.625 | 86.272431
```

Figure



[illegible]. [illegible] ([illegible] WGS84 4326 [illegible]). [illegible]. [illegible]. [illegible].

Code

```
SELECT round((ST_Area(geog) / 0.3048 ^ 2)::numeric, 2) AS sqft_spheroid,
       round((ST_Area(geog, false) / 0.3048 ^ 2)::numeric, 2) AS sqft_sphere,
       round(ST_Area(geog)::numeric, 2) AS sqm_spheroid
FROM (
  SELECT ST_Transform(
           'SRID=2249;POLYGON((743238 2967416,743238 2967450,
           743265 2967450,743265.625 2967416,743238 2967416))'::geometry,
           4326
         )::geography AS geog
) AS subquery;
```

Output

```
sqft_spheroid | sqft_sphere | sqm_spheroid
---------------+-------------+--------------
        928.68 |      926.61 |        86.28
```

If your data is in geography already:

Code

```
SELECT ST_Area(geog) / 0.3048 ^ 2 AS sqft,
                ST_Area(the_geog) AS sqm
        FROM somegeogtable;
```

□□

CG_3DArea, ST_GeomFromEWKT, ST_LengthSpheroid, ST_Perimeter, ST_Transform

7.12.2 ST_Azimuth

ST_Azimuth — □□□□□□ 2 □□□□□□□□□□□□.

□□

float **ST_Azimuth**(geometry origin, geometry target);
float **ST_Azimuth**(geography origin, geography target);

□□

Returns the azimuth in radians of the target point from the origin point, or NULL if the two points are coincident. The azimuth angle is a positive clockwise angle referenced from the positive Y axis (geometry) or the North meridian (geography): North = 0; Northeast = π/4; East = π/2; Southeast = 3π/4; South = π; Southwest 5π/4; West = 3π/2; Northwest = 7π/4.

For the geography type, the azimuth solution is known as the inverse geodesic problem.

The azimuth is a mathematical concept defined as the angle between a reference vector and a point, with angular units in radians. The result value in radians can be converted to degrees using the PostgreSQL function `degrees()`.

For offsetting lines relative to their direction, use ST_OffsetCurve. For projecting a point by distance and bearing, use ST_Project.

1.1.0 □□□□□□□□□□□□.

□□□□: 2.0.0 □□□□□□□□□□□□□□.

□□□□: 2.2.0 □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ GeographicLib □□□□□□□. □□□□□□□□□□□□□ Proj 4.9.0 □□□□□□□□□□□.

□□

□□□□□□□□□□

Code

```
SELECT degrees(ST_Azimuth(ST_Point(25, 45), ST_Point(75, 100))) AS degA_B,
       degrees(ST_Azimuth(ST_Point(75, 100), ST_Point(25, 45) )) AS degB_A;
```

Output

```
dega_b       |     degb_a
-----------------+-----------------
42.27368900609371 | 222.27368900609372
```

Compass direction labels can be computed from the azimuth.

Code

```
WITH directions(deg) AS (VALUES
  (0), (45), (90), (135), (180), (225), (270), (315)
),
bearings AS (
  SELECT deg,
         degrees(ST_Azimuth(ST_Point(0, 0),
                            ST_MakePoint(sin(radians(deg)),
                                         cos(radians(deg))))) AS azimuth
  FROM directions
)
SELECT deg,
       round(azimuth::numeric, 1) AS azimuth,
       (ARRAY['N', 'NE', 'E', 'SE', 'S', 'SW', 'W', 'NW', 'N'])
         [floor((azimuth + 22.5) / 45)::int + 1] AS direction
FROM bearings
ORDER BY deg;
```

Output

```
deg | azimuth | direction
-----+---------+-----------
   0 |     0.0 | N
  45 |    45.0 | NE
  90 |    90.0 | E
 135 |   135.0 | SE
 180 |   180.0 | S
 225 |   225.0 | SW
 270 |   270.0 | W
 315 |   315.0 | NW
```

Azimuth from Point(25, 45) to Point(75, 100), shown against the positive Y axis (north).

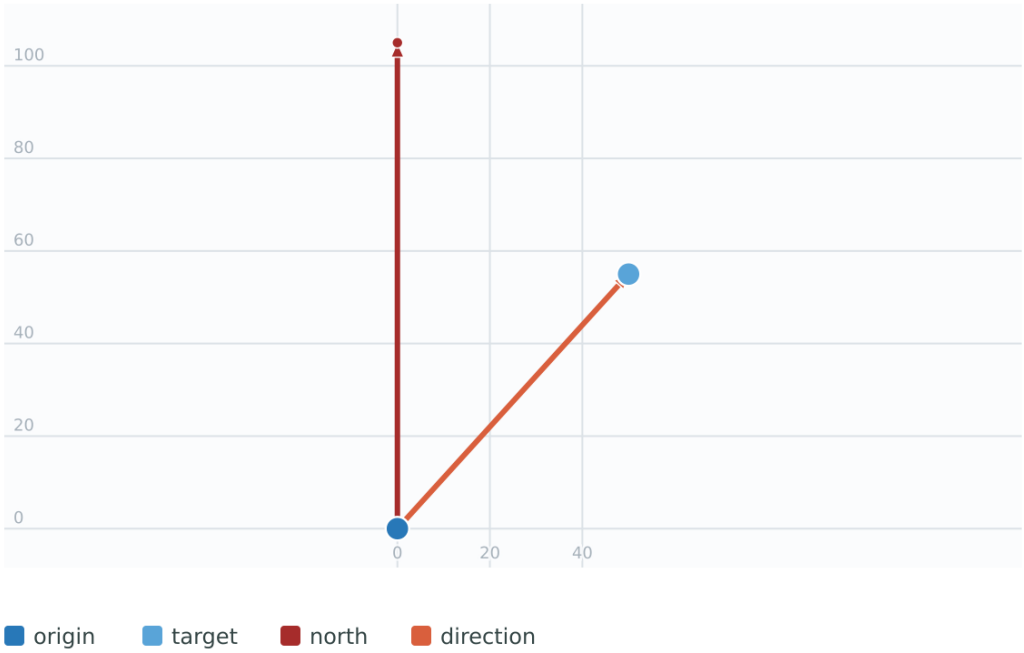
Code

```
WITH points AS (
  SELECT 'POINT(25 45)'::geometry AS origin,
         'POINT(75 100)'::geometry AS target
)
SELECT round(degrees(ST_Azimuth(origin, target))::numeric, 6) AS degrees,
       ST_MakeLine(origin, ST_Translate(origin, 0, 105)) AS north,
       ST_MakeLine(origin, target) AS direction
FROM points;
```

Output

```
42.273689 | LINESTRING(25 45,25 150) | LINESTRING(25 45,75 100)
```

Figure



Reverse azimuth from Point(75, 100) to Point(25, 45).

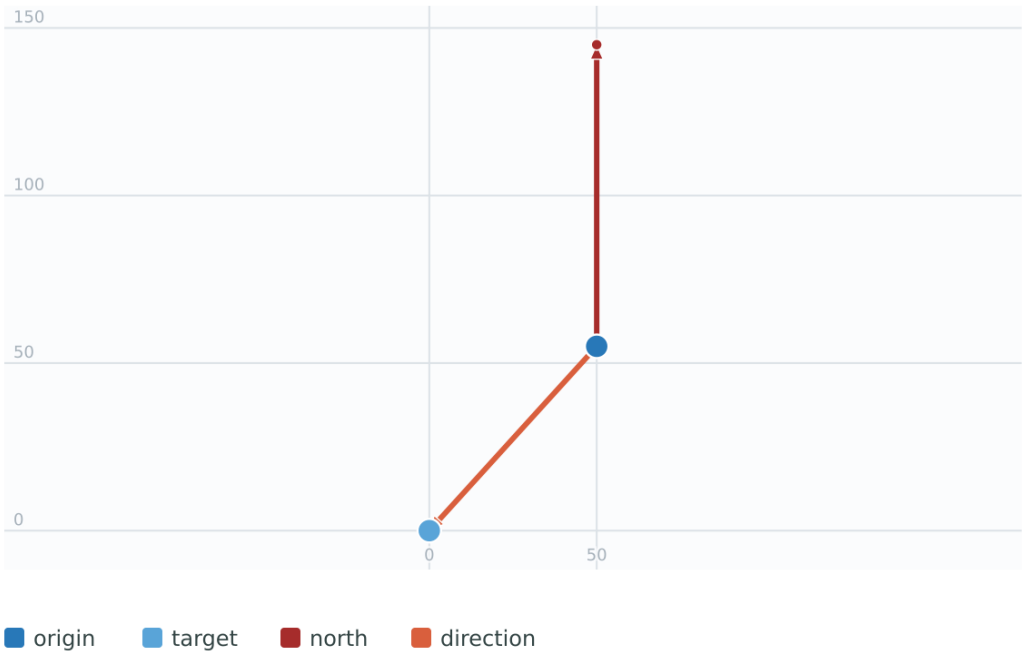
Code

```
WITH points AS (
  SELECT 'POINT(75 100)'::geometry AS origin,
         'POINT(25 45)'::geometry AS target
)
SELECT round(degrees(ST_Azimuth(origin, target))::numeric, 6) AS degrees,
       ST_MakeLine(origin, ST_Translate(origin, 0, 90)) AS north,
       ST_MakeLine(origin, target) AS direction
FROM points;
```

Output

```
222.273689 | LINESTRING(75 100,75 190) | LINESTRING(75 100,25 45)
```

Figure



□□

ST_Angle, ST_Translate, ST_Project, PostgreSQL Math Functions

7.12.3 ST_Angle

ST_Angle — □□□□□□ 3 □□□□ (longest) □□□□□□□□.

□□

float **ST_Angle**(geometry point1, geometry point2, geometry point3, geometry point4);
float **ST_Angle**(geometry line1, geometry line2);

□□

□□□□□□ 3 □□□□ (longest) □□□□□□□□.

Variant 1: computes the angle enclosed by the points P1-P2-P3. If a 4th point provided computes the angle points P1-P2 and P3-P4

Variant 2: computes the angle between two vectors S1-E1 and S2-E2, defined by the start and end points of the input lines

□□. PostgreSQL □□□□□ degrees() □□□□□□□□□□□□□□□□□□□□□□□□. □□□□□□□□□□□□.

Note that `ST_Angle(P1,P2,P3) = ST_Angle(P2,P1,P2,P3)`.

Availability: 2.5.0

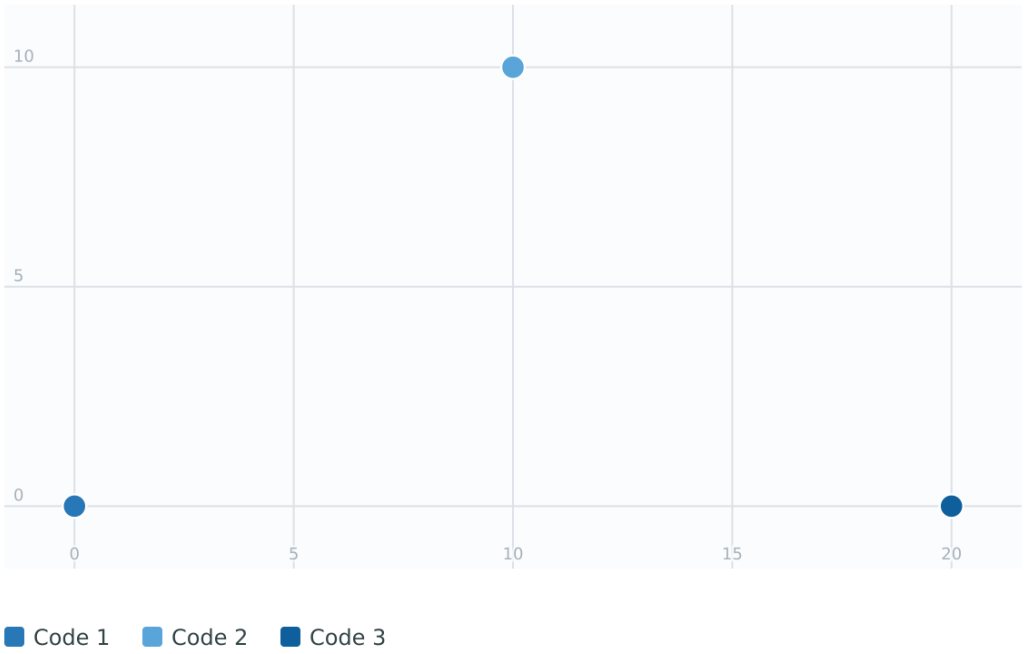
□□

□□□□□□□□□□□□□□□

Code

```
SELECT degrees(ST_Angle('POINT(0 0)', 'POINT(10 10)', 'POINT(20 0)'));
```

Figure

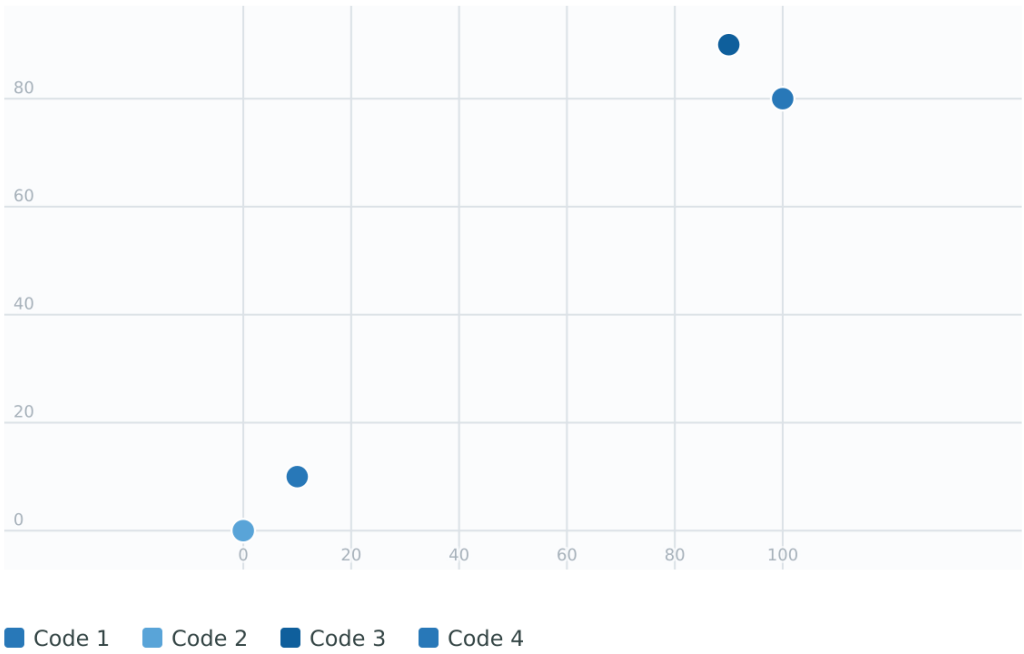


Angle between vectors defined by four points

Code

```
SELECT degrees(ST_Angle('POINT (10 10)', 'POINT (0 0)', 'POINT(90 90)', 'POINT (100 80)'));
```

Figure

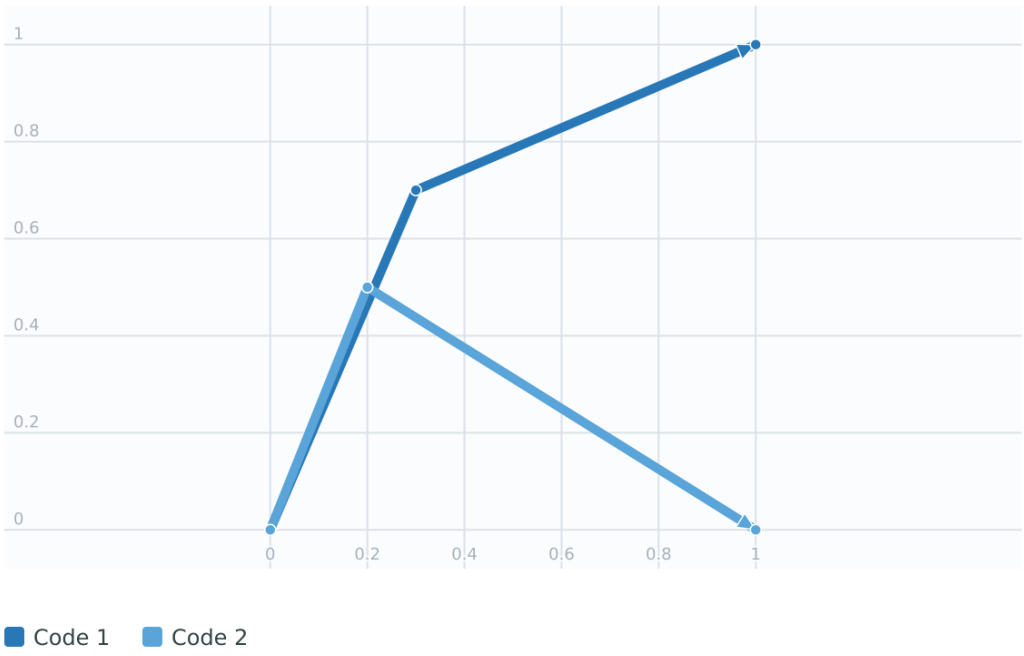


Angle between vectors defined by the start and end points of lines

Code

```
SELECT degrees(ST_Angle('LINESTRING(0 0,0.3 0.7,1 1)', 'LINESTRING(0 0,0.2 0.5,1 0)'));
```

Figure



□□

ST_Azimuth

7.12.4 ST_ClosestPoint

ST_ClosestPoint — Returns the 2D point on g1 that is closest to g2. This is the first point of the shortest line from one geometry to the other.

□□

geometry **ST_ClosestPoint**(geometry geom1, geometry geom2);
geography **ST_ClosestPoint**(geography geom1, geography geom2, boolean use_spheroid = true);

□□

Returns the 2-dimensional point on `geom1` that is closest to `geom2`. This is the first point of the shortest line between the geometries (as computed by ST_ShortestLine).



□□
3 □□□□□□□□ ST_3DClosestPoint □□□□□□□□□□□□□.

Enhanced: 3.4.0 - Support for geography.

1.5.0 □□□□□□□□□□□□.

□□

The closest point for a Point and a LineString is the point itself. The closest point for a LineString and a Point is a point on the line.

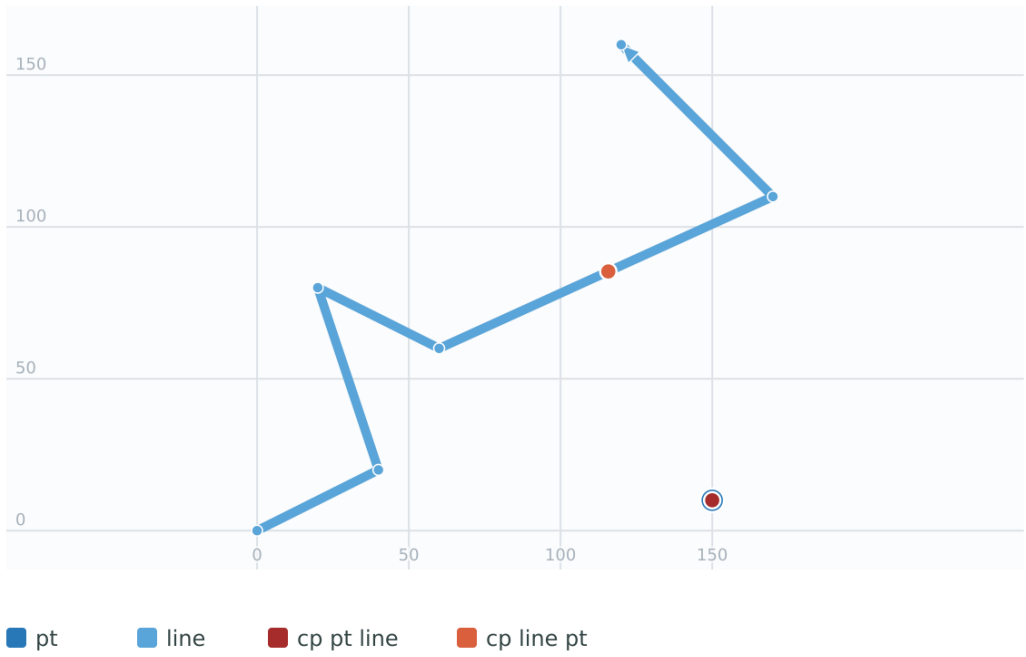
Code

```
SELECT ST_ClosestPoint(pt, line) AS cp_pt_line, ST_ClosestPoint(line, pt) AS cp_line_pt
    FROM (SELECT 'POINT (160 40)'::geometry AS pt,
                 'LINESTRING (10 30,50 50,30 110,70 90,180 140,130 190)'::geometry AS line ↩
    ) AS t;
```

Output

```
cp_pt_line   |               cp_line_pt
---------------+-----------------------------------------
 POINT(160 40)  | POINT(125.75342465753425 115.34246575342466)
```

Figure

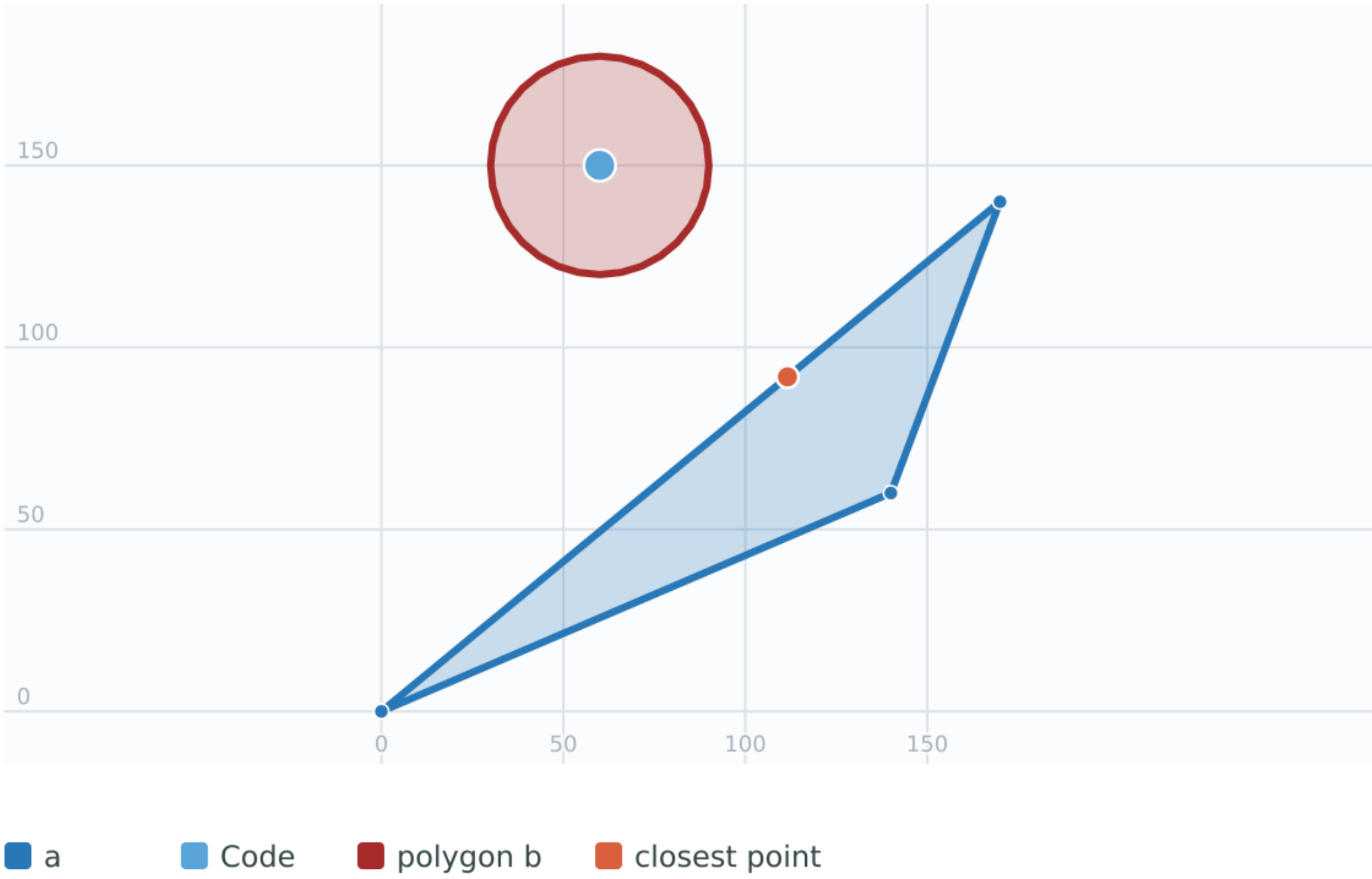


Find the closest point on polygon A to polygon B.

Code

```
WITH source AS (
  SELECT 'POLYGON ((190 150,20 10,160 70,190 150))'::geometry AS a,
         ST_Buffer('POINT(80 160)'::geometry, 30) AS b
)
SELECT b AS polygon_b,
       ST_ClosestPoint(a, b) AS closest_point
FROM source;
```

Figure



⊠⊠

ST_3DClosestPoint, ST_Distance, ST_LongestLine, ST_ShortestLine, ST_MaxDistance

7.12.5 ST_3DClosestPoint

ST_3DClosestPoint — g2 ⊠⊠⊠⊠⊠⊠ g1 ⊠⊠⊠⊠ 3 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠ 3D ⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

geometry **ST_3DClosestPoint**(geometry g1, geometry g2);

⊠⊠

g2 ⊠⊠⊠⊠⊠⊠ g1 ⊠⊠⊠⊠ 3 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠ 3D ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. 3D ⊠⊠⊠⊠⊠ 3D ⊠⊠⊠ 3D ⊠⊠⊠⊠⊠.

This function supports 3d and will not drop the z-index.

This function supports Polyhedral surfaces.

2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠⊠⊠: 2.2.0 ⊠⊠⊠⊠ 2D ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, (⊠⊠⊠⊠⊠⊠ Z ⊠ 0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠) 2D ⊠⊠ ⊠⊠⊠⊠⊠⊠⊠. 2D ⊠ 3D ⊠⊠⊠, ⊠⊠⊠ Z ⊠⊠⊠⊠ Z ⊠ 0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

Compare the 3D closest point to the 2D closest point for a LineString and a Point.

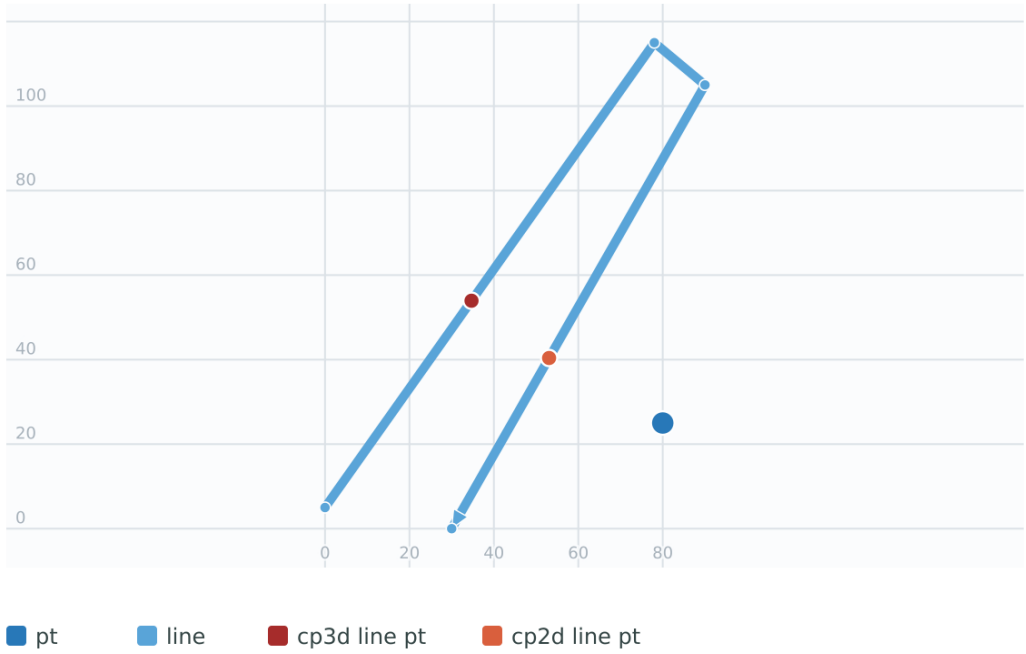
Code

```
WITH input AS (
  SELECT 'POINT(100 100 30)'::geometry AS pt,
         'LINESTRING (20 80 20,98 190 1,110 180 3,50 75 1000)'::geometry AS line
)
SELECT ST_3DClosestPoint(line, pt) AS cp3d_line_pt,
       ST_ClosestPoint(line, pt) AS cp2d_line_pt
FROM input;
```

Output

```
-[ RECORD 1 ]--+------------------------------------------------------
cp3d_line_pt   | POINT(54.69937988676193 128.93502291722837 11.547586950660556)
cp2d_line_pt   | POINT(73.07692307692307 115.38461538461539)
```

Figure



Compare the 3D closest point to the 2D closest point for a LineString and a MultiPoint.

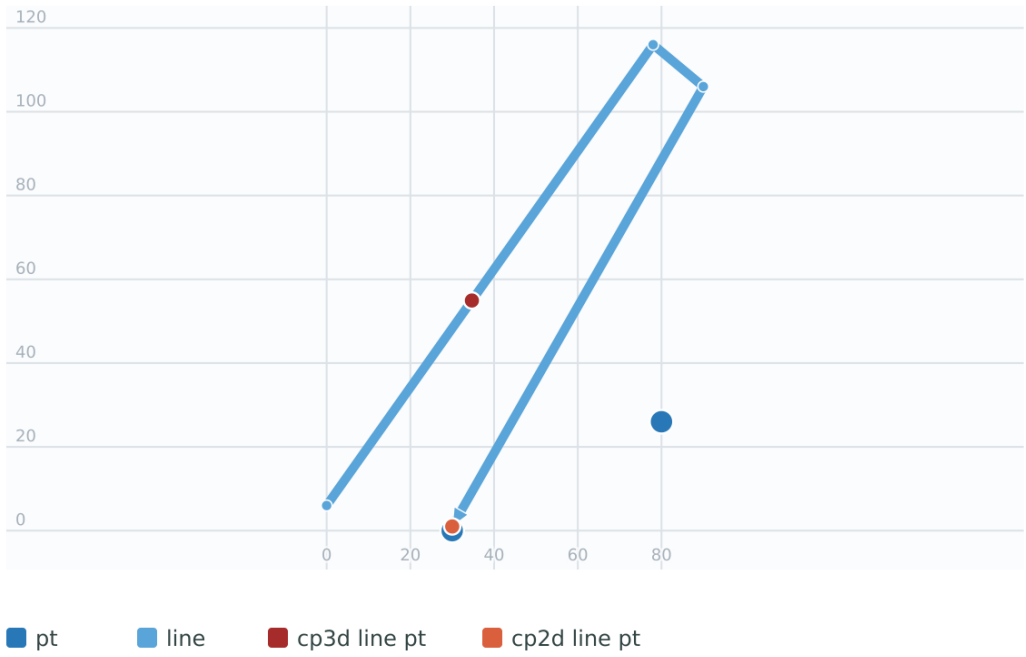
Code

```
WITH input AS (
  SELECT 'MULTIPOINT(100 100 30,50 74 1000)'::geometry AS pt,
         'LINESTRING (20 80 20,98 190 1,110 180 3,50 75 900)'::geometry AS line
)
SELECT ST_3DClosestPoint(line, pt) AS cp3d_line_pt,
       ST_ClosestPoint(line, pt) AS cp2d_line_pt
FROM input;
```

Output

```
-[ RECORD 1 ]--+------------------------------------------------------
cp3d_line_pt   | POINT(54.69937988676193 128.93502291722837 11.547586950660556)
cp2d_line_pt   | POINT(50 75)
```

Figure



Compare the 3D closest point to the 2D closest point for a Polygon and a MultiLineString.

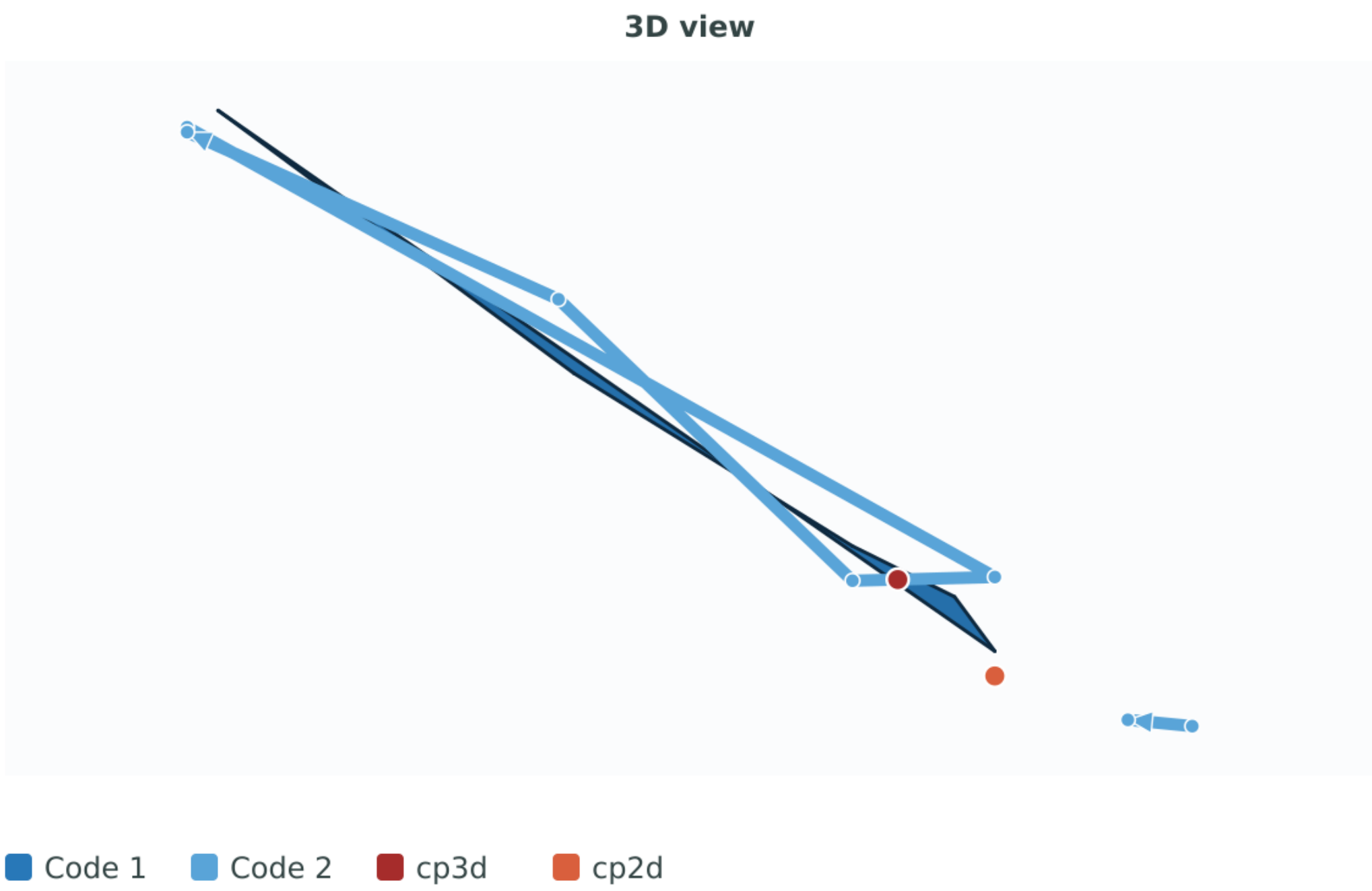
Code

```
WITH input AS (
  SELECT ST_GeomFromEWKT(
           'POLYGON((175 150 5,20 40 5,35 45 5,50 60 5,100 100 5,175 150 5))'
         ) AS poly,
         ST_GeomFromEWKT(
           'MULTILINESTRING((175 155 2,20 40 20,50 60 -2,125 100 1,175 155 1),
                            (1 10 2,5 20 1))'
         ) AS mline
)
SELECT ST_3DClosestPoint(poly, mline) AS cp3d,
       ST_ClosestPoint(poly, mline) AS cp2d
FROM input;
```

Output

```
-[ RECORD 1 ]--+-------------------------------------
cp3d           | POINT(40.45454545454545 53.63636363636364 5)
cp2d           | POINT(20 40)
```

Figure



⊠⊠

ST_AsEWKT, ST_ClosestPoint, ST_3DDistance, ST_3DShortestLine

7.12.6 ST_Distance

ST_Distance — ⊠⊠⊠⊠⊠⊠ 3 ⊠⊠⊠⊠ (longest) ⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

float **ST_Distance**(geometry g1, geometry g2);
float **ST_Distance**(geography geog1, geography geog2, boolean use_spheroid = true);

⊠⊠

⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠ 3 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (SRS ⊠⊠) ⊠⊠⊠⊠⊠⊠.

For geography types defaults to return the minimum geodesic distance between two geographies in meters, compute on the spheroid determined by the SRID. If `use_spheroid` is false, a faster spherical calculation is used.

This method implements the OGC Simple Features Implementation Specification for SQL 1.1.

This method implements the SQL/MM specification. SQL-MM 3: 5.1.23

This method supports Circular Strings and Curves.

1.5.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠.

Enhanced: 2.1.0 improved speed for geography. See Making Geography faster for details.

□□□□: 2.1.0 □□□□□□□□□□□□□□□□□□□□□.

□□□□: 2.2.0 □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ GeographicLib □□□□□□□. □□□□□□□□□□□□□ Proj 4.9.0 □□□□□□□□□□□.

Changed: 3.0.0 - does not depend on SFCGAL anymore.

□□□□

Geometry example - units in planar degrees 4326 is WGS 84 long lat, units are degrees.

Code

```
SELECT ST_Distance(
'SRID=4326;POINT(-72.1235 42.3521)'::geometry,
'SRID=4326;LINESTRING(-72.1260 42.45,-72.123 42.1546)'::geometry );
```

Output

```
0.00150567726382282
```

Geometry example - units in meters (SRID: 3857, proportional to pixels on popular web maps). Although the value is off, nearby ones can be compared correctly, which makes it a good choice for algorithms like KNN or KMeans.

Code

```
SELECT ST_Distance(
ST_Transform('SRID=4326;POINT(-72.1235 42.3521)'::geometry, 3857),
ST_Transform('SRID=4326;LINESTRING(-72.1260 42.45,-72.123 42.1546)'::geometry, 3857) );
```

Output

```
167.441410065196
```

Geometry example - units in meters (SRID: 3857 as above, but corrected by cos(lat) to account for distortion)

Code

```
SELECT ST_Distance(
ST_Transform('SRID=4326;POINT(-72.1235 42.3521)'::geometry, 3857),
ST_Transform('SRID=4326;LINESTRING(-72.1260 42.45,-72.123 42.1546)'::geometry, 3857)
        ) * cosd(42.3521);
```

Output

```
123.742351254151
```

Geometry example - units in meters (SRID: 26986 Massachusetts state plane meters) (most accurate for Massachusetts)

Code

```
SELECT ST_Distance(
ST_Transform('SRID=4326;POINT(-72.1235 42.3521)'::geometry, 26986),
ST_Transform('SRID=4326;LINESTRING(-72.1260 42.45,-72.123 42.1546)'::geometry, 26986) );
```

Output

```
123.797937878454
```

Geometry example - units in meters (SRID: 2163 US National Atlas Equal area) (least accurate)

Code

```
SELECT ST_Distance(
ST_Transform('SRID=4326;POINT(-72.1235 42.3521)'::geometry, 2163),
ST_Transform('SRID=4326;LINESTRING(-72.1260 42.45,-72.123 42.1546)'::geometry, 2163) );
```

Output

```
126.664256056812
```

□□□□□

Same as geometry example but note units in meters - use sphere for slightly faster and less accurate computation.

Code

```
SELECT ST_Distance(gg1, gg2) As spheroid_dist, ST_Distance(gg1, gg2, false) As sphere_dist
FROM (SELECT
    'SRID=4326;POINT(-72.1235 42.3521)'::geography as gg1,
    'SRID=4326;LINESTRING(-72.1260 42.45,-72.123 42.1546)'::geography as gg2
    ) As foo  ;
```

Output

```
spheroid_dist   |   sphere_dist
------------------+------------------
 123.802076746848 | 123.475736916397
```

□□

ST_3DDistance, ST_DWithin, ST_DistanceSphere, ST_DistanceSpheroid, ST_MaxDistance, ST_HausdorffDis
ST_FrechetDistance, ST_Transform

7.12.7 ST_3DDistance

ST_3DDistance — □□□□□□□, □□□□□□ (SRS □□□□) 3 □□□□□□□□□□□□□□□□□□□ □□.

□□

float **ST_3DDistance**(geometry g1, geometry g2);

□□

□□□□□□□, □□□□□□ 3 □□□□□□□□□□□□□□□□□□□□ (SRS □□) □□□□□□.

This function supports 3d and will not drop the z-index.

This function supports Polyhedral surfaces.

This method implements the SQL/MM specification. SQL-MM ISO/IEC 13249-3

2.0.0 □□□□□□□□□□□□.

□□□□: 2.2.0 □□□□, 2D □ 3D □□□□□□□ Z □□□□ Z □ 0 □□□□□□□□□□.

Changed: 3.0.0 - SFCGAL version removed

□□

Geometry example - units in meters (SRID: 2163 US National Atlas Equal area) (3D point and line compared 2D point and line) Note that currently no vertical datum support so Z is not transformed and assumed to be same units as final.

Code

```
SELECT ST_3DDistance(
            ST_Transform('SRID=4326;POINT(-72.1235 42.3521 4)'::geometry, 2163),
                ST_Transform('SRID=4326;LINESTRING(-72.1260 42.45 15,-72.123 42.1546 20)':: ↩
    geometry, 2163)
            ) As dist_3d,
            ST_Distance(
                ST_Transform('SRID=4326;POINT(-72.1235 42.3521)'::geometry, 2163),
                ST_Transform('SRID=4326;LINESTRING(-72.1260 42.45,-72.123 42.1546)':: ↩
    geometry, 2163)
            ) As dist_2d;
```

Output

```
dist_3d      |     dist_2d
------------------+-----------------
 127.295059324629 | 126.66425605671
```

Multilinestring and polygon both 3D and 2D distance. Same example as 3D closest point example.

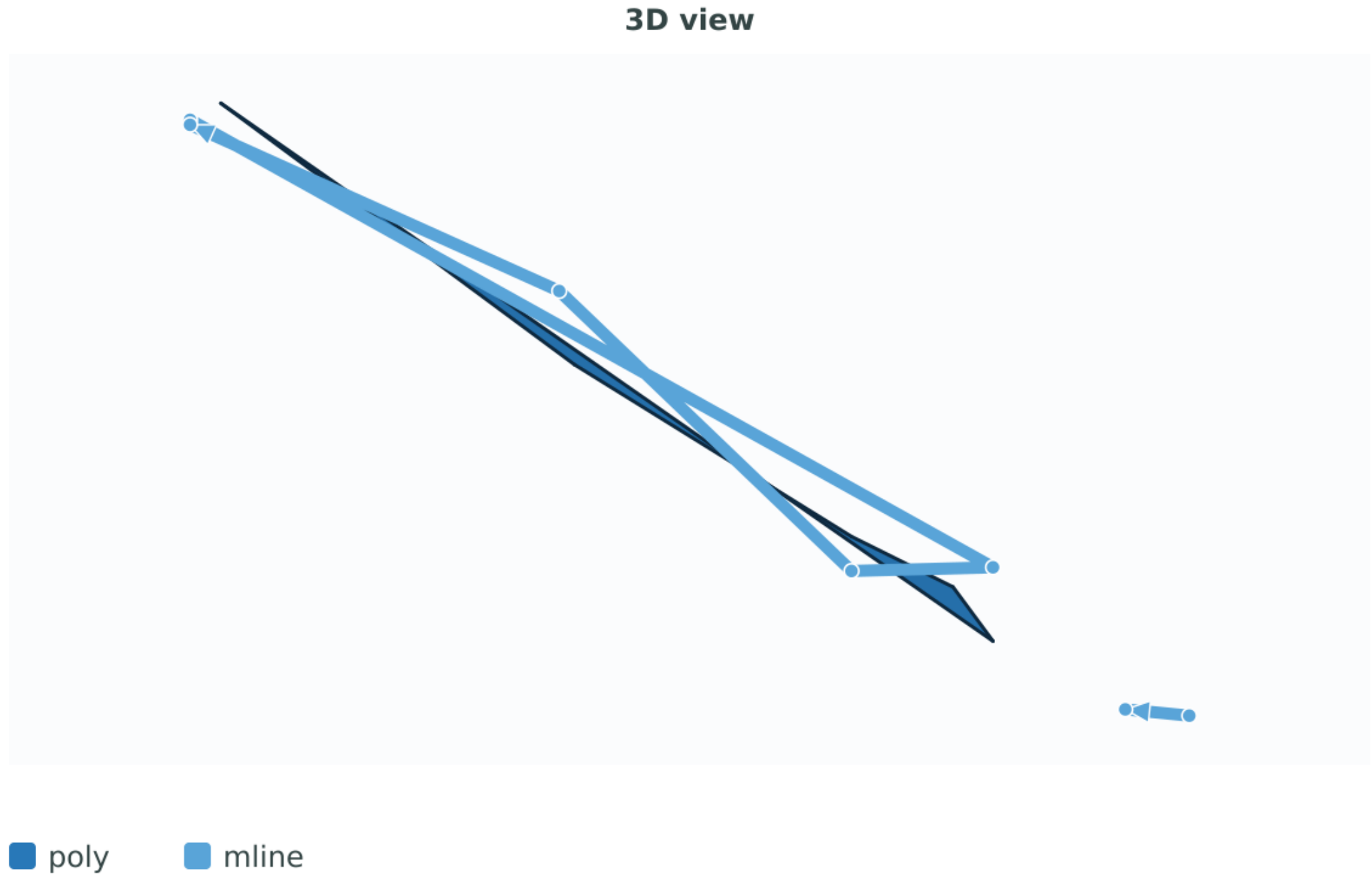
Code

```
SELECT ST_3DDistance(poly, mline) As dist3d,
    ST_Distance(poly, mline) As dist2d
        FROM (SELECT  'POLYGON((175 150 5,20 40 5,35 45 5,50 60 5,100 100 5,175 150 5))':: ↩
    geometry as poly,
               'MULTILINESTRING((175 155 2,20 40 20,50 60 -2,125 100 1,175 155 1),(1 10 2,5 ↩
     20 1))'::geometry as mline) as foo;
```

Output

```
dist3d       | dist2d
------------------+--------
                0 |      0
```

Figure



□□

ST_Distance, ST_3DClosestPoint, ST_3DDWithin, ST_3DMaxDistance, ST_3DShortestLine, ST_Transform

7.12.8 ST_DistanceSphere

ST_DistanceSphere — □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□. PostGIS 1.5 □□ □□□□□□□□□□□□□□.

□□

float **ST_DistanceSphere**(geometry geomlonlatA, geometry geomlonlatB, float8 radius=6371008);

□□

□□□□□□□ 2 □□□□□□□□□□□□□□□□□□□. SRID □□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□. ST_DistanceSpheroid □□□□□□□□□□□□, □□□□□□□□□□. PostGIS 1.5 □ □□□□□□□□□□□□□□□□□□□□□□.

1.5 □□□□□□□□□□□□□□□□□□□□□□□□□□□□. 1.5 □□□□□□□□□□□□□□□□□□□□ □□□□.

□□□□: 2.2.0 □□□□□□□ ST_Distance_Sphere □□□□□□□□□.

□□

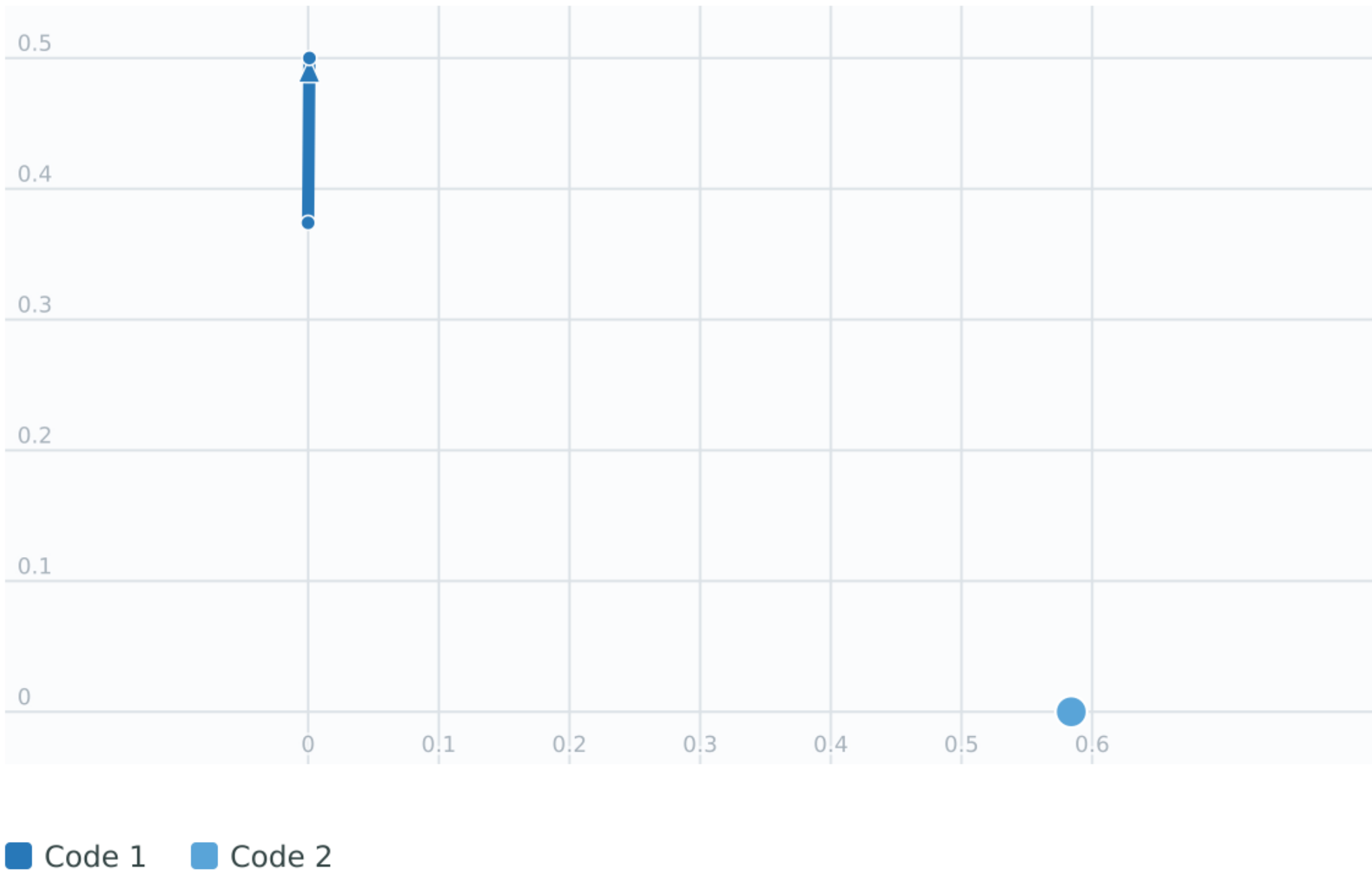
Code

```
WITH sample AS (
    SELECT
        ST_GeomFromText('LINESTRING(-118.584 38.374,-118.583 38.5)', 4326) AS geom,
        ST_GeomFromText('POINT(-118 38)', 4326) AS pt
)
SELECT
    round(ST_DistanceSphere(ST_Centroid(geom), pt)::numeric, 2) AS dist_meters,
    round(ST_Distance(
        ST_Transform(ST_Centroid(geom), 32611),
        ST_Transform(pt, 32611)
    )::numeric, 2) AS dist_utm11_meters,
    round(ST_Distance(ST_Centroid(geom), pt)::numeric, 5) AS dist_degrees,
    round(ST_Distance(
        ST_Transform(geom, 32611),
        ST_Transform(pt, 32611)
    )::numeric, 2) AS min_dist_line_point_meters
FROM sample;
```

Output

```
dist_meters | dist_utm11_meters | dist_degrees | min_dist_line_point_meters
------------+-------------------+--------------+---------------------------
   70424.71 |          70438.00 |      0.72900 |                   65871.18
```

Figure



⊠⊠

ST_Distance, ST_DistanceSpheroid

7.12.9 ST_DistanceSpheroid

ST_DistanceSpheroid — ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. PostGIS 1.5 ⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

float **ST_DistanceSpheroid**(geometry geomlonlatA, geometry geomlonlatB, spheroid measurement_sphero

⊠⊠

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ST_LengthSpheroi ⊠⊠⊠⊠⊠⊠⊠. PostGIS 1.5 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

> **⊠⊠**
> ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ SRID ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

1.5 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. 1.5 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠.

⊠⊠⊠⊠: 2.2.0 ⊠⊠⊠⊠⊠⊠⊠ ST_Distance_Spheroid ⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

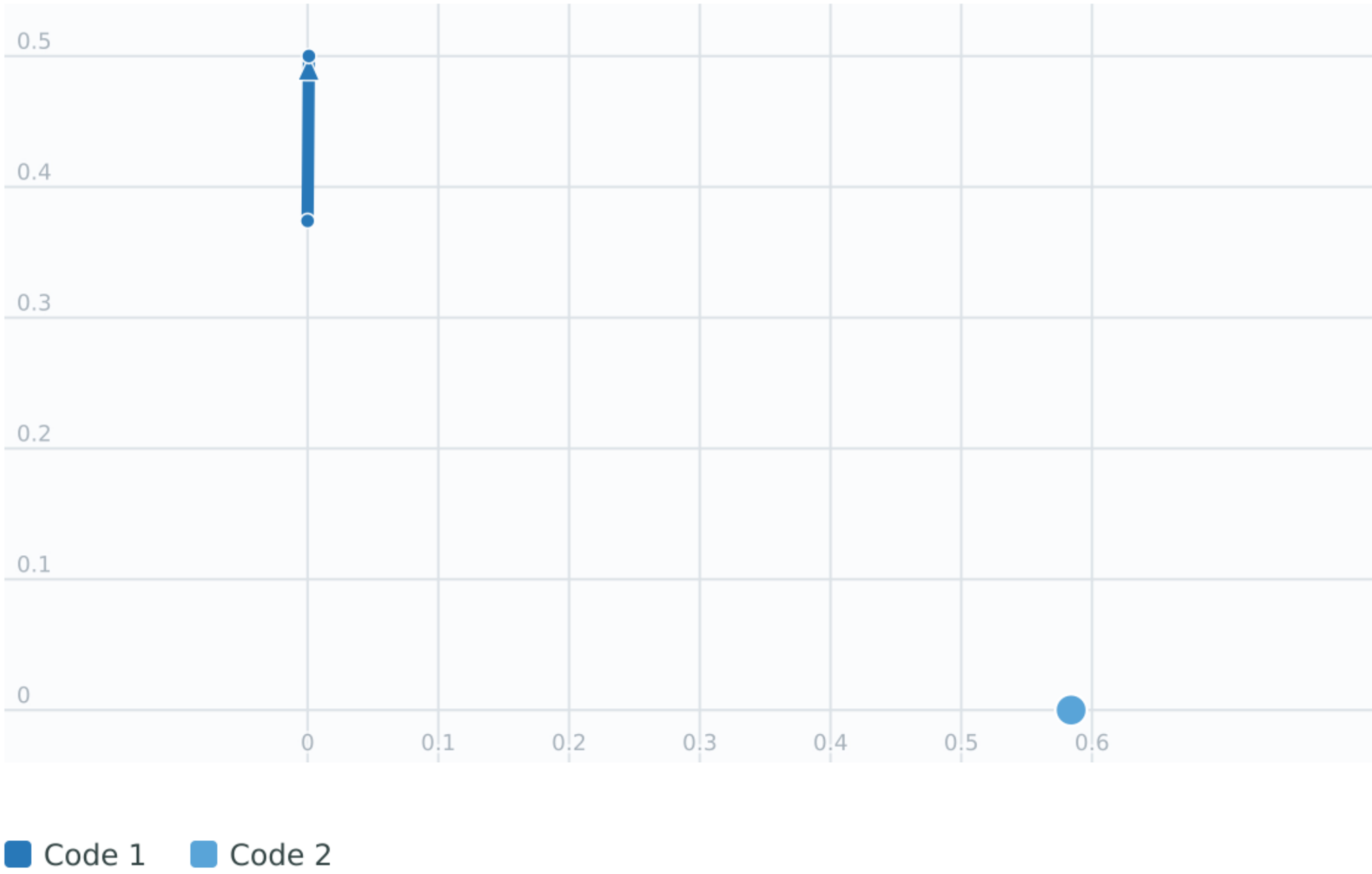
Code

```
WITH input AS (
  SELECT ST_Centroid(
      ST_GeomFromText(
        'LINESTRING(-118.584 38.374,-118.583 38.5)', 4326
      )
    ) AS source,
    ST_GeomFromText('POINT(-118 38)', 4326) AS target
)
SELECT round(
    ST_DistanceSpheroid(
      source,
      target,
      'SPHEROID["WGS 84",6378137,298.257223563]'
    )::numeric,
    2
  ) AS dist_meters_spheroid,
  round(ST_DistanceSphere(source, target)::numeric, 2)
    AS dist_meters_sphere,
  round(
    ST_Distance(
      ST_Transform(source, 32611),
      ST_Transform(target, 32611)
    )::numeric,
    2
  ) AS dist_utm11_meters
FROM input;
```

Output

```
dist_meters_spheroid | dist_meters_sphere | dist_utm11_meters
----------------------+--------------------+-------------------
             70454.92 |           70424.71 |          70438.00
```

Figure



□□

ST_Distance, ST_DistanceSphere

7.12.10 ST_FrechetDistance

ST_FrechetDistance — □□□□□□ 3 □□□□ (shortest) □□□□□□□□□.

□□

float **ST_FrechetDistance**(geometry g1, geometry g2, float densifyFrac = -1);

□□

Implements algorithm for computing the Fréchet distance restricted to discrete points for both geometries, based on Computing Discrete Fréchet Distance. The Fréchet distance is a measure of similarity between curves that takes into account the location and ordering of the points along the curves. Therefore it is often better than the Hausdorff distance.

When the optional densifyFrac is specified, this function performs a segment densification before computing the discrete Fréchet distance. The densifyFrac parameter sets the fraction by which to densify each segment. Each segment will be split into a number of equal-length subsegments, whose fraction of the total length is closest to the given fraction.

Units are in the units of the spatial reference system of the geometries.

> **Note!**
> □□
> □□□□□□□□□□□□□□□□□□□□□□□□□. □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□.



☒☒

The smaller densifyFrac we specify, the more accurate Fréchet distance we get. But, the computation time and the memory usage increase with the square of the number of subsegments.

GEOS ☒☒☒☒☒

Availability: 2.4.0 - requires GEOS >= 3.7.0

☒☒

Both lines are traversed from left to right. The first line is straight, while the second bows upward through its middle vertex. The point ordering matters, so keeping both inputs in the same direction makes the comparison clearer.

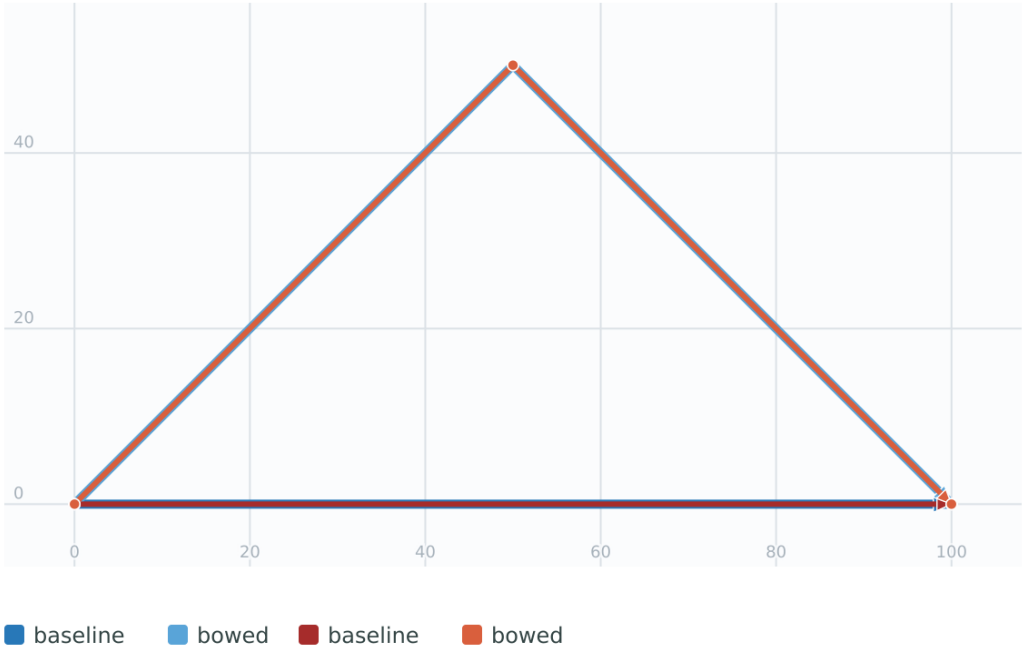
Code

```
WITH lines AS (
  SELECT 'LINESTRING (0 0,100 0)'::geometry AS baseline,
         'LINESTRING (0 0,50 50,100 0)'::geometry AS bowed
)
SELECT baseline AS baseline,
       bowed AS bowed,
       ST_FrechetDistance(baseline, bowed) AS discrete_distance,
       ST_FrechetDistance(baseline, bowed, 0.5) AS densified_distance
FROM lines;
```

Output

```
-[ RECORD 1 ]-----+---------------------------
baseline           | LINESTRING(0 0,100 0)
bowed              | LINESTRING(0 0,50 50,100 0)
discrete_distance  | 70.7106781186548
densified_distance | 50
```

Figure



□□

ST_HausdorffDistance

7.12.11 ST_HausdorffDistance

ST_HausdorffDistance — □□□□□□ 3 □□□□ (shortest) □□□□□□□□.

□□

float **ST_HausdorffDistance**(geometry g1, geometry g2);
float **ST_HausdorffDistance**(geometry g1, geometry g2, float densifyFrac);

□□

Returns the Hausdorff distance between two geometries. The Hausdorff distance is a measure of how similar or dissimilar 2 geometries are.

The function actually computes the "Discrete Hausdorff Distance". This is the Hausdorff distance computed at discrete points on the geometries. The *densifyFrac* parameter can be specified, to provide a more accurate answer by densifying segments before computing the discrete Hausdorff distance. Each segment is split into a number of equal-length subsegments whose fraction of the segment length is closest to the given fraction.

Units are in the units of the spatial reference system of the geometries.



□□
This algorithm is NOT equivalent to the standard Hausdorff distance. However, it computes an approximation that is correct for a large subset of useful cases. One important case is Linestrings that are roughly parallel to each other, and roughly equal in length. This is a useful metric for line matching.

1.5.0 □□□□□□□□□□□□.

□□

Hausdorff distance and minimum distance between two lines. The Hausdorff witness segment from the original illustration is retained and its length is checked against the computed distance.

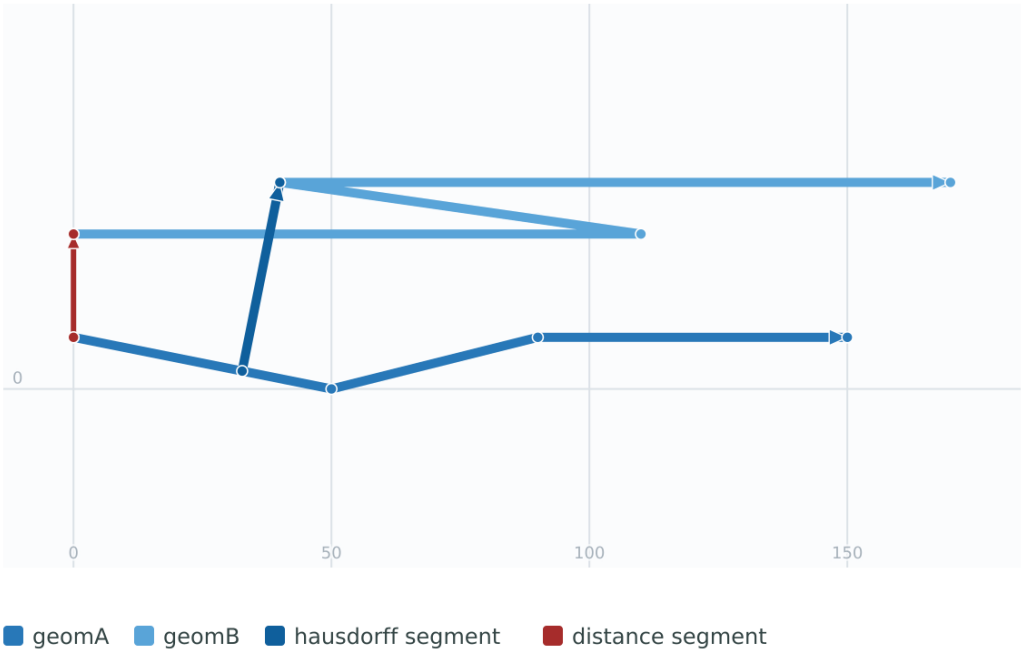
Code

```
WITH lines AS (
  SELECT 'LINESTRING (20 70,70 60,110 70,170 70)'::geometry AS geomA,
         'LINESTRING (20 90,130 90,60 100,190 100)'::geometry AS geomB,
         'LINESTRING (52.69230769230769 63.46153846153846,60 100)'::geometry AS ↩
    hausdorff_segment
)
SELECT round(ST_HausdorffDistance(geomA, geomB)::numeric, 6) AS hausdorff_distance,
       round(ST_Distance(geomA, geomB)::numeric, 6) AS distance,
       round(ST_Length(hausdorff_segment)::numeric, 6) AS witness_length,
       ST_ShortestLine(geomA, geomB) AS distance_segment
FROM lines;
```

Output

```
37.262066 | 20.000000 | 37.262066 | LINESTRING(20 70,20 90)
```

Figure

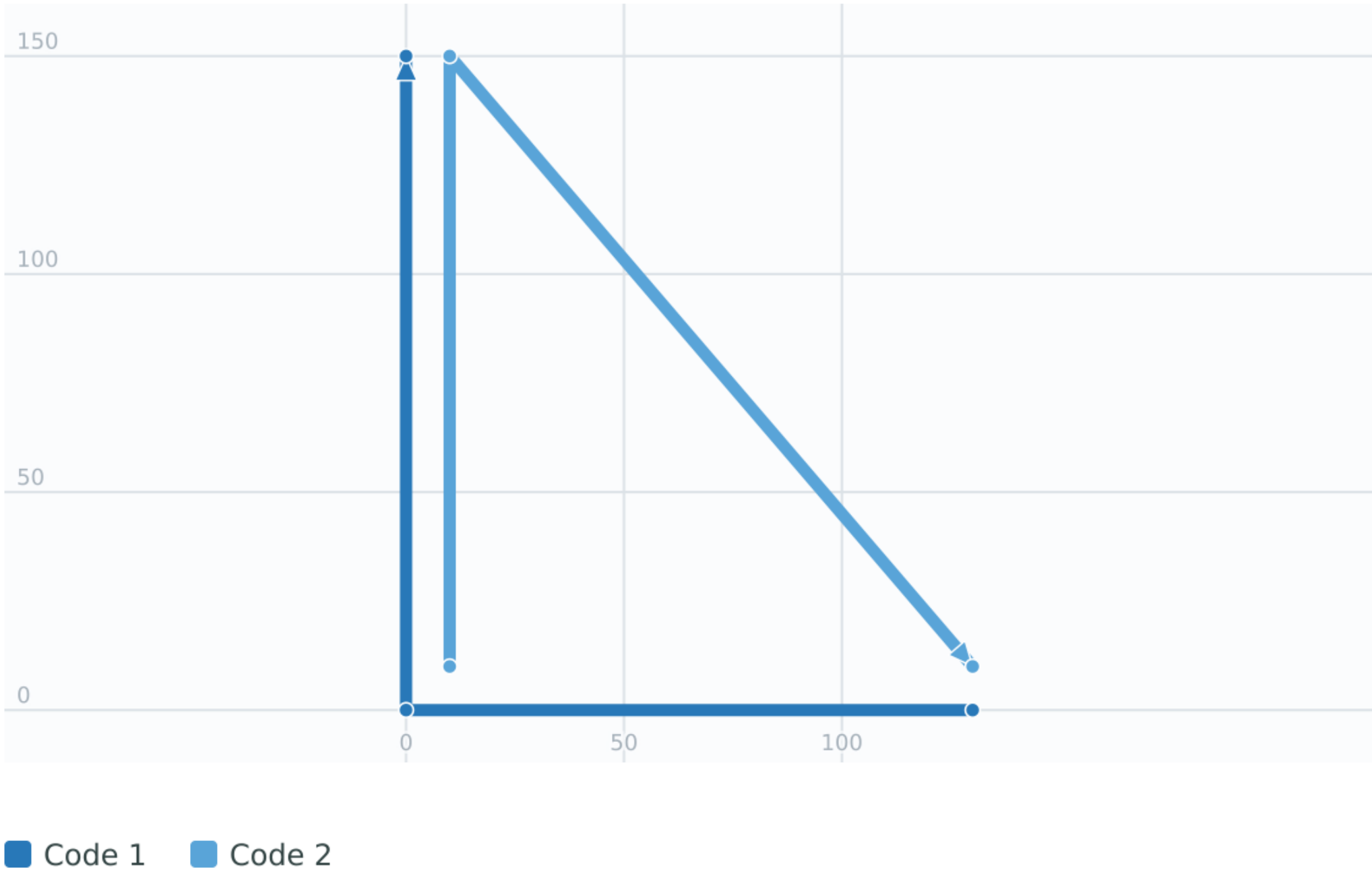


Example: Hausdorff distance with densification.

Code

```
SELECT ST_HausdorffDistance(
            'LINESTRING (130 0,0 0,0 150)'::geometry,
            'LINESTRING (10 10,10 150,130 10)'::geometry,
            0.5);
```

Figure



Example: For each building, find the parcel that best represents it. First we require that the parcel intersect with the building geometry. `DISTINCT ON` guarantees we get each building listed only once. `ORDER BY .. ST_HausdorffDistance` selects the parcel that is most similar to the building.

Code

```
SELECT DISTINCT ON (buildings.gid) buildings.gid, parcels.parcel_id
   FROM buildings
       INNER JOIN parcels
       ON ST_Intersects(buildings.geom, parcels.geom)
   ORDER BY buildings.gid, ST_HausdorffDistance(buildings.geom, parcels.geom);
```

⊠⊠

ST_FrechetDistance

7.12.12 ST_Length

ST_Length — ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

float **ST_Length**(geometry a_2dlinestring);
float **ST_Length**(geography geog, boolean use_spheroid = true);

⊠⊠

⊠⊠⊠⊠⊠: ⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠, ST_Curve, ST_MultiCurve ⊠⊠⊠⊠⊠⊠⊠ 2 ⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠ 0 ⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠ ST_Perimeter ⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

For geography types: computation is performed using the inverse geodesic calculation. Units of length are in meters. The spheroid is specified by the SRID; when no SRID is provided, WGS84 is used. If `use_spheroid = false`, then the calculation is based on a sphere instead of a spheroid.

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ST_Length2D ⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠⊠⊠: 2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. 2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠/⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠/⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. 2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ 0 ⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ST_Perimeter ⊠⊠⊠⊠⊠⊠⊠⊠⊠.

> **⊠⊠**
>
> ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ST_Length(gg,false); ⊠⊠⊠⊠⊠.

This method implements the OGC Simple Features Implementation Specification for SQL 1.1. s2.1.5.1

This method implements the SQL/MM specification. SQL-MM 3: 7.1.2, 9.3.4

1.5.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

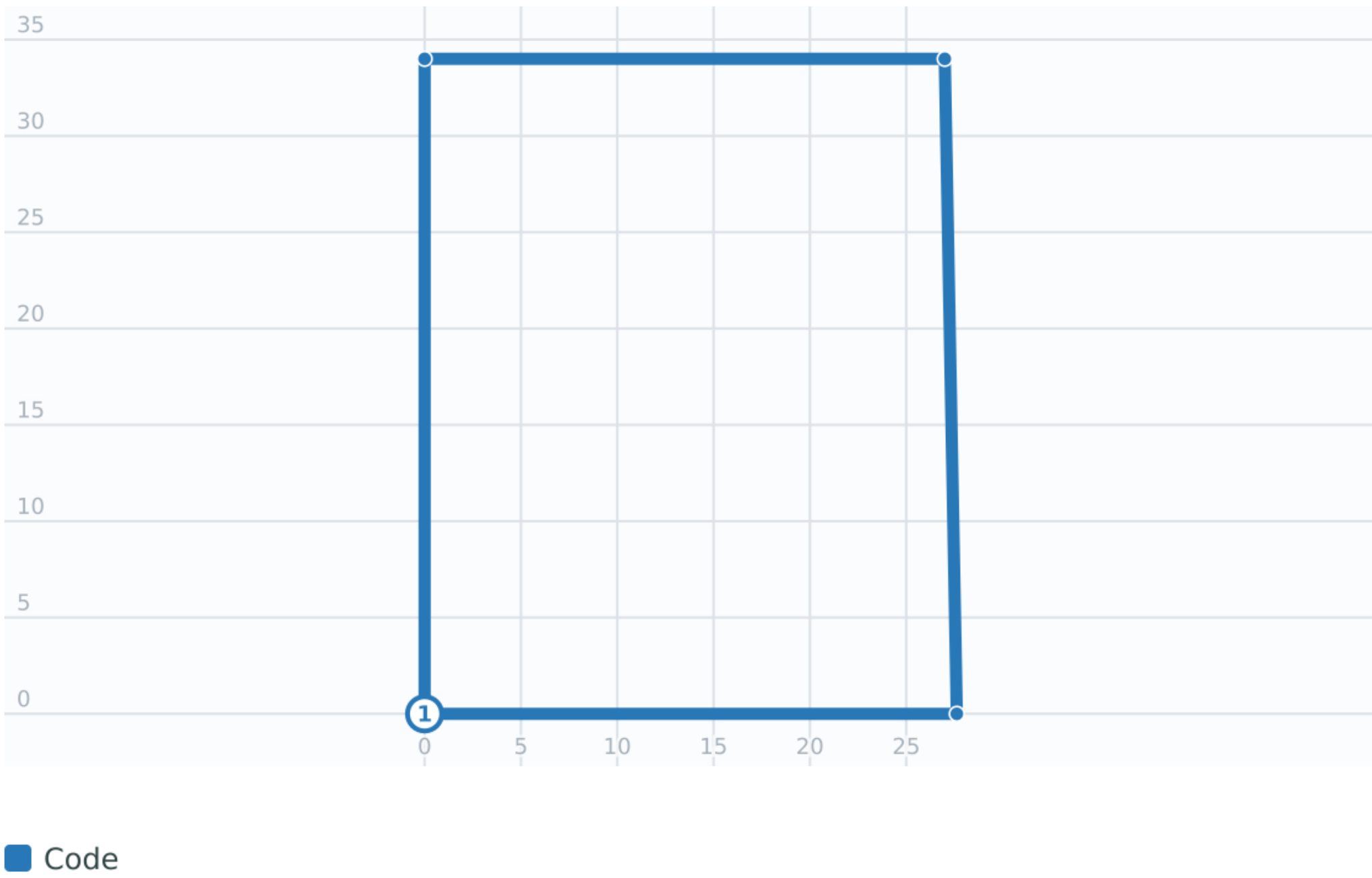
⊠⊠⊠⊠

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠ EPSG:2249 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

Code

```
SELECT ST_Length(ST_GeomFromText(
'LINESTRING(743238 2967416,743238 2967450,743265 2967450,
743265.625 2967416,743238 2967416)',
2249));
```

Figure

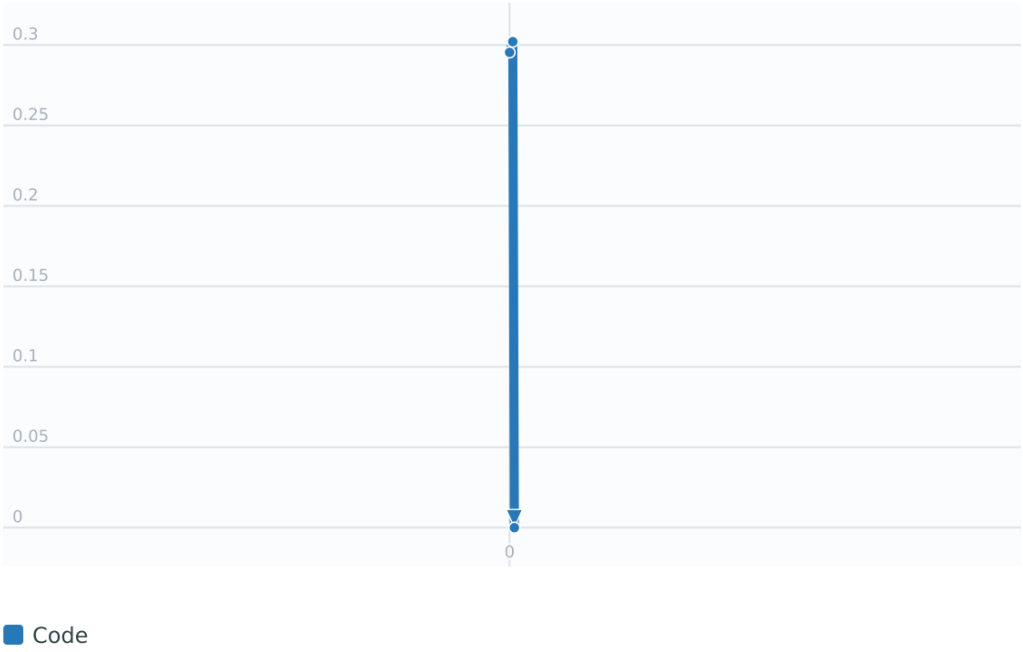


This example transforms a WGS 84 LineString to Massachusetts State Plane meters.

Code

```
SELECT ST_Length(ST_Transform(
        'SRID=4326;LINESTRING(-72.1260 42.45,-72.1240 42.45666,-72.123 42.1546)'::geometry,
            26986
    )
);
```

Figure



□□□□□

WGS84 □□□□□□□□□□□□□□□□□□□□.

This example uses the default spheroid calculation.

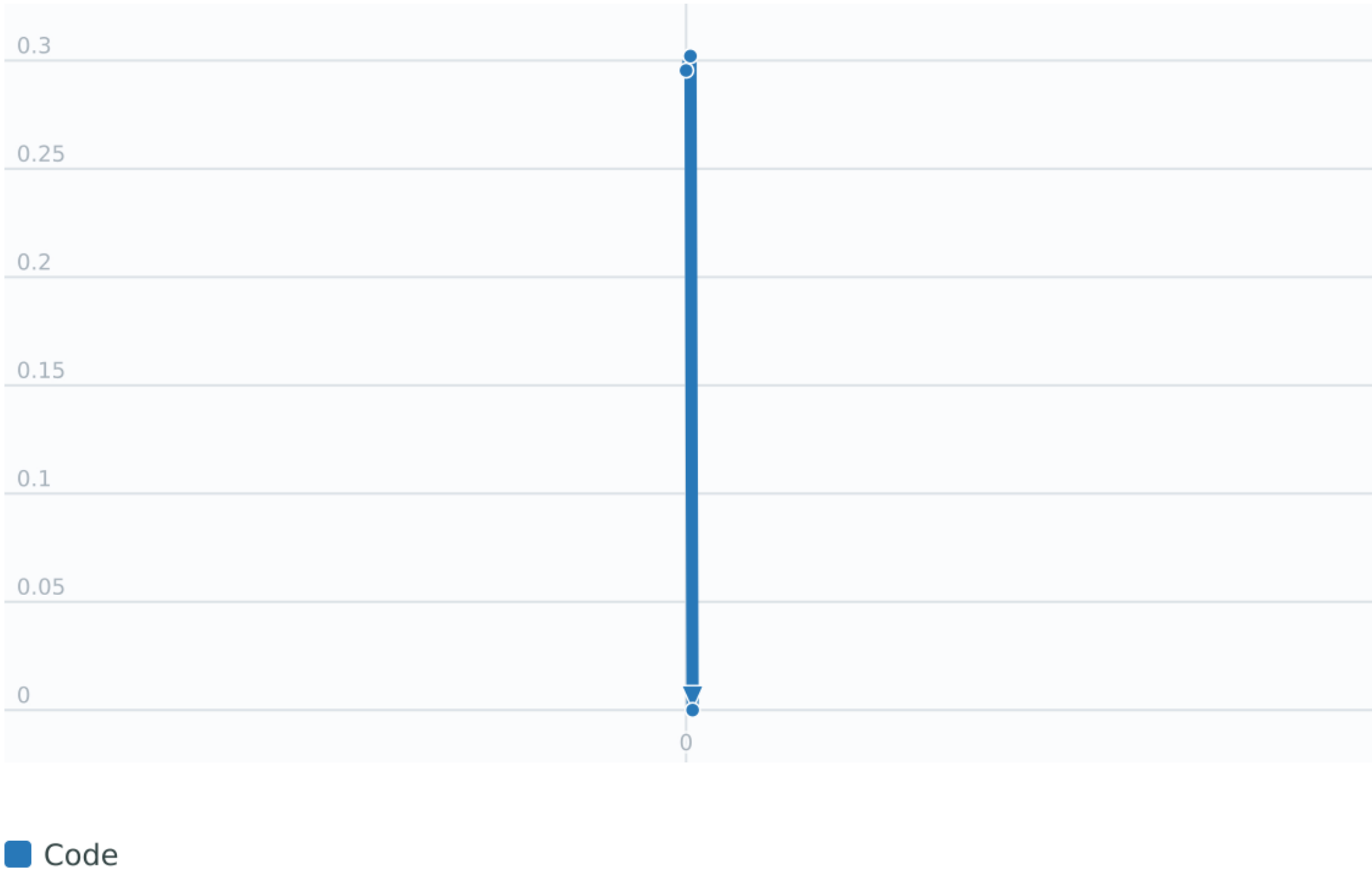
Code

```
SELECT ST_Length(the_geog) As length_spheroid, ST_Length(the_geog, false) As length_sphere
FROM (SELECT ST_GeographyFromText('SRID=4326;LINESTRING(-72.1260 42.45,-72.1240 ↩
    42.45666,-72.123 42.1546)') As the_geog)
 As foo;
```

Output

```
length_spheroid  |  length_sphere
-----------------+-----------------
34310.5703627284 | 34346.20609607416
```

Figure



☒☒

ST_GeographyFromText, ST_GeomFromEWKT, ST_LengthSpheroid, ST_Perimeter, ST_Transform

7.12.13 ST_Length2D

ST_Length2D — ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒ 2 ☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒ `ST_Length` ☒☒☒☒☒☒.

☒☒

float **ST_Length2D**(geometry a_2dlinestring);

☒☒

☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒ 2 ☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒ `ST_Length` ☒☒☒☒☒☒.

☒☒

ST_Length, ST_3DLength

7.12.14 ST_3DLength

ST_3DLength — ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.

⊠⊠

float **ST_3DLength**(geometry a_3dlinestring);

⊠⊠

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ 3 ⊠⊠⊠⊠ 2 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. 2 ⊠⊠⊠⊠⊠⊠⊠ 2 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (ST_Length ⊠ ST_Length2D ⊠⊠⊠⊠⊠⊠).

This function supports 3d and will not drop the z-index.

This method implements the SQL/MM specification. SQL-MM IEC 13249-3: 7.1, 10.3

⊠⊠⊠⊠: 2.0.0 ⊠⊠⊠⊠⊠⊠⊠ ST_Length3D ⊠⊠⊠⊠⊠⊠⊠⊠⊠.

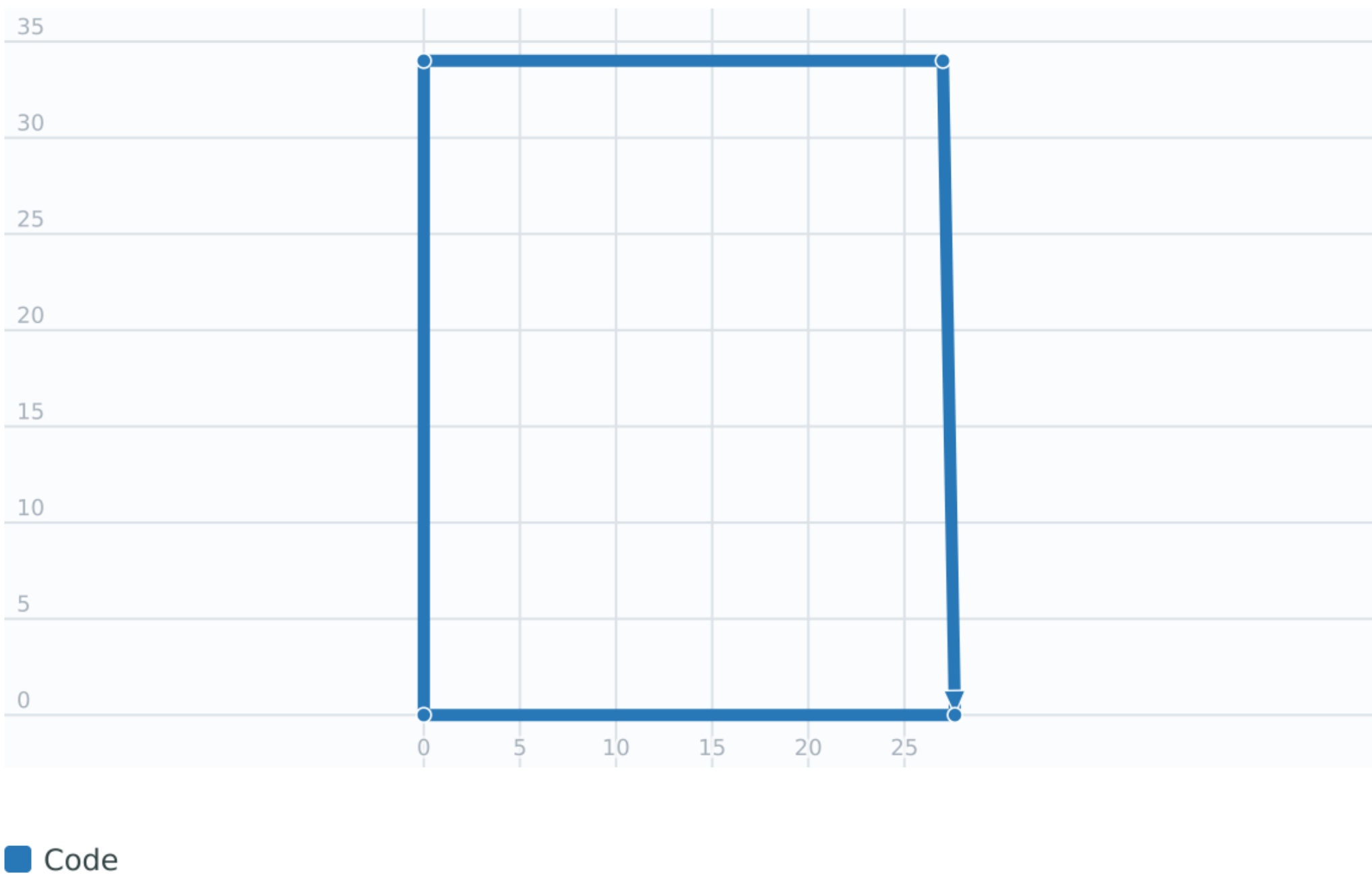
⊠⊠

3 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠ EPSG:2249 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

Code

```
SELECT ST_3DLength(ST_GeomFromText(
'LINESTRING(743238 2967416 1,743238 2967450 1,743265 2967450 3,
743265.625 2967416 3,743238 2967416 3)',
2249));
```

Figure



⊠⊠

ST_Length, ST_Length2D

7.12.15 ST_LengthSpheroid

ST_LengthSpheroid — □□□□□□□□□□□□□□□□□□.

□□

float **ST_LengthSpheroid**(geometry a_geometry, spheroid a_spheroid);

□□

Calculates the length or perimeter of a geometry on a spheroid. This is useful if the coordinates of the geometry are in longitude/latitude and a length is desired without reprojection. The spheroid is specified by a text value as follows:

```
SPHEROID[<NAME>,<SEMI-MAJOR AXIS>,<INVERSE FLATTENING>]
```

□□□□

```
SPHEROID["GRS_1980",6378137,298.257222101]
```

1.2.2 □□□□□□□□□□□□□□.

□□□□: 2.2.0 □□□□□□□□ ST_Length_Spheroid □□□□□□□□, ST_3DLength_Spheroid □□□□ □□□□□□□□□□.

This function supports 3d and will not drop the z-index.

□□

Measure a table geometry directly on a spheroid when the coordinates are stored in longitude and latitude.

Code

```
SELECT ST_LengthSpheroid(
    geometry_column,
    'SPHEROID["GRS_1980",6378137,298.257222101]'
    )
FROM geometry_table;
```

This example measures a 2D MultiLineString and each component separately on the same spheroid.

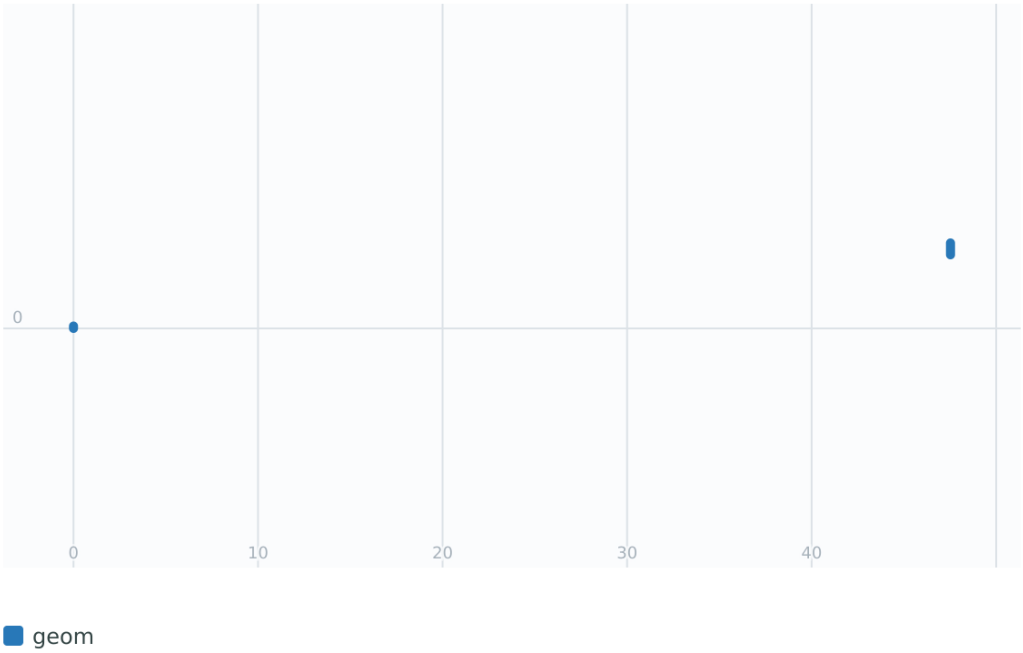
Code

```
WITH sample AS (
  SELECT 'MULTILINESTRING((-118.584 38.374,-118.583 38.5),
          (-71.05957 42.3589,-71.061 43))'::geometry AS geom,
         CAST('SPHEROID["GRS_1980",6378137,298.257222101]' AS spheroid) AS sph_m
)
SELECT ST_LengthSpheroid(geom, sph_m) AS tot_len,
       ST_LengthSpheroid(ST_GeometryN(geom, 1), sph_m) AS len_line1,
       ST_LengthSpheroid(ST_GeometryN(geom, 2), sph_m) AS len_line2
FROM sample;
```

Output

```
tot_len      |    len_line1     |    len_line2
-----------------+-----------------+-----------------
 85204.52077117894 | 13986.87252824332 | 71217.64824293563
```

Figure



This example uses 3D input; Z contributes to the measured result.

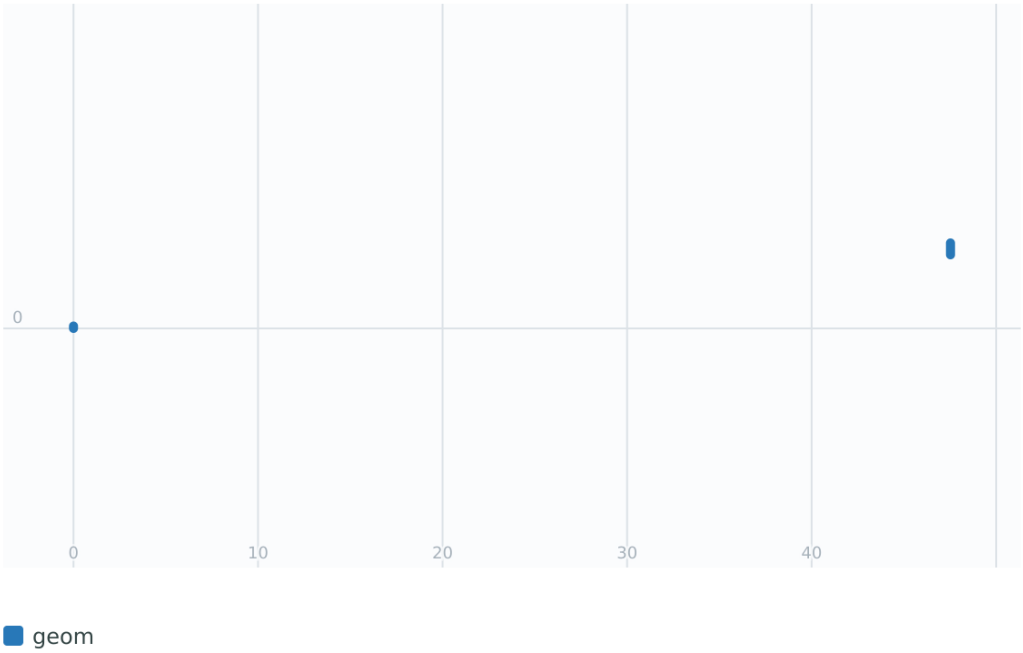
Code

```
WITH sample AS (
  SELECT 'MULTILINESTRING((-118.584 38.374 20,-118.583 38.5 30),
          (-71.05957 42.3589 75,-71.061 43 90))'::geometry AS geom,
         CAST('SPHEROID["GRS_1980",6378137,298.257222101]' AS spheroid) AS sph_m
)
SELECT ST_LengthSpheroid(geom, sph_m) AS tot_len,
       ST_LengthSpheroid(ST_GeometryN(geom, 1), sph_m) AS len_line1,
       ST_LengthSpheroid(ST_GeometryN(geom, 2), sph_m) AS len_line2
FROM sample;
```

Output

```
tot_len      |    len_line1    |    len_line2
-----------------+-----------------+-----------------
 85204.52592562366 | 13986.876103023424 | 71217.64982260024
```

Figure



☒☒

ST_GeometryN, ST_Length

7.12.16 ST_LongestLine

ST_LongestLine — ☒☒☒☒☒☒ 3 ☒☒☒☒ (longest) ☒☒☒☒☒☒☒☒.

☒☒

geometry **ST_LongestLine**(geometry g);
geometry **ST_LongestLine**(geometry g1, geometry g2);

☒☒

Returns the 2-dimensional longest line between the points of two geometries. The line returned starts on `g1` and ends on `g2`.

The longest line always occurs between two vertices. The function returns the first longest line if more than one is found. The length of the line is equal to the distance returned by ST_MaxDistance.

If only one geometry is provided, or g1 and g2 are the same geometry, returns the line between the two vertices farthest apart in the geometry. The endpoints of the line lie on the circle computed by ST_MinimumBoundingCircle.

1.5.0 ☒☒☒☒☒☒☒☒☒☒☒☒.

Enhanced: 3.7.0 - support for a single geometry input.

□□

Longest line between a Point and a LineString.

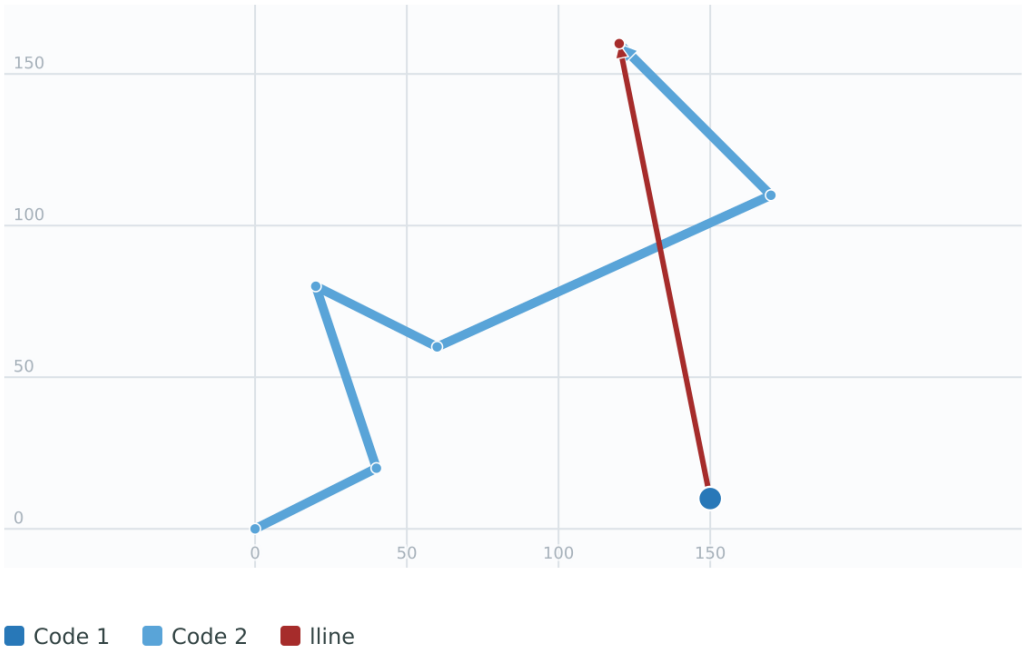
Code

```
SELECT ST_LongestLine(
        'POINT (160 40)',
        'LINESTRING (10 30,50 50,30 110,70 90,180 140,130 190)' ) AS lline;
```

Output

```
LINESTRING(160 40,130 190)
```

Figure



Longest line between two polygons.

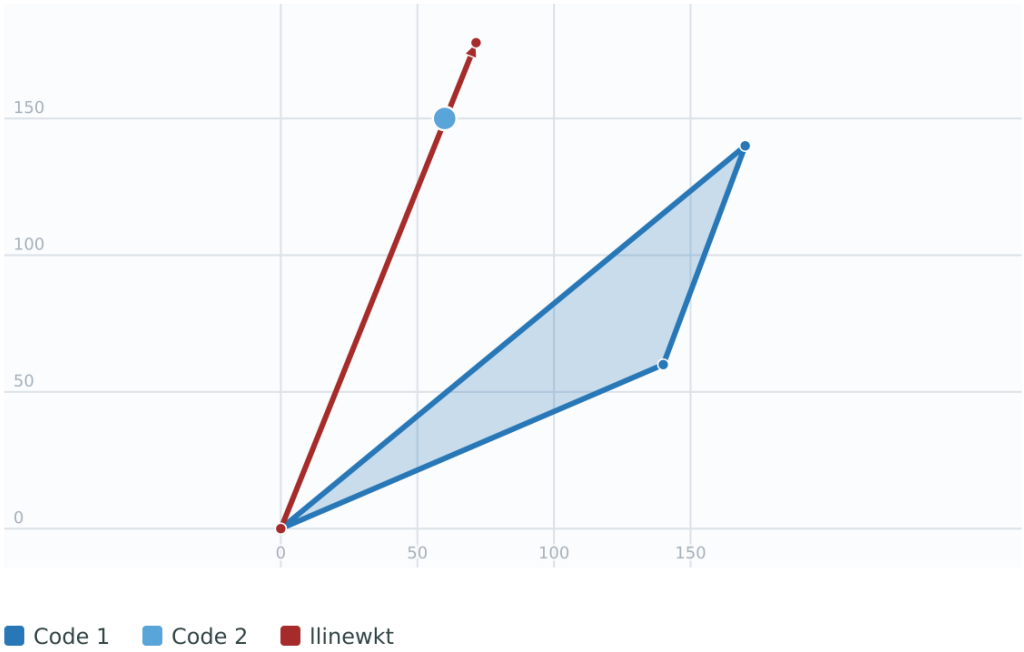
Code

```
SELECT ST_LongestLine(
        'POLYGON ((190 150,20 10,160 70,190 150))',
        ST_Buffer('POINT(80 160)', 30)
            ) AS llinewkt;
```

Output

```
LINESTRING(20 10,91.4805029709527 187.7163859753386)
```

Figure



Longest line across a single geometry. The length of the line is equal to the Maximum Distance. The endpoints of the line lie on the Minimum Bounding Circle.

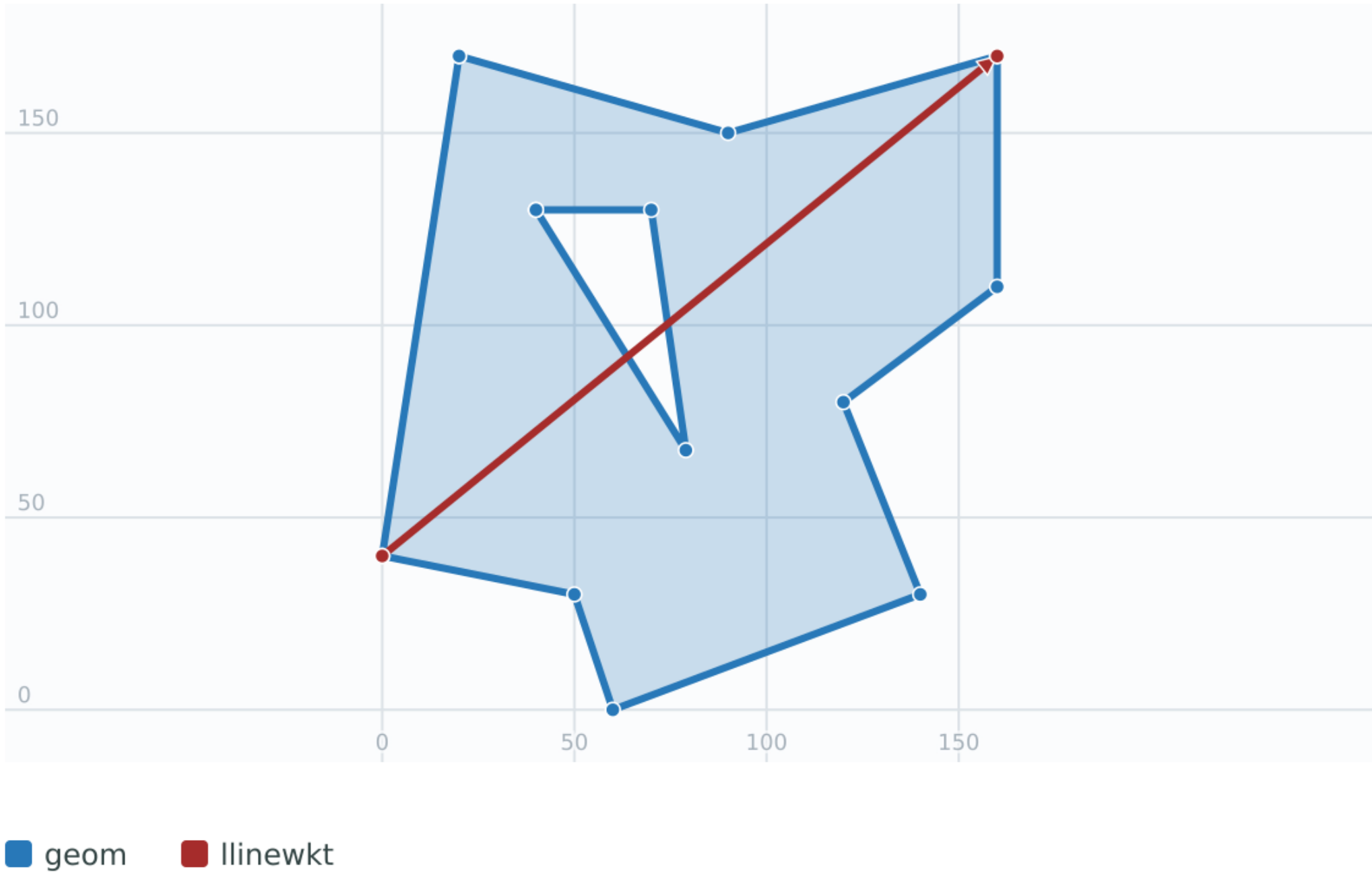
Code

```
SELECT ST_LongestLine(geom) AS llinewkt,
                  ST_MaxDistance(geom) AS max_dist,
                  ST_Length(ST_LongestLine(geom)) AS lenll
FROM (SELECT 'POLYGON ((40 180,110 160,180 180,180 120,140 90,160 40,80 10,70 40,20 50,40 ↩
    180),
              (60 140,99 77.5,90 140,60 140))'::geometry AS geom) AS t;
```

Output

```
llinewkt          |      max_dist      |       lenll
--------------------------+--------------------+--------------------
 LINESTRING(20 50,180 180) | 206.15528128088303 | 206.15528128088303
```

Figure



⌧⌧

ST_MaxDistance, ST_ShortestLine, ST_3DLongestLine, ST_MinimumBoundingCircle

7.12.17 ST_3DLongestLine

ST_3DLongestLine — ⌧⌧⌧⌧⌧⌧ 3 ⌧⌧⌧⌧ (longest) ⌧⌧⌧⌧⌧⌧⌧⌧.

⌧⌧

geometry **ST_3DLongestLine**(geometry g1, geometry g2);

⌧⌧

⌧⌧⌧⌧⌧⌧ 3 ⌧⌧⌧⌧ (longest) ⌧⌧⌧⌧⌧⌧⌧⌧. ⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧, ⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧. ⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧ g1 ⌧⌧⌧⌧⌧⌧⌧ g2 ⌧⌧⌧⌧⌧⌧⌧. ⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧ 3 ⌧⌧⌧⌧⌧⌧ ST_3DMaxDistance ⌧⌧⌧ g1 ⌧ g2 ⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧.

2.0.0 ⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧.

⌧⌧⌧⌧: 2.2.0 ⌧⌧⌧⌧ 2D ⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧, (⌧⌧⌧⌧⌧⌧⌧ Z ⌧ 0 ⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧) 2D ⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧. 2D ⌧ 3D ⌧⌧⌧, ⌧⌧⌧ Z ⌧⌧⌧⌧ Z ⌧ 0 ⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧.

This function supports 3d and will not drop the z-index.

This function supports Polyhedral surfaces.

⌧⌧

Longest line between a LineString and a Point, comparing the 3D and 2D results.

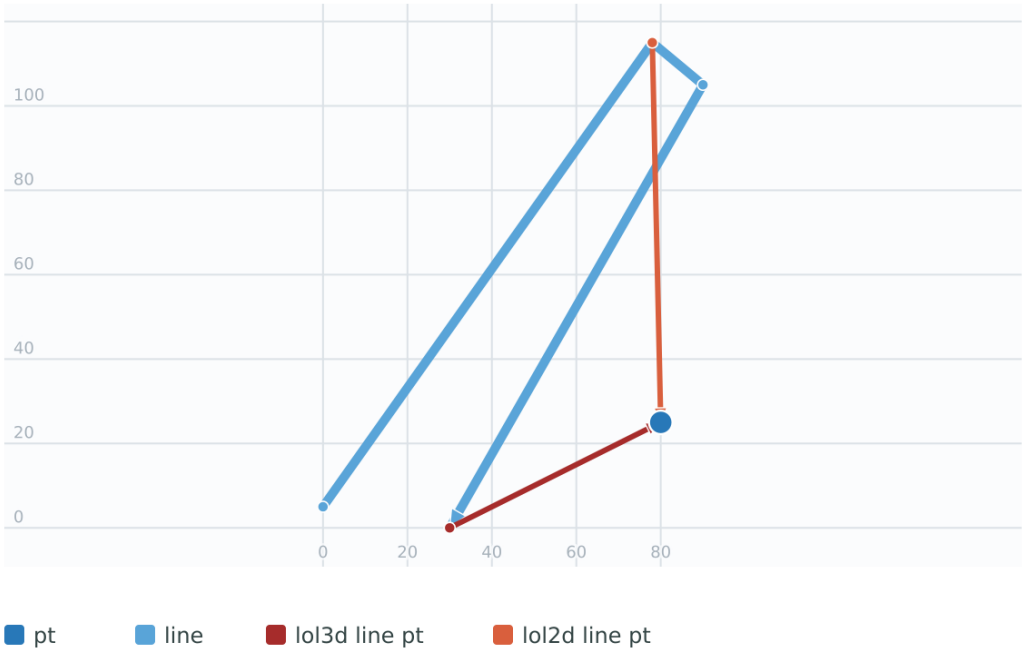
Code

```
SELECT ST_3DLongestLine(line, pt) AS lol3d_line_pt, ST_LongestLine(line, pt) AS  ↩
    lol2d_line_pt
FROM (
    SELECT 'POINT(100 100 30)'::geometry AS pt,
           'LINESTRING (20 80 20,98 190 1,110 180 3,50 75 1000)'::geometry AS line
) AS foo;
```

Output

```
lol3d_line_pt           |       lol2d_line_pt
--------------------------------+---------------------------
 LINESTRING(50 75 1000,100 100 30) | LINESTRING(98 190,100 100)
```

Figure



Longest line between a LineString and a MultiPoint, comparing the 3D and 2D results.

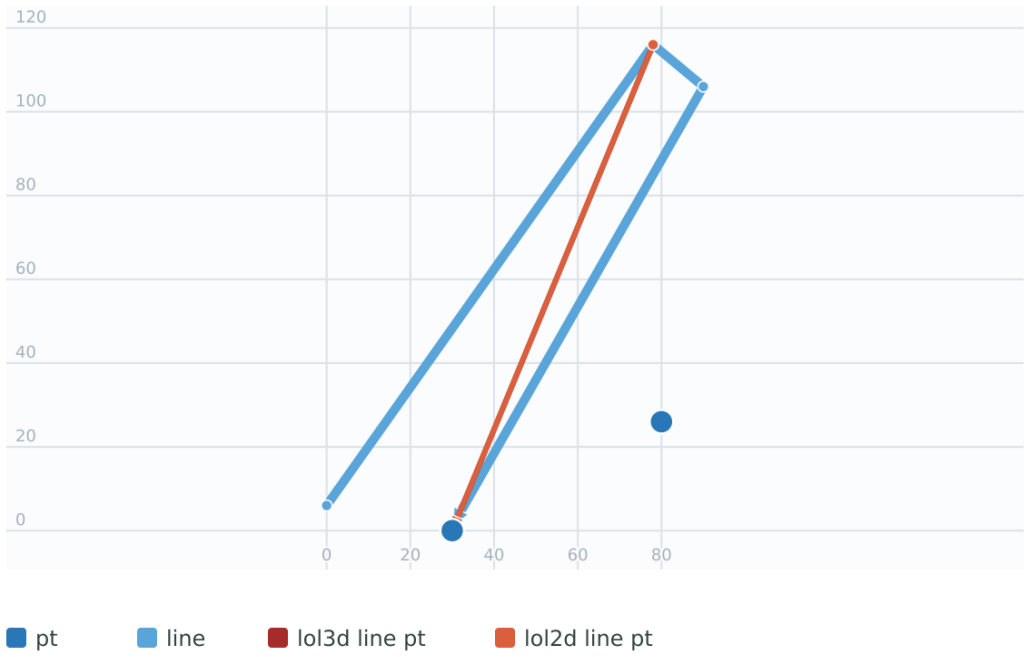
Code

```
SELECT ST_3DLongestLine(line, pt) AS lol3d_line_pt, ST_LongestLine(line, pt) AS  ↩
    lol2d_line_pt
FROM (
    SELECT 'MULTIPOINT(100 100 30,50 74 1000)'::geometry AS pt,
           'LINESTRING (20 80 20,98 190 1,110 180 3,50 75 900)'::geometry AS line
) AS foo;
```

Output

```
lol3d_line_pt          |      lol2d_line_pt
--------------------------------+-------------------------
 LINESTRING(98 190 1,50 74 1000) | LINESTRING(98 190,50 74)
```

Figure



Longest line between a Polygon and a MultiLineString, comparing the 3D and 2D results.

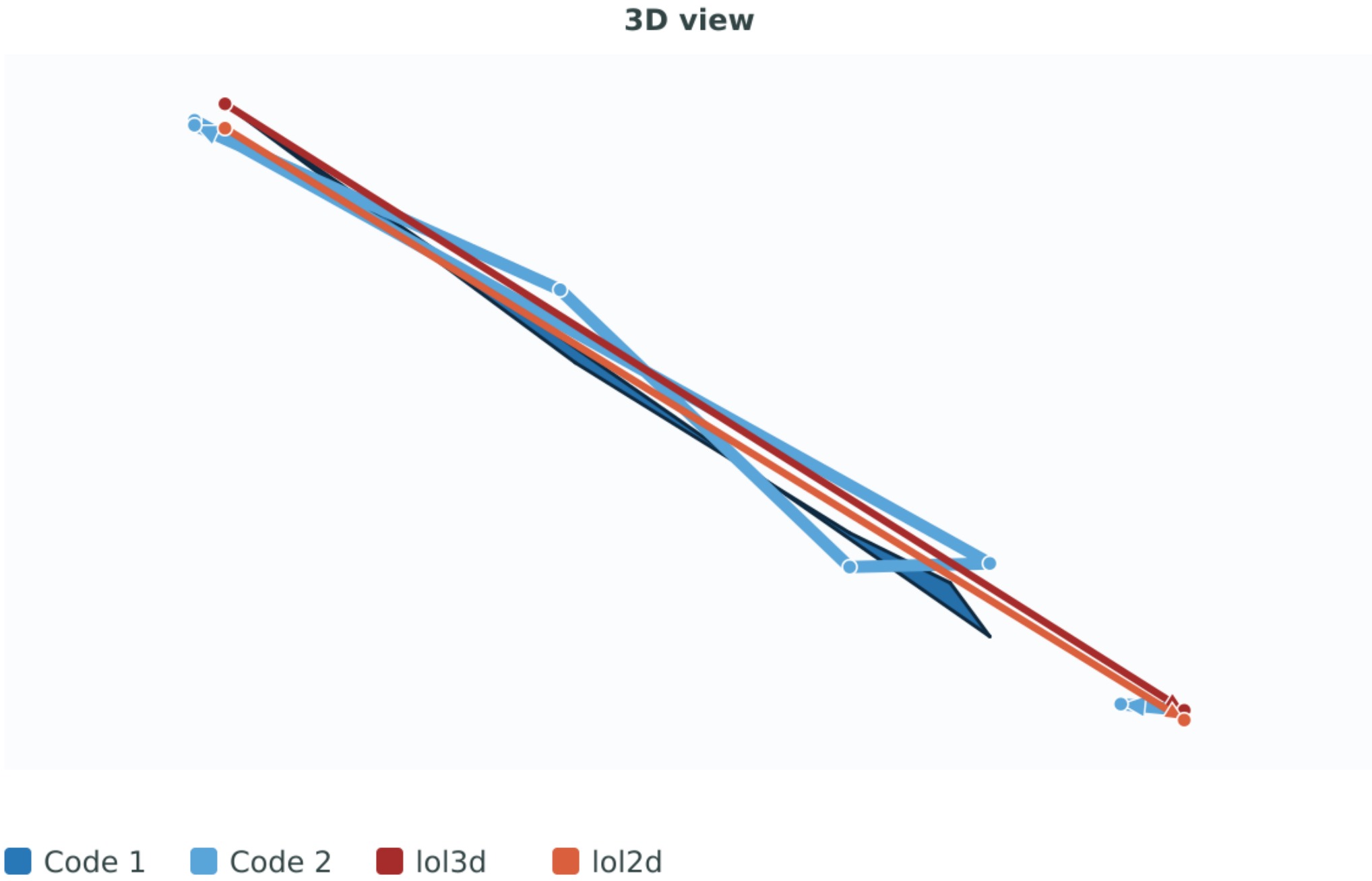
Code

```
SELECT ST_3DLongestLine(poly, mline) AS lol3d, ST_LongestLine(poly, mline) AS lol2d
FROM (
    SELECT ST_GeomFromEWKT(
                'POLYGON((175 150 5,20 40 5,35 45 5,50 60 5,100 100 5,175 150 5))'
            ) AS poly,
            ST_GeomFromEWKT(
                'MULTILINESTRING((175 155 2,20 40 20,50 60 -2,125 100 1,175 155 1),
                (1 10 2,5 20 1))'
            ) AS mline
) AS foo;
```

Output

```
lol3d             |          lol2d
------------------------------+-------------------------
 LINESTRING(175 150 5,1 10 2) | LINESTRING(175 150,1 10)
```

Figure



⊠⊠

ST_3DClosestPoint, ST_3DDistance, ST_LongestLine, ST_3DShortestLine, ST_3DMaxDistance

7.12.18 ST_MaxDistance

ST_MaxDistance — ⊠⊠⊠⊠⊠⊠ 2 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

float **ST_MaxDistance**(geometry g);
float **ST_MaxDistance**(geometry g1, geometry g2);

⊠⊠

⊠⊠⊠⊠⊠⊠ 2 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. g1 ⊠ g2 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

If only one geometry is provided, or g1 and g2 are the same geometry, returns the distance between the two vertices farthest apart in that geometry.

1.5.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

Enhanced: 3.7.0 - support for a single geometry input.

⊠⊠

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠

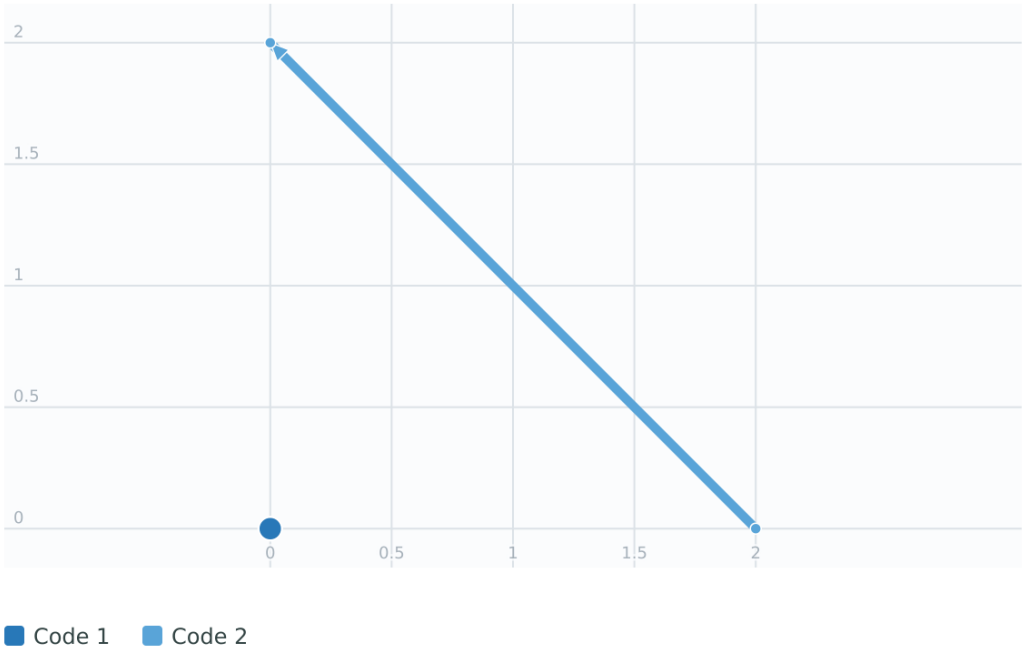
Code

```
SELECT ST_MaxDistance('POINT(0 0)'::geometry, 'LINESTRING ( 2 0,0 2 )'::geometry);
```

Output

```
2
```

Figure



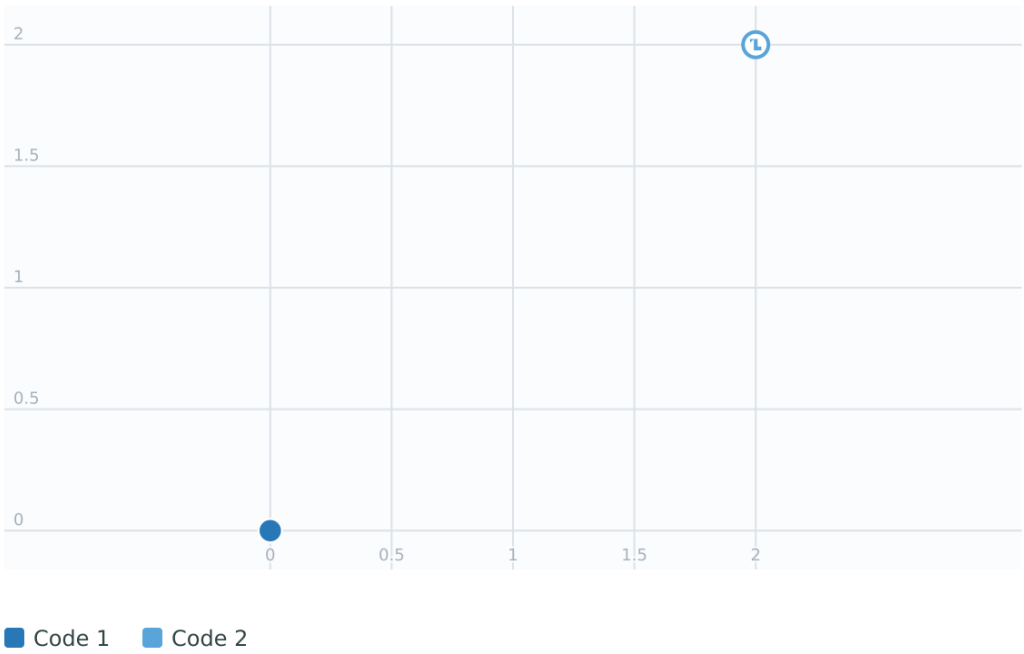
Code

```
SELECT ST_MaxDistance('POINT(0 0)'::geometry, 'LINESTRING ( 2 2,2 2 )'::geometry);
```

Output

```
2.82842712474619
```

Figure



Maximum distance between vertices of a single geometry.

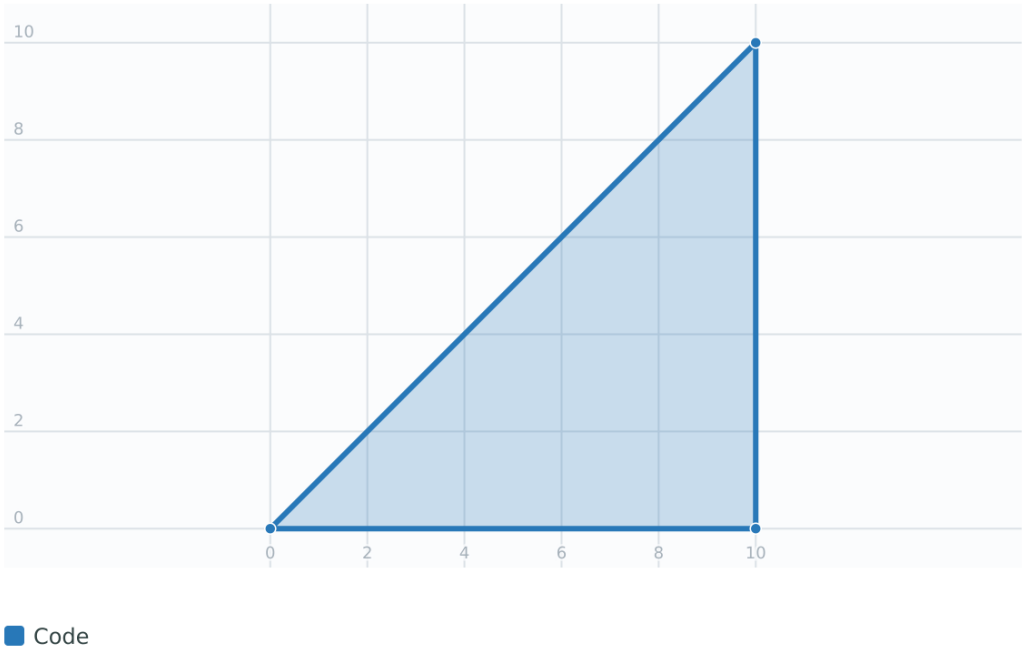
Code

```
SELECT ST_MaxDistance('POLYGON ((10 10,10 0,0 0,10 10))'::geometry);
```

Output

```
14.142135623730951
```

Figure



□□

ST_Distance, ST_LongestLine, ST_DFullyWithin

7.12.19 ST_3DMaxDistance

ST_3DMaxDistance — □□□□□□□□, □□□□□□□ (SRS □□□□□) 3 □□□□□□□□□□□□□□□□□□□□ □□□□□.

□□

float **ST_3DMaxDistance**(geometry g1, geometry g2);

□□

□□□□□□□□, □□□□□□□ 3 □□□□□□□□□□□□□□□□□□□□□ (SRS □□) □□□□□□□.

This function supports 3d and will not drop the z-index.

This function supports Polyhedral surfaces.

2.0.0 □□□□□□□□□□□□□.

□□□□: 2.2.0 □□□□□, 2D □ 3D □□□□□□□ Z □□□□□ Z □ 0 □□□□□□□□□□□.

□□

Geometry example - units in meters (SRID: 2163 US National Atlas Equal area) (3D point and line compared 2D point and line) Note that currently no vertical datum support so Z is not transformed and assumed to be same units as final.

Code

```
SELECT ST_3DMaxDistance(
            ST_Transform('SRID=4326;POINT(-72.1235 42.3521 10000)'::geometry, 2163),
                ST_Transform('SRID=4326;LINESTRING(-72.1260 42.45 15,-72.123 42.1546 20)':: ↩
    geometry, 2163)
            ) As dist_3d,
            ST_MaxDistance(
                ST_Transform('SRID=4326;POINT(-72.1235 42.3521 10000)'::geometry, 2163),
                ST_Transform('SRID=4326;LINESTRING(-72.1260 42.45 15,-72.123 42.1546 20)':: ↩
    geometry, 2163)
            ) As dist_2d;
```

Output

```
dist_3d      |     dist_2d
-----------------+-----------------
 24383.7467488441 | 22247.8472107251
```

□□

ST_Distance, ST_3DDWithin, ST_3DMaxDistance, ST_Transform

7.12.20 ST_MinimumClearance

ST_MinimumClearance — ⊠⊠⊠⊠⊠⊠ (robustness) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (clearance) ⊠⊠⊠⊠⊠⊠.

⊠⊠

float **ST_MinimumClearance**(geometry g);

⊠⊠

(⊠⊠⊠⊠⊠⊠⊠) ST_IsValid ⊠⊠ (⊠⊠⊠⊠⊠⊠) ST_IsSimple ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠ (WKT, KML, GML GeoJSON ⊠) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (MapInfo TAB ⊠) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠.

The minimum clearance is a quantitative measure of a geometry's robustness to change in coordinate precision. It is the largest distance by which vertices of the geometry can be moved without creating an invalid geometry. Larger values of minimum clearance indicate greater robustness.

⊠⊠⊠ e ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠:

- ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ 2 ⊠⊠ e ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ e ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠) ST_MinimumClearance ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

To avoid validity issues caused by precision loss, ST_ReducePrecision can reduce coordinate precision while ensuring that polygonal geometry remains valid.

2.3.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

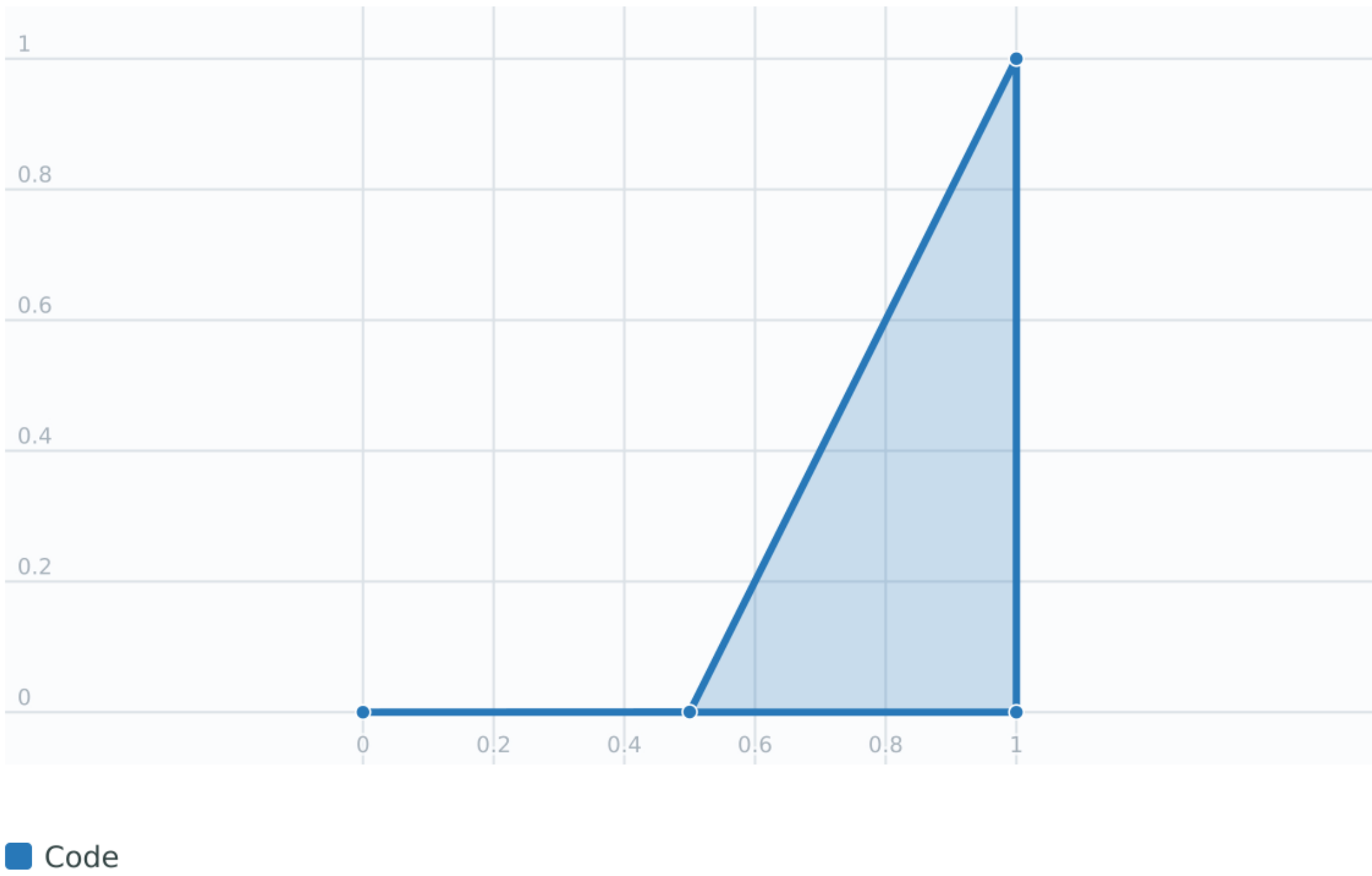
Code

```
SELECT ST_MinimumClearance('POLYGON ((0 0,1 0,1 1,0.5 3.2e-4,0 0))');
```

Output

```
0.00032
```

Figure



☒☒

ST_MinimumClearanceLine, ST_Crosses, ST_Dimension, ST_Intersects

7.12.21 ST_MinimumClearanceLine

ST_MinimumClearanceLine — ☒☒☒ 2 ☒☒☒☒☒☒, ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.

☒☒

Geometry **ST_MinimumClearanceLine**(geometry g);

☒☒

Returns the two-point LineString spanning a geometry's minimum clearance. If the geometry does not have a minimum clearance, `LINESTRING EMPTY` is returned.

GEOS ☒☒☒☒☒

2.3.0 ☒☒☒☒☒☒☒☒☒☒☒☒. GEOS 3.6.0 ☒☒☒☒☒☒☒☒☒☒.

☒☒

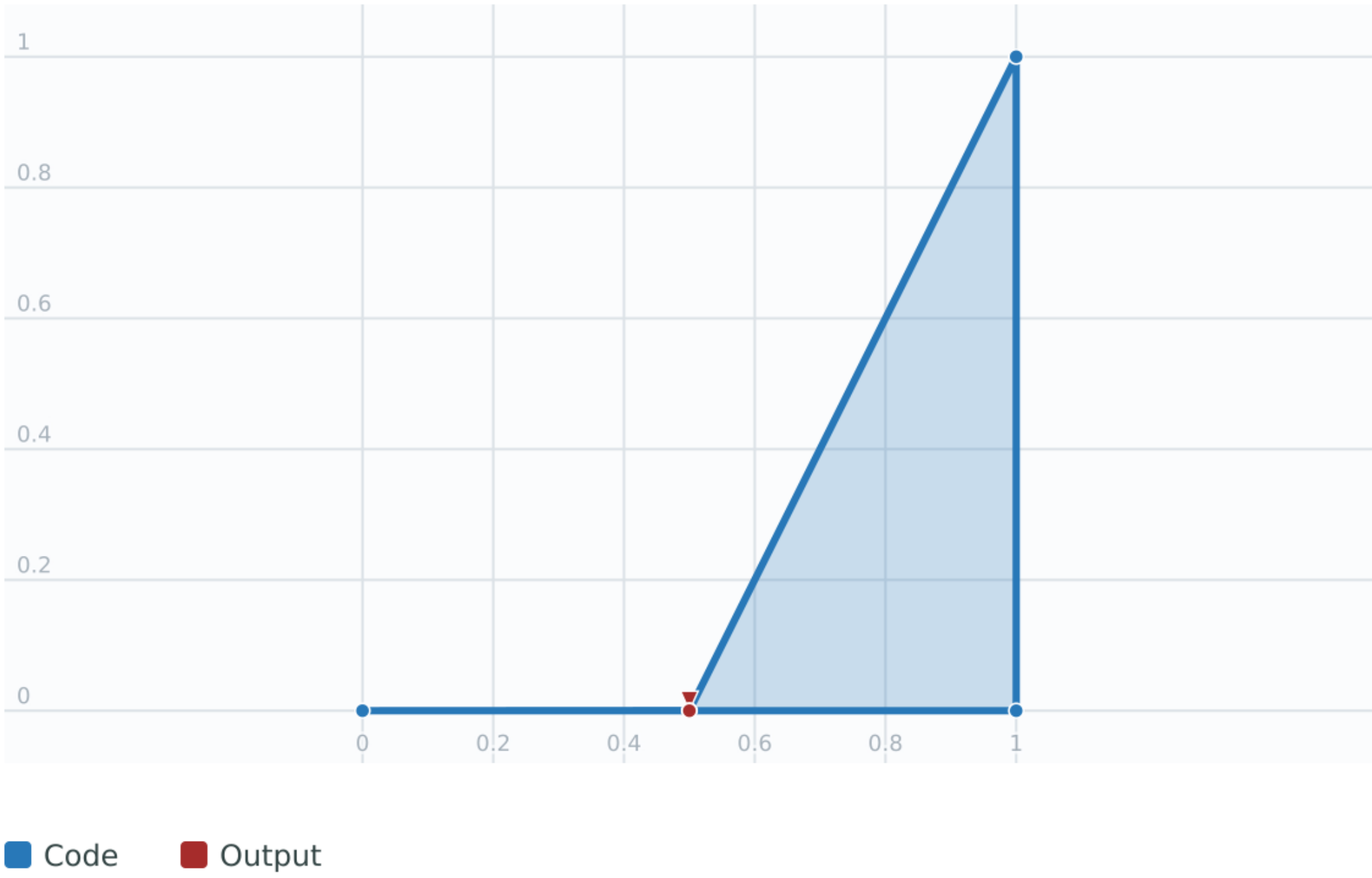
Code

```
SELECT ST_MinimumClearanceLine('POLYGON ((0 0,1 0,1 1,0.5 3.2e-4,0 0))');
```

Output

```
LINESTRING(0.5 0.00032,0.5 0)
```

Figure



⊠⊠

ST_MinimumClearance

7.12.22 ST_Perimeter

ST_Perimeter — Returns the length of the boundary of a polygonal geometry or geography.

⊠⊠

float **ST_Perimeter**(geometry g1);
float **ST_Perimeter**(geography geog, boolean use_spheroid = true);

⊠⊠

⊠⊠/⊠⊠⊠⊠ ST_Surface, ST_MultiSurface(⊠⊠⊠, ⊠⊠⊠⊠⊠) ⊠⊠⊠⊠⊠/⊠⊠⊠⊠ 2 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠ 0 ⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠ ST_Length ⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

For geography types, the calculations are performed using the inverse geodesic problem, where perimeter units are in meters. The spheroid is specified by the SRID; when no SRID is provided, WGS84 is used. If `use_spheroid = false`, then calculations will approximate a sphere instead of a spheroid.

⊠⊠⊠⊠⊠⊠ ST_Perimeter2D ⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

This method implements the OGC Simple Features Implementation Specification for SQL 1.1. s2.1.5.1

This method implements the SQL/MM specification. SQL-MM 3: 8.1.3, 9.5.4

⊠⊠⊠⊠: 2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

Geometry.

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠ EPSG:2249 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

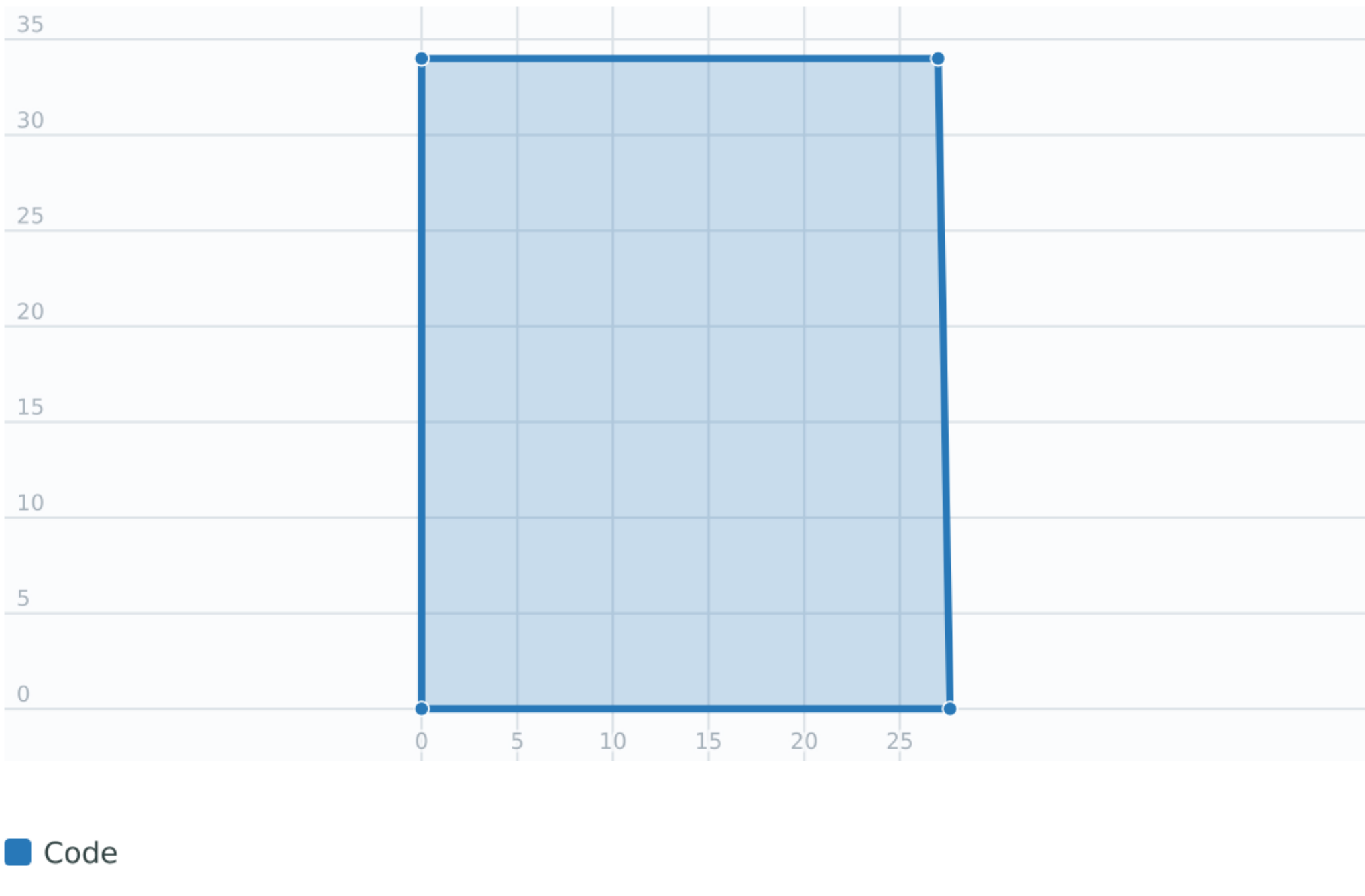
Code

```
SELECT ST_Perimeter(ST_GeomFromText(
'POLYGON((743238 2967416,743238 2967450,743265 2967450,
743265.625 2967416,743238 2967416))',
2249));
```

Output

```
122.63074400009504
```

Figure



Code

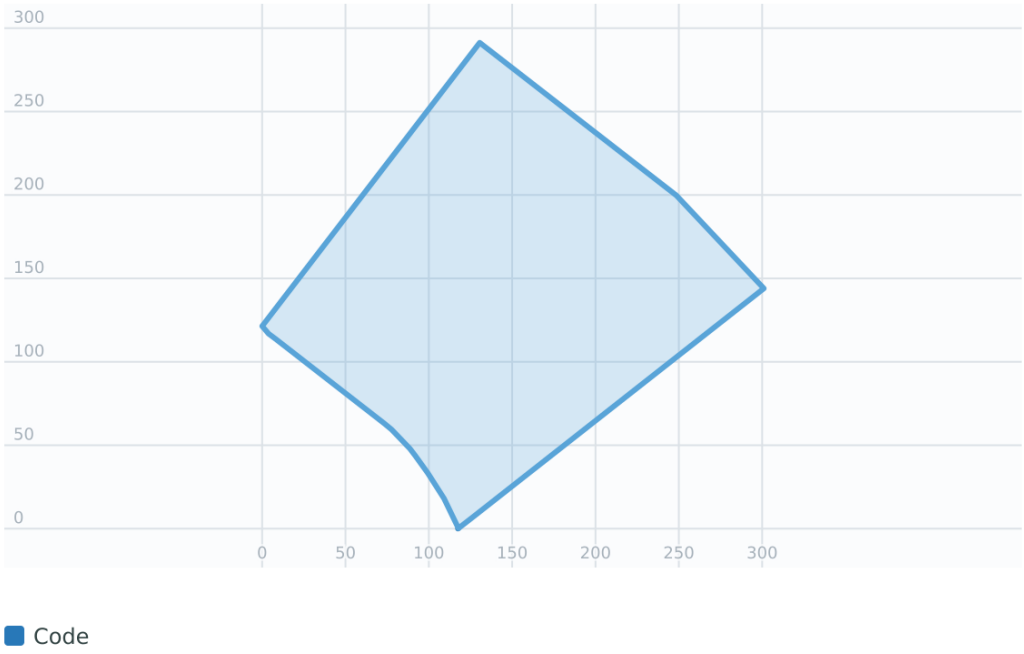
```
SELECT ST_Perimeter(ST_GeomFromText(
'MULTIPOLYGON(((763104.471273676 2949418.44119003,
763104.477769673 2949418.42538203,
763104.189609677 2949418.22343004,763104.471273676 2949418.44119003)),
((763104.471273676 2949418.44119003,763095.804579742 2949436.33850239,
763086.132105649 2949451.46730207,763078.452329651 2949462.11549407,
763075.354136904 2949466.17407812,763064.362142565 2949477.64291974,
```

```
763059.953961626 2949481.28983009,762994.637609571 2949532.04103014,
762990.568508415 2949535.06640477,762986.710889563 2949539.61421415,
763117.237897679 2949709.50493431,763235.236617789 2949617.95619822,
763287.718121842 2949562.20592617,763111.553321674 2949423.91664605,
763104.471273676 2949418.44119003)))',
2249));
```

Output

```
845.227713366825
```

Figure



□□□□□□□□□□□□□□□□□□□□□□□□□□□. □□□□□□□□□□□□ WGS84 □□□□□□□□□□□□□□□.

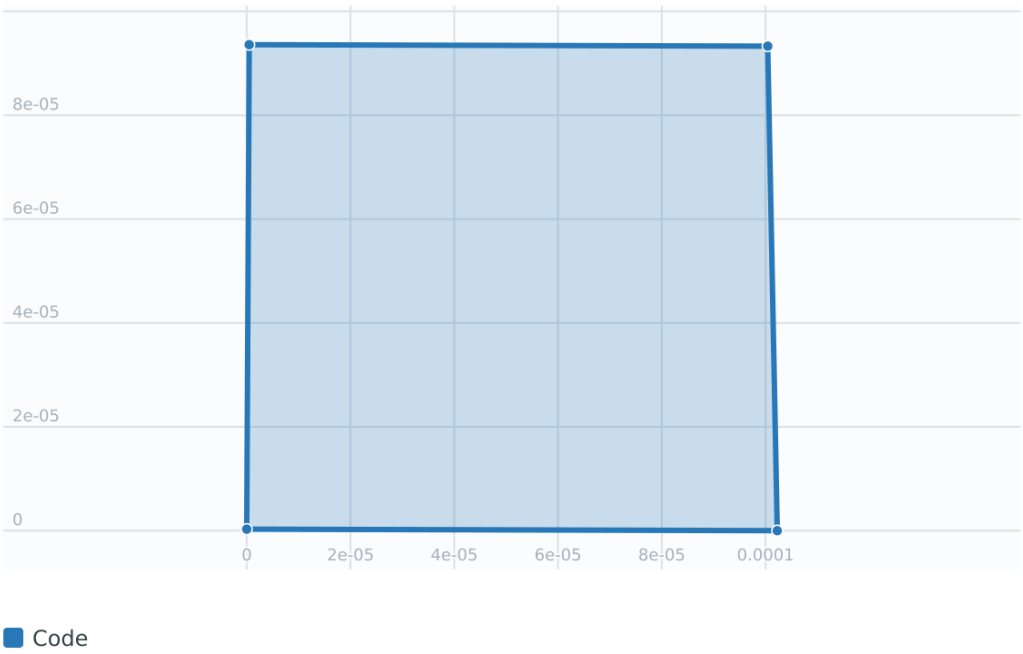
Code

```
SELECT round(ST_Perimeter(geog)::numeric, 8) As per_meters,
       round((ST_Perimeter(geog)/0.3048)::numeric, 8) As per_ft
FROM ST_GeogFromText('POLYGON((-71.1776848522251 42.3902896512902,-71.1776843766326 ↩
    42.3903829478009,
-71.1775844305465 42.3903826677917,-71.1775825927231 42.3902893647987,-71.1776848522251 ↩
    42.3902896512902))') As geog;
```

Output

```
per_meters  |    per_ft
------------+-------------
 37.37904625 | 122.63466619
```

Figure



This example uses a MultiPolygon.

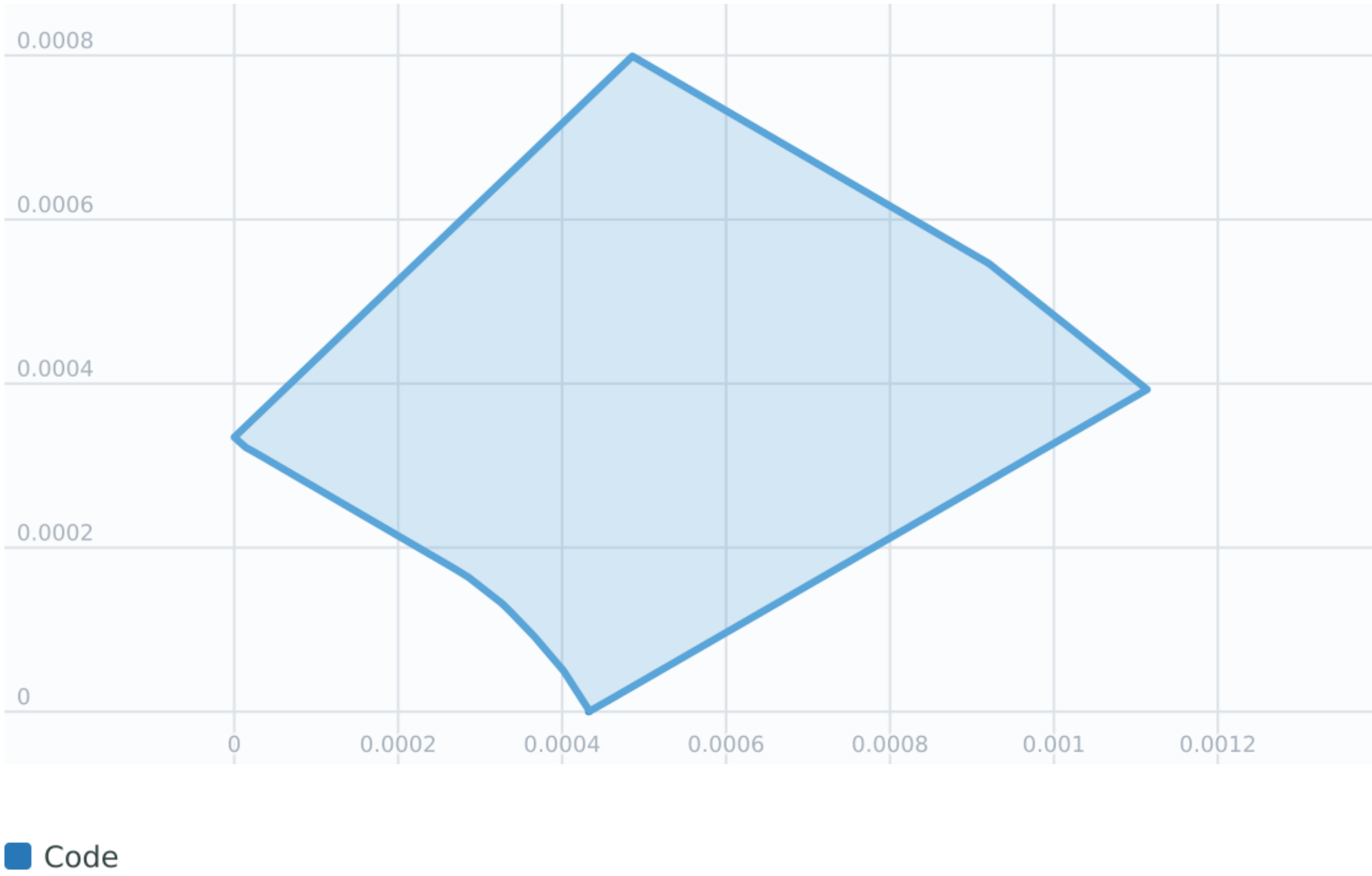
Code

```
SELECT round(ST_Perimeter(geog)::numeric, 8) As per_meters,
       round(ST_Perimeter(geog, false)::numeric, 8) As per_sphere_meters,
       round((ST_Perimeter(geog)/0.3048)::numeric, 8) As per_ft
FROM ST_GeogFromText('MULTIPOLYGON(((-71.1044543107478 42.340674480411,-71.1044542869917 ↩
    42.3406744369506,
-71.1044553562977 42.340673886454,-71.1044543107478 42.340674480411)),
((-71.1044543107478 42.340674480411,-71.1044860600303 42.3407237015564,-71.1045215770124 ↩
    42.3407653385914,
-71.1045498002983 42.3407946553165,-71.1045611902745 42.3408058316308,-71.1046016507427 ↩
    42.340837442371,
-71.104617893173 42.3408475056957,-71.1048586153981 42.3409875993595,-71.1048736143677 ↩
    42.3409959528211,
-71.1048878050242 42.3410084812078,-71.1044020965803 42.3414730072048,
-71.1039672113619 42.3412202916693,-71.1037740497748 42.3410666421308,
-71.1044280218456 42.3406894151355,-71.1044543107478 42.340674480411)))') As geog;
```

Output

```
per_meters  | per_sphere_meters |    per_ft
-------------+-------------------+-------------
 257.63428368 |      257.41231145 | 845.25683623
```

Figure



⊠⊠

ST_GeogFromText, ST_GeomFromText, ST_Length

7.12.23 ST_Perimeter2D

ST_Perimeter2D — Returns the 2D perimeter of a polygonal geometry. Alias for `ST_Perimeter`.

⊠⊠

float **ST_Perimeter2D**(geometry geomA);

⊠⊠

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ 2 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

> **⊠⊠**
> ⊠⊠⊠⊠⊠⊠ `ST_Perimeter` ⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠ `ST_Perimeter` ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

ST_Perimeter

7.12.24 ST_3DPerimeter

ST_3DPerimeter — □□□□□□□□□□□□□□□□.

□□

float **ST_3DPerimeter**(geometry geomA);

□□

□□□□□□□□□□□□□□□□□□□□ 3 □□□□□□□□□□. 2 □□□□□□□□ 2 □□□□□□□□□□□□□□.

This function supports 3d and will not drop the z-index.

This method implements the SQL/MM specification. SQL-MM ISO/IEC 13249-3: 8.1, 10.5

□□□□: 2.0.0 □□□□□□□□ ST_Perimeter3D □□□□□□□□□□.

□□

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

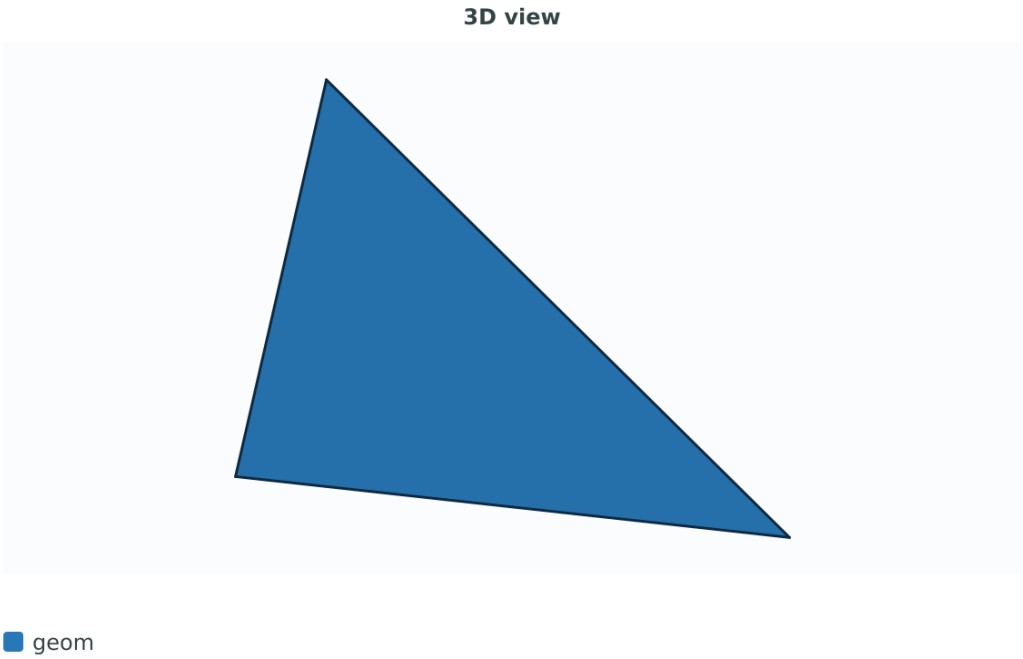
Code

```
SELECT ST_3DPerimeter(geom), ST_Perimeter2D(geom), ST_Perimeter(geom) FROM
            (SELECT 'SRID=2249;POLYGON((743238 2967416 2,743238 2967450 1,
743265.625 2967416 1,743238 2967416 2))'::geometry As geom) As foo;
```

Output

```
ST_3DPerimeter  |  st_perimeter2d  |   st_perimeter
------------------+------------------+------------------
105.46579359767418 | 105.43299727218763 | 105.43299727218763
```

Figure



□□

ST_GeomFromEWKT, ST_Perimeter, ST_Perimeter2D

7.12.25 ST_ShortestLine

ST_ShortestLine — □□□□□□ 2 □□□□□□□□□□□□.

□□

geometry **ST_ShortestLine**(geometry geom1, geometry geom2);
geography **ST_ShortestLine**(geography geom1, geography geom2, boolean use_spheroid = true);

□□

Returns the 2-dimensional shortest line between two geometries. The line returned starts in `geom1` and ends in `geom2`. If `geom1` and `geom2` intersect the result is a line with start and end at an intersection point. The length of the line is the same as ST_Distance returns for g1 and g2.

Enhanced: 3.4.0 - support for geography.

1.5.0 □□□□□□□□□□□□.

□□

Shortest line between a Point and a LineString.

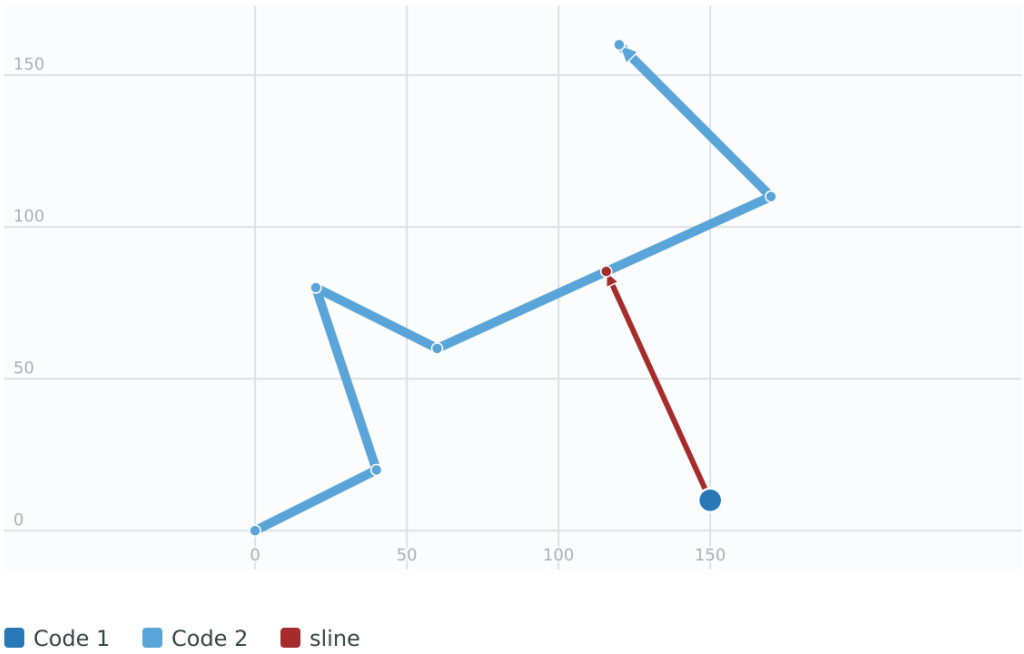
Code

```
SELECT ST_ShortestLine(
        'POINT (160 40)',
        'LINESTRING (10 30,50 50,30 110,70 90,180 140,130 190)') As sline;
```

Output

```
LINESTRING(160 40,125.75342465753425 115.34246575342466)
```

Figure



Shortest line between polygons.

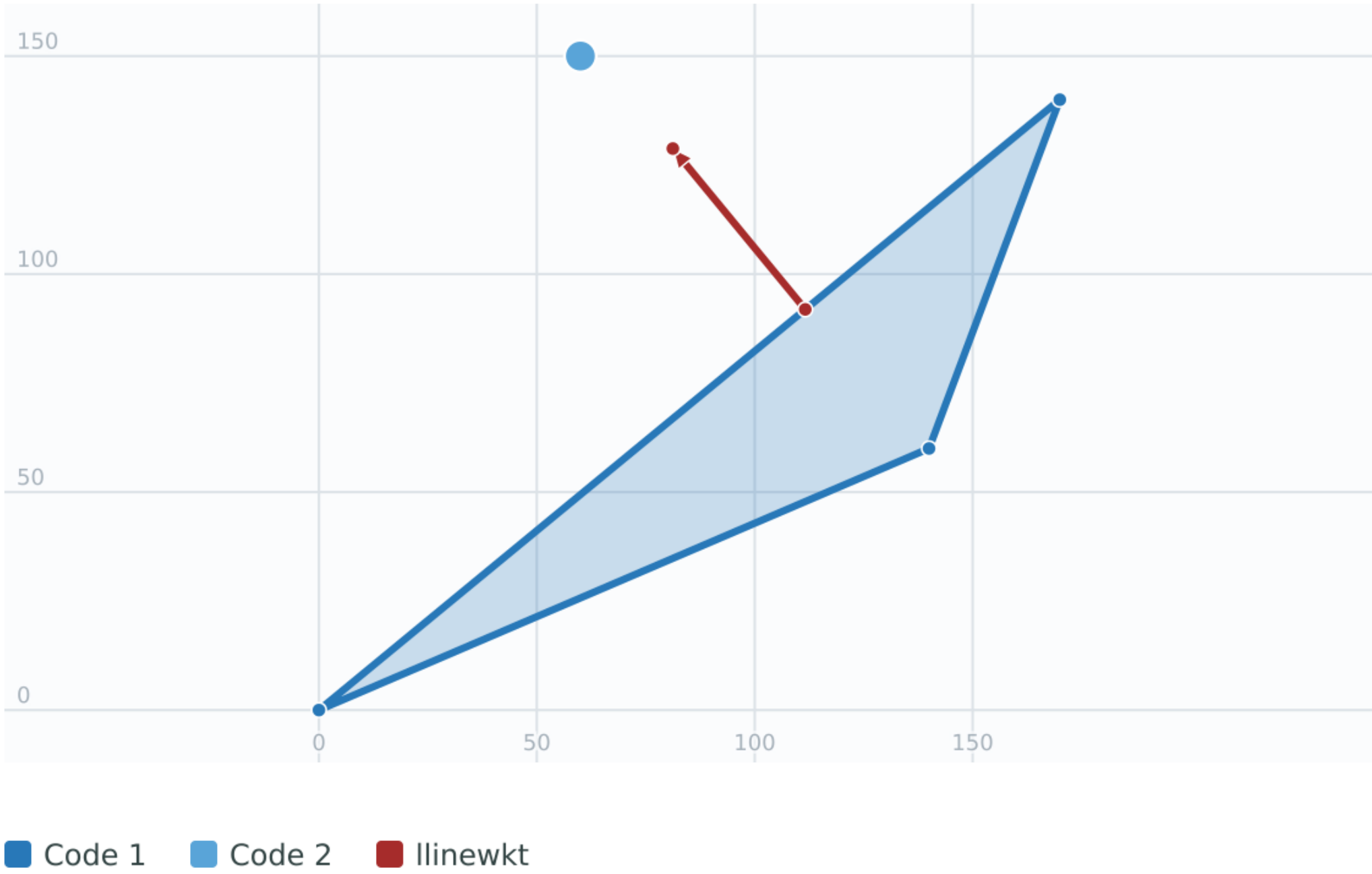
Code

```
SELECT ST_ShortestLine(
         'POLYGON ((190 150,20 10,160 70,190 150))',
         ST_Buffer('POINT(80 160)', 30)
                 ) AS llinewkt;
```

Output

```
LINESTRING(131.59149149528952 101.89887534906195,101.21320343559643 138.78679656440357)
```

Figure



⊠⊠

ST_ClosestPoint, ST_Distance, ST_LongestLine, ST_MaxDistance

7.12.26 ST_3DShortestLine

ST_3DShortestLine — ⊠⊠⊠⊠⊠⊠ 3 ⊠⊠⊠⊠ (shortest) ⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

geometry **ST_3DShortestLine**(geometry g1, geometry g2);

⊠⊠

⊠⊠⊠⊠⊠⊠ 3 ⊠⊠⊠⊠ (shortest) ⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠. g1 ⊠ g2 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠. g1 ⊠ g2 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠ g1 ⊠⊠⊠⊠⊠⊠ g2 ⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ 3 ⊠⊠⊠⊠⊠ ST_3DDistance ⊠⊠⊠ g1 ⊠ g2 ⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠.

2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠⊠⊠: 2.2.0 ⊠⊠⊠⊠ 2D ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, (⊠⊠⊠⊠⊠⊠ Z ⊠ 0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠) 2D ⊠⊠ ⊠⊠⊠⊠⊠⊠⊠. 2D ⊠ 3D ⊠⊠⊠, ⊠⊠⊠ Z ⊠⊠⊠⊠ Z ⊠ 0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

This function supports 3d and will not drop the z-index.

This function supports Polyhedral surfaces.

□□

Compare the 3D shortest line to the 2D shortest line for a LineString and a Point.

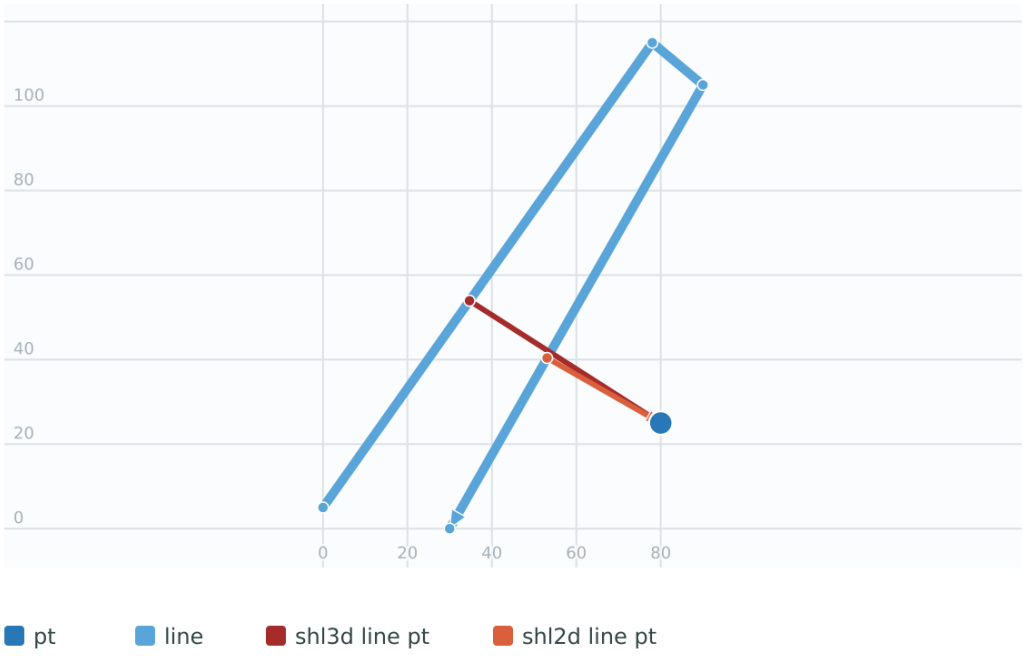
Code

```
WITH input AS (
  SELECT 'POINT(100 100 30)'::geometry AS pt,
         'LINESTRING (20 80 20,98 190 1,110 180 3,50 75 1000)'::geometry AS line
)
SELECT ST_3DShortestLine(line, pt) AS shl3d_line_pt,
       ST_ShortestLine(line, pt) AS shl2d_line_pt
FROM input;
```

Output

```
-[ RECORD 1 ]--+-------------------------------------------------------------------------------
shl3d_line_pt  | LINESTRING(54.69937988676193 128.93502291722837 11.547586950660556,100 100 30)
shl2d_line_pt  | LINESTRING(73.07692307692307 115.38461538461539,100 100)
```

Figure



Compare the 3D shortest line to the 2D shortest line for a LineString and a MultiPoint.

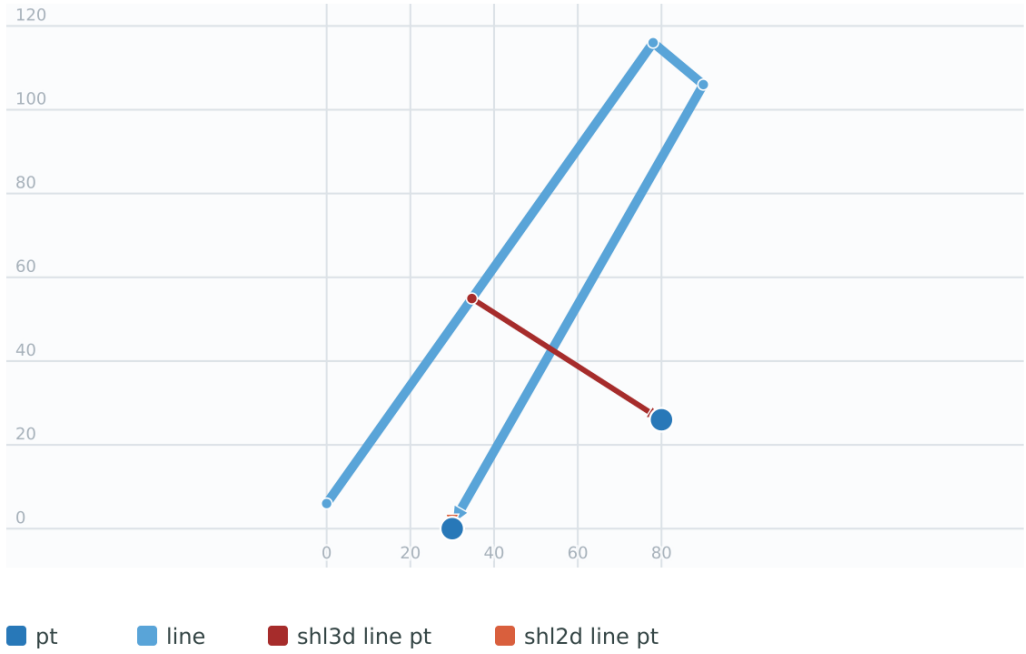
Code

```
WITH input AS (
  SELECT 'MULTIPOINT(100 100 30,50 74 1000)'::geometry AS pt,
         'LINESTRING (20 80 20,98 190 1,110 180 3,50 75 900)'::geometry AS line
)
SELECT ST_3DShortestLine(line, pt) AS shl3d_line_pt,
       ST_ShortestLine(line, pt) AS shl2d_line_pt
FROM input;
```

Output

```
-[ RECORD 1 ]--+-------------------------------------------------------------------------------
shl3d_line_pt  | LINESTRING(54.69937988676193 128.93502291722837 11.547586950660556,100 100 30)
shl2d_line_pt  | LINESTRING(50 75,50 74)
```

Figure



Compare the 3D shortest line to the 2D shortest line for a Polygon and a MultiLineString.

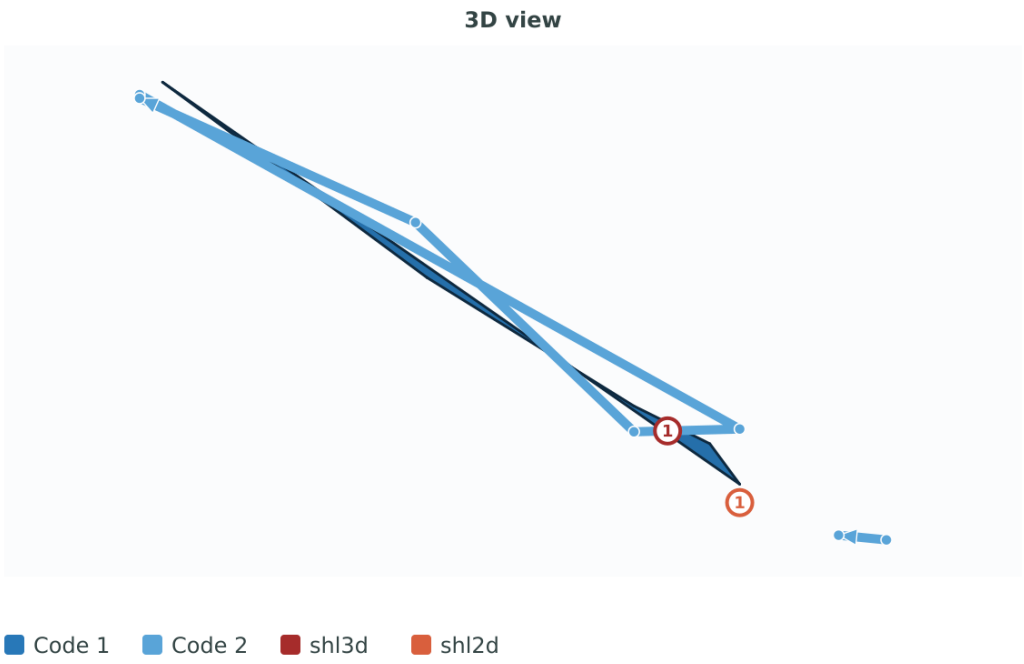
Code

```
WITH input AS (
  SELECT ST_GeomFromEWKT(
           'POLYGON((175 150 5,20 40 5,35 45 5,50 60 5,100 100 5,175 150 5))'
         ) AS poly,
         ST_GeomFromEWKT(
           'MULTILINESTRING((175 155 2,20 40 20,50 60 -2,125 100 1,175 155 1),
                            (1 10 2,5 20 1))'
         ) AS mline
)
SELECT ST_3DShortestLine(poly, mline) AS shl3d,
       ST_ShortestLine(poly, mline) AS shl2d
FROM input;
```

Output

```
-[ RECORD 1 ]--+-----------------------------------------------------------------------------------
shl3d          | LINESTRING(40.45454545454545 53.63636363636364 5,40.45454545454545 53.63636363636364 5)
shl2d          | LINESTRING(20 40,20 40)
```

Figure



□□

ST_3DClosestPoint, ST_3DDistance, ST_LongestLine, ST_ShortestLine, ST_3DMaxDistance

7.13 Overlay Functions

7.13.1 ST_ClipByBox2D

ST_ClipByBox2D — Computes the portion of a geometry falling within a rectangle.

□□

geometry **ST_ClipByBox2D**(geometry geom, box2d box);

□□

Clips a geometry by a 2D box in a fast and tolerant but possibly invalid way. Topologically invalid input geometries do not result in exceptions being thrown. The output geometry is not guaranteed to be valid (in particular, self-intersections for a polygon may be introduced).

GEOS □□□□□

2.2.0 □□□□□□□□□□□□.

□□

This example relies on the implicit cast from geometry to box2d for the second parameter.

Code

```
SELECT ST_ClipByBox2D(geom, ST_MakeEnvelope(0, 0, 10, 10)) FROM mytab;
```

□□

ST_Intersection, ST_MakeBox2D, ST_MakeEnvelope

7.13.2 ST_Difference

ST_Difference — Computes a geometry representing the part of geometry A that does not intersect geometry B.

□□

geometry **ST_Difference**(geometry geomA, geometry geomB, float8 gridSize = -1);

□□

Returns a geometry representing the part of geometry A that does not intersect geometry B. This is equivalent to `A - ST_Intersection(A,B)`. If A is completely contained in B then an empty atomic geometry of appropriate type is returned.



□□
This is the only overlay function where input order matters. ST_Difference(A, B) always returns a portion of A.

If the optional `gridSize` parameter is given (GEOS-3.9.0 or higher required), all result vertices are guaranteed to fall on a snap-rounded grid of the specified size. Note that operations performed on a grid may contain small artifacts produced during grid alignment, see ST_ReducePrecision.

GEOS □□□□□

Enhanced: 3.1.0 accept a gridSize parameter.

Requires GEOS >= 3.9.0 to use the gridSize parameter.

 This method implements the OGC Simple Features Implementation Specification for SQL 1.1. s2.1.1.3

 This method implements the SQL/MM specification. SQL-MM 3: 5.1.20

 This function supports 3d and will not drop the z-index. However, the result is computed using XY only. The result Z values are copied, averaged or interpolated.

□□

The difference of 2D linestrings.

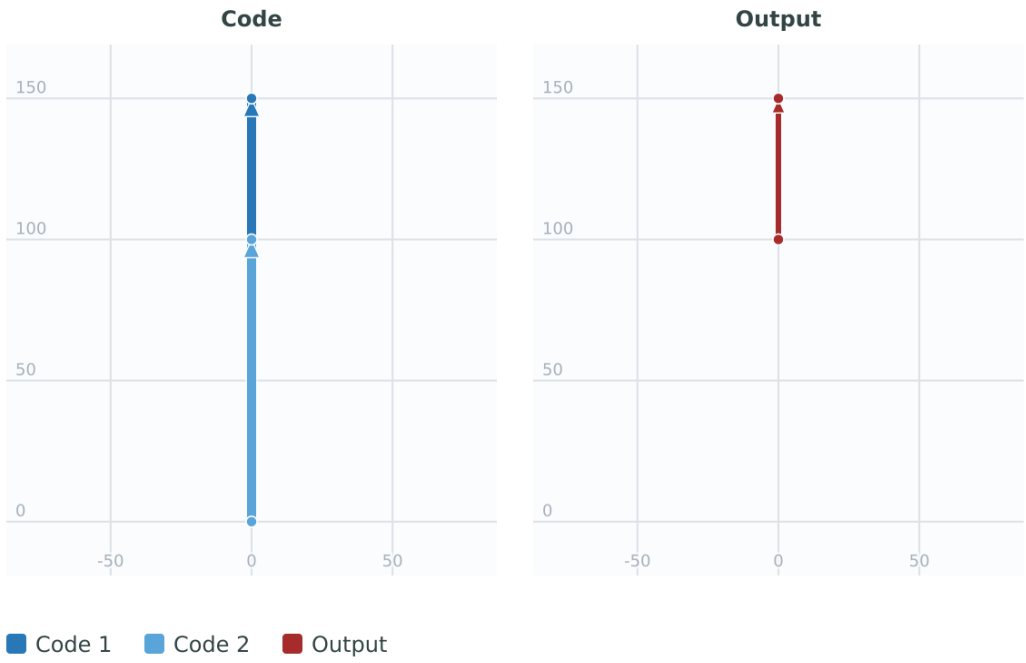
Code

```
SELECT ST_Difference('LINESTRING(50 100,50 200)'::geometry,
        'LINESTRING(50 50,50 150)'::geometry
    );
```

Output

```
LINESTRING(50 150,50 200)
```

Figure



The difference of 3D points.

Code

```
SELECT ST_Difference('MULTIPOINT(-118.58 38.38 5,-118.60 38.329 6,-118.614 38.281 7)' :: ↩
    geometry,
 'POINT(-118.614 38.281 5)' :: geometry
);
```

Output

```
MULTIPOINT(-118.6 38.329 6,-118.58 38.38 5)
```

Figure



□□

ST_SymDifference, ST_Intersection, ST_Union, ST_ReducePrecision

7.13.3 ST_Intersection

ST_Intersection — Computes a geometry representing the shared portion of geometries A and B.

□□

geometry **ST_Intersection**(geometry geomA , geometry geomB , float8 gridSize = -1);
geography **ST_Intersection**(geography geogA , geography geogB);

□□

Returns a geometry representing the point-set intersection of two geometries. In other words, that portion of geometry A and geometry B that is shared between the two geometries.

If the geometries have no points in common (i.e. are disjoint) then an empty atomic geometry of appropriate type is returned.

If the optional `gridSize` parameter is given (GEOS-3.9.0 or higher required), all result vertices are guaranteed to fall on a snap-rounded grid of the specified size. Note that operations performed on a grid may contain small artifacts produced during grid alignment, see ST_ReducePrecision.

ST_Intersection in conjunction with ST_Intersects is useful for clipping geometries such as in bounding box, buffer, or region queries where you only require the portion of a geometry that is inside a country or region of interest.



□□
Spatial predicates such as ST_Intersects and overlay functions such as ST_Intersection are computed by different algorithms. For very close or nearly coincident coordinates, floating-point precision and geometric robustness may make a predicate return true while the computed intersection is empty, or place an intersection point slightly away from the input line. For repeatable overlay results, snap inputs to a suitable grid first, for example with ST_ReducePrecision, or use the gridSize parameter when all input vertices already lie on the grid.



□□

For geography this is a thin wrapper around the geometry implementation. It first determines the best SRID that fits the bounding box of the 2 geography objects (if geography objects are within one half zone UTM but not same UTM will pick one of those) (favoring UTM or Lambert Azimuthal Equal Area (LAEA) north/south pole, and falling back on mercator in worst case scenario) and then intersection in that best fit planar spatial ref and retransforms back to WGS84 geography.



□□
This function will drop the M coordinate values if present.



□□
If working with 3D geometries, you may want to use SFCGAL based CG_3DIntersection which does a proper 3D intersection for 3D geometries. Although this function works with Z-coordinate, it does an averaging of Z-Coordinate.

GEOS □□□□□

Enhanced: 3.1.0 accept a gridSize parameter

Requires GEOS >= 3.9.0 to use the gridSize parameter

Changed: 3.0.0 does not depend on SFCGAL.

Availability: 1.5 support for geography data type was introduced.


This method implements the OGC Simple Features Implementation Specification for SQL 1.1. s2.1.1.3


This method implements the SQL/MM specification. SQL-MM 3: 5.1.18


This function supports 3d and will not drop the z-index. However, the result is computed using XY only. The result Z values are copied, averaged or interpolated.

□□

Code

```
SELECT ST_Intersection('POINT(0 0)'::geometry, 'LINESTRING ( 2 0,0 2 )'::geometry);
```

Output

```
POINT EMPTY
```

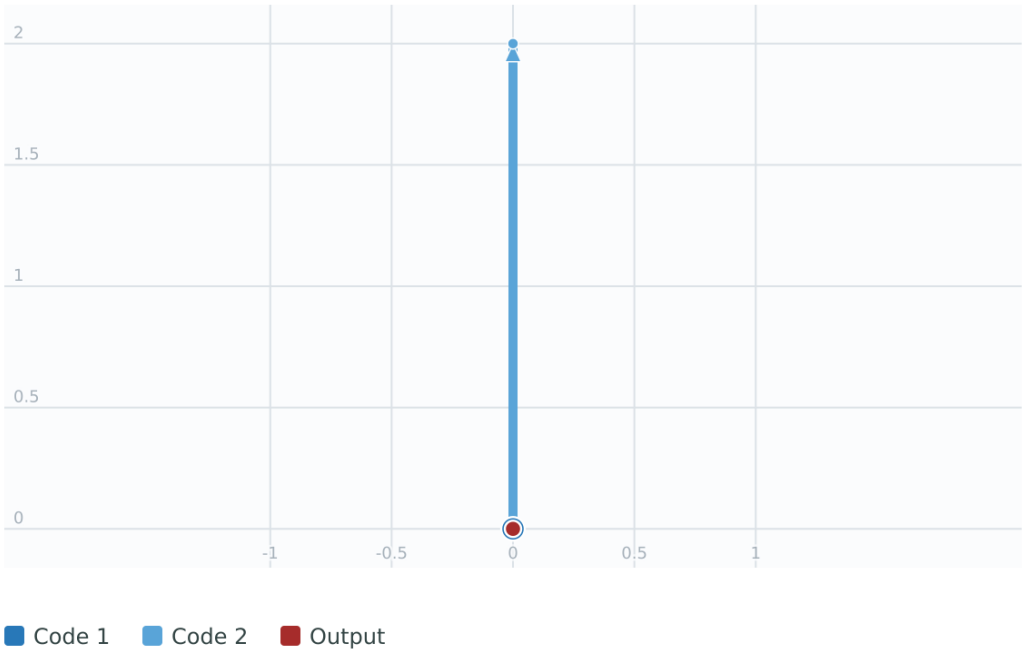
Code

```
SELECT ST_Intersection('POINT(0 0)'::geometry, 'LINESTRING ( 0 0,0 2 )'::geometry);
```

Output

```
POINT(0 0)
```

Figure



Clip all lines (trails) by country. Here we assume country geom are POLYGON or MULTIPOLYGONS. NOTE: we are only keeping intersections that result in a LINESTRING or MULTILINESTRING because we don't care about trails that just share a point. The dump is needed to expand a geometry collection into individual single MULT* parts. The below is fairly generic and will work for polys, etc. by just changing the where clause.

Code

```
select clipped.gid, clipped.f_name, clipped_geom
from (
         select trails.gid, trails.f_name,
             (ST_Dump(ST_Intersection(country.geom, trails.geom))).geom clipped_geom
         from country
              inner join trails on ST_Intersects(country.geom, trails.geom)
     ) as clipped
where ST_Dimension(clipped.clipped_geom) = 1;
```

For polys e.g. polygon landmarks, you can also use the sometimes faster hack that buffering anything by 0.0 except a polygon results in an empty geometry collection. (So a geometry collection containing polys, lines and points buffered by 0.0 would only leave the polygons and dissolve the collection shell.)

Code

```
select poly.gid,
    ST_Multi(ST_Buffer(ST_Intersection(country.geom, poly.geom),
            0.0
        )
    ) clipped_geom
from country
     inner join poly on ST_Intersects(country.geom, poly.geom)
where not ST_IsEmpty(ST_Buffer(ST_Intersection(country.geom, poly.geom), 0.0));
```

Note this is not a true 3D intersection. It uses 2.5D geometries (geometries with Z ordinates, but where calculations are performed in 2D). The same input is compared with CG_3DIntersection, which performs a true 3D intersection.

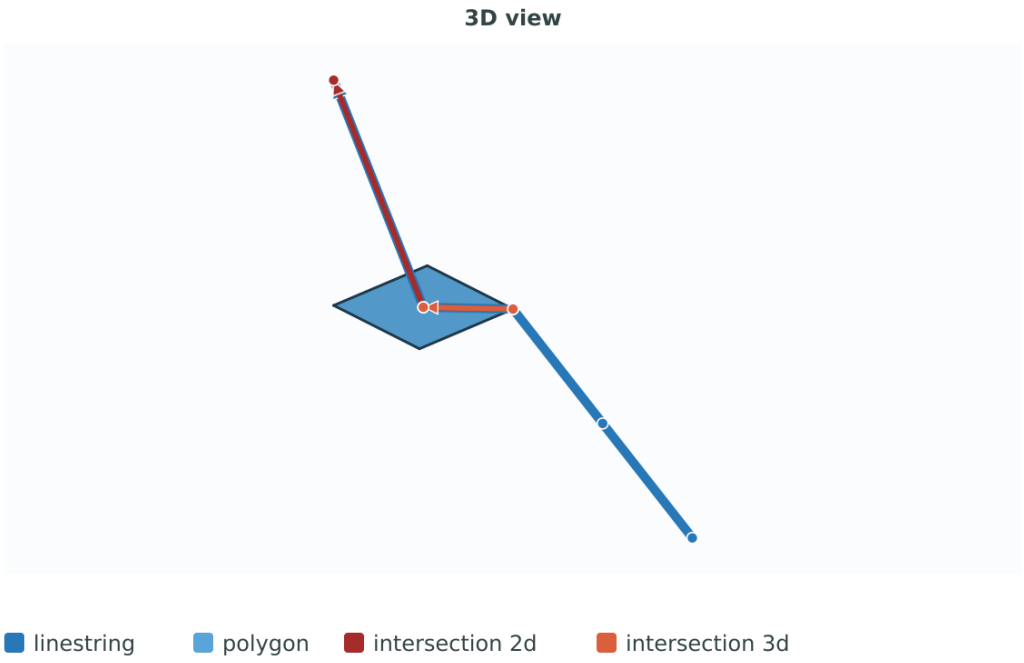
Code

```
WITH data AS (
  SELECT 'LINESTRING Z (2 2 6,1.5 1.5 7,1 1 8,0.5 0.5 8,0 0 10)'::geometry AS  ↩
    input_linestring,
         'POLYGON Z ((0 0 8,0 1 8,1 1 8,1 0 8,0 0 8))'::geometry AS input_polygon
)
SELECT input_linestring AS input_linestring,
       input_polygon AS input_polygon,
       ST_Intersection(input_linestring, input_polygon) AS intersection_2d,
       CG_3DIntersection(input_linestring, input_polygon) AS intersection_3d
FROM data;
```

Output

```
LINESTRING Z (2 2 6,1.5 1.5 7,1 1 8,0.5 0.5 8,0 0 10) | POLYGON Z ((0 0 8,0 1 8,1 1 8,1 0 8,0 0 8)) | LINESTRING Z (1  ↩
    1 8,0.5 0.5 8,0 0 10) | LINESTRING Z (1 1 8,0.5 0.5 8)
```

Figure



□□

CG_3DIntersection, ST_Difference, ST_Union, ST_ClipByBox2D, ST_Dimension, ST_Dump, ST_Force2D, ST_SymDifference, ST_Intersects, ST_Multi, ST_ReducePrecision

7.13.4 ST_MemUnion

ST_MemUnion — Aggregate function which unions geometries in a memory-efficent but slower way

□□

geometry **ST_MemUnion**(geometry set geomfield);

□□

An aggregate function that unions the input geometries, merging them to produce a result geometry with no overlaps. The output may be a single geometry, a MultiGeometry, or a Geometry Collection.



□□
Produces the same result as ST_Union, but uses less memory and more processor time. This aggregate function works by unioning the geometries incrementally, as opposed to the ST_Union aggregate which first accumulates an array and then unions the contents using a fast algorithm.



This function supports 3d and will not drop the z-index. However, the result is computed using XY only. The result Z values are copied, averaged or interpolated.

□□

Code

```
SELECT id,
       ST_MemUnion(geom) as singlegeom
FROM sometable f
GROUP BY id;
```

□□

ST_Union

7.13.5 ST_Node

ST_Node — Nodes a collection of lines.

□□

geometry **ST_Node**(geometry geom);

□□

Returns a (Multi)LineString representing the fully noded version of a collection of linestrings. The noding preserves all of the input nodes, and introduces the least possible number of new nodes. The resulting linework is dissolved (duplicate lines are removed).

This is a good way to create fully-noded linework suitable for use as input to ST_Polygonize.

ST_UnaryUnion can also be used to node and dissolve linework. It provides an option to specify a gridSize, which can provide simpler and more robust output. See also ST_Union for an aggregate variant.

This function supports 3d and will not drop the z-index.

GEOS □□□□□

2.0.0 □□□□□□□□□□□□.

Changed: 2.4.0 this function uses GEOSNode internally instead of GEOSUnaryUnion. This may cause the resulting linestrings to have a different order and direction compared to PostGIS < 2.4.

□□

Noding a 3D LineString which self-intersects.

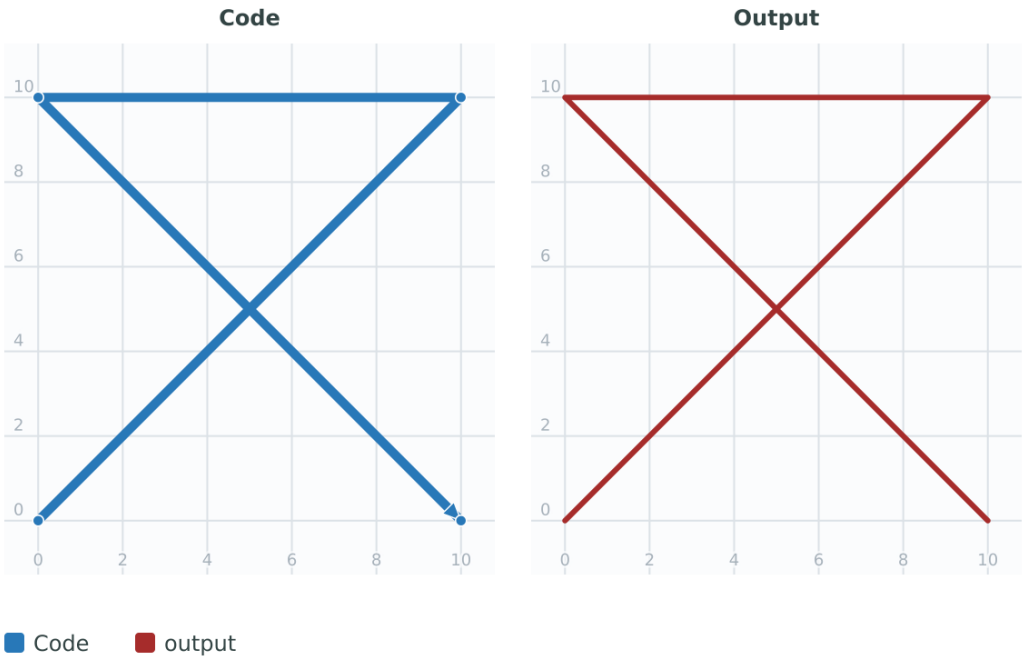
Code

```
SELECT ST_Node('LINESTRINGZ(0 0 0, 10 10 10, 0 10 5, 10 0 3)'::geometry) As  output;
```

Output

```
MULTILINESTRING Z ((0 0 0,5 5 4.5),(5 5 4.5,10 10 10,0 10 5,5 5 4.5),(5 5 4.5,10 0 3))
```

Figure



Noding two LineStrings which share common linework. Note that the result linework is dissolved.

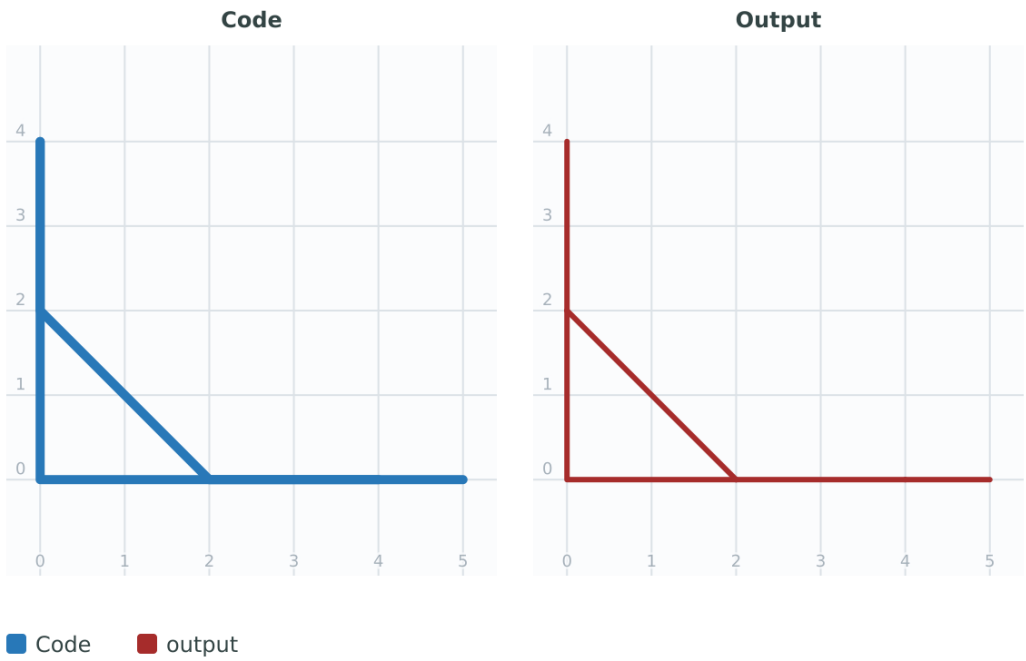
Code

```
SELECT ST_Node('MULTILINESTRING ((2 5,2 1,7 1),(6 1,4 1,2 3,2 5))'::geometry) As  output;
```

Output

```
MULTILINESTRING((2 5,2 3),(2 3,2 1,4 1),(4 1,2 3),(4 1,6 1),(6 1,7 1))
```

Figure



☒☒

ST_UnaryUnion, ST_AsBinary

7.13.6 ST_Split

ST_Split — Returns a collection of geometries created by splitting a geometry by another geometry.

☒☒

geometry **ST_Split**(geometry input, geometry blade);

☒☒

The function supports splitting a LineString by a (Multi)Point, (Multi)LineString or (Multi)Polygon boundary, or a (Multi)Polygon by a LineString. When a (Multi)Polygon is used as as the blade, its linear components (the boundary) are used for splitting the input. The result geometry is always a collection.

This function is in a sense the opposite of ST_Union. Applying ST_Union to the returned collection should theoretically yield the original geometry (although due to numerical rounding this may not be exactly the case).



□□
If the the input and blade do not intersect due to numerical precision issues, the input may not be split as expected. To avoid this situation it may be necessary to snap the input to the blade first, using ST_Snap with a small tolerance.

Availability: 2.0.0 requires GEOS

Enhanced: 2.2.0 support for splitting a line by a multiline, a multipoint or (multi)polygon boundary was introduced.

Enhanced: 2.5.0 support for splitting a polygon by a multiline was introduced.

□□

Split a circular polygon by a diagonal line.

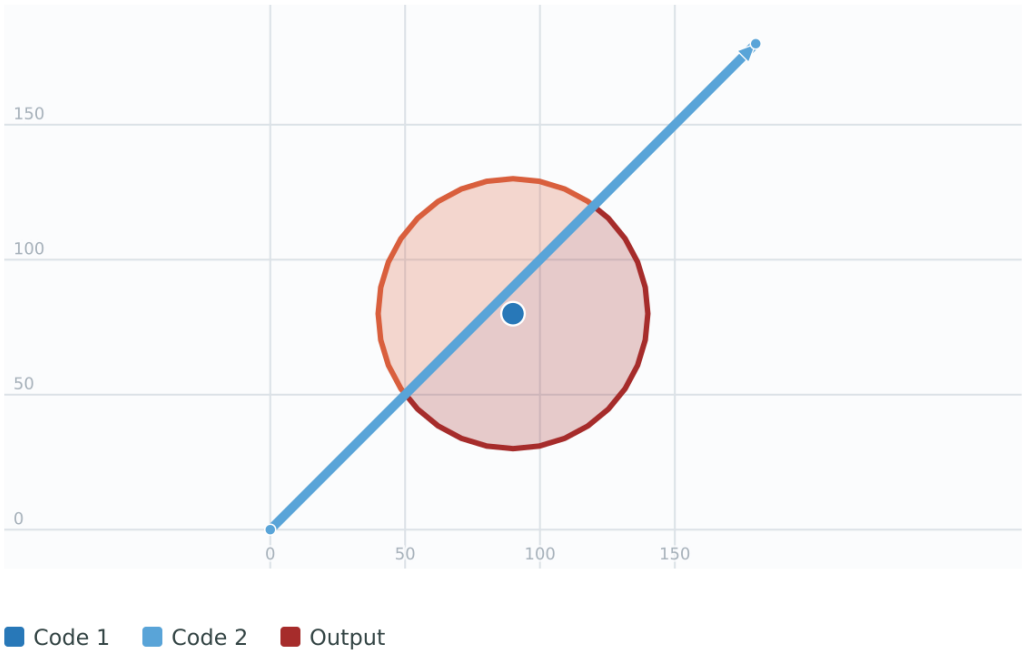
Code

```
SELECT
         ST_Split(
           ST_Buffer('POINT(100 90)'::geometry, 50),
           'LINESTRING(10 10,190 190)'::geometry
         );
```

Output

```
GEOMETRYCOLLECTION(POLYGON((150 90,149 80.2,146.2 70.9,141.6 62.2,135.4 54.6,127.8 48.4,119.1 43.8,109.8 41,100 ↩
    40,90.2 41,80.9 43.8,72.2 48.4,64.6 54.6,60.1 60.1,129.9 129.9,135.4 125.4,141.6 117.8,146.2 109.1,149 99.8,150 ↩
    90)), POLYGON((60.1 60.1,58.4 62.2,53.8 70.9,51 80.2,50 90,51 99.8,53.8 109.1,58.4 117.8,64.6 125.4,72.2 ↩
    131.6,80.9 136.2,90.2 139,100 140,109.8 139,119.1 136.2,127.8 131.6,129.9 129.9,60.1 60.1)))
```

Figure



Split a MultiLineString by a Point, where the point lies exactly on both LineStrings elements.

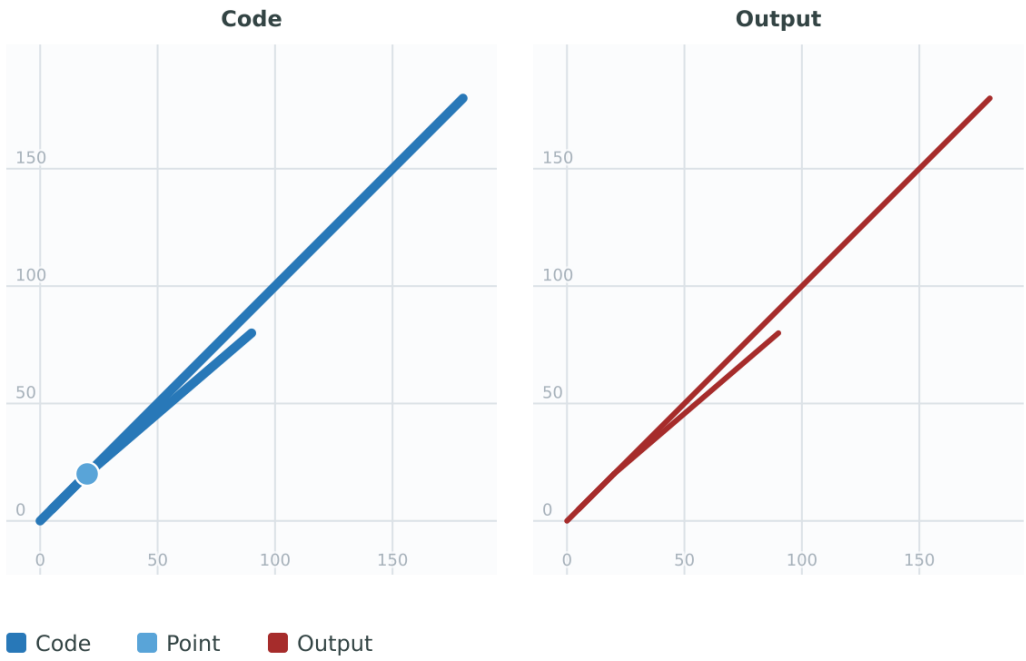
Code

```
SELECT ST_Split('MULTILINESTRING((10 10,190 190),(15 15,30 30,100 90))',
    ST_Point(30, 30));
```

Output

```
GEOMETRYCOLLECTION(
    LINESTRING(10 10,30 30),
    LINESTRING(30 30,190 190),
    LINESTRING(15 15,30 30),
    LINESTRING(30 30,100 90)
)
```

Figure

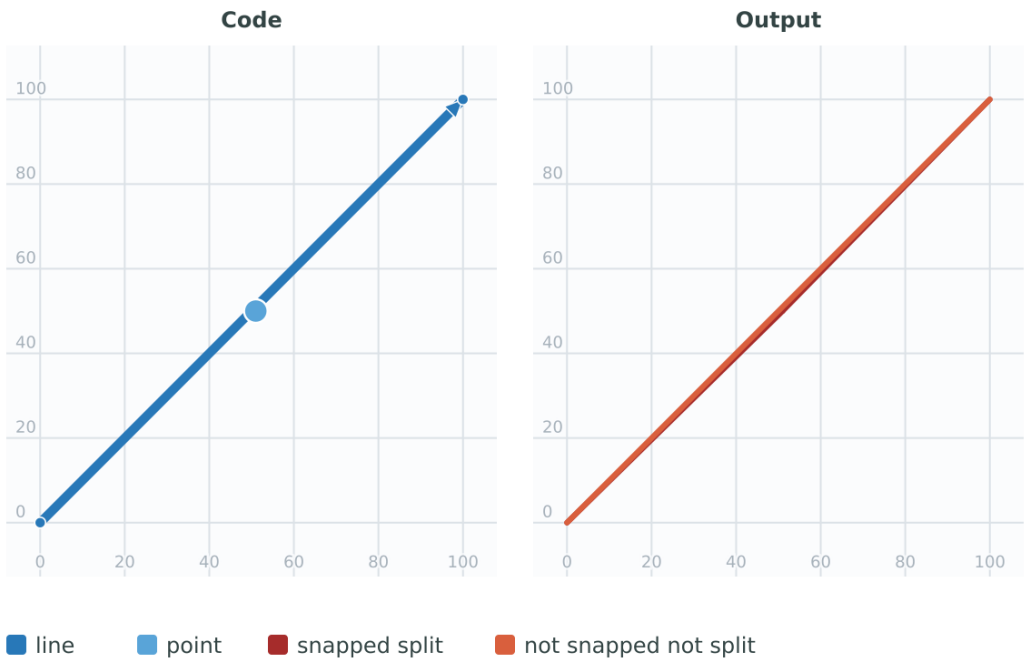


Split a LineString by a Point, where the point does not lie exactly on the line. Shows using ST_Snap to snap the line to the point to allow it to be split.

Code

```
WITH data AS (SELECT
  'LINESTRING(0 0,100 100)'::geometry AS line,
  'POINT(51 50)':: geometry AS point
)
SELECT ST_Split(ST_Snap(line, point, 1), point) AS snapped_split,
       ST_Split(line, point) AS not_snapped_not_split
       FROM data;
```

Figure



☒☒

ST_Snap, ST_AsBinary

7.13.7 ST_Subdivide

ST_Subdivide — Computes a rectilinear subdivision of a geometry.

☒☒

setof geometry **ST_Subdivide**(geometry geom, integer max_vertices=256, float8 gridSize = -1);

☒☒

Returns a set of geometries that are the result of dividing `geom` into parts using rectilinear lines, with each part containing no more than `max_vertices`.

`max_vertices` must be 5 or more, as 5 points are needed to represent a closed box.

If the optional `gridSize` parameter is given (GEOS-3.9.0 or higher required), all result vertices are guaranteed to fall on a snap-rounded grid of the specified size. Note that operations performed on a grid may contain small artifacts produced during grid alignment, see ST_ReducePrecision.

Point-in-polygon and other spatial operations are normally faster for indexed subdivided datasets. Since the bounding boxes for the parts usually cover a smaller area than the original geometry bbox, index queries produce fewer ”hit” cases. The ”hit” cases are faster because the spatial operations executed by the index recheck process fewer points.



☒☒

When casting a subdivided geometry to geography, the resulting geography may differ from the original. Subdivision adds vertices in planar (geometry) space. If vertices are inserted along the boundary, they will alter the geographical representation, where edges are interpreted as geodesic segments. To minimize distortion, first densify the geography using ST_Segmentize to add geodesic vertices, then cast to geometry before subdivision.



☒☒

This is a set-returning function (SRF) that return a set of rows containing single geometry values. It can be used in a SELECT list or a FROM clause to produce a result set with one record for each result geometry.

GEOS ☒☒☒☒☒

2.2.0 ☒☒☒☒☒☒☒☒☒☒☒☒.

Enhanced: 2.5.0 reuses existing points on polygon split, vertex count is lowered from 8 to 5.

Enhanced: 3.1.0 accept a gridSize parameter.

Requires GEOS >= 3.9.0 to use the gridSize parameter

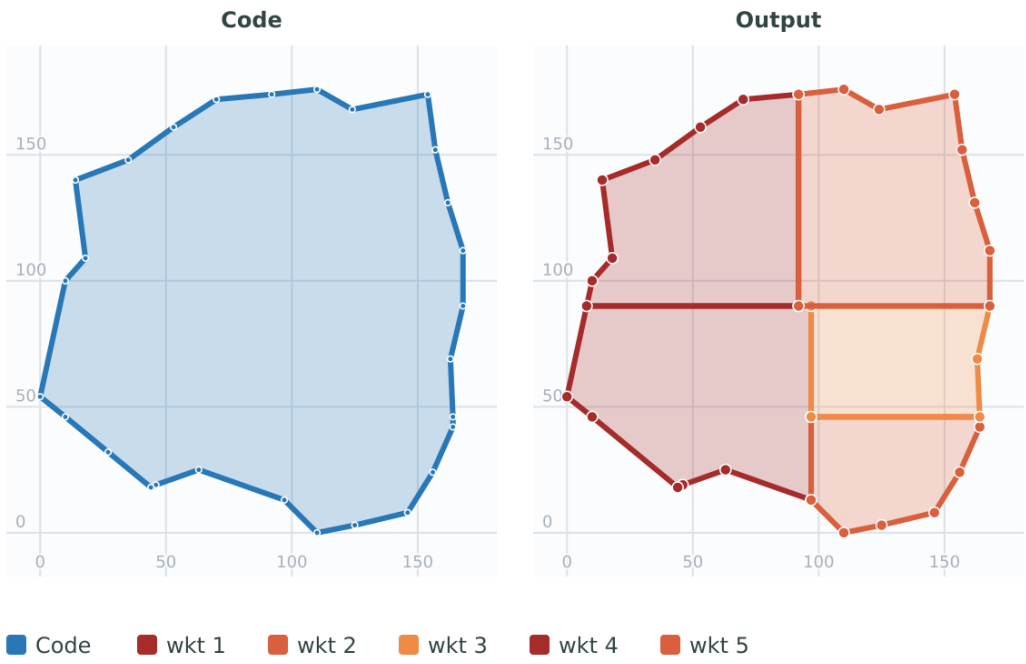
☒☒

Example: Subdivide a polygon into parts with no more than 10 vertices, and assign each part a unique id.

Code

```
SELECT row_number() OVER() As rn,geom As wkt
    FROM (SELECT ST_Subdivide('POLYGON((132 10,119 23,85 35,68 29,66 28,49 42,32 56,22 ↩
    64,32 110,40 119,36 150,
        57 158,75 171,92 182,114 184,132 186,146 178,176 184,179 162,184 141,190 122,
        190 100,185 79,186 56,186 52,178 34,168 18,147 13,132 10))'::geometry, 10))  AS f( ↩
    geom);
```

Figure



Example: Densify a long geography line using ST_Segmentize(geography, distance), and use ST_Subdivide to split the resulting line into sublines of 8 vertices. Densification minimizes the impact of changes to the geography representation of a geometry when subdividing.

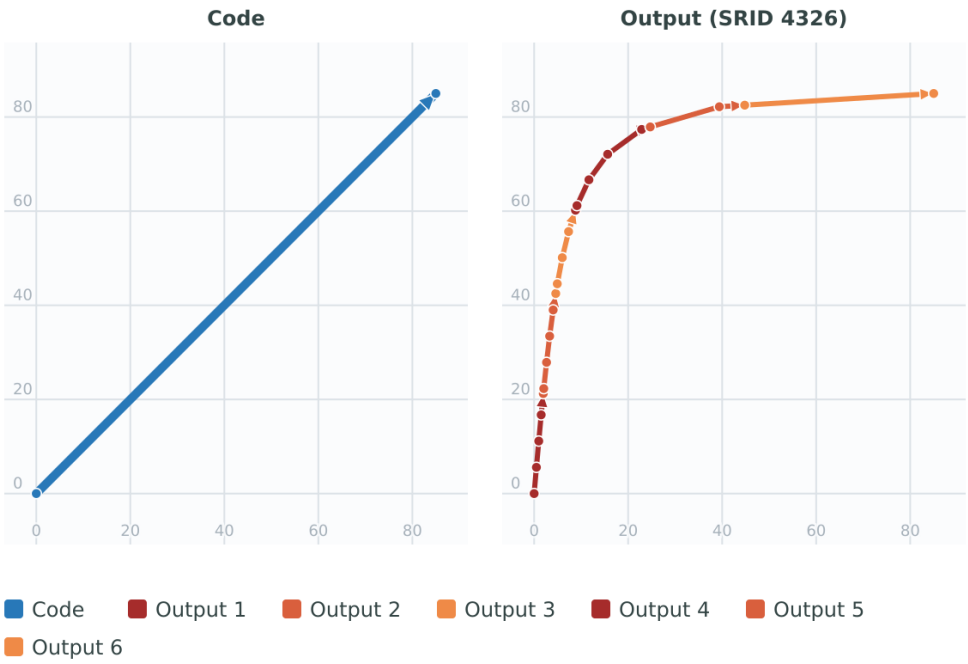
Code

```
SELECT ST_Subdivide(ST_Segmentize('LINESTRING(0 0,85 85)'::geography,
                            1200000)::geometry, 8);
```

Output

```
LINESTRING(0 0,0.487578359029357 5.57659056746196,0.984542144675897 11.1527721155093,1.50101059639722  ↩
    16.7281035483571,1.94532113630331 21.25)
LINESTRING(1.94532113630331 21.25,2.04869538062779 22.3020741387339,2.64204641967673 27.8740533545155,3.29994062412787 ↩
     33.443216802941,4.04836719489742 39.0084282520239,4.59890468420694 42.5)
LINESTRING(4.59890468420694 42.5,4.92498503922732 44.5680389206321,5.98737409390639 50.1195229244701,7.3290919767674  ↩
    55.6587646879025,8.79638749938413 60.1969505994924)
LINESTRING(8.79638749938413 60.1969505994924,9.11375579533779 61.1785363177625,11.6558166691368  ↩
    66.6648504160202,15.642041247655 72.0867690601745,22.8716627200212 77.3609628116894,24.6991785131552  ↩
    77.8939011989848)
LINESTRING(24.6991785131552 77.8939011989848,39.4046096622744 82.1822848017636,44.7994523421035 82.5156766227011)
LINESTRING(44.7994523421035 82.5156766227011,85 85)
```

Figure



Example: Subdivide the complex geometries of a table in-place. The original geometry records are deleted from the source table, and new records for each subdivided result geometry are inserted.

Code

```
WITH complex_areas_to_subdivide AS (
    DELETE from polygons_table
    WHERE ST_NPoints(geom)
> 255
    RETURNING id, column1, column2, column3, geom
)
INSERT INTO polygons_table (fid, column1, column2, column3, geom)
    SELECT fid, column1, column2, column3,
           ST_Subdivide(geom, 255) as geom
    FROM complex_areas_to_subdivide;
```

Example: Create a new table containing subdivided geometries, retaining the key of the original geometry so that the new table can be joined to the source table. Since ST_Subdivide is a set-returning (table) function that returns a set of single-value rows, this syntax automatically produces a table with one row for each result part.

Code

```
CREATE TABLE subdivided_geoms AS
    SELECT pkey, ST_Subdivide(geom) AS geom
    FROM original_geoms;
```

□□

ST_ClipByBox2D, ST_Segmentize, ST_Split, ST_NPoints, ST_ReducePrecision

7.13.8 ST_SymDifference

ST_SymDifference — Computes a geometry representing the portions of geometries A and B that do not intersect.

□□

geometry **ST_SymDifference**(geometry geomA, geometry geomB, float8 gridSize = -1);

□□

Returns a geometry representing the portions of geometries A and B that do not intersect. This is equivalent to `ST_Union(A,B) - ST_Intersection(A,B)`. It is called a symmetric difference because `ST_SymDifference(A,B) = ST_SymDifference(B,A)`.

If the optional `gridSize` parameter is given (GEOS-3.9.0 or higher required), all result vertices are guaranteed to fall on a snap-rounded grid of the specified size. Note that operations performed on a grid may contain small artifacts produced during grid alignment, see ST_ReducePrecision.

GEOS □□□□□

Enhanced: 3.1.0 accept a gridSize parameter.

Requires GEOS >= 3.9.0 to use the gridSize parameter

This method implements the OGC Simple Features Implementation Specification for SQL 1.1. s2.1.1.3

This method implements the SQL/MM specification. SQL-MM 3: 5.1.21

This function supports 3d and will not drop the z-index. However, the result is computed using XY only. The result Z values are copied, averaged or interpolated.

□□

This example shows a 2D-safe symmetric difference of two LineStrings.

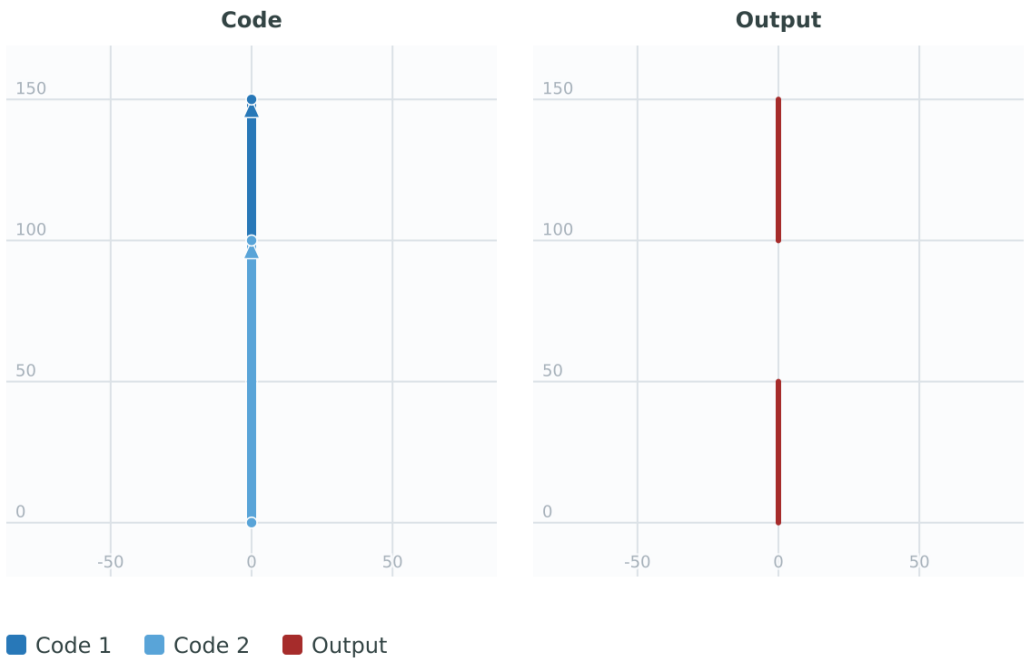
Code

```
SELECT ST_SymDifference('LINESTRING(50 100,50 200)',
        'LINESTRING(50 50,50 150)'
    );
```

Output

```
MULTILINESTRING((50 150,50 200),(50 50,50 100))
```

Figure



In 3D, this example does not produce the expected result.

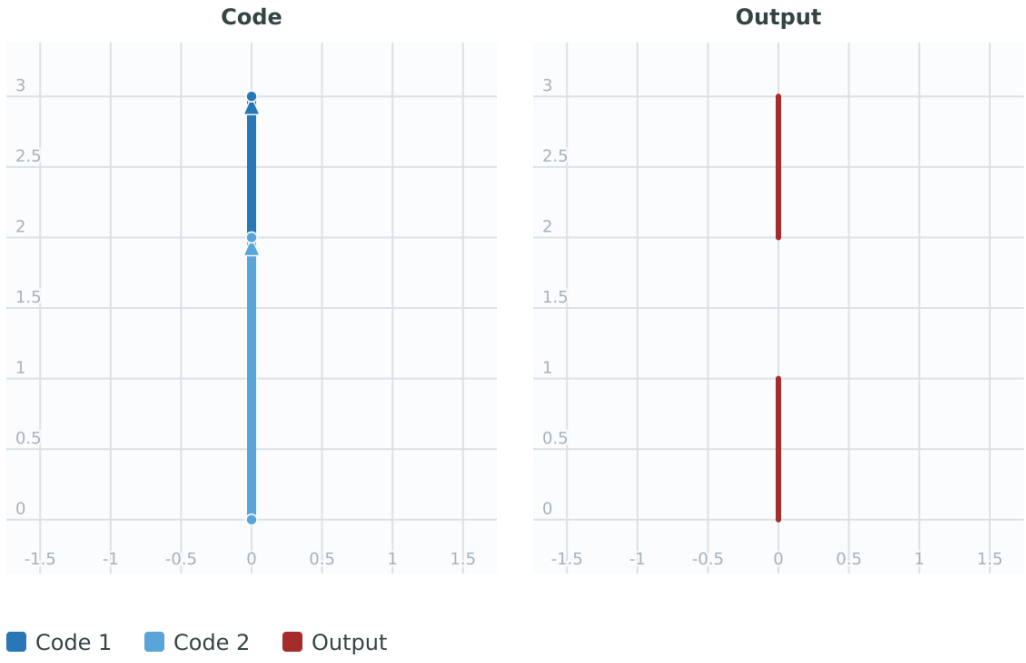
Code

```
SELECT ST_SymDifference('LINESTRING(1 2 1,1 4 2)',
    'LINESTRING(1 1 3,1 3 4)');
```

Output

```
MULTILINESTRING((1 3 4,1 4 2),(1 1 3,1 2 1))
```

Figure



□□

ST_Difference, ST_Intersection, ST_Union, ST_ReducePrecision

7.13.9 ST_UnaryUnion

ST_UnaryUnion — Computes the union of the components of a single geometry.

□□

geometry **ST_UnaryUnion**(geometry geom, float8 gridSize = -1);

□□

A single-input variant of ST_Union. The input may be a single geometry, a MultiGeometry, or a GeometryCollection. The union is applied to the individual elements of the input.

This function can be used to fix MultiPolygons which are invalid due to overlapping components. However, the input components must each be valid. An invalid input component such as a bow-tie polygon may cause an error. For this reason it may be better to use ST_MakeValid.

Another use of this function is to node and dissolve a collection of linestrings which cross or overlap to make them simple. (ST_Node also does this, but it does not provide the `gridSize` option.)

It is possible to combine ST_UnaryUnion with ST_Collect to fine-tune how many geometries are be unioned at once. This allows trading off between memory usage and compute time, striking a balance between ST_Union and ST_MemUnion.

If the optional `gridSize` parameter is given (GEOS-3.9.0 or higher required), all result vertices are guaranteed to fall on a snap-rounded grid of the specified size. Note that operations performed on a grid may contain small artifacts produced during grid alignment, see ST_ReducePrecision.

 This function supports 3d and will not drop the z-index. However, the result is computed using XY only. The result Z values are copied, averaged or interpolated.

Enhanced: 3.1.0 accept a gridSize parameter.

Requires GEOS >= 3.9.0 to use the gridSize parameter

2.0.0 □□□□□□□□□□□□.

□□

ST_Union, ST_MemUnion, ST_MakeValid, ST_Collect, ST_Node, ST_ReducePrecision

7.13.10 ST_Union

ST_Union — Computes a geometry representing the point-set union of the input geometries.

□□

geometry **ST_Union**(geometry g1, geometry g2);
geometry **ST_Union**(geometry g1, geometry g2, float8 gridSize);
geometry **ST_Union**(geometry[] g1_array);
geometry **ST_Union**(geometry set g1field);
geometry **ST_Union**(geometry set g1field, float8 gridSize);

□□

Unions the input geometries, merging geometry to produce a result geometry with no overlaps. The output may be an atomic geometry, a MultiGeometry, or a Geometry Collection. Comes in several variants:

Two-input variant: returns a geometry that is the union of two input geometries. If either input is NULL, then NULL is returned.

Array variant: returns a geometry that is the union of an array of geometries.

Aggregate variant: returns a geometry that is the union of a rowset of geometries. The ST_Union() function is an "aggregate" function in the terminology of PostgreSQL. That means that it operates on rows of data, in the same way the SUM() and AVG() functions do and like most aggregates, it also ignores NULL geometries.

See ST_UnaryUnion for a non-aggregate, single-input variant.

The ST_Union array and set variants use the fast Cascaded Union algorithm described in http://blog.cleverele 2009/01/must-faster-unions-in-postgis-14.html

If the optional `gridSize` parameter is given (GEOS-3.9.0 or higher required), all result vertices are guaranteed to fall on a snap-rounded grid of the specified size. Note that operations performed on a grid may contain small artifacts produced during grid alignment, see ST_ReducePrecision.



□□
ST_Collect may sometimes be used in place of ST_Union, if the result is not required to be non-overlapping. ST_Collect is usually faster than ST_Union because it performs no processing on the collected geometries.

GEOS □□□□□

ST_Union creates MultiLineString and does not sew LineStrings into a single LineString. Use ST_LineMerge to sew LineStrings.

NOTE: this function was formerly called GeomUnion(), which was renamed from "Union" because UNION is an SQL reserved word.

Enhanced: 3.1.0 accept a gridSize parameter.

Requires GEOS >= 3.9.0 to use the gridSize parameter

Changed: 3.0.0 does not depend on SFCGAL.

Availability: 1.4.0 - ST_Union was enhanced. ST_Union(geomarray) was introduced and also faster aggregate collection in PostgreSQL.

 This method implements the OGC Simple Features Implementation Specification for SQL 1.1. s2.1.1.3



□□
Aggregate version is not explicitly defined in OGC SPEC.

 This method implements the SQL/MM specification. SQL-MM 3: 5.1.19 the z-index (elevation) when polygons are involved.

 This function supports 3d and will not drop the z-index. However, the result is computed using XY only. The result Z values are copied, averaged or interpolated.

□□

Aggregate example

Code

```
SELECT id,
       ST_Union(geom) as singlegeom
FROM sometable f
GROUP BY id;
```

Non-Aggregate example

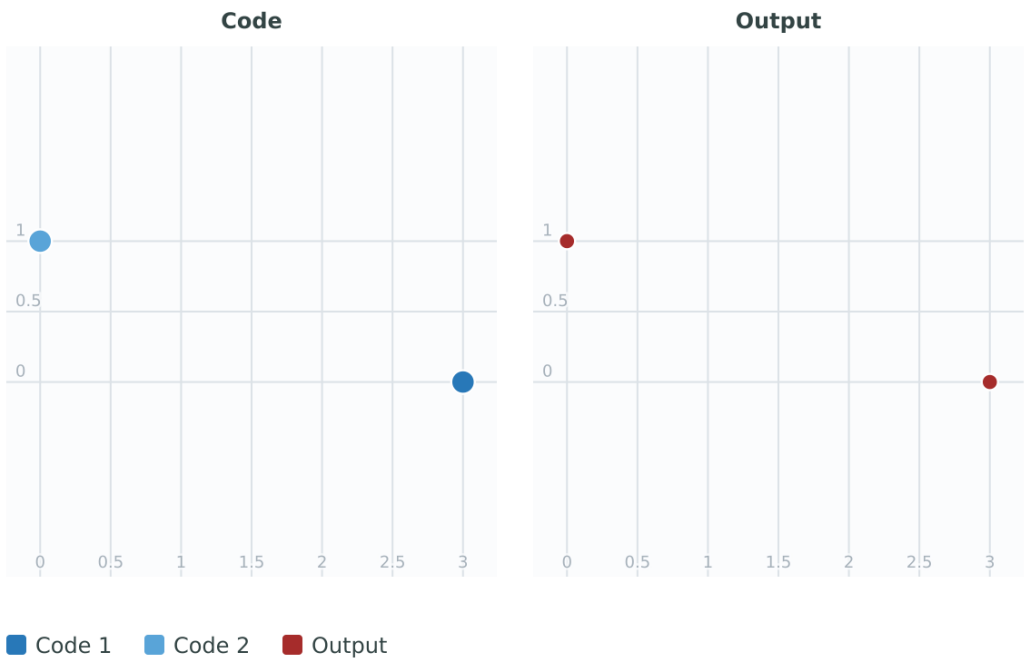
Code

```
SELECT ST_Union('POINT(1 2)'::geometry, 'POINT(-2 3)'::geometry);
```

Output

```
MULTIPOINT((1 2),(-2 3))
```

Figure



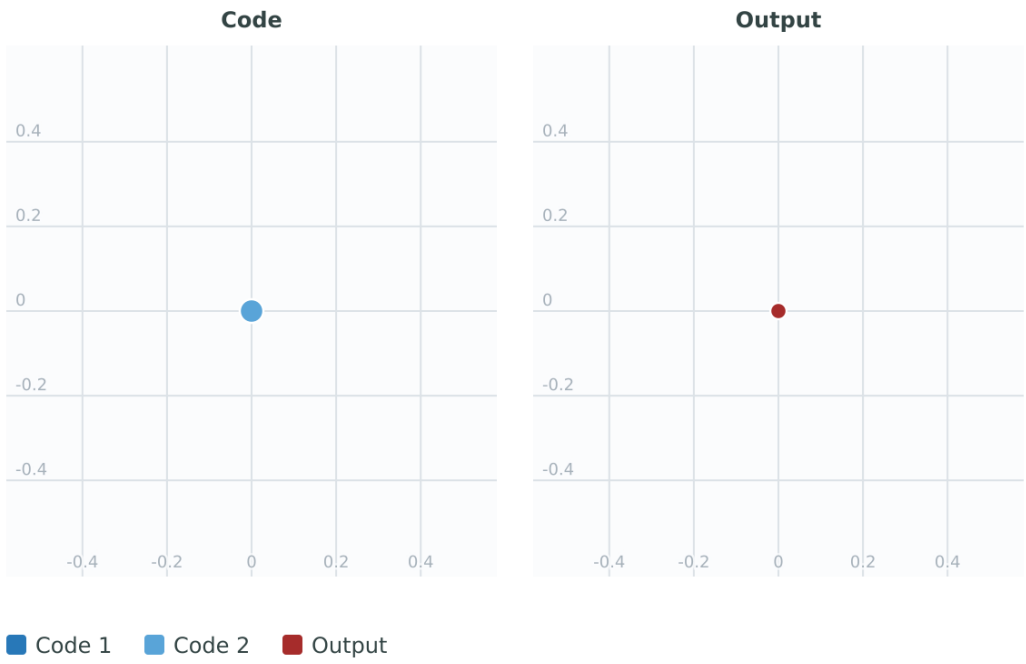
Code

```
SELECT ST_Union('POINT(1 2)'::geometry, 'POINT(1 2)'::geometry);
```

Output

```
POINT(1 2)
```

Figure



3D example - sort of supports 3D (and with mixed geometry types!)

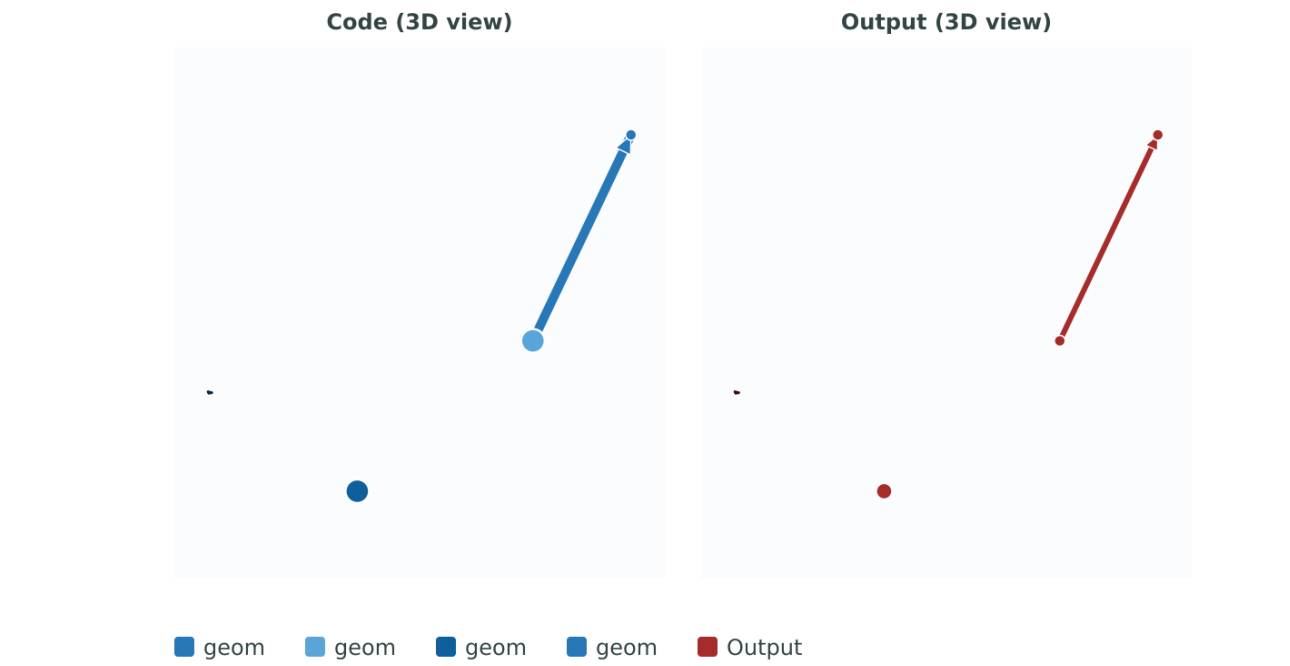
Code

```
SELECT ST_Union(geom)
FROM (
    SELECT 'POLYGON Z((-7 4.2 5,-7.1 4.2 5,-7.1 4.3 5,-7 4.2 5))'::geometry geom
    UNION ALL
    SELECT 'POINT Z(5 5 5)'::geometry geom
    UNION ALL
    SELECT 'POINT Z(-2 3 1)'::geometry geom
    UNION ALL
    SELECT 'LINESTRING Z(5 5 5,10 10 10)'::geometry geom
) AS foo;
```

Output

```
GEOMETRYCOLLECTION(POINT(-2 3 1), LINESTRING(5 5 5,10 10 10), POLYGON((-7 4.2 5,-7.1 4.2 5,-7.1 4.3 5,-7 4.2 5)))
```

Figure



This 3D example does not mix dimensions.

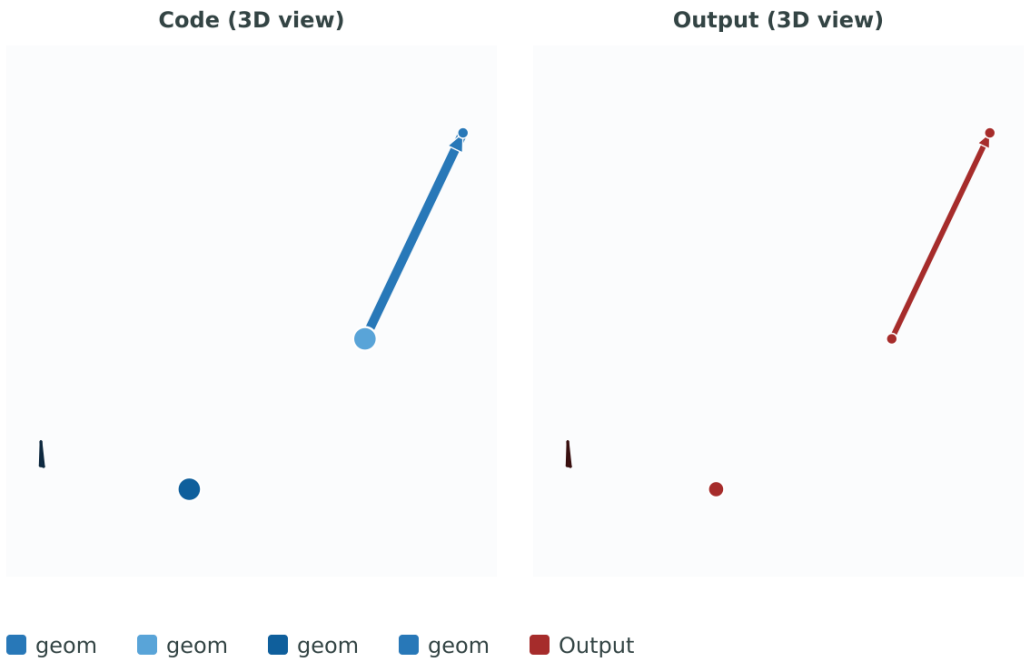
Code

```
SELECT ST_Union(geom)
FROM (
    SELECT 'POLYGON((-7 4.2 2,-7.1 4.2 3,-7.1 4.3 2,-7 4.2 2))'::geometry geom
    UNION ALL
    SELECT 'POINT(5 5 5)'::geometry geom
    UNION ALL
    SELECT 'POINT(-2 3 1)'::geometry geom
    UNION ALL
    SELECT 'LINESTRING(5 5 5,10 10 10)'::geometry geom
) AS foo;
```

Output

```
GEOMETRYCOLLECTION(POINT(-2 3 1), LINESTRING(5 5 5,10 10 10), POLYGON((-7 4.2 2,-7.1 4.2 3,-7.1 4.3 2,-7 4.2 2)))
```

Figure



This example unions an array built from a query.

Code

```
SELECT ST_Union(ARRAY(SELECT geom FROM sometable));
```

This example unions an array of linestrings.

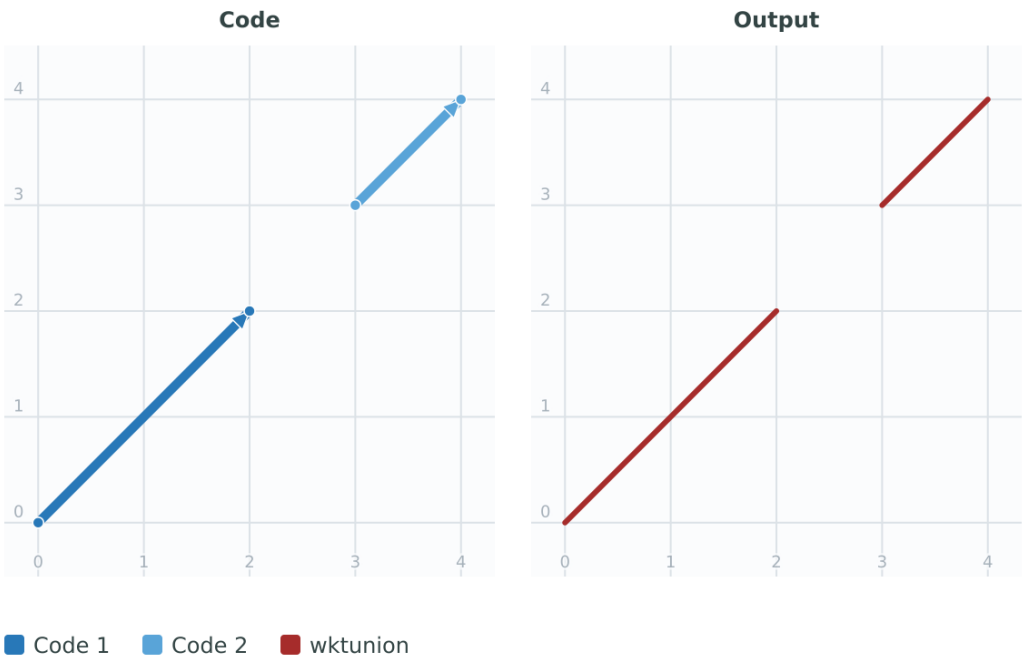
Code

```
SELECT ST_Union(ARRAY[
    'LINESTRING(1 2,3 4)'::geometry,
    'LINESTRING(4 5,5 6)'::geometry
]) As wktunion;
```

Output

```
MULTILINESTRING((1 2,3 4),(4 5,5 6))
```

Figure



□□

ST_CoverageUnion, ST_UnaryUnion, ST_MemUnion, ST_Collect, ST_Intersection, ST_Difference, ST_SymDi
ST_LineMerge

7.14 □□□□□□

7.14.1 ST_Buffer

ST_Buffer — Computes a geometry covering all points within a given distance from a geometry.

□□

geometry **ST_Buffer**(geometry g1, float radius_of_buffer, text buffer_style_parameters = ”);
geometry **ST_Buffer**(geometry g1, float radius_of_buffer, integer num_seg_quarter_circle);
geography **ST_Buffer**(geography g1, float radius_of_buffer, text buffer_style_parameters);
geography **ST_Buffer**(geography g1, float radius_of_buffer, integer num_seg_quarter_circle);

□□

Computes a POLYGON or MULTIPOLYGON that represents all points whose distance from a geometry/geography is less than or equal to a given distance. A negative distance shrinks the geometry rather than expanding it. A negative distance may shrink a polygon completely, in which case POLYGON EMPTY is returned. For points and lines negative distances always return empty results.

For geometry, the distance is specified in the units of the Spatial Reference System of the geometry. For geography, the distance is specified in meters.

The optional third parameter controls the buffer accuracy and style. The accuracy of circular arcs in the buffer is specified as the number of line segments used to approximate a quarter circle (default is 8). The buffer style can be specified by providing a list of blank-separated key=value pairs as follows:

- 'quad_segs=#' : number of line segments used to approximate a quarter circle (default is 8).
- 'endcap=round|flat|square' : endcap style (defaults to "round"). 'butt' is accepted as a synonym for 'flat'.
- 'join=round|mitre|bevel' : join style (defaults to "round"). 'miter' is accepted as a synonym for 'mitre'.
- 'mitre_limit=#.#' : mitre ratio limit (only affects mitered join style). 'miter_limit' is accepted as a synonym for 'mitre_limit'.
- 'side=both|left|right' : defaults to 'both'. 'left' or 'right' performs a single-sided buffer on the geometry, with the buffered side relative to the direction of the line. This is only applicable to LINESTRING geometry and does not affect POINT or POLYGON geometries. By default end caps are square when 'left' or 'right' are specified.



☒☒

For geography this is a thin wrapper around the geometry implementation. It determines a planar spatial reference system that best fits the bounding box of the geography object (trying UTM, Lambert Azimuthal Equal Area (LAEA) North/South pole, and finally Mercator). The buffer is computed in the planar space, and then transformed back to WGS84. This may not produce the desired behavior if the input object is much larger than a UTM zone or crosses the dateline

---



☒☒
Buffer can handle invalid inputs and the output is always a valid polygonal geometry. Buffering by distance 0 is sometimes used as a way of repairing invalid polygons. ST_MakeValid is more suitable for this process as it can handle multi-polygons.



☒☒
Buffering is sometimes used to perform a within-distance search. For this use case it is more efficient to use ST_DWithin.



☒☒
This function ignores the Z dimension. It always gives a 2D result even when used on a 3D geometry.

Enhanced: 2.5.0 - ST_Buffer geometry support was enhanced to allow for side buffering specification `side=both|left|right`.

Availability: 1.5 - ST_Buffer was enhanced to support different endcaps and join types. These are useful for example to convert road linestrings into polygon roads with flat or square edges instead of rounded edges. Thin wrapper for geography was added.

GEOS ☒☒☒☒☒

 This method implements the OGC Simple Features Implementation Specification for SQL 1.1. s2.1.1.3

 This method implements the SQL/MM specification. SQL-MM IEC 13249-3: 5.1.30

□□

The textual results are rounded to whole coordinate units for readability. Each result remains available as text, while the manual initially favors the shared input/output figure.

quad_segs=8 (□□□)

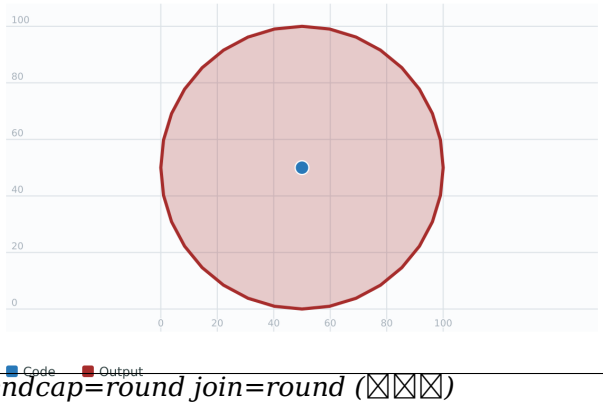
Code

```
SELECT ST_Buffer(
  50, 'quad_segs=8');
```

Output

```
POLYGON((150 90,149 80,146 71,142 62,135 55,128 48,119
44,110 41,100 40,90 41,81 44,72 48,65 55,58 62,54 71,51
80,50 90,51 100,54 109,58 118,65 125,72 132,81 136,90
139,100 140,110 139,119 136,128 132,135 125,142 118,146
109,149 100,150 90))
```

Figure



quad_segs=2

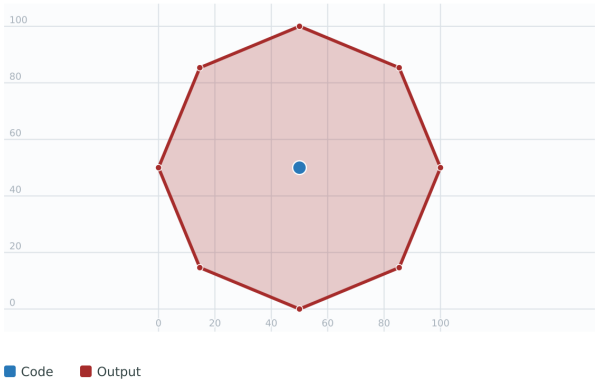
Code

```
ST_GeomFromText('POINT(100 90)'),
SELECT ST_Buffer(                                ST_Geo
  50, 'quad_segs=2');
```

Output

```
POLYGON((150 90,135 55,100 40,65 55,50 90,65 125,100
140,135 125,150 90))
```

Figure



endcap=round join=round (□□□)

Code

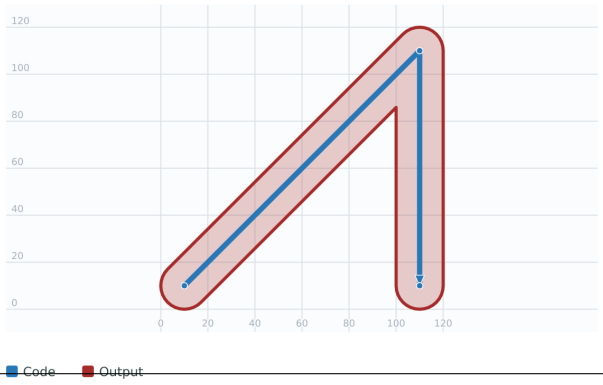
```
SELECT ST_Buffer(

  10, 'endcap=round join=round');
```

Output

```
POLYGON((143 157,144 158,146 159,148 160,150 160,152
160,154 159,156 158,157 157,158 156,159 154,160 152,160
150,160 50,160 48,159 46,158 44,157 43,156 42,154 41,152
40,150 40,148 40,146 41,144 42,143 43,142 44,141 46,140
48,140 50,140 126,57 43,56 42,54 41,52 40,50 40,48 40,46
41,44 42,43 43,42 44,41 46,40 48,40 50,40 52,41 54,42
56,43 57,143 157))
```

Figure



endcap=square

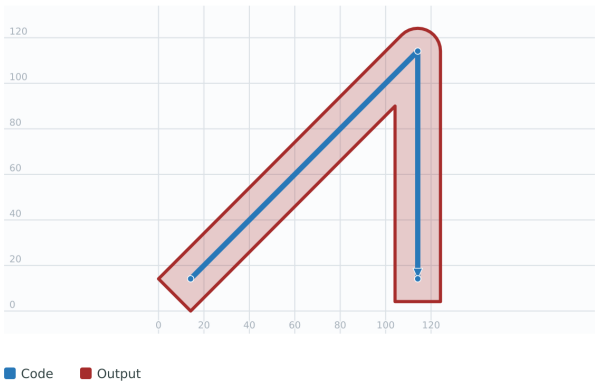
Code

```
ST_GeomFromText('LINESTRING(50 50,150 ↩
SELECT ST_Buffer(                                ST_Geo
  10, 'endcap=square join=round');                 150,
```

Output

```
POLYGON((143 157,144 158,146 159,148 160,150 160,152
160,154 159,156 158,157 157,158 156,159 154,160 152,160
150,160 50,160 40,140 40,140 126,57 43,50 36,36 50,143
157))
```

Figure



endcap=flat

Code

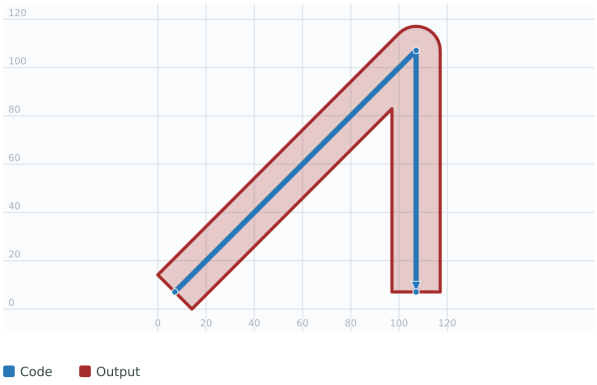
```
SELECT ST_Buffer(

  10, 'endcap=flat join=round');
```

Output

```
POLYGON((143 157,144 158,146 159,148 160,150 160,152
160,154 159,156 158,157 157,158 156,159 154,160 152,160
150,160 50,140 50,140 126,57 43,43 57,143 157))
```

Figure



join=bevel

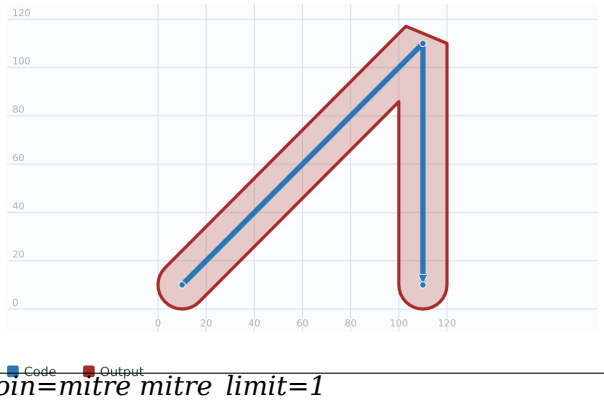
Code

```
SELECT ST_Buffer(
ST_GeomFromText('LINESTRING(50 50,150 ↩
  150,150 50)'),
  10, 'join=bevel');
```

Output

```
POLYGON((143 157,160 150,160 50,160 48,159 46,158 44,157
43,156 42,154 41,152 40,150 40,148 40,146 41,144 42,143
43,142 44,141 46,140 48,140 50,140 126,57 43,56 42,54
41,52 40,50 40,48 40,46 41,44 42,43 43,42 44,41 46,40
48,40 50,40 52,41 54,42 56,43 57,143 157))
```

Figure



join=mitre mitre_limit=5.0 (☒☒☒☒☒☒☒☒)

Code

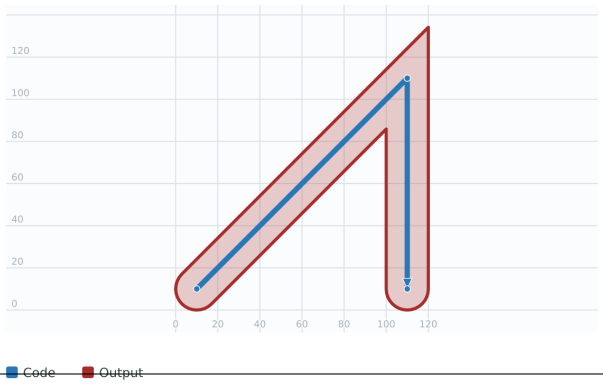
```
SELECT ST_Buffer(

  10, 'join=mitre mitre_limit=5.0');
```

Output

```
POLYGON((160 174,160 50,160 48,159 46,158 44,157 43,156
42,154 41,152 40,150 40,148 40,146 41,144 42,143 43,142
44,141 46,140 48,140 50,140 126,57 43,56 42,54 41,52 40,
50 40,48 40,46 41,44 42,43 43,42 44,41 46,40 48,40 50,40
52,41 54,42 56,43 57,160 174))
```

Figure



join=mitre mitre_limit=1

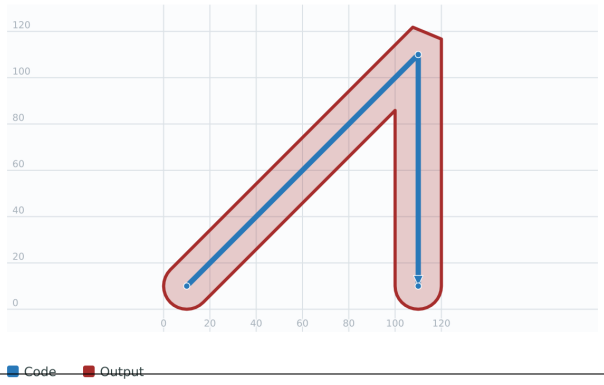
Code

```
SELECT ST_Buffer(ST_GeomFromText('LINESTRING(50 50,150 ↩
  150,150 50)'),
  10, 'join=mitre mitre_limit=1.0');
```

Output

```
POLYGON((148 162,160 157,160 50,160 48,159 46,158 44,157
43,156 42,154 41,152 40,150 40,148 40,146 41,144 42,143
43,142 44,141 46,140 48,140 50,140 126,57 43,56 42,54
41,52 40,50 40,48 40,46 41,44 42,43 43,42 44,41 46,40
48,40 50,40 52,41 54,42 56,43 57,148 162))
```

Figure



side=left

Code

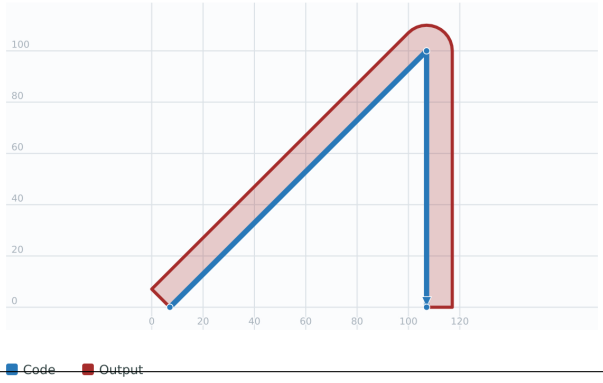
```
SELECT ST_Buffer(

  10, 'side=left');
```

Output

```
POLYGON((150 50,150 150,50 50,43 57,143 157,144 158,146
159,148 160,150 160,152 160,154 159,156 158,157 157,158
156,159 154,160 152,160 150,160 50,150 50))
```

Figure



side=right

Code

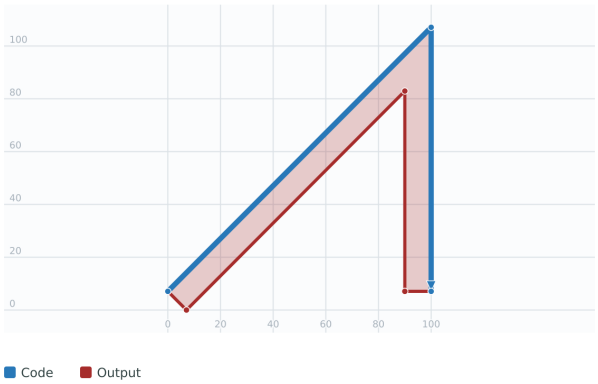
```
SELECT ST_Buffer(
  ST_GeomFromText('LINESTRING(50 50,150 ↩
  150,150 50)'),

   10, 'side=right');
```

Output

```
POLYGON((50 50,150 150,150 50,140 50,140 126,57 43,50
50))
```

Figure



side=left join=mitre

Code

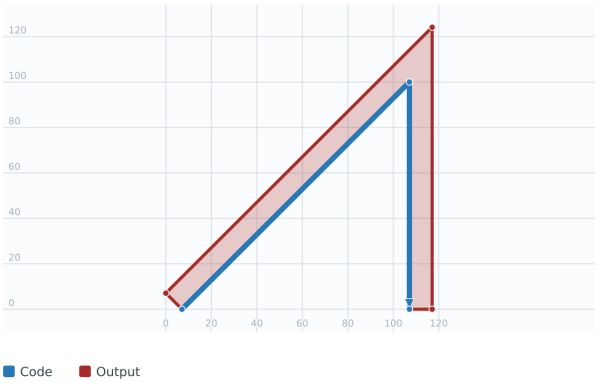
```
SELECT ST_Buffer(

  10, 'side=left join=mitre');
```

Output

```
POLYGON((150 50,150 150,50 50,43 57,160 174,160 50,150
50))
```

Figure



right-hand winding, polygon boundary side=left

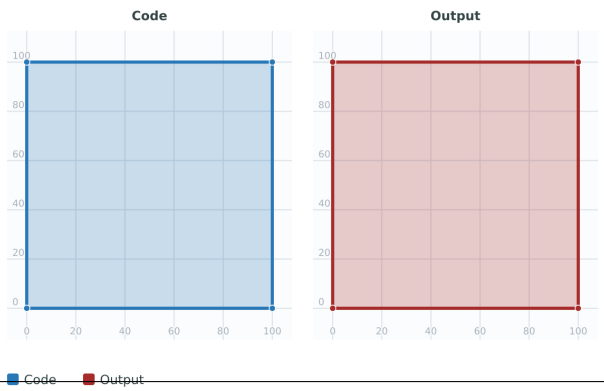
Code

```
SELECT ST_Buffer(
  ST_GeomFromText('LINESTRING(50 50,150 ↩
  150,150 50)'),
     'POLYGON ((50 50,50 150,150 150,150 ↩
    50,50 50))')),
   20, 'side=left');
```

Output

```
POLYGON((50 50,50 150,150 150,150 50,50 50))
```

Figure



right-hand winding, polygon boundary side=right

Code

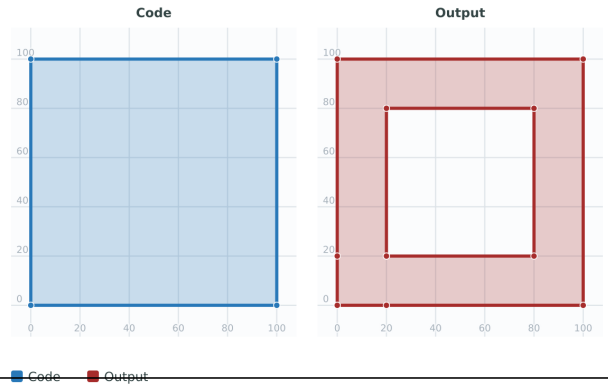
```
SELECT ST_Buffer(

    'POLYGON ((50 50,50 150,150 150,150  ↩
   50,50 50))')),
  20, 'side=right');
```

Output

```
POLYGON((50 50,50 70,50 150,150 150,150 50,70 50,50 50),
(70 70,130 70,130 130,70 130,70 70))
```

Figure




```
ST_ForceRHR(ST_Boundary(ST_GeomFromText ↩
  (
```

A buffered point approximates a circle. A buffered point forcing approximation of (see diagram) 2 points per quarter circle is poly with 8 sides (see diagram)

Code

```
SELECT ST_NPoints(ST_Buffer('POINT(100 90)'::geometry, 50)) As promisingcircle_pcount,
ST_NPoints(ST_Buffer('POINT(100 90)'::geometry, 50, 2)) As lamecircle_pcount;
```

Output

```
promisingcircle_pcount | lamecircle_pcount
-----------------------+-------------------
             33 |                 9
```

A lighter, less smooth circle using only 2 points per quarter circle is an octagon.

The following example creates a 100 meter octagon in NAD83 longitude/latitude by transforming to the Massachusetts state plane meter projection and then buffering. The figure shows the input and output in separate panels because the coordinates are in different SRIDs.

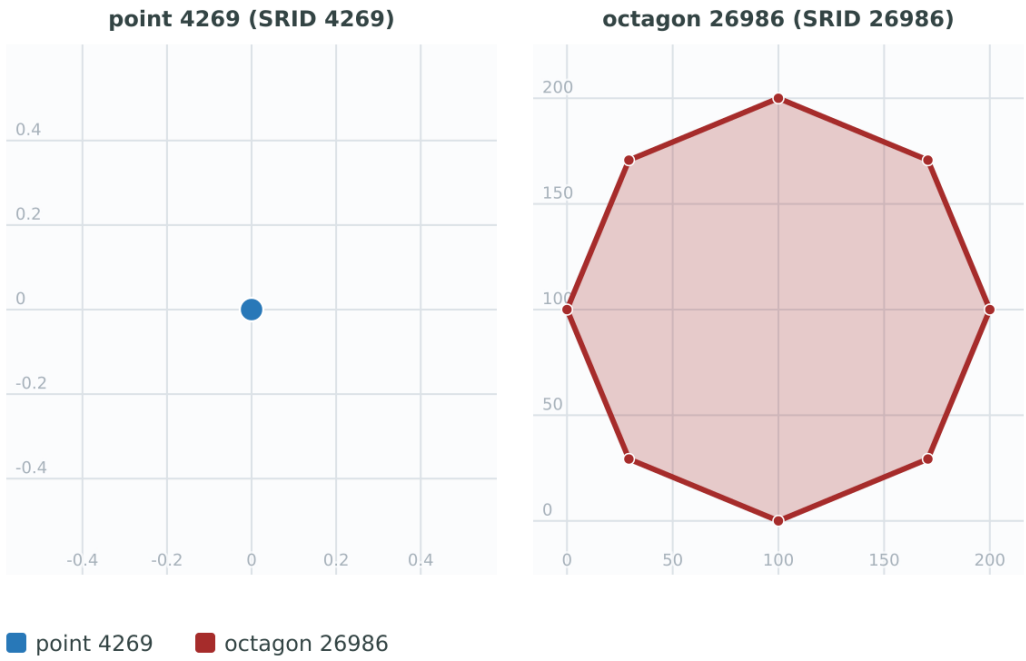
Code

```
WITH input AS (
  SELECT ST_SetSRID(ST_Point(-71.063526, 42.35785), 4269) AS point_4269
), projected AS (
  SELECT point_4269,
         ST_Transform(point_4269, 26986) AS point_26986
  FROM input
)
SELECT point_4269 AS input_point_4269,
       ST_Buffer(point_26986, 100, 2) AS octagon_26986
FROM projected;
```

Output

```
SRID=4269;POINT(-71.063526 42.35785) | SRID=26986;POLYGON((236057.5905746494 900908.7599186979,236028.30125276805 ↩
    900838.0492405792,235957.5905746494 900808.7599186979,235886.87989653074 900838.0492405792,235857.5905746494 ↩
    900908.7599186979,235886.87989653074 900979.4705968165,235957.5905746494 901008.7599186979,236028.30125276805 ↩
    900979.4705968165,236057.5905746494 900908.7599186979))
```

Figure



Buffering a polygon by a positive distance and then by the same negative distance is a morphological closing operation. It can be useful for smoothing noisy boundaries, such as simplifying a coastline by removing narrow inlets and bays. The result intentionally changes the shape and area of the input, and the distance must be chosen in the units of the input SRS. For data in a geographic SRS, transform to a projected SRS suited to the operation first; for cartographic visualization in Web Mercator, using that display SRS can be appropriate.

Smooth a coastline using a 1 km closing operation, then simplify the result. The example uses Massachusetts State Plane meter units for the operation.

Code

```
WITH projected AS (
  SELECT id, ST_Transform(geom, 26986) AS geom
  FROM coast
),
closed AS (
  SELECT id, ST_Buffer(ST_Buffer(geom, 1000), -1000) AS geom
  FROM projected
)
SELECT id, ST_Transform(ST_SimplifyPreserveTopology(geom, 50), 4326) AS geom
FROM closed;
```

The same technique is described in Paul Ramsey's Removing Complexities. The final simplification step is optional; use ST_SimplifyPreserveTopology when polygonal validity matters, and check the result against the cartographic or analytical tolerance required by the application.

A small negative buffer can also be used as a heuristic to find narrow polygon spikes. The query below compares how much area and perimeter remain after erosion. A large `erosion_index` indicates that the perimeter dropped proportionally more than the area. This is only a preselection aid; the threshold and buffer distance must be chosen for the data units and the kind of spike being checked.

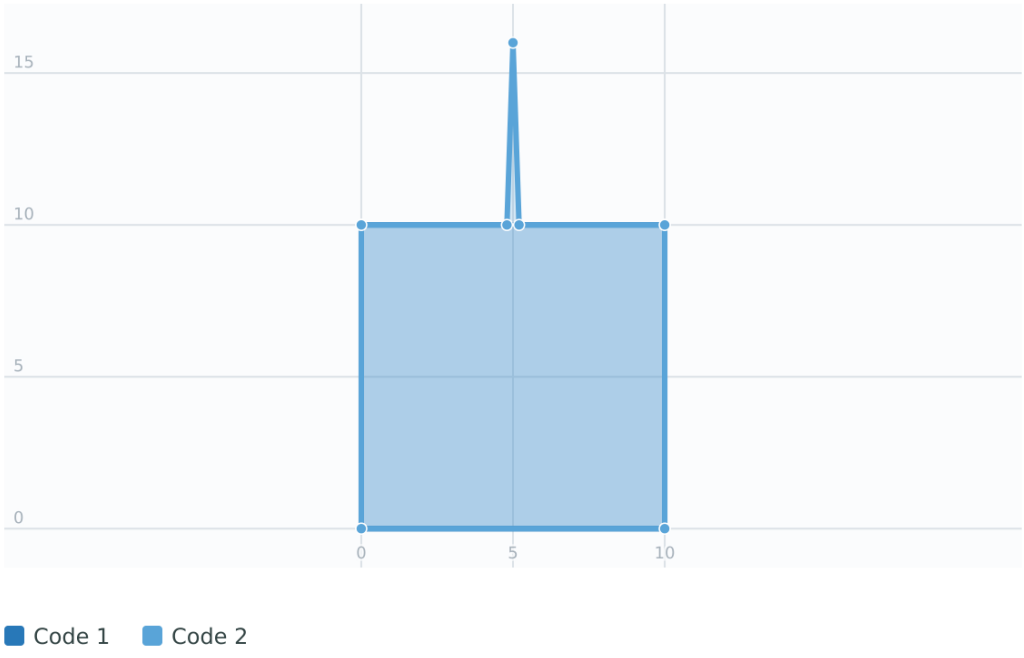
Code

```
WITH sample(id, geom) AS (
  VALUES
    ('plain', 'POLYGON((0 0,10 0,10 10,0 10,0 0))'::geometry),
    ('spike', 'POLYGON((0 0,10 0,10 10,5.2 10,5 16,4.8 10,0 10,0 0))'::geometry)
),
eroded AS (
  SELECT id, geom, ST_Buffer(geom, -0.5) AS eroded_geom
  FROM sample
),
scored AS (
  SELECT id,
         round((
           ST_Area(eroded_geom) / ST_Area(geom) /
           NULLIF(ST_Perimeter(eroded_geom) / ST_Perimeter(geom), 0)
         )::numeric, 3) AS erosion_index
  FROM eroded
  WHERE NOT ST_IsEmpty(eroded_geom)
)
SELECT id, erosion_index, erosion_index > 1.1 AS possible_spike
FROM scored
ORDER BY id;
```

Output

```
id    | erosion_index | possible_spike
-------+---------------+----------------
 plain |         0.900 | f
 spike |         1.147 | t
```

Figure



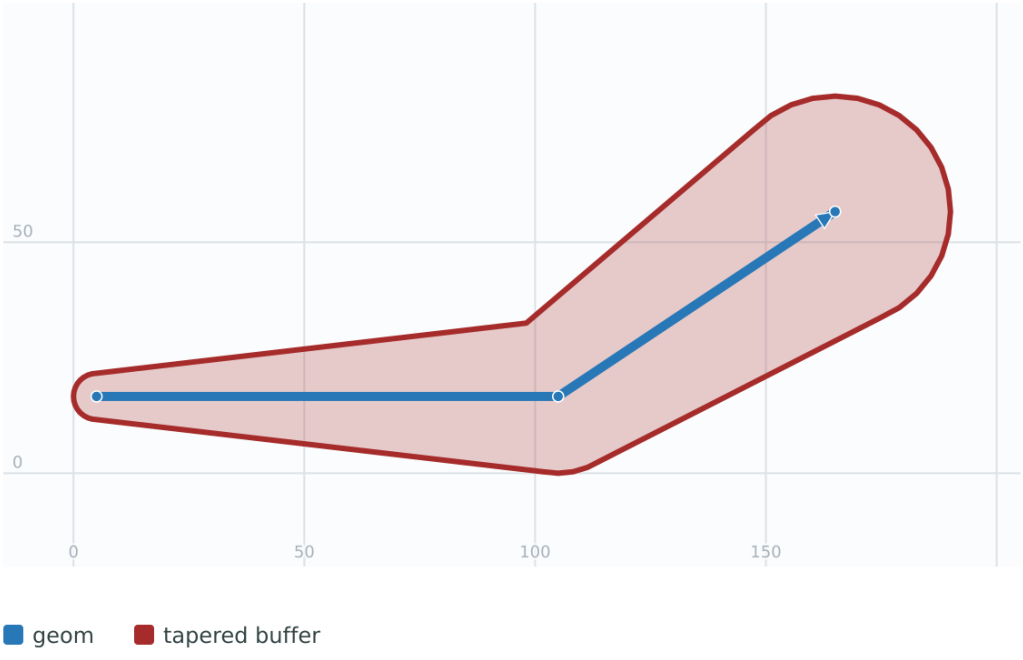
A variable-distance buffer can be approximated by buffering each vertex of a line by a distance interpolated along the line, building the convex hull for each consecutive pair of vertex buffers, and unioning the segment hulls. As with any geometry buffer, distances are in the units of the input SRS.

Taper a line buffer from 5 to 25 units along the line.

Code

```
WITH line AS (
  SELECT 'LINESTRING(0 0, 100 0, 160 40)'::geometry AS geom
),
vertices AS (
  SELECT (dump).path[1] AS n,
         (dump).geom AS geom,
         line.geom AS line_geom
  FROM line
  CROSS JOIN LATERAL ST_DumpPoints(line.geom) AS dump
),
radii AS (
  SELECT n,
         geom,
         5 + (25 - 5) * ST_LineLocatePoint(line_geom, geom) AS radius
  FROM vertices
),
segments AS (
  SELECT n,
         geom,
         radius,
         lead(geom) OVER (ORDER BY n) AS next_geom,
         lead(radius) OVER (ORDER BY n) AS next_radius
  FROM radii
)
SELECT ST_Union(
         ST_ConvexHull(
           ST_Collect(
             ST_Buffer(geom, radius),
             ST_Buffer(next_geom, next_radius)
           )
         )
       ) AS tapered_buffer
FROM segments
WHERE next_geom IS NOT NULL;
```

Figure



□□

ST_Collect, ST_ConvexHull, ST_DWithin, ST_DumpPoints, ST_LineLocatePoint, ST_SetSRID, ST_SimplifyPre
ST_Transform, ST_Union, ST_MakeValid

7.14.2 ST_BuildArea

ST_BuildArea — Creates a polygonal geometry formed by the linework of a geometry.

□□

geometry **ST_BuildArea**(geometry geom);

□□

Creates an areal geometry formed by the constituent linework of the input geometry. The input can be a LineString, MultiLineString, Polygon, MultiPolygon or a GeometryCollection. The result is a Polygon or MultiPolygon, depending on input. If the input linework does not form polygons, NULL is returned.

Unlike ST_MakePolygon, this function accepts rings formed by multiple lines, and can form any number of polygons.

This function converts inner rings into holes. To turn inner rings into polygons as well, use ST_Polygonize.



□□
Input linework must be correctly noded for this function to work properly. ST_Node can be used to node lines.
If the input linework crosses, this function will produce invalid polygons. ST_MakeValid can be used to ensure the output is valid.

1.1.0 □□□□□□□□□□□□.

□□

Building an area from five input lines.

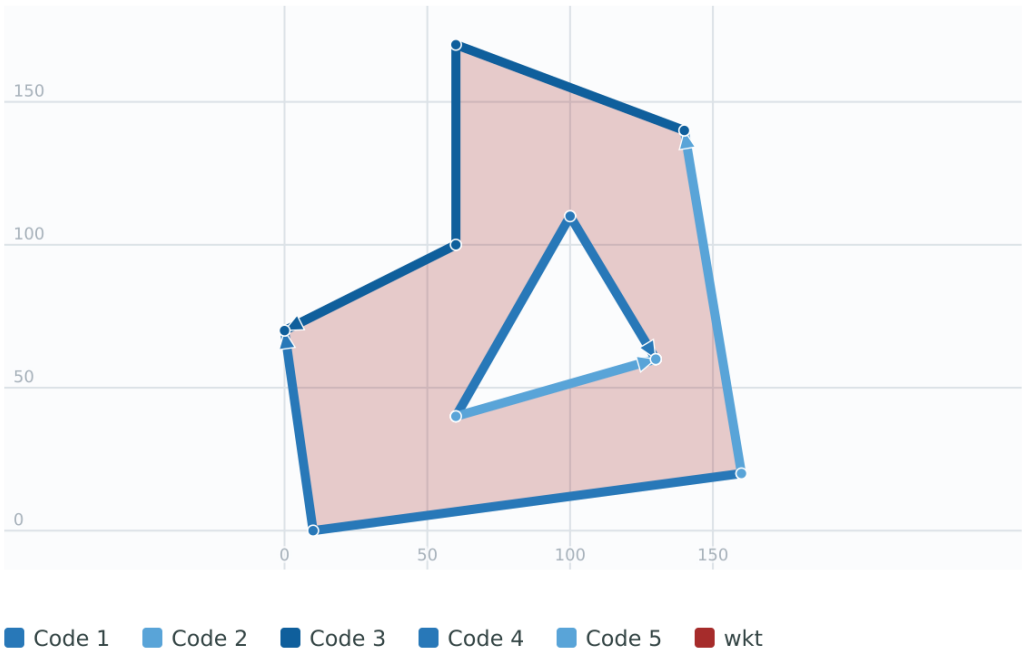
Code

```
WITH data(geom) AS (VALUES
   ('LINESTRING (180 40,30 20,20 90)'::geometry),
  ('LINESTRING (180 40,160 160)'::geometry),
  ('LINESTRING (160 160,80 190,80 120,20 90)'::geometry),
  ('LINESTRING (80 60,120 130,150 80)'::geometry),
  ('LINESTRING (80 60,150 80)'::geometry)
)
SELECT ST_BuildArea(ST_Collect(geom )) AS wkt
    FROM data;
```

Output

```
POLYGON((180 40,30 20,20 90,80 120,80 190,160 160,180 40),(150 80,120 130,80 60,150 80))
```

Figure

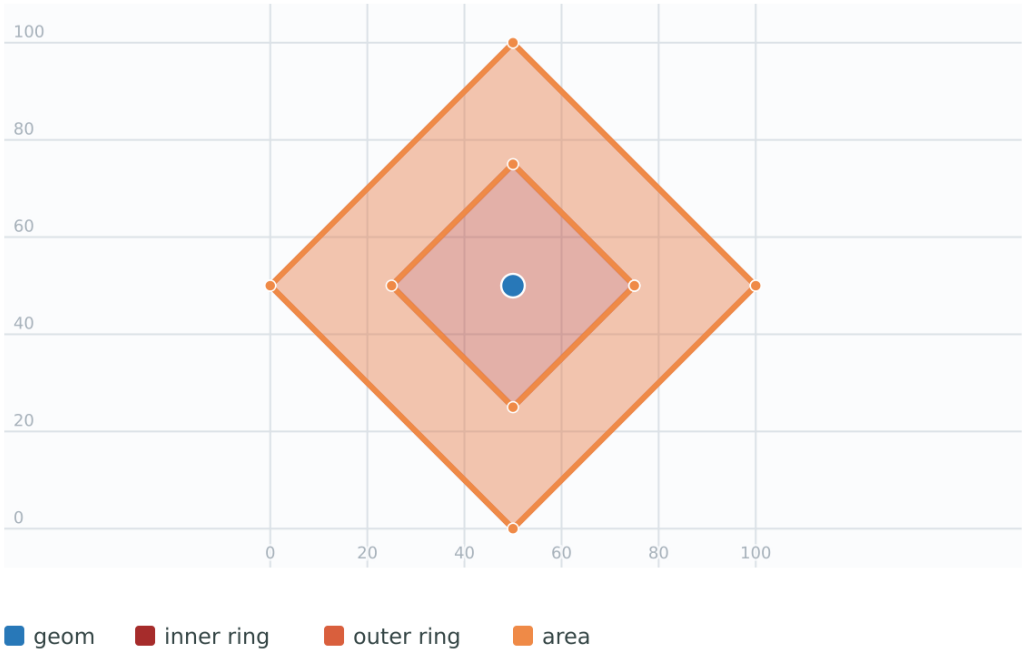


Create a donut from two circular polygons.

Code

```
WITH input AS (
  SELECT 'POINT(100 90)'::geometry AS geom
),
rings AS (
  SELECT ST_Buffer(geom, 25, 'quad_segs=1') AS inring,
         ST_Buffer(geom, 50, 'quad_segs=1') AS outring
  FROM input
)
SELECT inring AS inner_ring,
       outring AS outer_ring,
       ST_BuildArea(ST_Collect(inring, outring)) AS area
FROM rings;
```

Figure



Input linework that crosses at non-endpoint intersections must be noded before building areas:

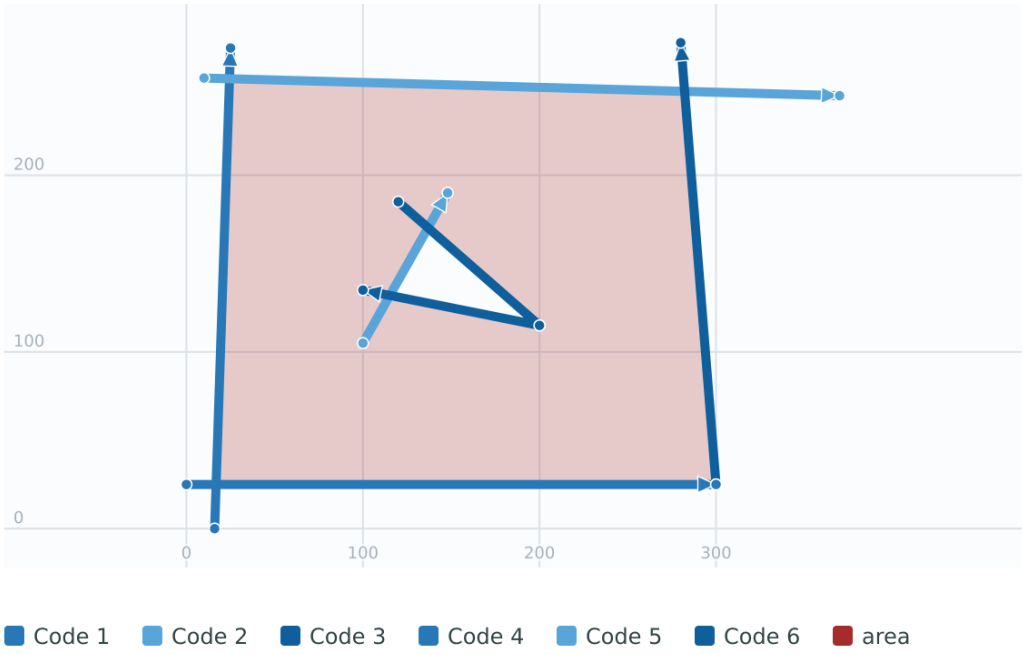
Code

```
WITH lines(geom) AS (VALUES
  ('LINESTRING(36 35,45 307)'::geometry),
  ('LINESTRING(30 290,390 280)'::geometry),
  ('LINESTRING(320 60,300 310)'::geometry),
  ('LINESTRING(20 60,320 60)'::geometry),
  ('LINESTRING(120 140,168 225)'::geometry),
  ('LINESTRING(140 220,220 150,120 170)'::geometry)
),
noded AS (
  SELECT ST_Node(ST_UnaryUnion(ST_Collect(geom))) AS geom FROM lines
)
SELECT ST_BuildArea(geom) AS area
FROM noded;
```

Output

```
POLYGON((320 60,36.8 60,44.4 289.6,302.2 282.4,320 60),(156.9 205.3,135.2 167,220 150,156.9 205.3))
```

Figure



☒☒

ST_Collect, ST_MakePolygon, ST_MakeValid, ST_Node, ST_Polygonize, ST_BdPolyFromText, ST_BdMPolyFr
(wrappers to this function with standard OGC interface)

7.14.3 ST_Centroid

ST_Centroid — ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.

☒☒

geometry **ST_Centroid**(geometry g1);
geography **ST_Centroid**(geography g1, boolean use_spheroid = true);

☒☒

Computes a point which is the geometric center of mass of a geometry. For `[MULTI]POINT`s, the centroid is the arithmetic mean of the input coordinates. For `[MULTI]LINESTRING`s, the centroid is computed using the weighted length of each line segment. For `[MULTI]POLYGON`s, the centroid is computed in terms of area. If an empty geometry is supplied, an empty `GEOMETRYCOLLECTION` is returned. If `NULL` is supplied, `NULL` is returned. If `CIRCULARSTRING` or `COMPOUNDCURVE` are supplied, they are converted to linestring with CurveToLine first, then same than for `LINESTRING`

For mixed-dimension input, the result is equal to the centroid of the component Geometries of highest dimension (since the lower-dimension geometries contribute zero ”weight” to the centroid).

Note that for polygonal geometries the centroid does not necessarily lie in the interior of the polygon. For example, see the diagram below of the centroid of a C-shaped polygon. To construct a point guaranteed to lie in the interior of a polygon use ST_PointOnSurface.

New in 2.3.0 : supports CIRCULARSTRING and COMPOUNDCURVE (using CurveToLine)

Availability: 2.4.0 support for geography was introduced.

This method implements the OGC Simple Features Implementation Specification for SQL 1.1.

This method implements the SQL/MM specification. SQL-MM 3: 8.1.4, 9.5.5

□□

Centroid of a MultiPoint.

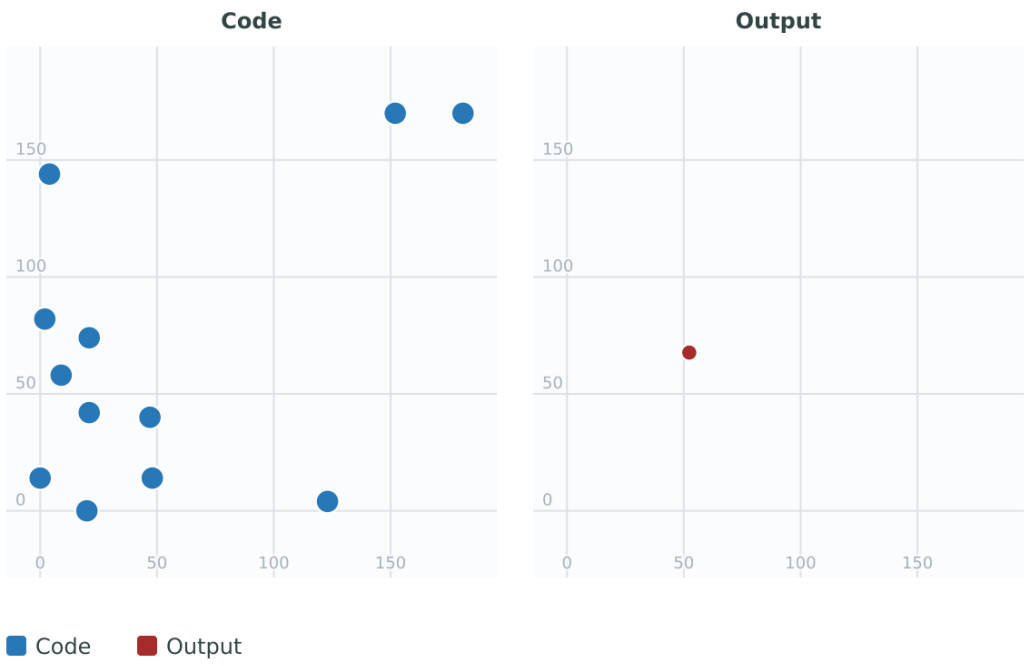
Code

```
SELECT ST_Centroid(
  'MULTIPOINT (8 24,10 92,12 154,17 68,28 10,29 52,29 84,55 50,56 24,131 14,160 180,189 ↩
    180)');
```

Output

```
POINT(60.333 77.667)
```

Figure



Centroid of a LineString.

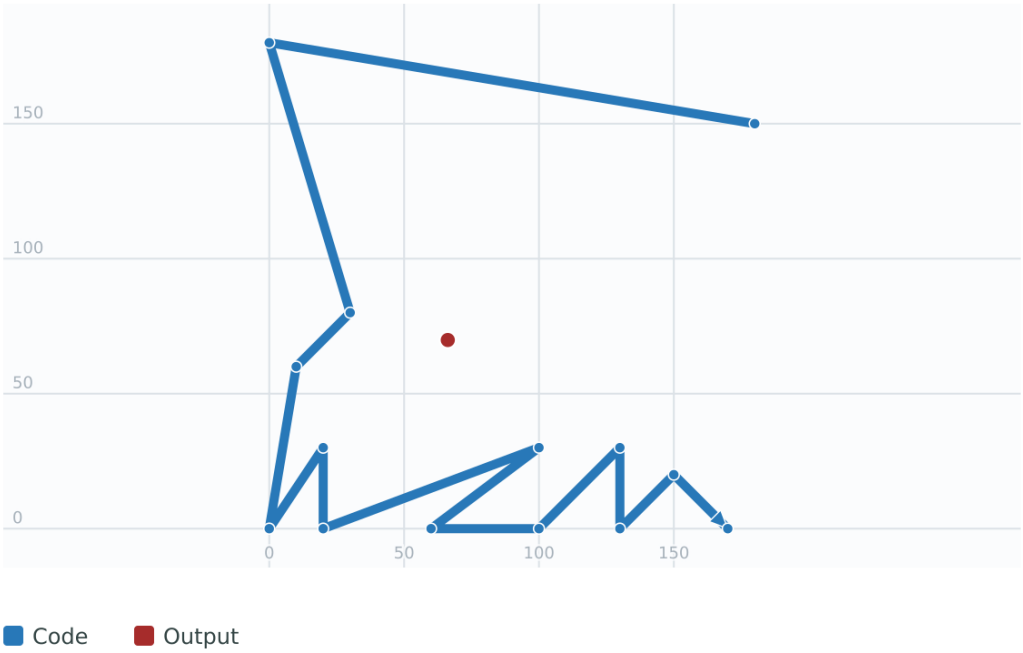
Code

```
SELECT ST_Centroid(
  'LINESTRING (190 160,10 190,40 90,20 70,10 10,30 40,30 10,110 40,70 10,110 10,140 40,140 ↩
    10,160 30,180 10)');
```

Output

```
POINT(76.191 79.876)
```

Figure



Centroid of a Polygon.

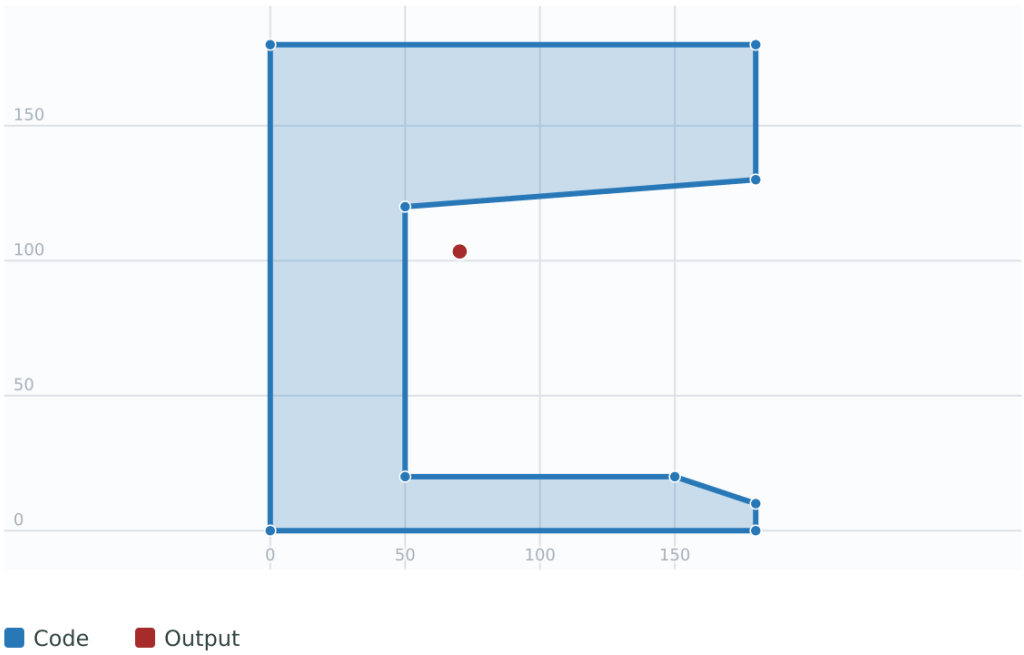
Code

```
SELECT ST_Centroid(
  'POLYGON ((190 190,10 190,10 10,190 10,190 20,160 30,60 30,60 130,190 140,190 190))');
```

Output

```
POINT(80.251 113.405)
```

Figure



Centroid of the highest-dimension components of a GeometryCollection.

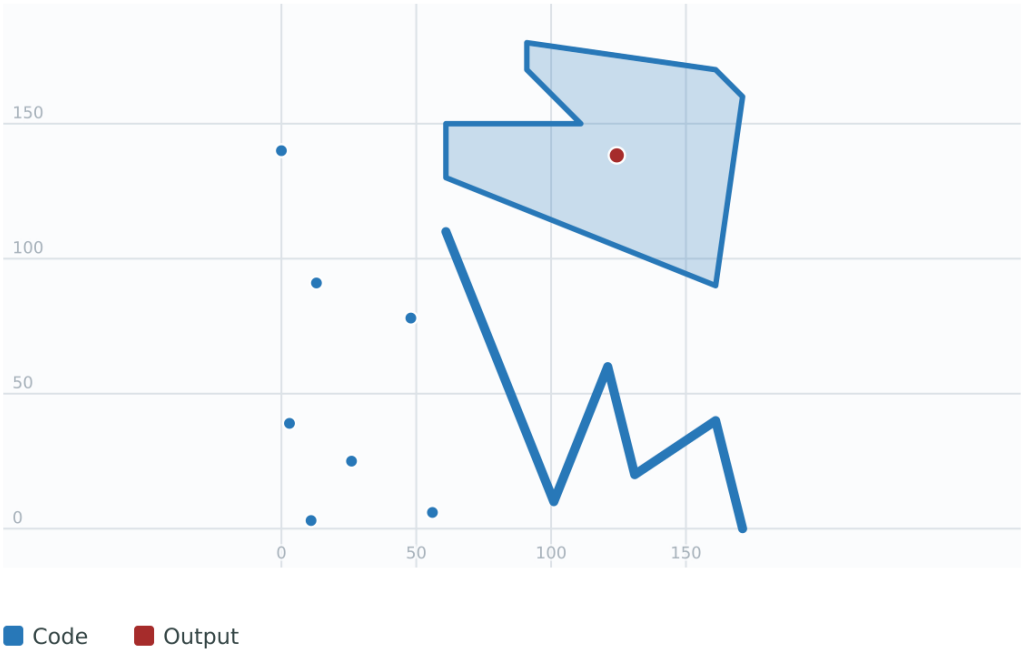
Code

```
SELECT ST_Centroid(
  'GEOMETRYCOLLECTION (POLYGON ((190 170,180 100,80 140,80 160,130 160,110 180,110 190,180 ↩
   180,190 170)),
    LINESTRING (80 120,120 20,140 70,150 30,180 50,190 10),
    MULTIPOINT (19 150,22 49,30 13,32 101,45 35,67 88,75 16))');
```

Output

```
POINT(143.361 148.263)
```

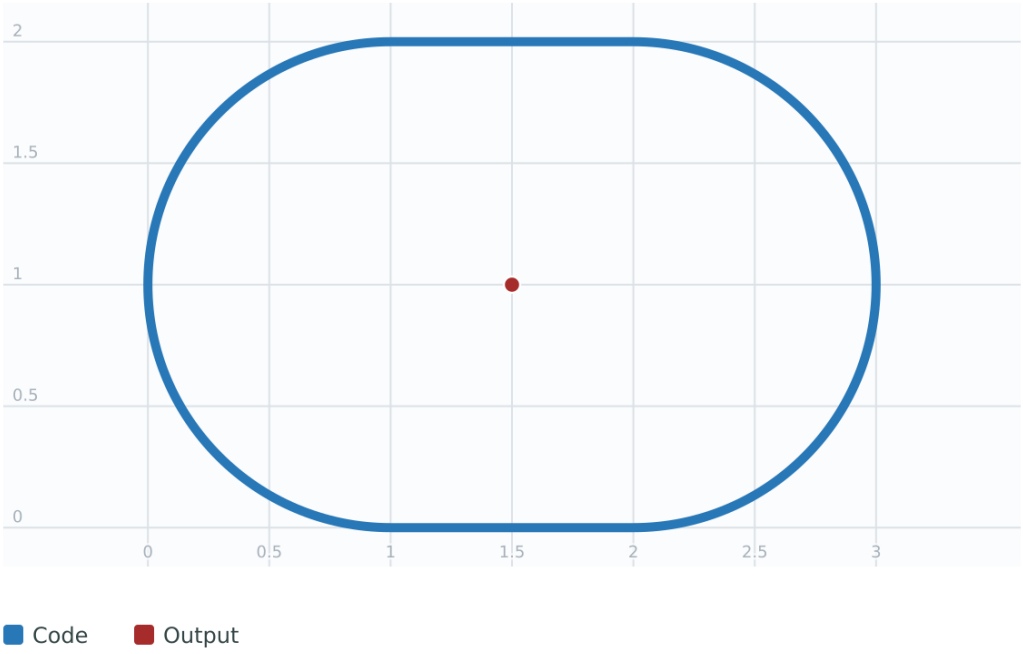
Figure



Code

```
SELECT ST_Centroid(g)
FROM  ST_GeomFromText('CIRCULARSTRING(0 2,-1 1,0 0,0.5 0,1 0,2 1,1 2,0.5 2,0 2)')  AS g ;
```

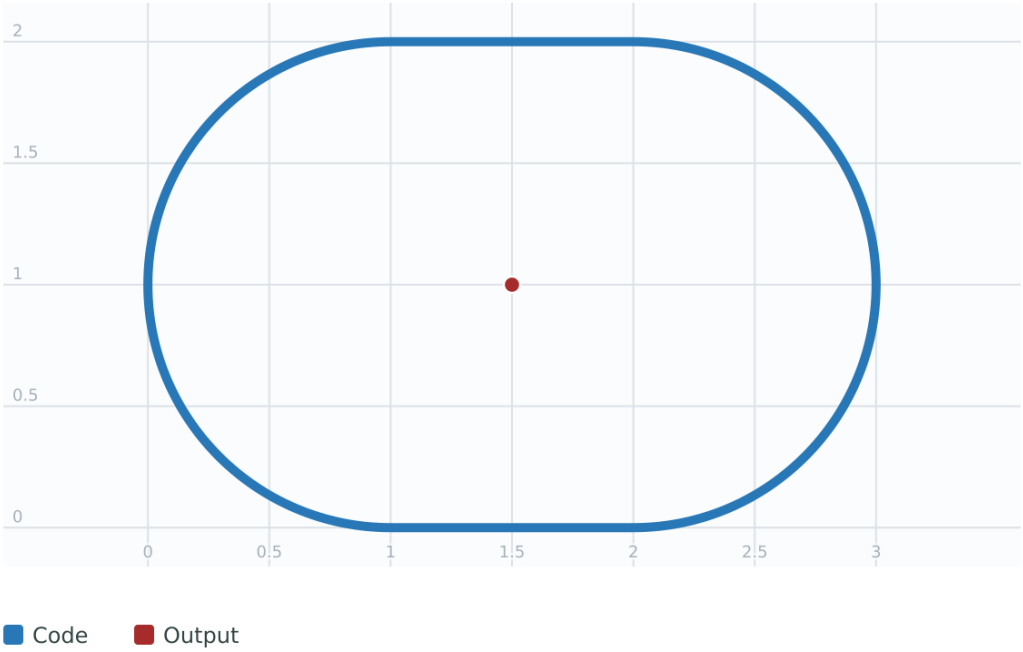
Figure



Code

```
SELECT ST_Centroid(g)
FROM  ST_GeomFromText('COMPOUNDCURVE(CIRCULARSTRING(0 2,-1 1,0 0),(0 0,0.5 0,1 0),  ↩
    CIRCULARSTRING( 1 0,2 1,1 2),(1 2,0.5 2,0 2))' ) AS g;
```

Figure



Centroids of the parts of a MultiPolygon.

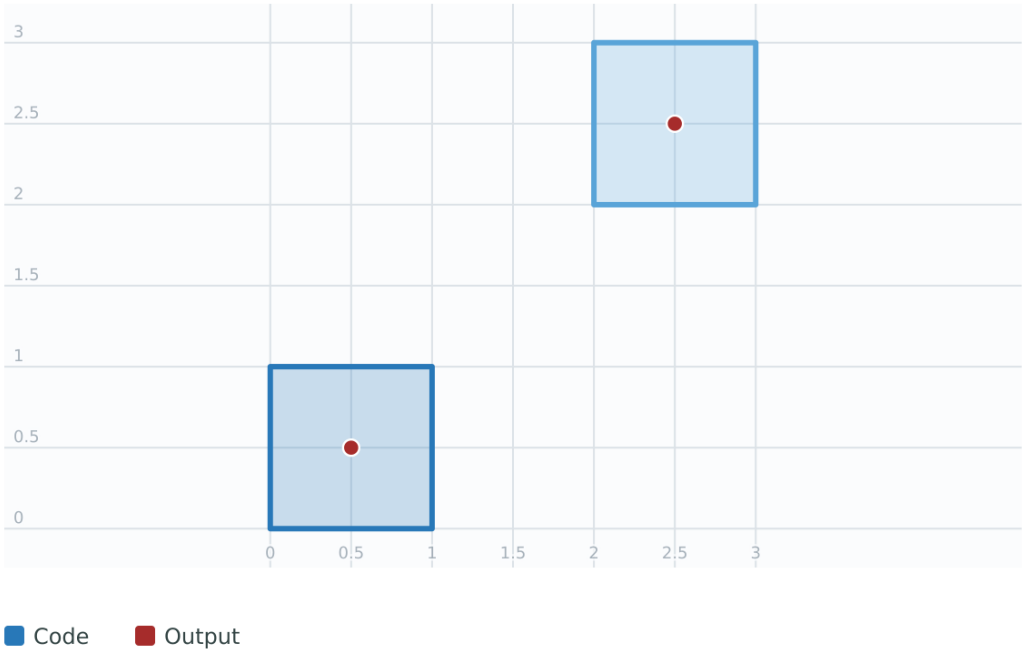
Code

```
SELECT ST_Collect(ST_Centroid(geom) ORDER BY path)
FROM ST_Dump('MULTIPOLYGON (
    ((0 0,0 1,1 1,1 0,0 0)),
    ((2 2,2 3,3 3,3 2,2 2))
  )'::geometry);
```

Output

```
MULTIPOINT((0.5 0.5),(2.5 2.5))
```

Figure



□□

ST_Collect, ST_Dump, ST_PointOnSurface, ST_GeometricMedian

7.14.4 ST_ChaikinSmoothing

ST_ChaikinSmoothing — Returns a smoothed version of a geometry, using the Chaikin algorithm

□□

geometry **ST_ChaikinSmoothing**(geometry geom, integer nIterations = 1, boolean preserveEndPoints = false);

□□

Smoothes a linear or polygonal geometry using Chaikin's algorithm. The degree of smoothing is controlled by the `nIterations` parameter. On each iteration, each interior vertex is replaced by two vertices located at 1/4 of the length of the line segments before and after the vertex. A reasonable degree of smoothing is provided by 3 iterations; the maximum is limited to 5.

If `preserveEndPoints` is true, the endpoints of Polygon rings are not smoothed. The endpoints of LineStrings are always preserved.



□□
The number of vertices doubles with each iteration, so the result geometry may have many more points than the input. To reduce the number of points use a simplification function on the result (see ST_Simplify, ST_SimplifyPreserveTopology and ST_SimplifyVW).

The result has interpolated values for the Z and M dimensions when present.

This function supports 3d and will not drop the z-index.

Availability: 2.5.0

□□

Smoothing a Polygon using 1, 2 and 3 iterations.

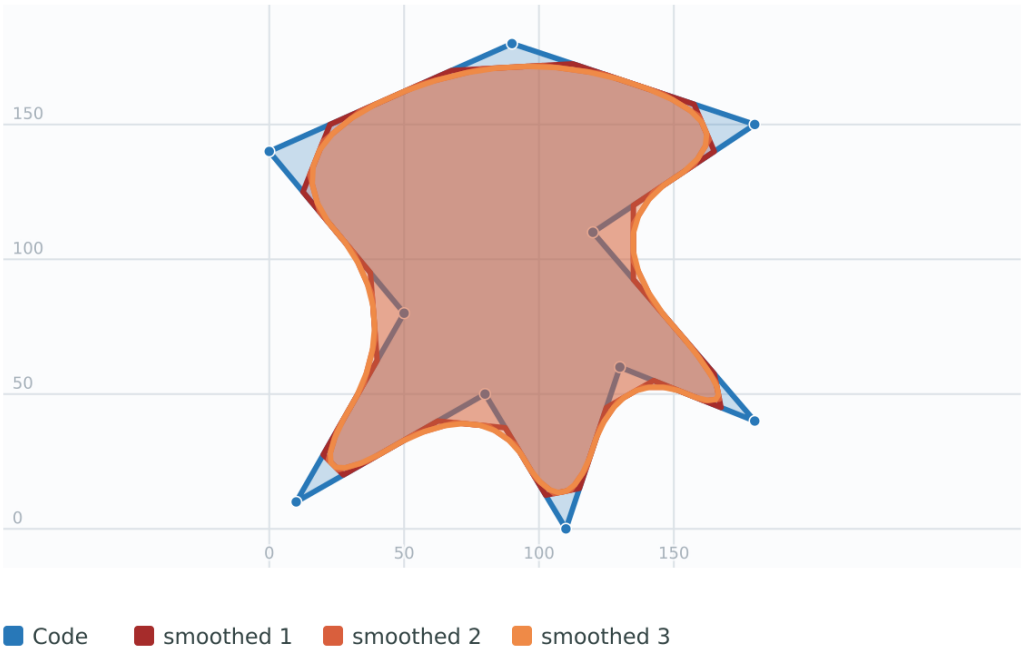
Code

```
SELECT n AS iterations, ST_ChaikinSmoothing(
    'POLYGON ((20 20,60 90,10 150,100 190,190 160,130 120,190 50,140 70,120 10,90 60,20 20)
    )', n
  ) AS smoothed
FROM generate_series(1, 3) AS n;
```

Output

```
1 | POLYGON((30 37.5,50 72.5,47.5 105,22.5 135,32.5 160,77.5 180,122.5 182.5,167.5 167.5,175 150,145 130,145 102.5,175
     67.5,177.5 55,152.5 65,135 55,125 25,112.5 22.5,97.5 47.5,72.5 50,37.5 30,30 37.5))
2 | POLYGON((35 46.25,45 63.75,49.375 80.625,48.125 96.875,41.25 112.5,28.75 127.5,25 141.25,30 153.75,43.75 165,66.25
     175,88.75 180.625,111.25 181.875,133.75 178.75,156.25 171.25,169.375 163.125,173.125 154.375,167.5 145,152.5
    135,145 123.125,145 109.375,152.5 93.75,167.5 76.25,175.625 64.375,176.875 58.125,171.25 57.5,158.75 62.5,148.125
     62.5,139.375 57.5,132.5 47.5,127.5 32.5,121.875 24.375,115.625 23.125,108.75 28.75,101.25 41.25,91.25
    48.125,78.75 49.375,63.75 45,46.25 35,35.625 31.875,31.875 35.625,35 46.25))
3 | POLYGON((37.5 50.625,42.5 59.375,46.09375 67.96875,48.28125 76.40625,49.0625 84.6875,48.4375 92.8125,46.40625
    100.78125,42.96875 108.59375,38.125 116.25,31.875 123.75,27.8125 130.9375,25.9375 137.8125,26.25 144.375,28.75
    150.625,33.4375 156.5625,40.3125 162.1875,49.375 167.5,60.625 172.5,71.875 176.40625,83.125 179.21875,94.375
    180.9375,105.625 181.5625,116.875 181.09375,128.125 179.53125,139.375 176.875,150.625 173.125,159.53125
    169.21875,166.09375 165.15625,170.3125 160.9375,172.1875 156.5625,171.71875 152.03125,168.90625 147.34375,163.75
    142.5,156.25 137.5,150.625 132.03125,146.875 126.09375,145 119.6875,145 112.8125,146.875 105.46875,150.625
    97.65625,156.25 89.375,163.75 80.625,169.53125 73.28125,173.59375 67.34375,175.9375 62.8125,176.5625
    59.6875,175.46875 57.96875,172.65625 57.65625,168.125 58.75,161.875 61.25,156.09375 62.5,150.78125 62.5,145.9375
    61.25,141.5625 58.75,137.65625 55,134.21875 50,131.25 43.75,128.75 36.25,126.09375 30.46875,123.28125
    26.40625,120.3125 24.0625,117.1875 23.4375,113.90625 24.53125,110.46875 27.34375,106.875 31.875,103.125
    38.125,98.75 42.96875,93.75 46.40625,88.125 48.4375,81.875 49.0625,75 48.28125,67.5 46.09375,59.375 42.5,50.625
    37.5,43.59375 34.21875,38.28125 32.65625,34.6875 32.8125,32.8125 34.6875,32.65625 38.28125,34.21875 43.59375,37.5
     50.625))
```

Figure



Smoothing a LineString using 1, 2 and 3 iterations.

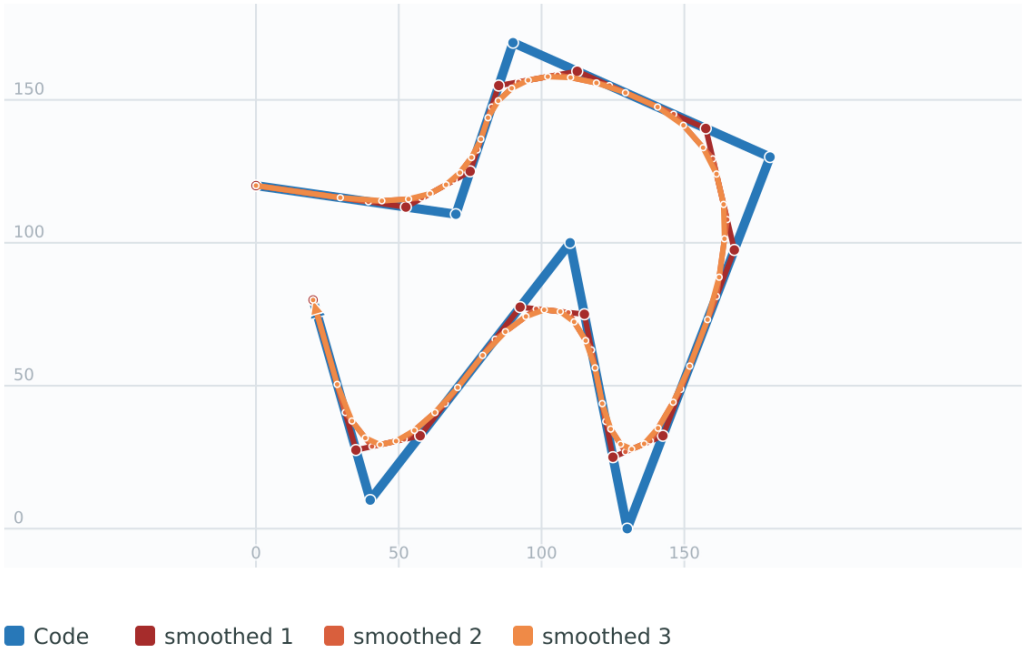
Code

```
SELECT n AS iterations, ST_ChaikinSmoothing(
    'LINESTRING (10 140,80 130,100 190,190 150,140 20,120 120,50 30,30 100)', n
  ) AS smoothed
FROM generate_series(1, 3) AS n;
```

Output

```
1 | LINESTRING(10 140,62.5 132.5,85 145,95 175,122.5 180,167.5 160,177.5 117.5,152.5 52.5,135 45,125 95,102.5 ↩
    97.5,67.5 52.5,45 47.5,30 100)
2 | LINESTRING(10 140,49.375 134.375,68.125 135.625,79.375 141.875,87.5 152.5,92.5 167.5,101.875 176.25,115.625 ↩
    178.75,133.75 175,156.25 165,170 149.375,175 128.125,171.25 101.25,158.75 68.75,148.125 50.625,139.375 ↩
    46.875,132.5 57.5,127.5 82.5,119.375 95.625,108.125 96.875,93.75 86.25,76.25 63.75,61.875 51.25,50.625 ↩
    48.75,41.25 60.625,30 100)
3 | LINESTRING(10 140,39.53125 135.78125,54.0625 134.6875,63.4375 135.3125,70.9375 137.1875,76.5625 140.3125,81.40625 ↩
    144.53125,85.46875 149.84375,88.75 156.25,91.25 163.75,94.84375 169.6875,99.53125 174.0625,105.3125 ↩
    176.875,112.1875 178.125,120.15625 177.8125,129.21875 175.9375,139.375 172.5,150.625 167.5,159.6875 ↩
    161.09375,166.5625 153.28125,171.25 144.0625,173.75 133.4375,174.0625 121.40625,172.1875 107.96875,168.125 ↩
    93.125,161.875 76.875,156.09375 64.21875,150.78125 55.15625,145.9375 49.6875,141.5625 47.8125,137.65625 ↩
    49.53125,134.21875 54.84375,131.25 63.75,128.75 76.25,125.46875 85.78125,121.40625 92.34375,116.5625 ↩
    95.9375,110.9375 96.5625,104.53125 94.21875,97.34375 88.90625,89.375 80.625,80.625 69.375,72.65625 ↩
    60.625,65.46875 54.375,59.0625 50.625,53.4375 49.375,48.28125 51.71875,43.59375 57.65625,38.4375 70.46875,30 100)
```

Figure



Smoothing a triangle shows the weakest visual change, so it is kept last.

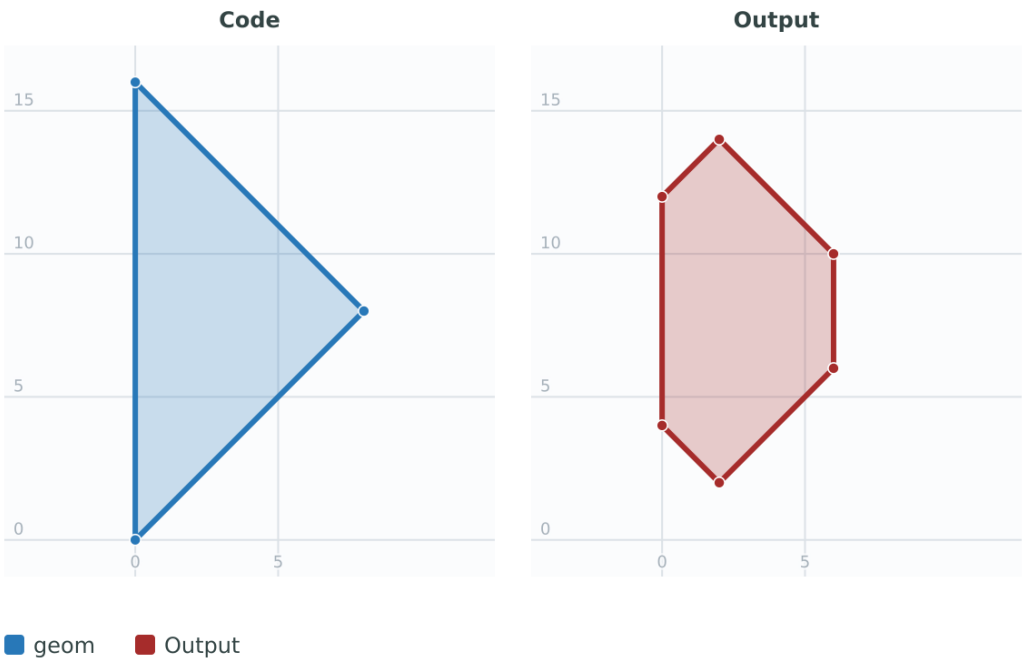
Code

```
SELECT ST_ChaikinSmoothing(geom) smoothed
FROM (SELECT  'POLYGON((0 0,8 8,0 16,0 0))'::geometry geom) AS foo;
```

Output

```
POLYGON((2 2,6 6,6 10,2 14,0 12,0 4,2 2))
```

Figure



☒☒

ST_Simplify, ST_SimplifyPreserveTopology, ST_SimplifyVW

7.14.5 ST_CatmullRomSmoothing

ST_CatmullRomSmoothing — Returns a smoothed version of a geometry, using the Catmull-Rom spline algorithm

☒☒

geometry **ST_CatmullRomSmoothing**(geometry geom, integer nSegments = 5);

☒☒

Smoothes a linear or polygonal geometry using the Catmull-Rom spline algorithm. Unlike ST_ChaikinSmooth this is an *interpolating* spline: the output curve passes through every original vertex. Between each pair of consecutive original vertices, `nSegments` - 1 new intermediate points are inserted.

The `nSegments` parameter controls the density of the output. With `nSegments` = 5 (the default), each original span is divided into 5 sub-segments (inserting 4 new points per span). The minimum value is 2.

At least 4 input vertices are required to apply smoothing; geometries with fewer vertices are returned unchanged. Points and multipoints are always returned unchanged.



□□

The output vertex count grows as 1 + (N-1) * nSegments for open lines, and 1 + N * nSegments for closed rings, where N is the number of input vertices. For large geometries, use a simplification function on the result to reduce the number of points (see ST_Simplify, ST_SimplifyPreserveTopology and ST_SimplifyVW).

The result has interpolated values for the Z and M dimensions when present.

 This function supports 3d and will not drop the z-index.

Availability: 3.7.0

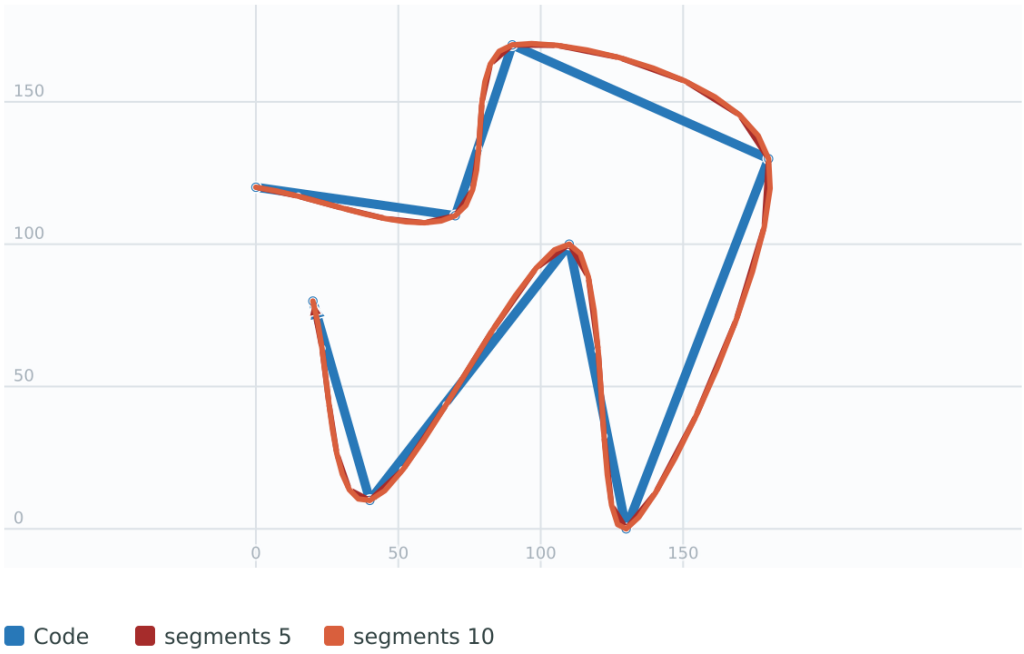
□□

Smoothing a LineString using nSegments = 5 and 10.

Code

```
WITH input(geom) AS (VALUES
  ('LINESTRING (10 140,80 130,100 190,190 150,140 20,120 120,50 30,30 100)'::geometry)
)
SELECT ST_CatmullRomSmoothing(geom, 5) AS segments_5,
  ST_CatmullRomSmoothing(geom, 10) AS segments_10
FROM input;
```

Figure



Smoothing a 4-point collinear line with default nSegments=5 shows the minimal change case.

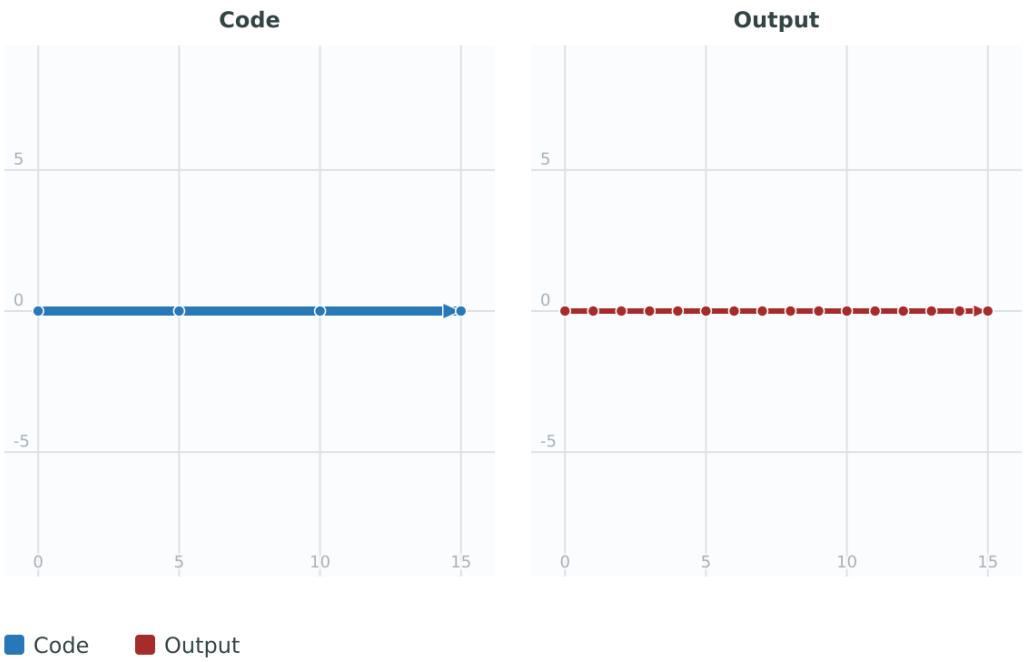
Code

```
SELECT ST_CatmullRomSmoothing('LINESTRING(0 0,5 0,10 0,15 0)');
```

Output

```
LINESTRING(0 0,1 0,2 0,3 0,4 0,5 0,6 0,7 0,8 0,9 0,10 0,11 0,12 0,13 0,14 0,15 0)
```

Figure

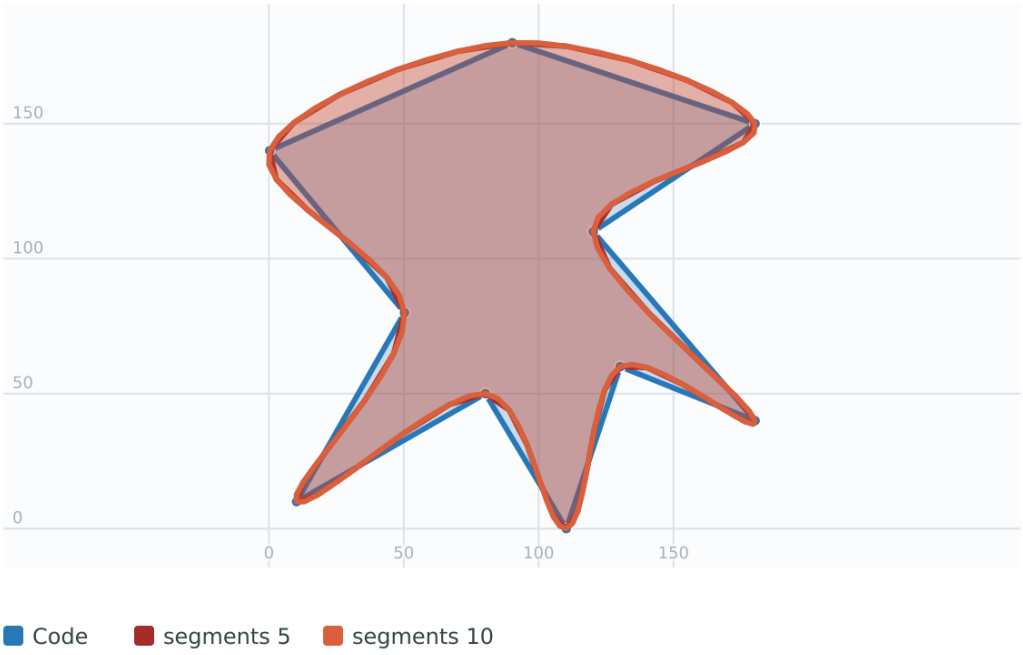


Smoothing a Polygon using nSegments = 5 and 10. This larger example is kept last.

Code

```
WITH input(geom) AS (VALUES
  ('POLYGON ((20 20,60 90,10 150,100 190,190 160,130 120,190 50,140 70,120 10,90 60,20 20))
    '::geometry)
)
SELECT ST_CatmullRomSmoothing(geom, 5) AS segments_5,
  ST_CatmullRomSmoothing(geom, 10) AS segments_10
FROM input;
```

Figure



□□

ST_ChaikinSmoothing, ST_Simplify, ST_SimplifyPreserveTopology, ST_SimplifyVW

7.14.6 ST_ConcaveHull

ST_ConcaveHull — Computes a possibly concave geometry that contains all input geometry vertices

□□

geometry **ST_ConcaveHull**(geometry param_geom, float param_pctconvex, boolean param_allow_holes = false);

□□

A concave hull is a (usually) concave geometry which contains the input, and whose vertices are a subset of the input vertices. In the general case the concave hull is a Polygon. The concave hull of two or more collinear points is a two-point LineString. The concave hull of one or more identical points is a Point. The polygon will not contain holes unless the optional `param_allow_holes` argument is specified as true.

One can think of a concave hull as ”shrink-wrapping” a set of points. This is different to the convex hull, which is more like wrapping a rubber band around the points. A concave hull generally has a smaller area and represents a more natural boundary for the input points.

The `param_pctconvex` controls the concaveness of the computed hull. A value of 1 produces the convex hull. Values between 1 and 0 produce hulls of increasing concaveness. A value of 0 produces a hull with maximum concaveness (but still a single polygon). Choosing a suitable value depends on the nature of the input data, but often values between 0.3 and 0.1 produce reasonable results.



□□
Technically, the param_pctconvex determines a length as a fraction of the difference between the longest and shortest edges in the Delaunay Triangulation of the input points. Edges longer than this length are ”eroded” from the triangulation. The triangles remaining form the concave hull.

For point and linear inputs, the hull will enclose all the points of the inputs. For polygonal inputs, the hull will enclose all the points of the input *and also* all the areas covered by the input. If you want a point-wise hull of a polygonal input, convert it to points first using ST_Points.

This is not an aggregate function. To compute the concave hull of a set of geometries use ST_Collect (e.g. `ST_ConcaveHull(ST_Collect(geom ), 0.80)`.

2.0.0 □□□□□□□□□□□□.

Enhanced: 3.3.0, GEOS native implementation enabled for GEOS 3.11+

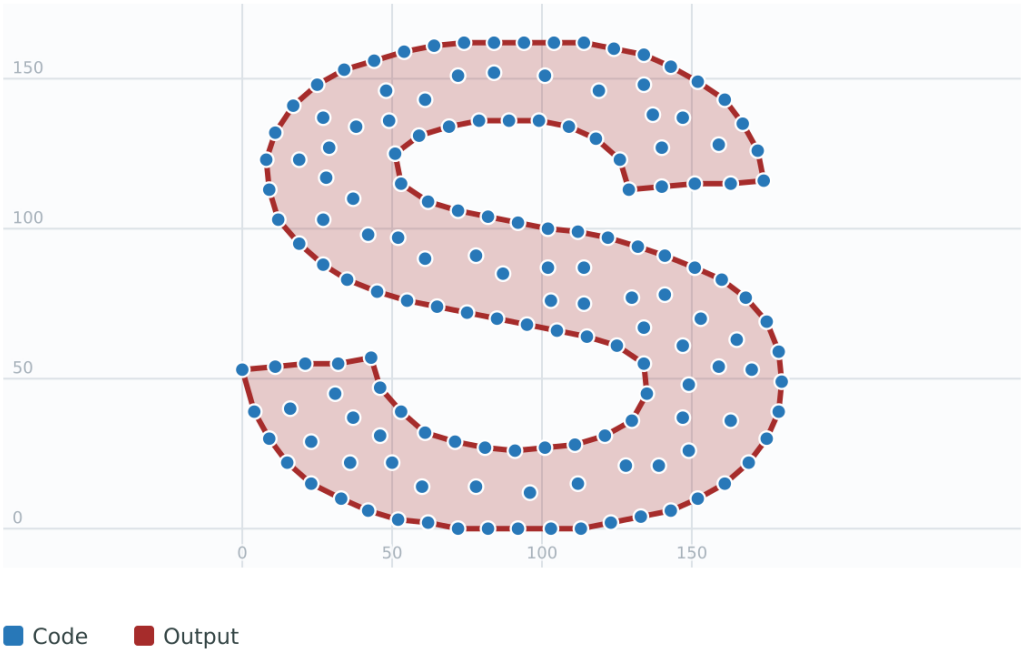
□□

Concave hull of a MultiPoint.

Code

```
SELECT ST_ConcaveHull('MULTIPOINT ((10 72),(53 76),(56 66),(63 58),(71 51),(81 48),(91 46) ↩
    ,(101 45),(111 46),(121 47),(131 50),(140 55),(145 64),(144 74),(135 80),(125 83),(115  ↩
    85),(105 87),(95 89),(85 91),(75 93),(65 95),(55 98),(45 102),(37 107),(29 114),(22 122) ↩
    ,(19 132),(18 142),(21 151),(27 160),(35 167),(44 172),(54 175),(64 178),(74 180),(84  ↩
    181),(94 181),(104 181),(114 181),(124 181),(134 179),(144 177),(153 173),(162 168),(171 ↩
     162),(177 154),(182 145),(184 135),(139 132),(136 142),(128 149),(119 153),(109 155) ↩
    ,(99 155),(89 155),(79 153),(69 150),(61 144),(63 134),(72 128),(82 125),(92 123),(102  ↩
    121),(112 119),(122 118),(132 116),(142 113),(151 110),(161 106),(170 102),(178 96),(185 ↩
     88),(189 78),(190 68),(189 58),(185 49),(179 41),(171 34),(162 29),(153 25),(143 23) ↩
    ,(133 21),(123 19),(113 19),(102 19),(92 19),(82 19),(72 21),(62 22),(52 25),(43 29),(33 ↩
     34),(25 41),(19 49),(14 58),(21 73),(31 74),(42 74),(173 134),(161 134),(150 133),(97  ↩
    104),(52 117),(157 156),(94 171),(112 106),(169 73),(58 165),(149 40),(70 33),(147 157) ↩
    ,(48 153),(140 96),(47 129),(173 55),(144 86),(159 67),(150 146),(38 136),(111 170),(124 ↩
     94),(26 59),(60 41),(71 162),(41 64),(88 110),(122 34),(151 97),(157 56),(39 146),(88  ↩
    33),(159 45),(47 56),(138 40),(129 165),(33 48),(106 31),(169 147),(37 122),(71 109) ↩
    ,(163 89),(37 156),(82 170),(180 72),(29 142),(46 41),(59 155),(124 106),(157 80),(175  ↩
    82),(56 50),(62 116),(113 95),(144 167))',
         0.1 );
```

Figure

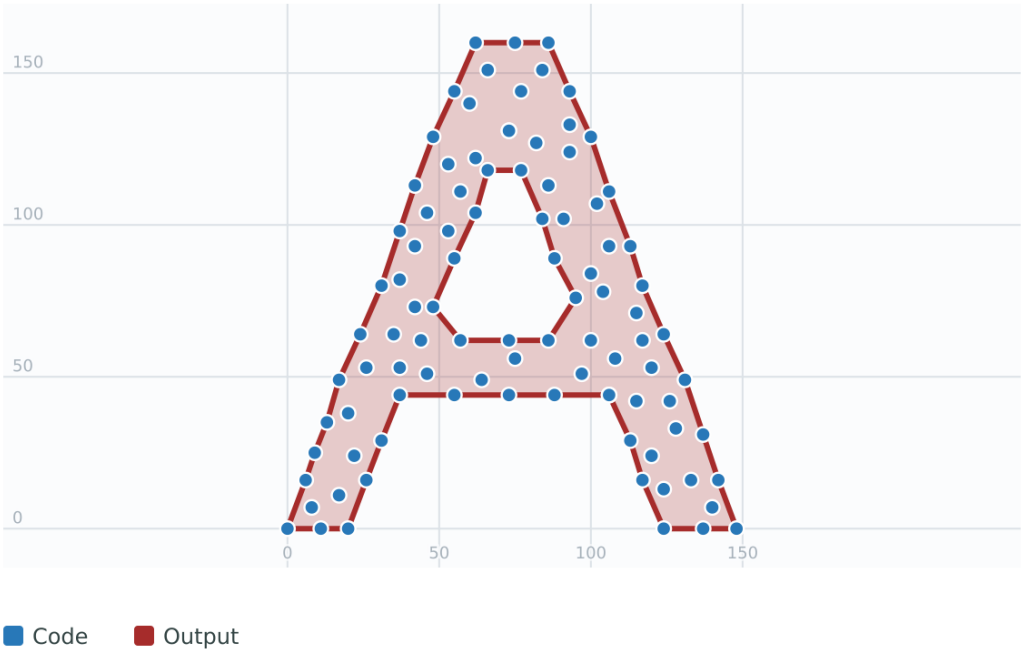


Concave hull of a MultiPoint, allowing holes.

Code

```
SELECT ST_ConcaveHull('MULTIPOINT ((132 64),(114 64),(99 64),(81 64),(63 64),(57 49),(52
    36),(46 20),(37 20),(26 20),(32 36),(39 55),(43 69),(50 84),(57 100),(63 118),(68 133)
    ,(74 149),(81 164),(88 180),(101 180),(112 180),(119 164),(126 149),(132 131),(139 113)
    ,(143 100),(150 84),(157 69),(163 51),(168 36),(174 20),(163 20),(150 20),(143 36),(139
    49),(132 64),(99 151),(92 138),(88 124),(81 109),(74 93),(70 82),(83 82),(99 82),(112
    82),(126 82),(121 96),(114 109),(110 122),(103 138),(99 151),(34 27),(43 31),(48 44),(46
     58),(52 73),(63 73),(61 84),(72 71),(90 69),(101 76),(123 71),(141 62),(166 27),(150
    33),(159 36),(146 44),(154 53),(152 62),(146 73),(134 76),(143 82),(141 91),(130 98)
    ,(126 104),(132 113),(128 127),(117 122),(112 133),(119 144),(108 147),(119 153),(110
    171),(103 164),(92 171),(86 160),(88 142),(79 140),(72 124),(83 131),(79 118),(68 113)
    ,(63 102),(68 93),(35 45))',
         0.15, true );
```

Figure



Comparing a concave hull of a Polygon to the concave hull of the constituent points. The hull respects the boundary of the polygon, whereas the points-based hull does not.

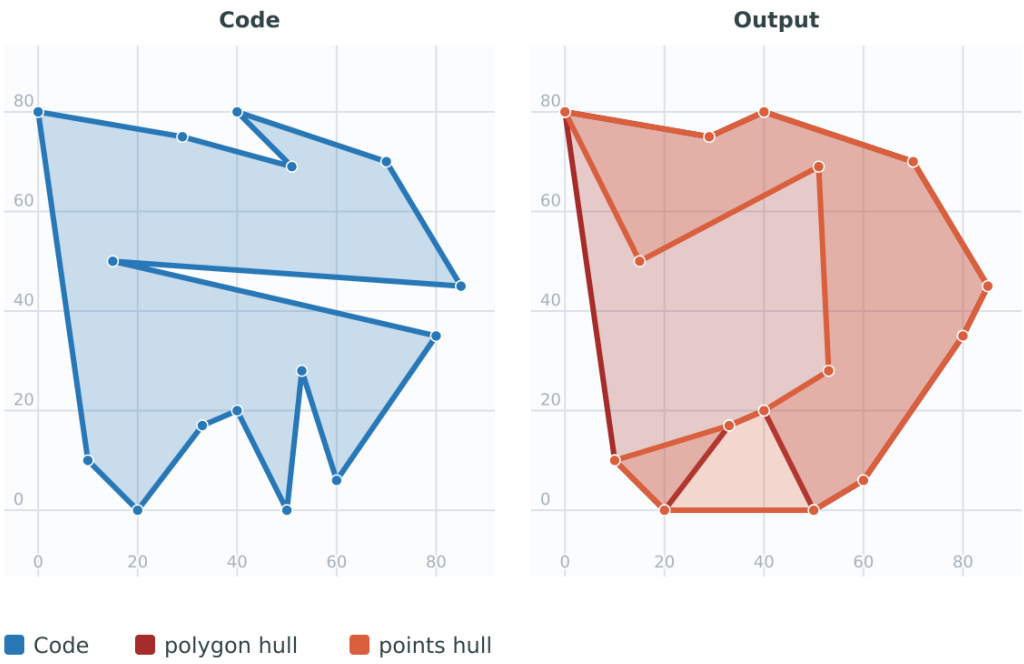
Code

```
WITH data(geom) AS (VALUES
   ('POLYGON ((10 90,39 85,61 79,50 90,80 80,95 55,25 60,90 45,70 16,63 38,60 10,50 30,43 ↩
    27,30 10,20 20,10 90))'::geometry)
)
SELECT ST_ConcaveHull(geom, 0.1) AS polygon_hull,
       ST_ConcaveHull(ST_Points(geom), 0.1) AS points_hull
    FROM data;
```

Output

```
POLYGON((60 10,50 30,43 27,30 10,20 20,10 90,39 85,50 90,80 80,95 55,90 45,70 16,60 10)) | POLYGON((25 60,10 90,39 ↩
    85,50 90,80 80,95 55,90 45,70 16,60 10,30 10,20 20,43 27,50 30,63 38,61 79,25 60))
```

Figure



Using with ST_Collect to compute the concave hull of a geometry set.

This example estimates an infected area from point observations.

Code

```
SELECT disease_type,
    ST_ConcaveHull(ST_Collect(obs_pnt), 0.3 ) AS geom
  FROM disease_obs
  GROUP BY disease_type;
```

☒☒

ST_ConvexHull, ST_Collect, CG_AlphaShape, CG_OptimalAlphaShape

7.14.7 ST_ConvexHull

ST_ConvexHull — Computes the convex hull of a geometry.

☒☒

geometry **ST_ConvexHull**(geometry geomA);

☒☒

Computes the convex hull of a geometry. The convex hull is the smallest convex geometry that encloses all geometries in the input.

One can think of the convex hull as the geometry obtained by wrapping an rubber band around a set of geometries. This is different from a concave hull which is analogous to ”shrink-wrapping”

the geometries. A convex hull is often used to determine an affected area based on a set of point observations.

In the general case the convex hull is a Polygon. The convex hull of two or more collinear points is a two-point LineString. The convex hull of one or more identical points is a Point.

This is not an aggregate function. To compute the convex hull of a set of geometries, use ST_Collect to aggregate them into a geometry collection (e.g. `ST_ConvexHull(ST_Collect(geom))`.

GEOS □□□□□

✅ This method implements the OGC Simple Features Implementation Specification for SQL 1.1. s2.1.1.3

✅ This method implements the SQL/MM specification. SQL-MM IEC 13249-3: 5.1.16

✅ This function supports 3d and will not drop the z-index.

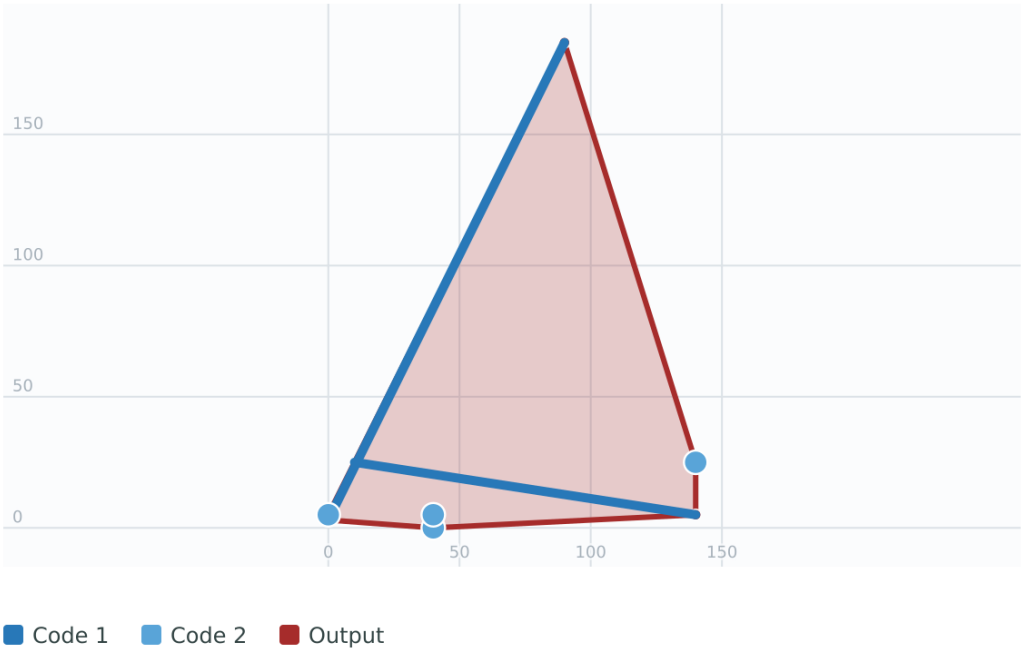
□□

Compute the convex hull of a MultiLineString and a MultiPoint.

Code

```
SELECT ST_ConvexHull(ST_Collect(ST_GeomFromText('MULTILINESTRING((100 190,10 8),(150 10,20
    30))'),
            ST_GeomFromText('MULTIPOINT(50 5,150 30,50 10,10 10)')
            ));
```

Figure



Using with ST_Collect to compute the convex hulls of geometry sets.

This example returns an estimated infected area from point observations.

Code

```
SELECT d.disease_type,
    ST_ConvexHull(ST_Collect(d.geom)) As geom
    FROM disease_obs As d
    GROUP BY d.disease_type;
```

□□

ST_Collect, ST_ConcaveHull, ST_MinimumBoundingCircle

7.14.8 ST_DelaunayTriangles

ST_DelaunayTriangles — Returns the Delaunay triangulation of the vertices of a geometry.

□□

geometry **ST_DelaunayTriangles**(geometry g1, float tolerance = 0.0, int4 flags = 0);

□□

Computes the Delaunay triangulation of the vertices of the input geometry. The optional `tolerance` can be used to snap nearby input vertices together, which improves robustness in some situations. The result geometry is bounded by the convex hull of the input vertices. The result geometry representation is determined by the `flags` code:

- `0` - a GEOMETRYCOLLECTION of triangular POLYGONs (default)
- `1` - a MULTILINESTRING of the edges of the triangulation
- `2` - A TIN of the triangulation

GEOS □□□□□
2.1.0 □□□□□□□□□□□□.

This function supports 3d and will not drop the z-index.

This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).

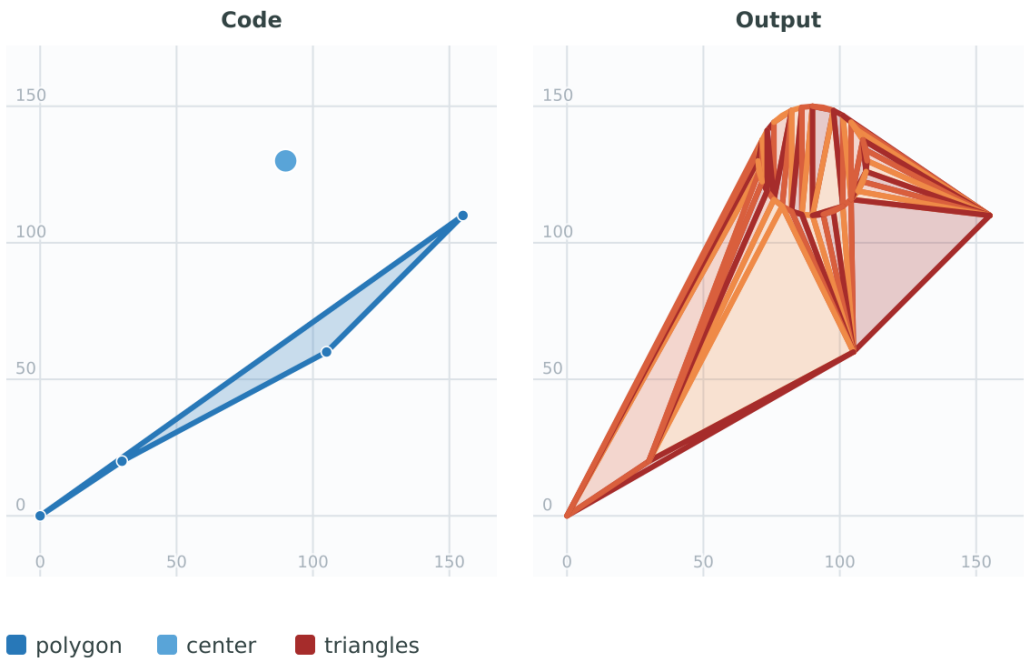
□□

Triangulation of the vertices of a polygon and a buffered point.

Code

```
WITH data AS (
  SELECT polygon, ST_Buffer(center, 20) AS point_buffer
  FROM (SELECT
    'POLYGON((175 150,20 40,50 60,125 100,175 150))'::geometry AS polygon,
    'POINT(110 170)'::geometry AS center
  ) AS input
)
SELECT ST_DelaunayTriangles(ST_Union(polygon, point_buffer)) AS triangles
FROM data;
```

Figure

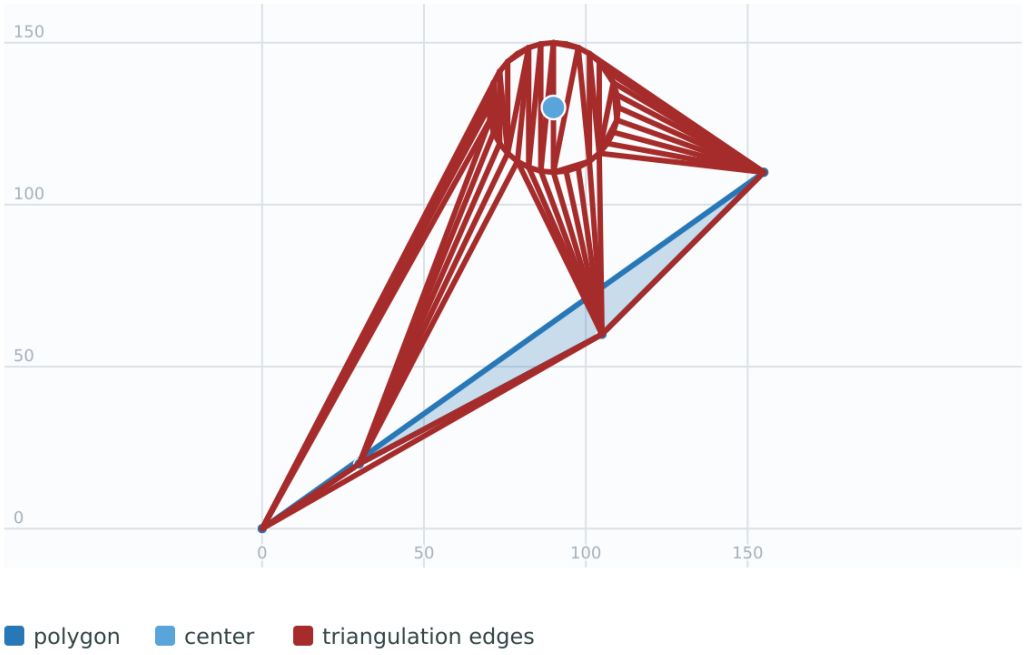


Triangulation edges returned as a `MULTILINESTRING`.

Code

```
WITH data AS (
  SELECT polygon, ST_Buffer(center, 20) AS point_buffer
  FROM (SELECT
    'POLYGON((175 150,20 40,50 60,125 100,175 150))'::geometry AS polygon,
    'POINT(110 170)'::geometry AS center
  ) AS input
)
SELECT ST_DelaunayTriangles(ST_Union(polygon, point_buffer), 0.001, 1) AS ↩
    triangulation_edges
FROM data;
```

Figure

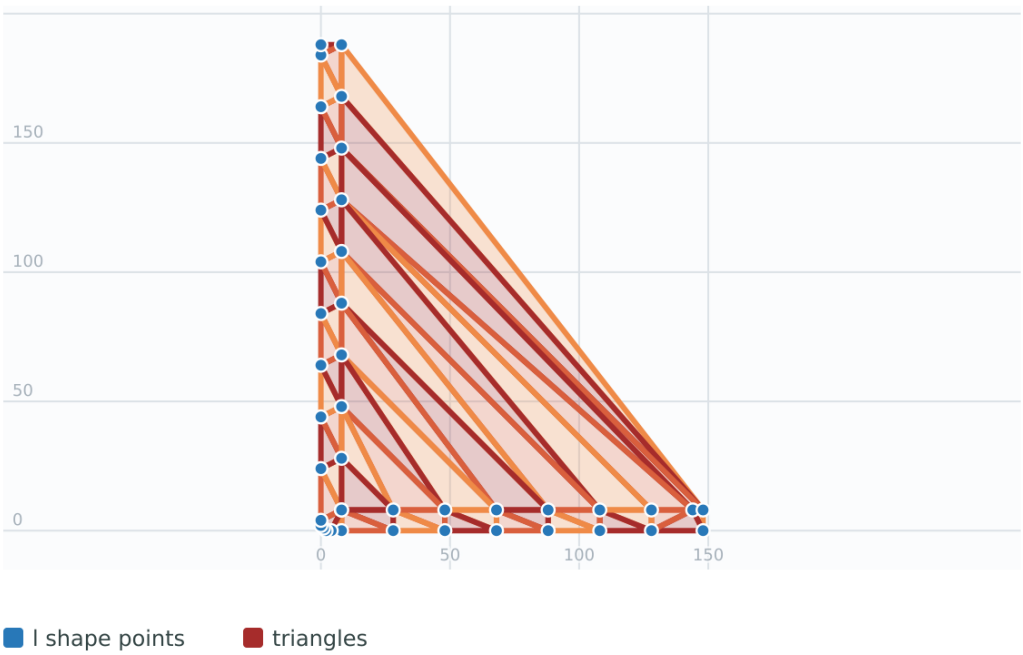


Triangulation of 42 points forming an L shape.

Code

```
WITH data AS (
  SELECT 'MULTIPOINT(14 14,34 14,54 14,74 14,94 14,114 14,134 14,
    150 14,154 14,154 6,134 6,114 6,94 6,74 6,54 6,34 6,
    14 6,10 6,8 6,7 7,6 8,6 10,6 30,6 50,6 70,6 90,6 110,6 130,
    6 150,6 170,6 190,6 194,14 194,14 174,14 154,14 134,14 114,
    14 94,14 74,14 54,14 34,14 14)'::geometry AS l_shape_points
)
SELECT ST_DelaunayTriangles(l_shape_points) AS triangles
FROM data;
```

Figure



Example using vertices with Z values.

Code

```
SELECT ST_DelaunayTriangles(
  'MULTIPOINT Z(14 14 10,150 14 100,34 6 25,20 10 150)'::geometry);
```

Output

```
GEOMETRYCOLLECTION Z (POLYGON Z ((14 14 10,20 10 150,34 6 25,14 14 10)),POLYGON Z ((14 14 ↩
    10,34 6 25,150 14 100,14 14 10)))
```

☒☒

ST_VoronoiPolygons, ST_TriangulatePolygon, CG_ConstrainedDelaunayTriangles, ST_VoronoiLines, ST_ConvexHull

7.14.9 ST_FilterByM

ST_FilterByM — Removes vertices based on their M value

☒☒

geometry **ST_FilterByM**(geometry geom, double precision min, double precision max = null, boolean returnM = false);

□□

Filters out vertex points based on their M-value. Returns a geometry with only vertex points that have a M-value larger or equal to the min value and smaller or equal to the max value. If max-value argument is left out only min value is considered. If fourth argument is left out the m-value will not be in the resulting geometry. If resulting geometry have too few vertex points left for its geometry type an empty geometry will be returned. In a geometry collection geometries without enough points will just be left out silently.

This function is mainly intended to be used in conjunction with ST_SetEffectiveArea. ST_EffectiveArea sets the effective area of a vertex in its m-value. With ST_FilterByM it then is possible to get a simplified version of the geometry without any calculations, just by filtering



□□
There is a difference in what ST_SimplifyVW returns when not enough points meet the criteria compared to ST_FilterByM. ST_SimplifyVW returns the geometry with enough points while ST_FilterByM returns an empty geometry



□□
Note that the returned geometry might be invalid



□□
This function returns all dimensions, including the Z and M values

Availability: 2.5.0

□□

A linestring is filtered

Code

```
SELECT ST_FilterByM(geom, 30) simplified
FROM (SELECT  ST_SetEffectiveArea('LINESTRING(5 2,3 8,6 20,7 25,10 10)'::geometry) geom) As ↩
     foo;
```

Output

```
LINESTRING(5 2,7 25,10 10)
```

Figure



□□

ST_SetEffectiveArea, ST_SimplifyVW

7.14.10 ST_GeneratePoints

ST_GeneratePoints — Generates a multipoint of random points contained in a Polygon or MultiPolygon.

□□

geometry **ST_GeneratePoints**(geometry g, integer npoints, integer seed = 0);

□□

ST_GeneratePoints generates a multipoint consisting of a given number of pseudo-random points which lie within the input area. The optional `seed` is used to regenerate a deterministic sequence of points, and must be greater than zero.

2.3.0 □□□□□□□□□□□□.

Enhanced: 3.0.0, added seed parameter

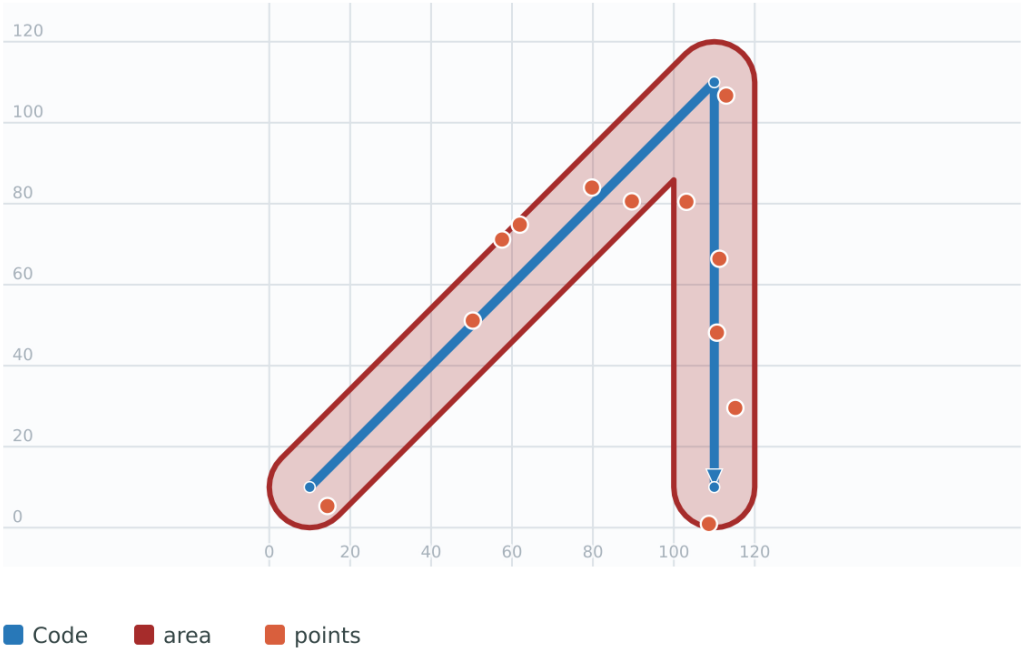
□□

Generate 12 points inside a buffered LineString using the deterministic seed 1996.

Code

```
WITH source AS (
  SELECT ST_Buffer(ST_GeomFromText('LINESTRING(50 50,150 150,150 50)'),
           10, 'endcap=round join=round') AS geom
)
SELECT geom AS area,
       ST_GeneratePoints(geom, 12, 1996) AS points
FROM source;
```

Figure



Given a table of polygons s, return 12 individual points per polygon. Results will be different each time you run.

Code

```
SELECT s.id, dp.path[1] AS pt_id, dp.geom
FROM s, ST_DumpPoints(ST_GeneratePoints(s.geom, 12)) AS dp;
```

Generate dot-density points from a polygon table, using the absolute value of an attribute as the number of points and keeping the sign of the attribute for styling.

Code

```
SELECT z.id,
       CASE WHEN z.count_delta < 0 THEN -1 ELSE 1 END AS sign,
       dp.path[1] AS pt_id,
       dp.geom
FROM zones AS z
CROSS JOIN LATERAL ST_DumpPoints(
    ST_GeneratePoints(z.geom, abs(z.count_delta)::integer, z.id + 1)
) AS dp
WHERE z.count_delta <> 0;
```

☒☒

ST_DumpPoints

7.14.11 ST_GeometricMedian

ST_GeometricMedian — □□□□□□□□□□□□□□ (median) □□□□□□.

□□

geometry **ST_GeometricMedian** (geometry geom, float8 tolerance = NULL, int max_iter = 10000, boolean fail_if_not_converged = false);

□□

Computes the approximate geometric median of a MultiPoint geometry using the Weiszfeld algorithm. The geometric median is the point minimizing the sum of distances to the input points. It provides a centrality measure that is less sensitive to outlier points than the centroid (center of mass).

The algorithm iterates until the distance change between successive iterations is less than the supplied `tolerance` parameter. If this condition has not been met after `max_iterations` iterations, the function produces an error and exits, unless `fail_if_not_converged` is set to `false` (the default).

If a `tolerance` argument is not provided, the tolerance value is calculated based on the extent of the input geometry.

If present, the input point M values are interpreted as their relative weights.

2.3.0 □□□□□□□□□□□□.

Enhanced: 2.5.0 Added support for M as weight of points.

This function supports 3d and will not drop the z-index.

This function supports M coordinates.

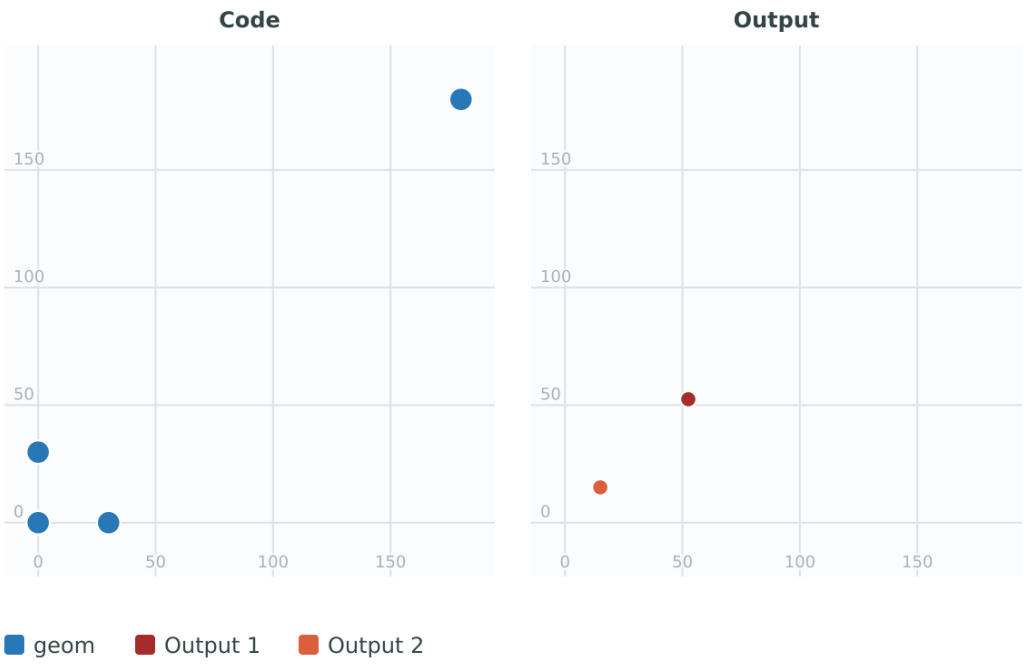
□□

Compare the geometric median and centroid of a MultiPoint.

Code

```
WITH test AS (
SELECT 'MULTIPOINT((10 10),(10 40),(40 10),(190 190))'::geometry geom)
SELECT
  ST_Centroid(geom) centroid,
  ST_GeometricMedian(geom) median
FROM test;
```

Figure



□□

ST_Centroid

7.14.12 ST_LineMerge

ST_LineMerge — Return the lines formed by sewing together a MultiLineString.

□□

geometry **ST_LineMerge**(geometry amultilinestring);
geometry **ST_LineMerge**(geometry amultilinestring, boolean directed);

□□

Returns a LineString or MultiLineString formed by joining together the line elements of a MultiLineString. Lines are joined at their endpoints at 2-way intersections. Lines are not joined across intersections of 3-way or greater degree.

If **directed** is TRUE, then ST_LineMerge will not change point order within LineStrings, so lines with opposite directions will not be merged



□□
Only use with MultiLineString/LineStrings. Other geometry types return an empty GeometryCollection

GEOS □□□□□

Enhanced: 3.3.0 accept a directed parameter.

Requires GEOS >= 3.11.0 to use the directed parameter.

1.1.0 □□□□□□□□□□□□.



□□
This function strips the M dimension.

□□

Cardinality 2

Nodes with cardinality 2 are merged away.

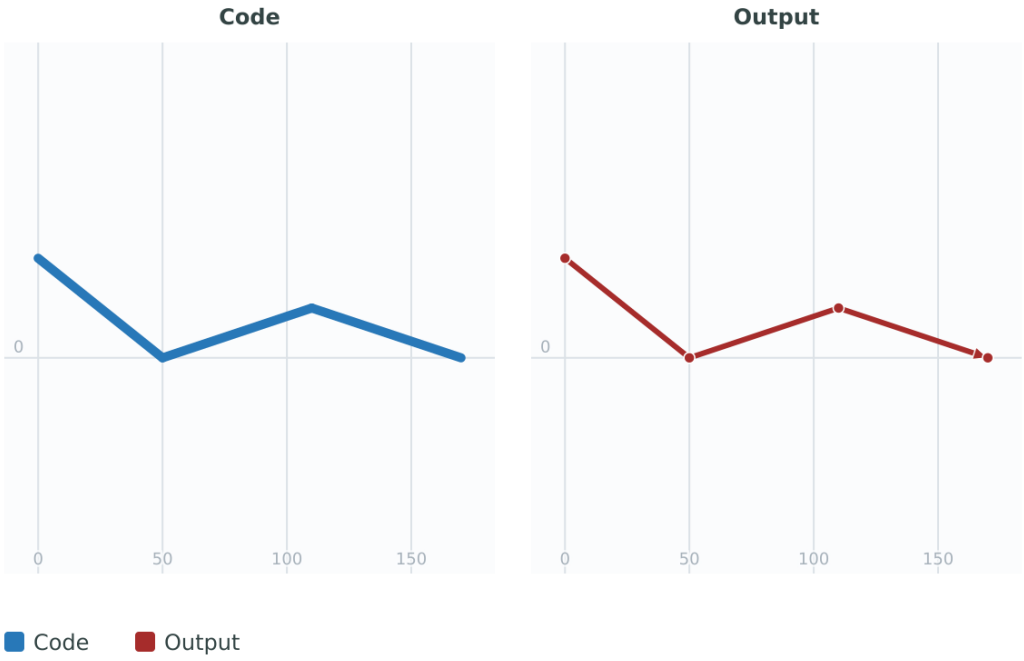
Code

```
SELECT ST_LineMerge('MULTILINESTRING((10 160,60 120),(120 140,60 120),(120 140,180 120))'
);
```

Output

```
LINESTRING(10 160,60 120,120 140,180 120)
```

Figure



Cardinality 3 or More

Lines are not merged across intersections with degree > 2.

Code

```
SELECT ST_LineMerge('MULTILINESTRING((10 160,60 120),(120 140,60 120),(120 140,180 120) ↩
    ,(100 180,120 140))'
);
```

Output

```
MULTILINESTRING((10 160,60 120,120 140),(100 180,120 140),(120 140,180 120))
```

Figure



Non-Touching Lines

If merging is not possible due to non-touching lines, the original MultiLineString is returned.

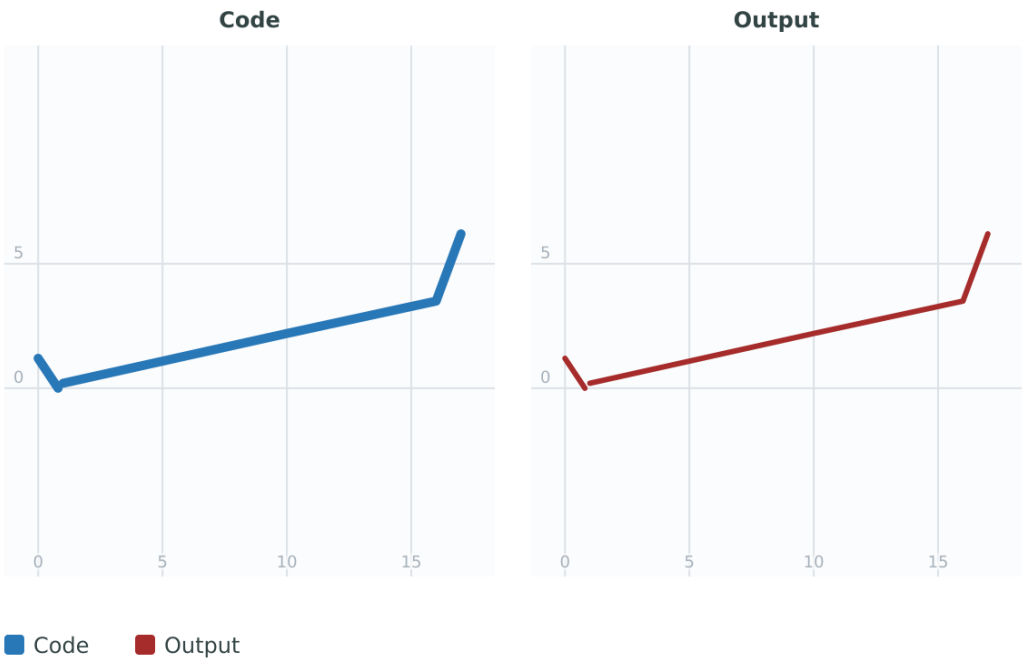
Code

```
SELECT ST_LineMerge('MULTILINESTRING((-29 -27,-30 -29.7,-36 -31,-45 -33),(-45.2 -33.2,-46 ↩
    -32))'
);
```

Output

```
MULTILINESTRING((-45.2 -33.2,-46 -32),(-29 -27,-30 -29.7,-36 -31,-45 -33))
```

Figure



Directed Parameter

Lines with opposite directions are not merged if *directed* is TRUE.

Code

```
SELECT ST_LineMerge('MULTILINESTRING((60 30,10 70),(120 50,60 30),(120 50,180 30))',
TRUE);
```

Output

```
MULTILINESTRING((120 50,60 30,10 70),(120 50,180 30))
```

Figure



Z-dimension Handling

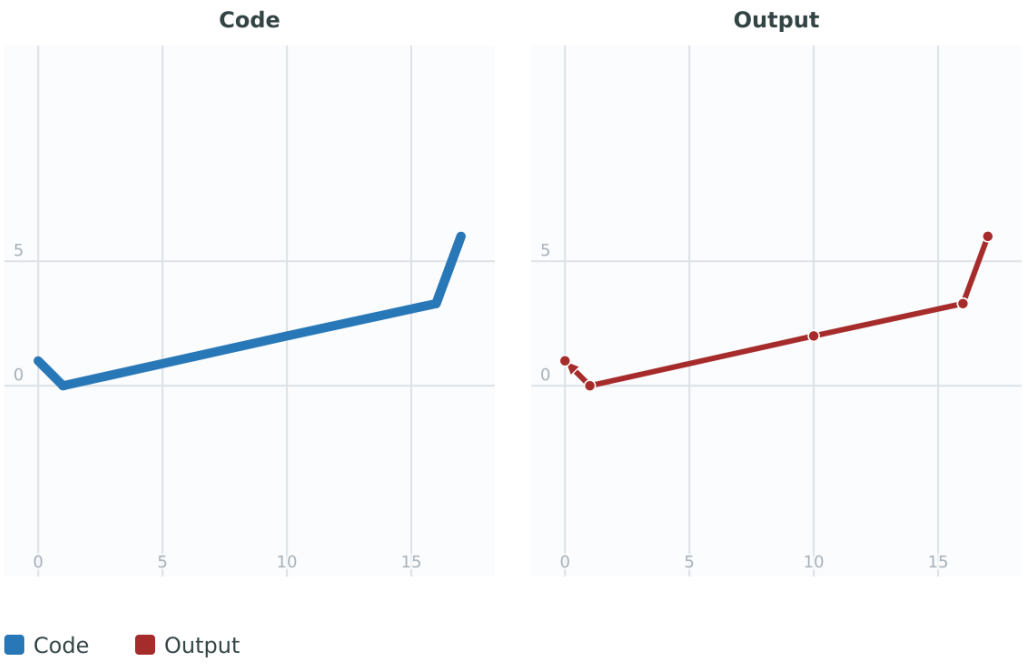
Code

```
SELECT ST_LineMerge('MULTILINESTRING((-29 -27 11,-30 -29.7 10,-36 -31 5,-45 -33 6),(-29 -27 ↩
     12,-30 -29.7 5),(-45 -33 1,-46 -32 11))'
         );
```

Output

```
LINESTRING Z (-30 -29.7 5,-29 -27 11,-30 -29.7 10,-36 -31 5,-45 -33 1,-46 -32 11)
```

Figure



□□

ST_Segmentize, ST_LineSubstring

7.14.13 ST_MaximumInscribedCircle

ST_MaximumInscribedCircle — □□□□□□□□□□□□□□□□.

□□

(geometry, geometry, double precision) **ST_MaximumInscribedCircle**(geometry geom);

□□

Finds the largest circle that is contained within a (multi)polygon, or which does not overlap any lines and points. Returns a record with fields:

- `center` - center point of the circle
- `nearest` - a point on the geometry nearest to the center
- `radius` - radius of the circle

For polygonal inputs, the circle is inscribed within the boundary rings, using the internal rings as boundaries. For linear and point inputs, the circle is inscribed within the convex hull of the input, using the input lines and points as further boundaries.

Availability: 3.1.0.

Requires GEOS >= 3.9.0.

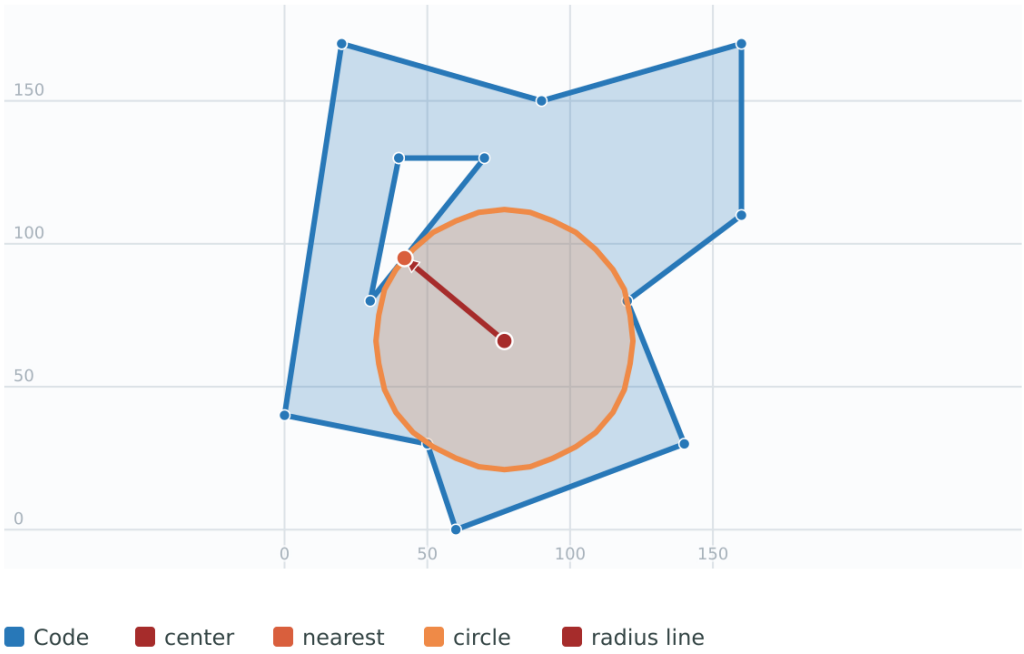
□□

Maximum inscribed circle of a polygon. Center, nearest point, radius, circle, and radius line are returned.

Code

```
WITH mic AS (
  SELECT * FROM ST_MaximumInscribedCircle(
    'POLYGON ((40 180,110 160,180 180,180 120,140 90,160 40,80 10,70 40,20 50,40 180),
              (60 140,50 90,90 140,60 140))')
)
SELECT round(radius::numeric, 3) AS radius,
       ST_SnapToGrid(center, 1) AS center,
       ST_SnapToGrid(nearest, 1) AS nearest,
       ST_SnapToGrid(ST_Buffer(center, radius), 1) AS circle,
       ST_SnapToGrid(ST_MakeLine(center, nearest), 1) AS radius_line
FROM mic;
```

Figure

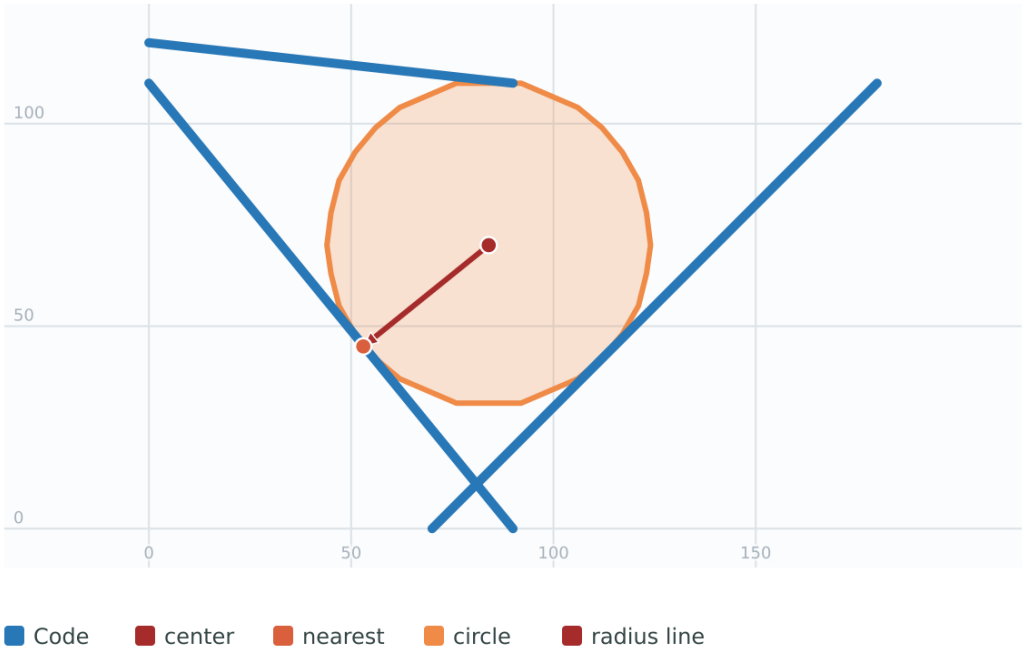


Maximum inscribed circle of a MultiLineString.

Code

```
WITH mic AS (
  SELECT * FROM ST_MaximumInscribedCircle(
    'MULTILINESTRING((100 40,10 150),(10 160,100 150),(190 150,80 40))')
)
SELECT round(radius::numeric, 3) AS radius,
       ST_SnapToGrid(center, 1) AS center,
       ST_SnapToGrid(nearest, 1) AS nearest,
       ST_SnapToGrid(ST_Buffer(center, radius), 1) AS circle,
       ST_SnapToGrid(ST_MakeLine(center, nearest), 1) AS radius_line
FROM mic;
```

Figure



☒☒

ST_MinimumBoundingRadius, ST_LargestEmptyCircle

7.14.14 ST_LargestEmptyCircle

ST_LargestEmptyCircle — Computes the largest circle not overlapping a geometry.

☒☒

(geometry, geometry, double precision) **ST_LargestEmptyCircle**(geometry geom, double precision tolerance=0.0, geometry boundary=POINT EMPTY);

☒☒

Finds the largest circle which does not overlap a set of point and line obstacles. (Polygonal geometries may be included as obstacles, but only their boundary lines are used.) The center of the circle is constrained to lie inside a polygonal boundary, which by default is the convex hull of the input geometry. The circle center is the point in the interior of the boundary which has the farthest distance from the obstacles. The circle itself is provided by the center point and a nearest point lying on an obstacle determining the circle radius.

If a boundary is supplied it must be a valid polygonal geometry with positive area.

The circle center is determined to a given accuracy specified by a distance tolerance, using an iterative algorithm. If the accuracy distance is not specified a reasonable default is used.

Returns a record with fields:

- `center` - center point of the circle

- `nearest` - a point on the geometry nearest to the center
- `radius` - radius of the circle

To find the largest empty circle in the interior of a polygon, see ST_MaximumInscribedCircle.

Availability: 3.4.0.

Requires GEOS >= 3.9.0.

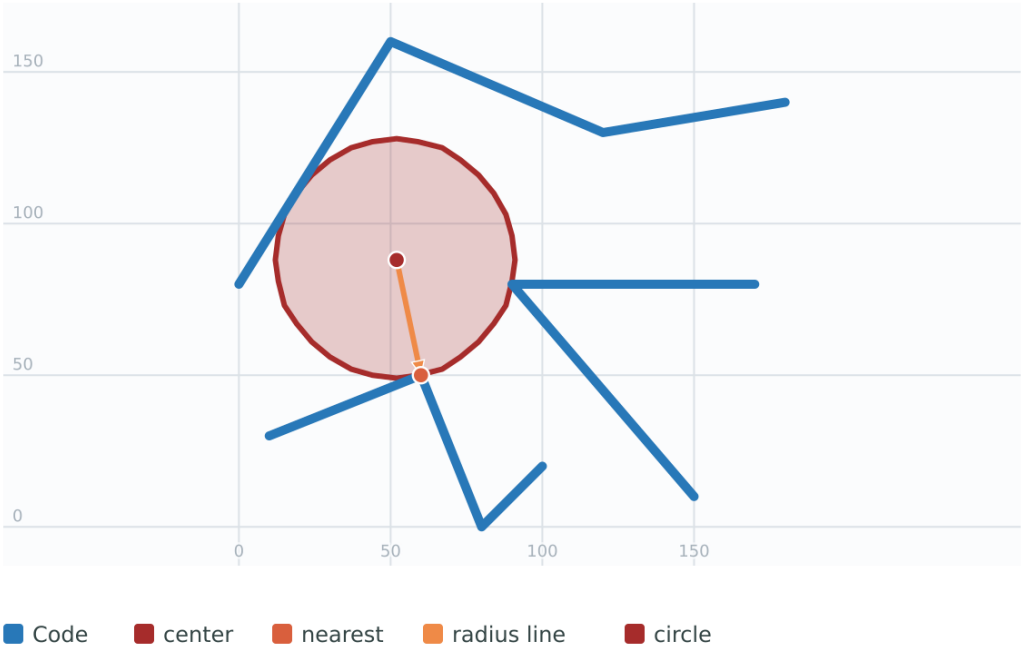
□□

Largest empty circle within a set of lines.

Code

```
WITH obstacles(geom) AS (VALUES ('MULTILINESTRING (
        (10 100,60 180,130 150,190 160),
        (20 50,70 70,90 20,110 40),
        (160 30,100 100,180 100))'::geometry)),
circle AS (
  SELECT result.*
  FROM obstacles
  CROSS JOIN LATERAL ST_LargestEmptyCircle(obstacles.geom) AS result
)
SELECT round(radius::numeric, 6) AS radius,
       ST_SnapToGrid(center, 1) AS center,
       ST_SnapToGrid(nearest, 1) AS nearest,
       ST_SnapToGrid(ST_MakeLine(center, nearest), 1) AS radius_line,
       ST_SnapToGrid(ST_Buffer(center, radius), 1) AS circle
FROM circle;
```

Figure

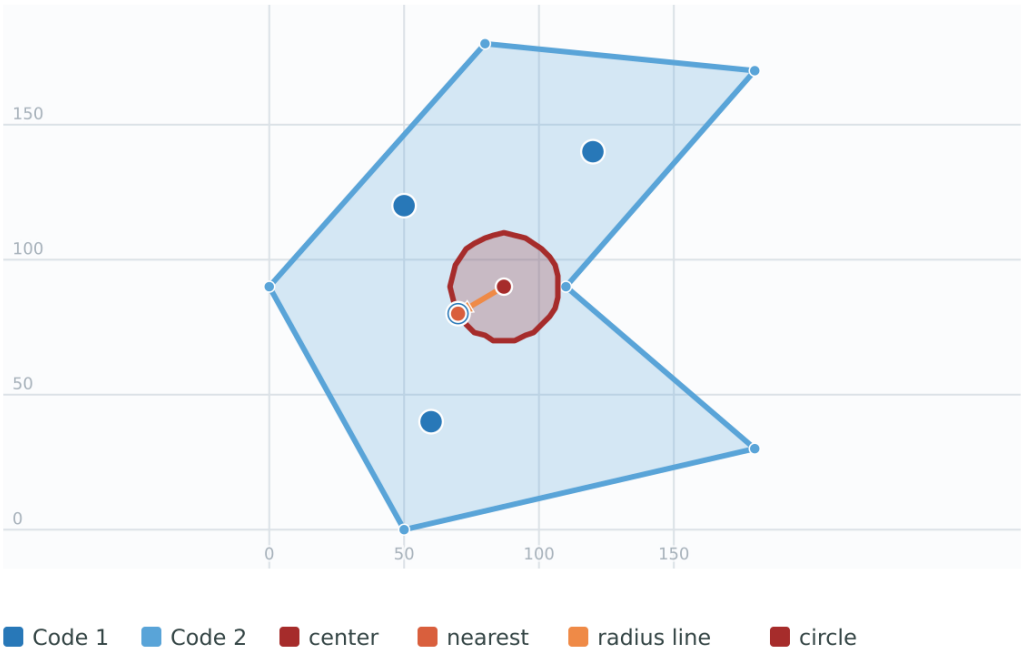


Largest empty circle within a set of points, constrained to lie in a polygon. The constraint polygon boundary is included both as an obstacle and as the constraint for the circle center.

Code

```
WITH inputs(points, boundary) AS (VALUES (
  'MULTIPOINT ((70 50),(60 130),(130 150),(80 90))'::geometry,
  'POLYGON ((90 190,10 100,60 10,190 40,120 100,190 180,90 190))'::geometry
)),
circle AS (
  SELECT result.*
  FROM inputs
  CROSS JOIN LATERAL ST_LargestEmptyCircle(
    ST_Collect(points, boundary), 0, boundary
  ) AS result
)
SELECT round(radius::numeric, 6) AS radius,
       ST_SnapToGrid(center, 1) AS center,
       ST_SnapToGrid(nearest, 1) AS nearest,
       ST_SnapToGrid(ST_MakeLine(center, nearest), 1) AS radius_line,
       ST_SnapToGrid(ST_Buffer(center, radius), 1) AS circle
FROM circle;
```

Figure



☒☒

ST_MinimumBoundingRadius

7.14.15 ST_MinimumBoundingCircle

ST_MinimumBoundingCircle — Returns the smallest circle polygon that contains a geometry.

☒☒

geometry **ST_MinimumBoundingCircle**(geometry geomA, integer num_segs_per_qt_circ=48);

⊠⊠

Returns the smallest circle polygon that contains a geometry.

> ⊠⊠
> ⊠⊠⊠⊠⊠⊠⊠ 48 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (minimum bounding circle) ⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠, ST_MinimumBoundingRadius ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

Use with ST_Collect to get the minimum bounding circle of a set of geometries.

To compute two points lying on the minimum circle (the ”maximum diameter”) use ST_LongestLine.

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (Roeck) ⊠⊠⊠⊠⊠⊠⊠⊠⊠.

GEOS ⊠⊠⊠⊠⊠

1.4.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

Code

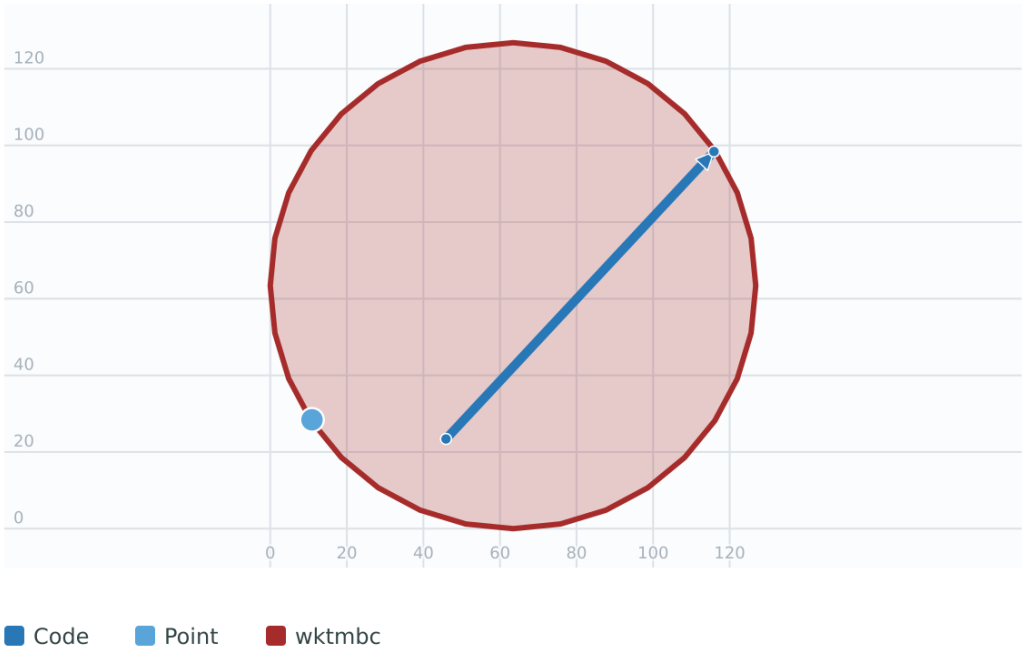
```
SELECT d.disease_type,
ST_MinimumBoundingCircle(ST_Collect(d.geom)) As geom
FROM disease_obs As d
GROUP BY d.disease_type;
```

Minimum bounding circle of a point and linestring, using 8 segments to approximate a quarter circle.

Code

```
SELECT ST_MinimumBoundingCircle(ST_Collect('LINESTRING(55 75,125 150)',
ST_Point(20, 80)), 8
) As wktmbc;
```

Figure



□□

ST_Collect, ST_MinimumBoundingRadius, ST_LargestEmptyCircle, ST_LongestLine

7.14.16 ST_MinimumBoundingRadius

ST_MinimumBoundingRadius — Returns the center point and radius of the smallest circle that contains a geometry.

□□

(geometry, double precision) **ST_MinimumBoundingRadius**(geometry geom);

□□

Computes the center point and radius of the smallest circle that contains a geometry. Returns a record with fields:

- `center` - center point of the circle
- `radius` - radius of the circle

Use with ST_Collect to get the minimum bounding circle of a set of geometries.

To compute two points lying on the minimum circle (the ”maximum diameter”) use ST_LongestLine.

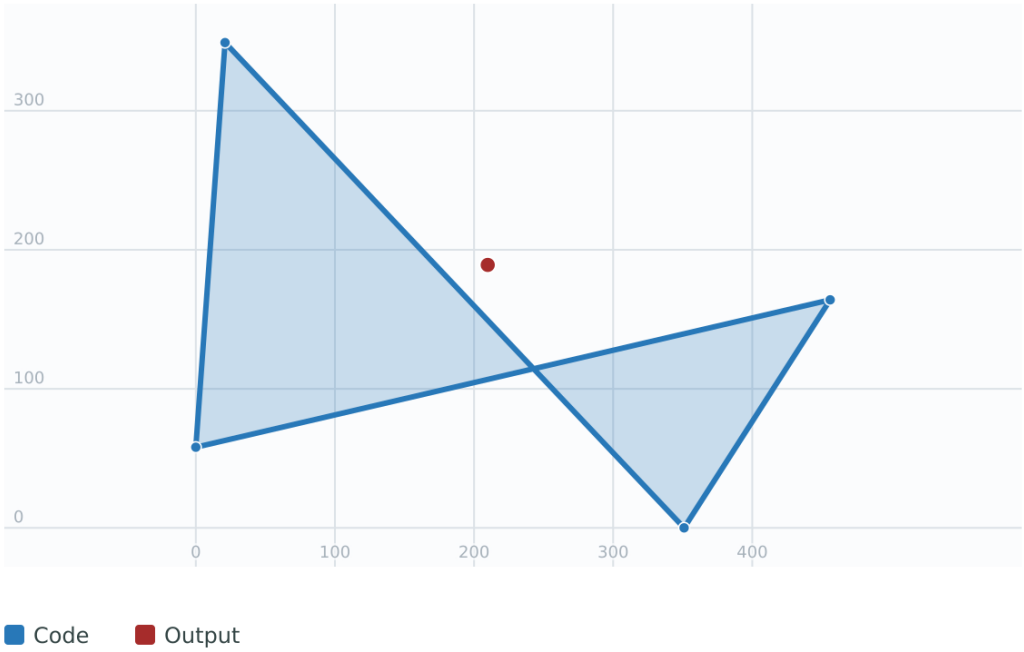
2.3.0 □□□□□□□□□□□□.

□□

Code

```
SELECT center, radius FROM ST_MinimumBoundingRadius('POLYGON((26426 65078,26531 65242,26075 ↩
    65136,26096 65427,26426 65078))');
```

Figure



☒☒

ST_Collect, ST_MinimumBoundingCircle, ST_LongestLine

7.14.17 ST_OrientedEnvelope

ST_OrientedEnvelope — Returns a minimum-area rectangle containing a geometry.

☒☒

geometry **ST_OrientedEnvelope**(geometry geom);

☒☒

Returns the minimum-area rotated rectangle enclosing a geometry. Note that more than one such rectangle may exist. May return a Point or LineString in the case of degenerate inputs.

Availability: 2.5.0.

Requires GEOS >= 3.6.0.

☒☒

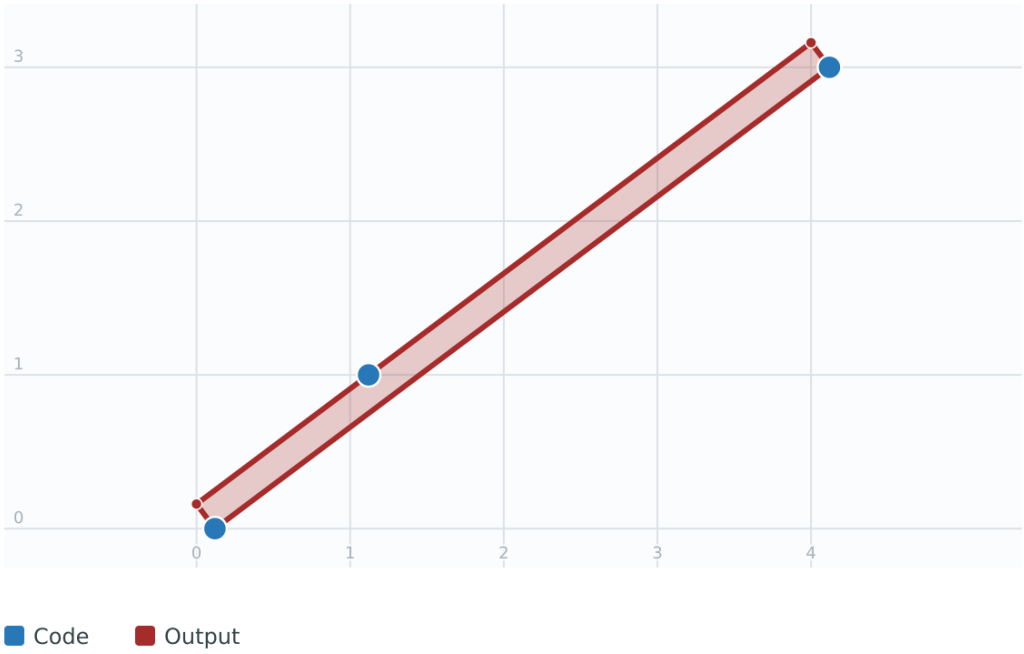
Code

```
SELECT ST_OrientedEnvelope('MULTIPOINT ((0 0),(-1 -1),(3 2))');
```

Output

```
POLYGON((3 2,-1 -1,-1.12 -0.84,2.88 2.16,3 2))
```

Figure



Oriented envelope of a Point and LineString.

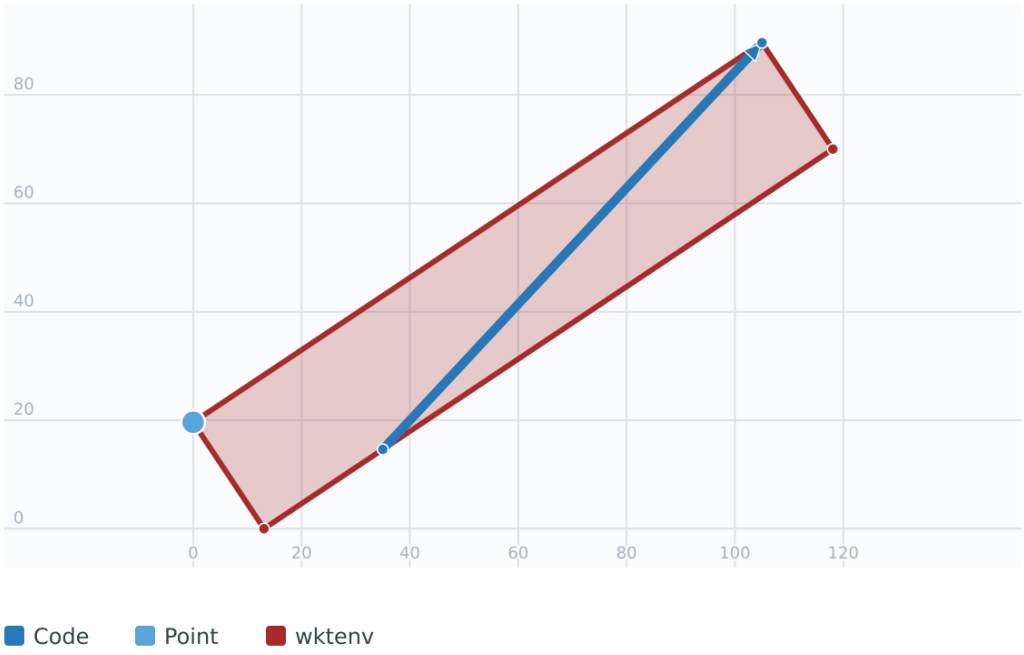
Code

```
SELECT ST_OrientedEnvelope(ST_Collect('LINESTRING(55 75,125 150)',
                ST_Point(20, 80))
                ) As wktenv;
```

Output

```
POLYGON((20 80,125 150,138.0769230769241 130.38461538461573,33.07692307692291 60.384615384615195,20 80))
```

Figure



□□

ST_Envelope ST_MinimumBoundingCircle

7.14.18 ST_OffsetCurve

ST_OffsetCurve — Returns an offset line at a given distance and side from an input line.

□□

geometry **ST_OffsetCurve**(geometry line, float signed_distance, text style_parameters='');

□□

Return an offset line at a given distance and side from an input line. All points of the returned geometries are not further than the given distance from the input geometry. Useful for computing parallel lines about a center line.

For positive distance the offset is on the left side of the input line and retains the same direction. For a negative distance it is on the right side and in the opposite direction.

□□□□□□□□□□□□□□□□□□□□□□□□□.

Note that output may be a MULTILINESTRING or EMPTY for some jigsaw-shaped input geometries.

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ = □□□□□□□□□□□□□:

- 'quad_segs=#' : □□□ (quarter circle) □□□□□□□□□□□□□□□□□□□□ (□□□□ 8)
- 'join=round|mitre|bevel' : □□□□□ (□□□□" □□ (round)"). ' □□□ (mitre)' □□□□□' □□□ (miter)' □□□□□□□□.

- 'mitre_limit=#.#' : □□□□□□□ (□□□□□□□□□□□□□□□□□□). 'mitre_limit' □□□□□'miter_limit' □□□□□□□.

GEOS □□□□□

Behavior changed in GEOS 3.11 so offset curves now have the same direction as the input line, for both positive and negative offsets.

2.0 □□□□□□□□□□□□.

Enhanced: 2.5 - added support for GEOMETRYCOLLECTION and MULTILINESTRING



□□
This function ignores the Z dimension. It always gives a 2D result even when used on a 3D geometry.

□□

Compute an open buffer around roads.

Code

```
SELECT ST_Union(
 ST_OffsetCurve(f.geom, f.width/2, 'quad_segs=4 join=round'),
 ST_OffsetCurve(f.geom, -f.width/2, 'quad_segs=4 join=round')
) as track
FROM someroadstable;
```

Offset 15 units with `quad_segs=4 join=round`.

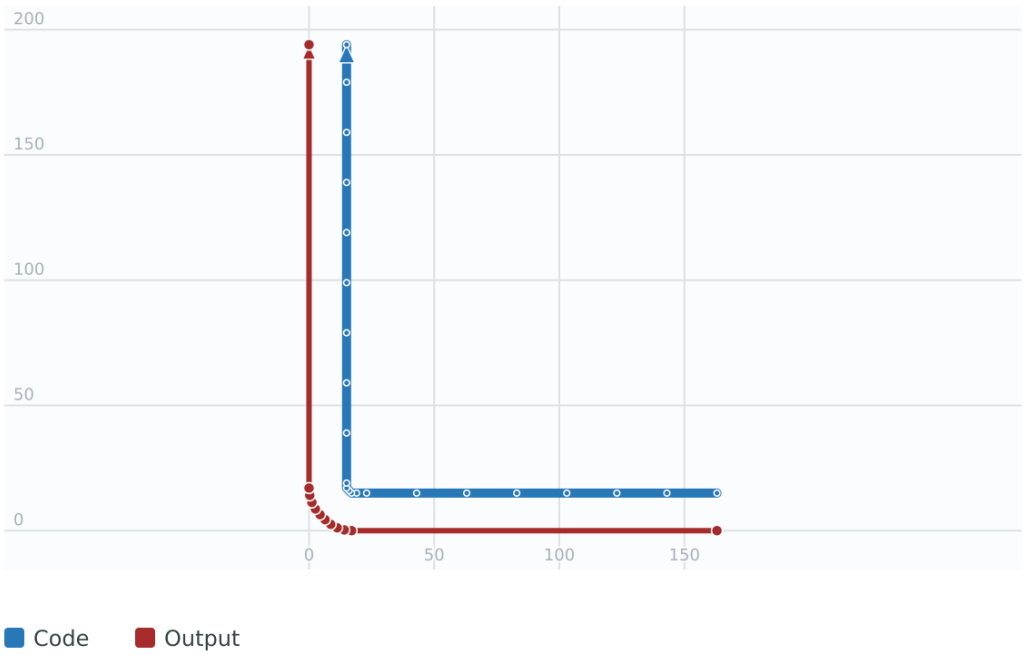
Code

```
SELECT ST_OffsetCurve(
    ST_GeomFromText('LINESTRING(164 16,144 16,124 16,104 16,84 16,64 16,
    44 16,24 16,20 16,18 16,17 17,
    16 18,16 20,16 40,16 60,16 80,16 100,
    16 120,16 140,16 160,16 180,16 195)'),
    15, 'quad_segs=4 join=round');
```

Output

```
LINESTRING(164 1,18 1,15.073645169758077 1.288220793951544,
    12.259748514523654 2.141807012330698,9.66644650470597 3.527955815461818,
    7.393398282201788 5.393398282201788,5.393398282201788 7.393398282201788,
    3.527955815461818 9.66644650470597,2.141807012330696 12.259748514523654,
    1.288220793951544 15.073645169758075,1 18,1 195)
```

Figure



Offset -15 units with `quad_segs=4 join=round`.

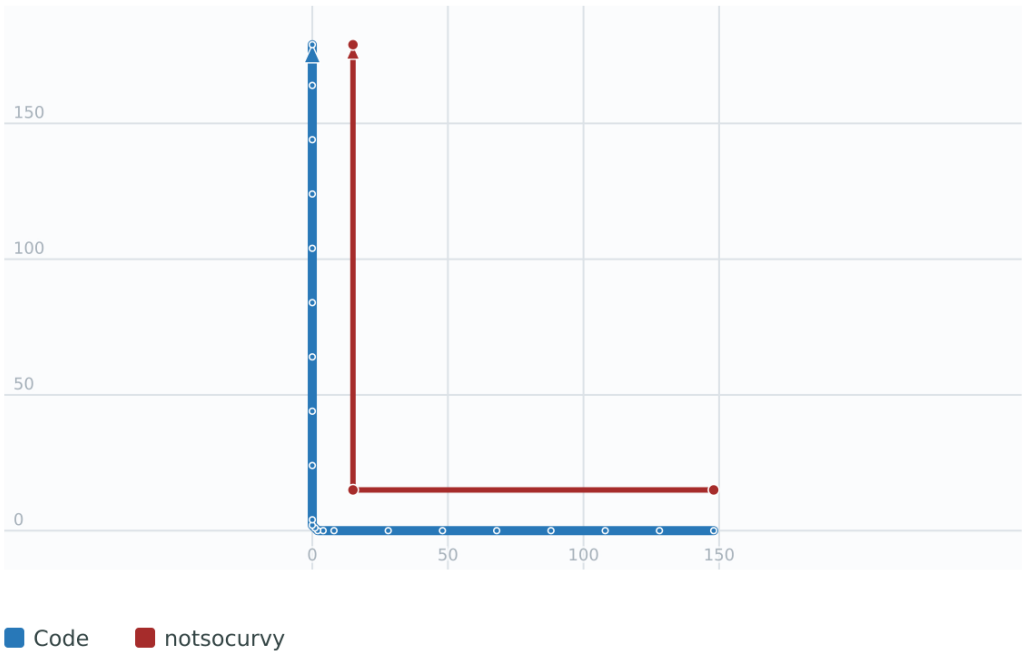
Code

```
SELECT ST_OffsetCurve(
    geom,
    -15,
    'quad_segs=4 join=round') As notsocurvy
    FROM ST_GeomFromText('LINESTRING(164 16,144 16,124 16,104 16,84 16,64 16,
    44 16,24 16,20 16,18 16,17 17,
    16 18,16 20,16 40,16 60,16 80,16 100,
    16 120,16 140,16 160,16 180,16 195)') As geom;
```

Output

```
LINESTRING(164 31,31 31,31 195)
```

Figure



Apply two negative offsets to obtain a smoother curve; the distances combine to -45 units.

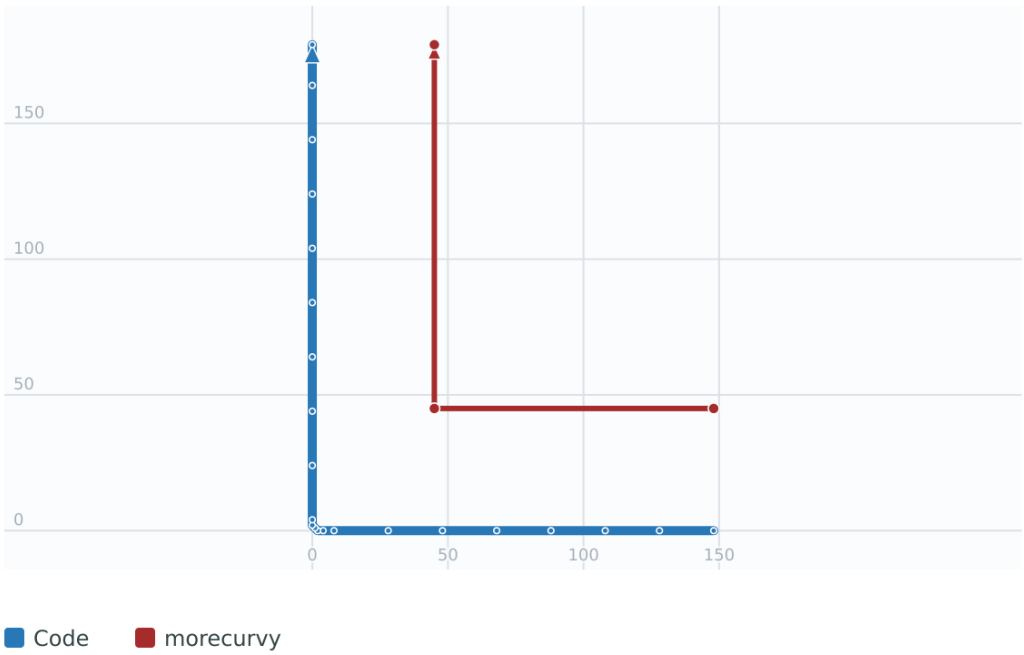
Code

```
SELECT ST_OffsetCurve(
    ST_OffsetCurve(
        geom,
        -30,
        'quad_segs=4 join=round'),
    -15,
    'quad_segs=4 join=round') As morecurvy
    FROM ST_GeomFromText('LINESTRING(164 16,144 16,124 16,104 16,84 16,64 16,
    44 16,24 16,20 16,18 16,17 17,
    16 18,16 20,16 40,16 60,16 80,16 100,
    16 120,16 140,16 160,16 180,16 195)') As geom;
```

Output

```
LINESTRING(164 61,61 61,61 195)
```

Figure



Collect the positive offset and the double-offset to produce parallel curves.

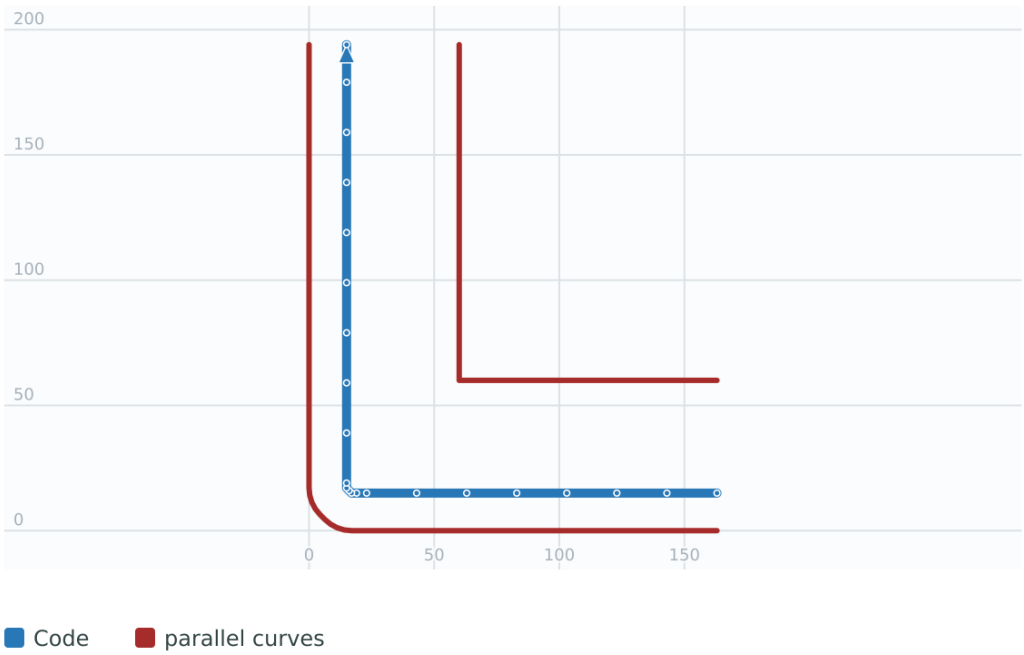
Code

```
SELECT ST_Collect(
    ST_OffsetCurve(geom, 15, 'quad_segs=4 join=round'),
    ST_OffsetCurve(
        ST_OffsetCurve(
            geom,
            -30,
            'quad_segs=4 join=round'),
        -15,
        'quad_segs=4 join=round')
) As parallel_curves
    FROM ST_GeomFromText('LINESTRING(164 16,144 16,124 16,104 16,84 16,64 16,
    44 16,24 16,20 16,18 16,17 17,
    16 18,16 20,16 40,16 60,16 80,16 100,
    16 120,16 140,16 160,16 180,16 195)') As geom;
```

Output

```
MULTILINESTRING((164 1,18 1,15.073645169758077 1.288220793951544,
12.259748514523654 2.141807012330698,9.66644650470597 3.527955815461818,
7.393398282201788 5.393398282201788,5.393398282201788 7.393398282201788,
3.527955815461818 9.66644650470597,2.141807012330696 12.259748514523654,
1.288220793951544 15.073645169758075,1 18,1 195),(164 61,61 61,61 195))
```

Figure



Offset 15 units with a bevel join.

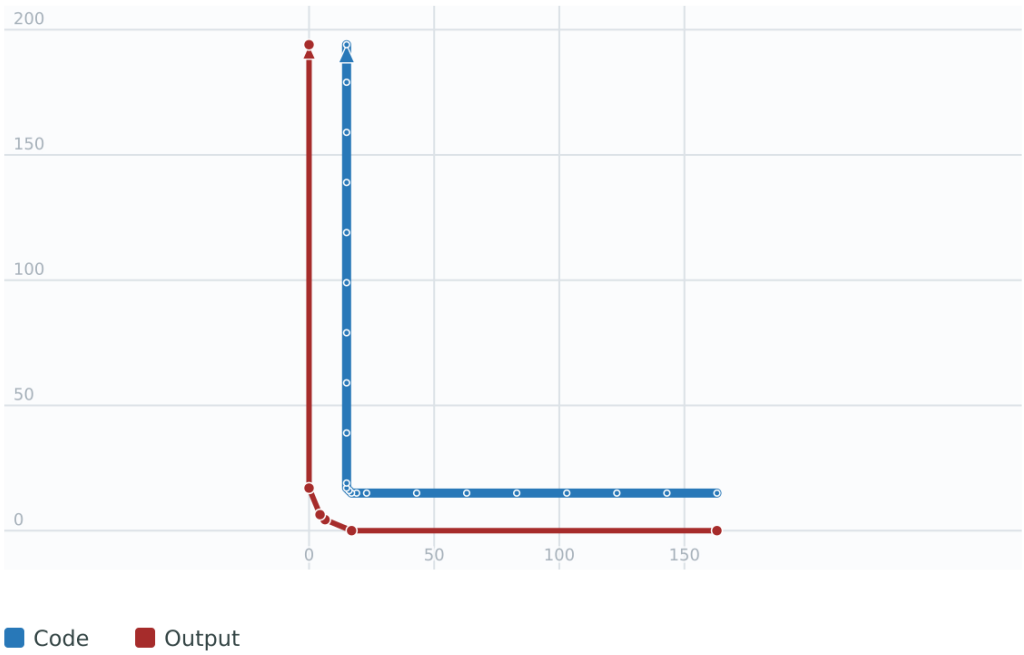
Code

```
SELECT ST_OffsetCurve(
    ST_GeomFromText('LINESTRING(164 16,144 16,124 16,104 16,84 16,64 16,
    44 16,24 16,20 16,18 16,17 17,
    16 18,16 20,16 40,16 60,16 80,16 100,
    16 120,16 140,16 160,16 180,16 195)'),
        15, 'quad_segs=4 join=bevel');
```

Output

```
LINESTRING(164 1,18 1,7.393398282201788 5.393398282201788,
    5.393398282201788 7.393398282201788,1 18,1 195)
```

Figure



Collect offsets on both sides using a mitre join and a mitre limit of 2.2.

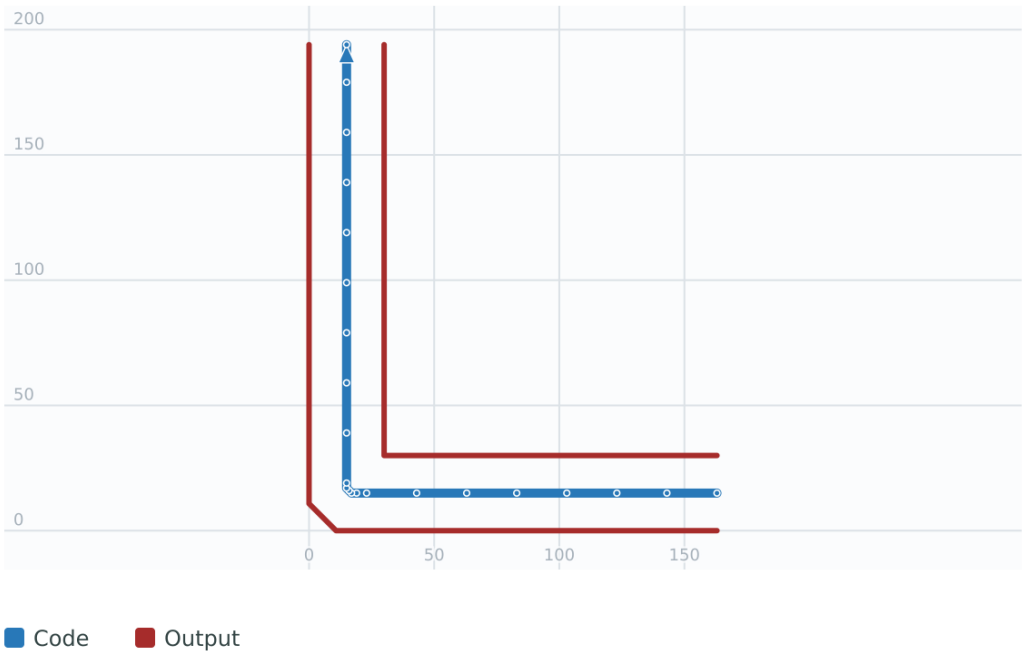
Code

```
SELECT ST_Collect(
    ST_OffsetCurve(geom, 15, 'quad_segs=4 join=mitre mitre_limit=2.2'),
    ST_OffsetCurve(geom, -15, 'quad_segs=4 join=mitre mitre_limit=2.2')
    )
    FROM ST_GeomFromText('LINESTRING(164 16,144 16,124 16,104 16,84 16,64 16,
    44 16,24 16,20 16,18 16,17 17,
    16 18,16 20,16 40,16 60,16 80,16 100,
    16 120,16 140,16 160,16 180,16 195)') As geom;
```

Output

```
MULTILINESTRING((164 1,11.786796564403577 1,1 11.786796564403577,1 195),
    (164 31,31 31,31 195))
```

Figure



□□

ST_Buffer

7.14.19 ST_PointOnSurface

ST_PointOnSurface — Computes a point guaranteed to lie in a polygon, or on a geometry.

□□

geometry **ST_PointOnSurface**(geometry g1);

□□

Returns a `POINT` which is guaranteed to lie in the interior of a surface (`POLYGON`, `MULTIPOLYGON`, and `CURVEPOLYGON`). In PostGIS this function also works on line and point geometries.

This method implements the OGC Simple Features Implementation Specification for SQL 1.1. s3.2.14.2 // s3.2.18.2

This method implements the SQL/MM specification. SQL-MM 3: 8.1.5, 9.5.6. The specifications define ST_PointOnSurface for surface geometries only. PostGIS extends the function to support all common geometry types. Other databases (Oracle, DB2, ArcSDE) seem to support this function only for surfaces. SQL Server 2008 supports all common geometry types.

This function supports 3d and will not drop the z-index.

□□

Point on surface of a MultiPoint.

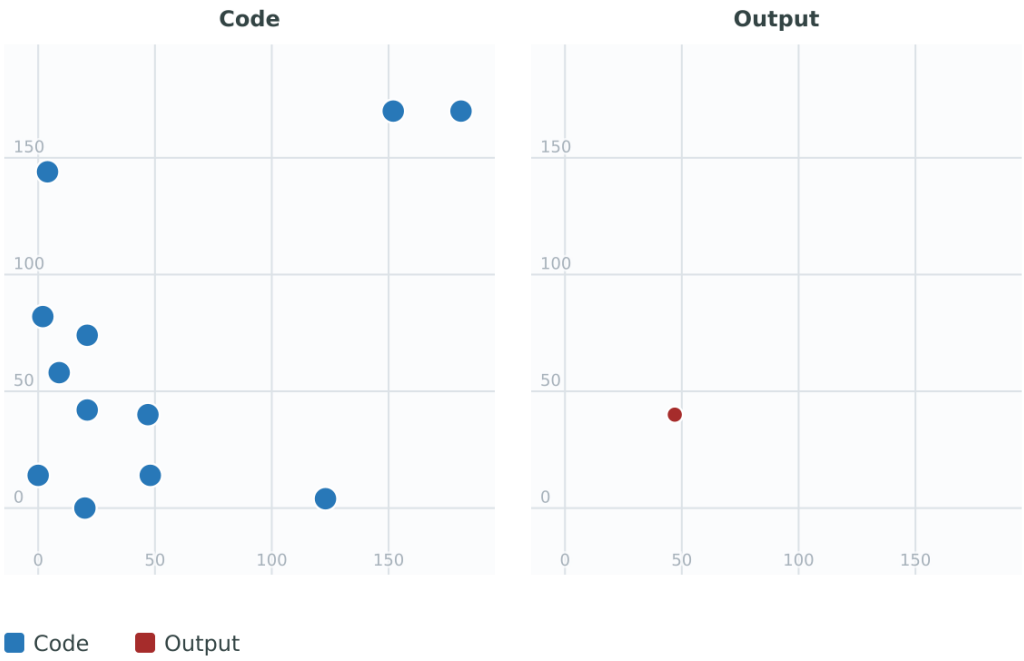
Code

```
SELECT ST_PointOnSurface(
  'MULTIPOINT (8 24,10 92,12 154,17 68,28 10,29 52,29 84,55 50,56 24,131 14,160 180,189 ↩
   180)');
```

Output

```
POINT(55 50)
```

Figure



Point on surface of a LineString.

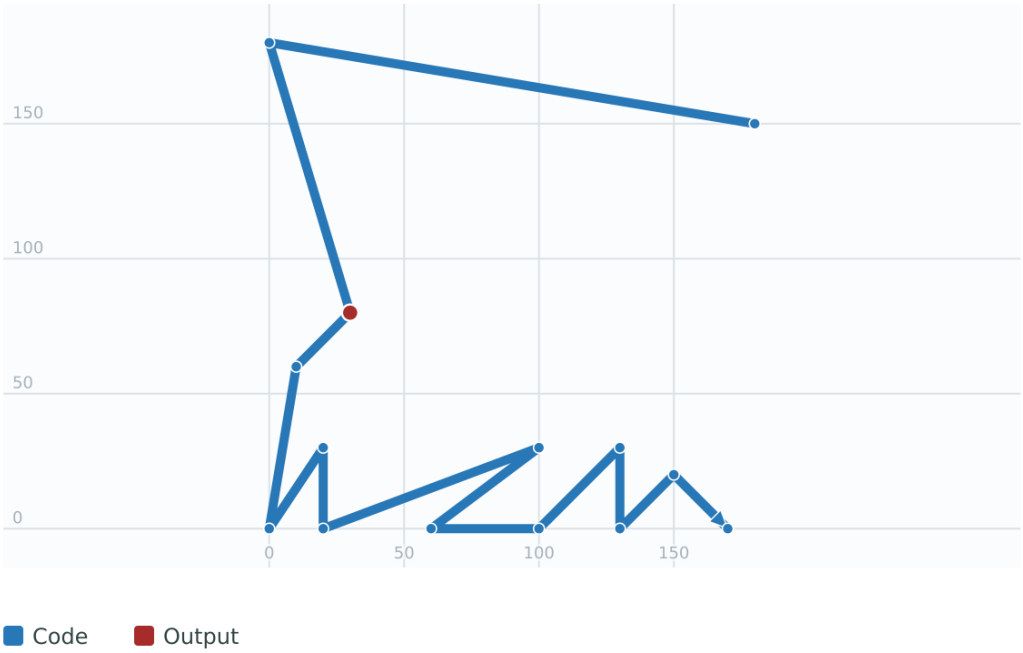
Code

```
SELECT ST_PointOnSurface(
  'LINESTRING (190 160,10 190,40 90,20 70,10 10,30 40,30 10,110 40,70 10,110 10,140 40,140 ↩
   10,160 30,180 10)');
```

Output

```
POINT(40 90)
```

Figure



Point on surface of a Polygon.

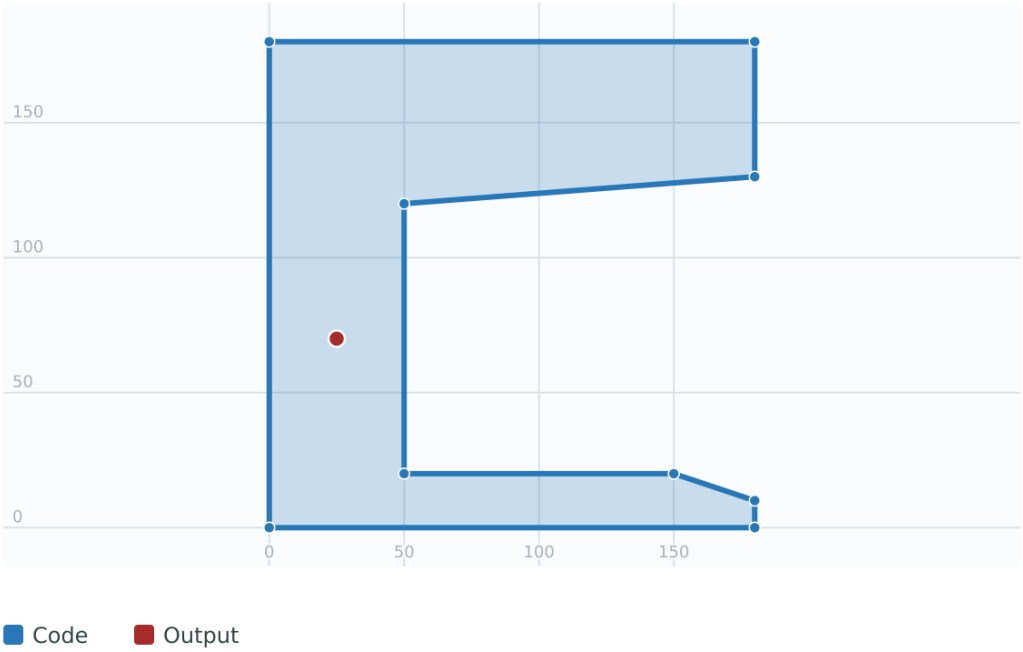
Code

```
SELECT ST_PointOnSurface(
  'POLYGON ((190 190,10 190,10 10,190 10,190 20,160 30,60 30,60 130,190 140,190 190))');
```

Output

```
POINT(35 80)
```

Figure



Point on surface of the highest-dimension components of a GeometryCollection.

Code

```
SELECT ST_PointOnSurface(
  'GEOMETRYCOLLECTION (POLYGON ((190 170,180 100,80 140,80 160,130 160,110 180,110 190,180 ↩
   180,190 170)),
   LINESTRING (80 120,120 20,140 70,150 30,180 50,190 10),
   MULTIPOINT (19 150,22 49,30 13,32 101,45 35,67 88,75 16))');
```

Output

```
POINT(133.571 150)
```

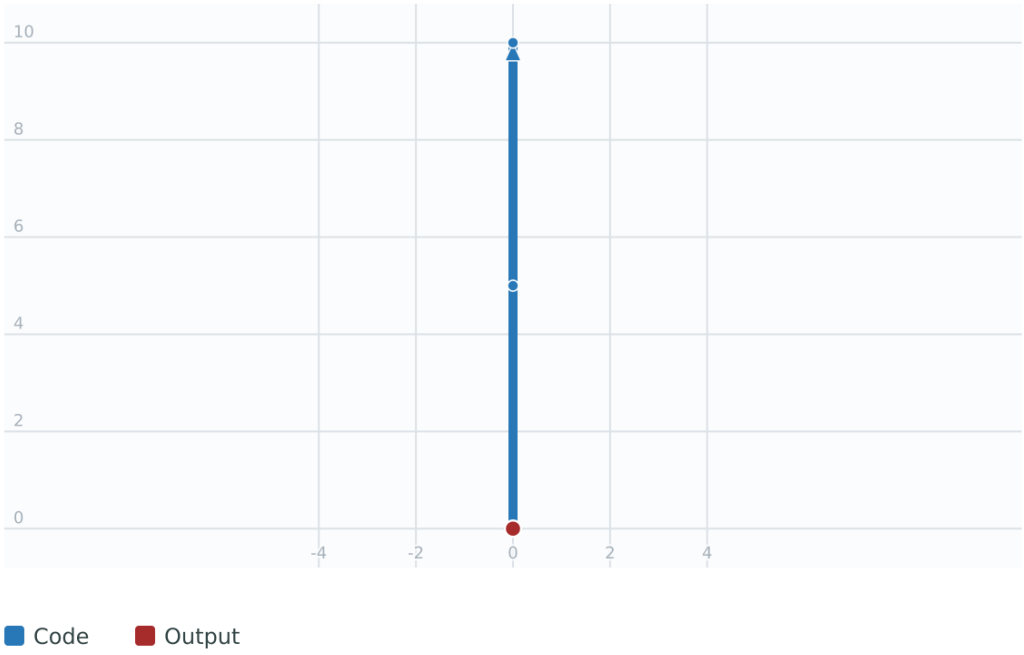
Figure



Code

```
SELECT ST_PointOnSurface('LINESTRING(0 5 1,0 0 1,0 10 2)');
```

Figure

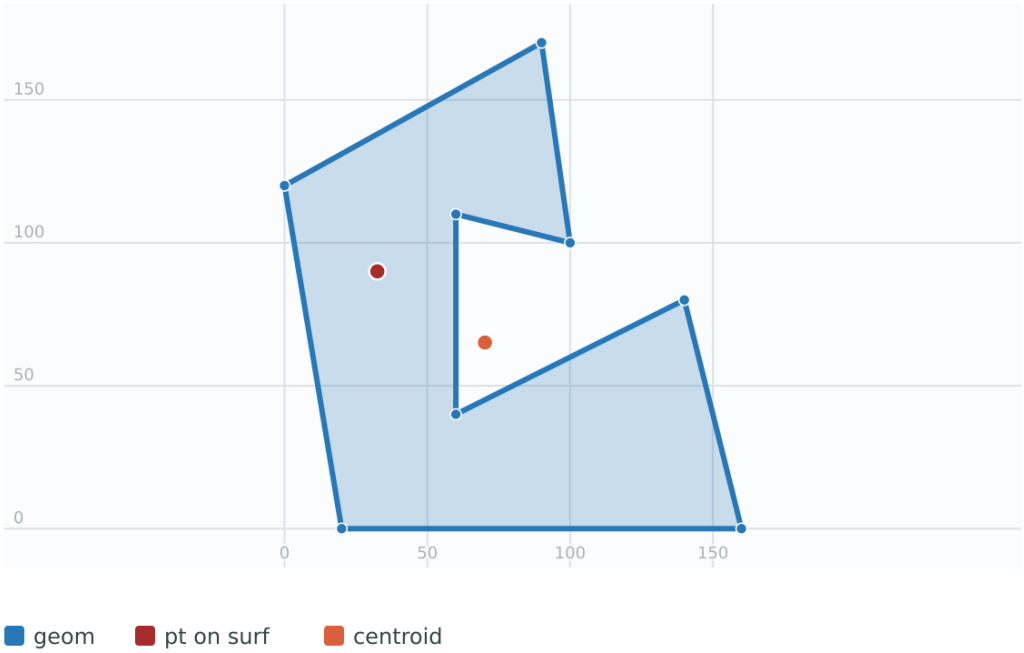


Example: The result of ST_PointOnSurface is guaranteed to lie within polygons, whereas the point computed by ST_Centroid may be outside.

Code

```
SELECT ST_PointOnSurface(geom) AS pt_on_surf, ST_Centroid(geom) AS centroid
    FROM (SELECT 'POLYGON ((130 120,120 190,30 140,50 20,190 20,
                        170 100,90 60,90 130,130 120))'::geometry AS geom) AS t;
```

Figure



□□

ST_Centroid, ST_MaximumInscribedCircle

7.14.20 ST_Polygonize

ST_Polygonize — Computes a collection of polygons formed from the linework of a set of geometries.

□□

geometry **ST_Polygonize**(geometry set geomfield);
geometry **ST_Polygonize**(geometry[] geom_array);

□□

Creates a GeometryCollection containing the polygons formed by the linework of a set of geometries. If the input linework does not form any polygons, an empty GeometryCollection is returned.

This function creates polygons covering all delimited areas. If the result is intended to form a valid polygonal geometry, use ST_BuildArea to prevent holes being filled.



□□
The input linework must be correctly noded for this function to work properly. To ensure input is noded use ST_Node on the input geometry before polygonizing.



□□
GeometryCollections can be difficult to handle with external tools. Use ST_Dump to convert the polygonized result into separate polygons.

GEOS □□□□□
1.0.0RC1 □□□□□□□□□□□□.

□□

Polygonize a set of input lines.

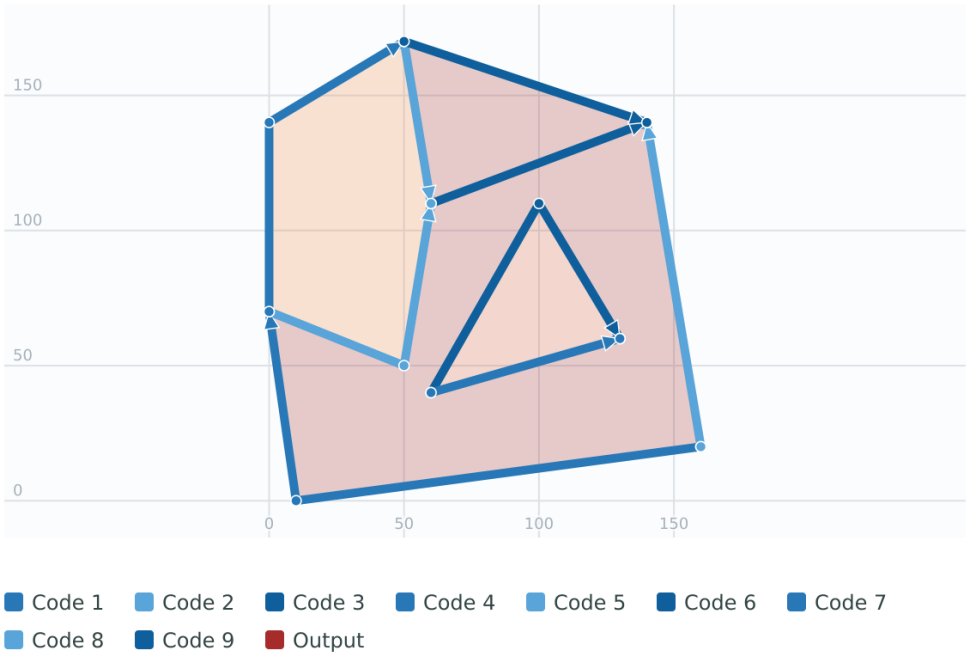
Code

```
WITH data(geom) AS (
  VALUES
    ('LINESTRING (180 40,30 20,20 90)'::geometry),
    ('LINESTRING (180 40,160 160)'::geometry),
    ('LINESTRING (80 60,120 130,150 80)'::geometry),
    ('LINESTRING (80 60,150 80)'::geometry),
    ('LINESTRING (20 90,70 70,80 130)'::geometry),
    ('LINESTRING (80 130,160 160)'::geometry),
    ('LINESTRING (20 90,20 160,70 190)'::geometry),
    ('LINESTRING (70 190,80 130)'::geometry),
    ('LINESTRING (70 190,160 160)'::geometry)
)
SELECT ST_Polygonize(geom)
FROM data;
```

Output

```
GEOMETRYCOLLECTION(POLYGON((180 40,30 20,20 90,70 70,80 130,160 160,180 40),(150 80,120 130,80 60,150 80)), POLYGON ↩
    ((80 60,120 130,150 80,80 60)), POLYGON((80 130,70 70,20 90,20 160,70 190,80 130)), POLYGON((160 160,80 130,70  ↩
    190,160 160)))
```

Figure

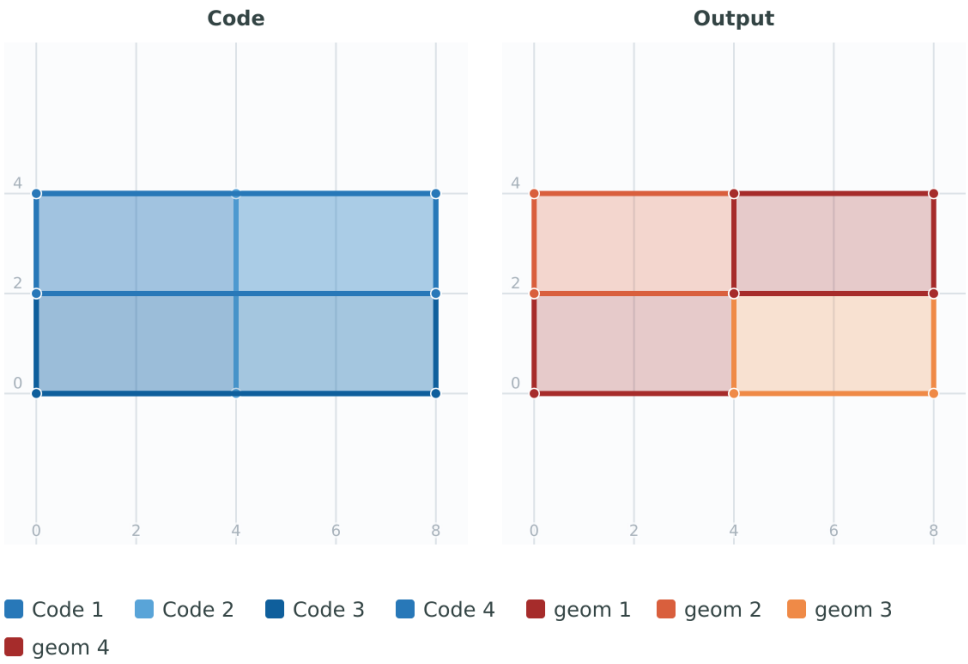


Polygonizing two polygon coverages and transferring attributes to the resultants:

Code

```
WITH poly_a(id, geom) AS (VALUES
  ('a1', 'POLYGON ((0 0, 4 0, 4 4, 0 4, 0 0))'::geometry),
  ('a2', 'POLYGON ((4 0, 8 0, 8 4, 4 4, 4 0))'::geometry)
),
poly_b(id, geom) AS (VALUES
  ('b1', 'POLYGON ((0 0, 8 0, 8 2, 0 2, 0 0))'::geometry),
  ('b2', 'POLYGON ((0 2, 8 2, 8 4, 0 4, 0 2))'::geometry)
),
lines AS (
  SELECT ST_Boundary(geom) AS geom FROM poly_a
  UNION ALL
  SELECT ST_Boundary(geom) FROM poly_b
),
noded AS (
  SELECT ST_UnaryUnion(ST_Collect(geom)) AS geom FROM lines
),
cells AS (
  SELECT (ST_Dump(ST_Polygonize(geom))).geom AS geom FROM noded
)
SELECT a.id AS id_a,
       b.id AS id_b,
       c.geom AS geom
FROM cells AS c
LEFT JOIN poly_a AS a ON ST_Within(ST_PointOnSurface(c.geom), a.geom)
LEFT JOIN poly_b AS b ON ST_Within(ST_PointOnSurface(c.geom), b.geom)
WHERE a.id IS NOT NULL OR b.id IS NOT NULL
ORDER BY id_a, id_b;
```

Figure



Polygonizing a table of linestrings:

Code

```
SELECT ST_Polygonize(geom_4269) As geomtextrep
FROM (SELECT geom_4269 FROM ma.suffolk_edges) As foo;
```

Output

```
geomtextrep
-------------
 SRID=4269;GEOMETRYCOLLECTION(POLYGON((-71.040878 42.285678,-71.040943 42.2856,-71.04096  ↩
    42.285752,-71.040878 42.285678)),
 POLYGON((-71.17166 42.353675,-71.172026 42.354044,-71.17239 42.354358,-71.171794  ↩
    42.354971,-71.170511 42.354855,
 -71.17112 42.354238,-71.17166 42.353675)))
(1 row)
```

Use ST_Dump to dump out the polygonize geoms into individual polygons.

Code

```
SELECT (ST_Dump(t.polycoll)).geom AS geomtextrep
FROM (SELECT ST_Polygonize(geom_4269) AS polycoll
    FROM (SELECT geom_4269 FROM ma.suffolk_edges)
        As foo) AS t;
```

Output

```
geomtextrep
-------------
 SRID=4269;POLYGON((-71.040878 42.285678,-71.040943 42.2856,-71.04096 42.285752,
-71.040878 42.285678))
 SRID=4269;POLYGON((-71.17166 42.353675,-71.172026 42.354044,-71.17239 42.354358,
-71.171794 42.354971,-71.170511 42.354855,-71.17112 42.354238,-71.17166 42.353675))
(2 rows)
```

□□

ST_BuildArea, ST_Boundary, ST_Collect, ST_Dump, ST_Node, ST_PointOnSurface, ST_UnaryUnion, ST_Within

7.14.21 ST_ReducePrecision

ST_ReducePrecision — Returns a valid geometry with points rounded to a grid tolerance.

□□

geometry **ST_ReducePrecision**(geometry g, float8 gridsize);

□□

Returns a valid geometry with all points rounded to the provided grid tolerance, and features below the tolerance removed.

Unlike ST_SnapToGrid the returned geometry will be valid, with no ring self-intersections or collapsed components.

Precision reduction can be used to:

- match coordinate precision to the data accuracy
- reduce the number of coordinates needed to represent a geometry
- ensure valid geometry output to formats which use lower precision (e.g. text formats such as WKT, GeoJSON or KML when the number of output decimal places is limited).
- export valid geometry to systems which use lower or limited precision (e.g. SDE, Oracle tolerance value)

Availability: 3.1.0.

Requires GEOS >= 3.9.0.

□□

Code

```
SELECT ST_ReducePrecision('POINT(1.412 19.323)', 0.1);
```

Output

```
POINT(1.4 19.3)
```

Figure



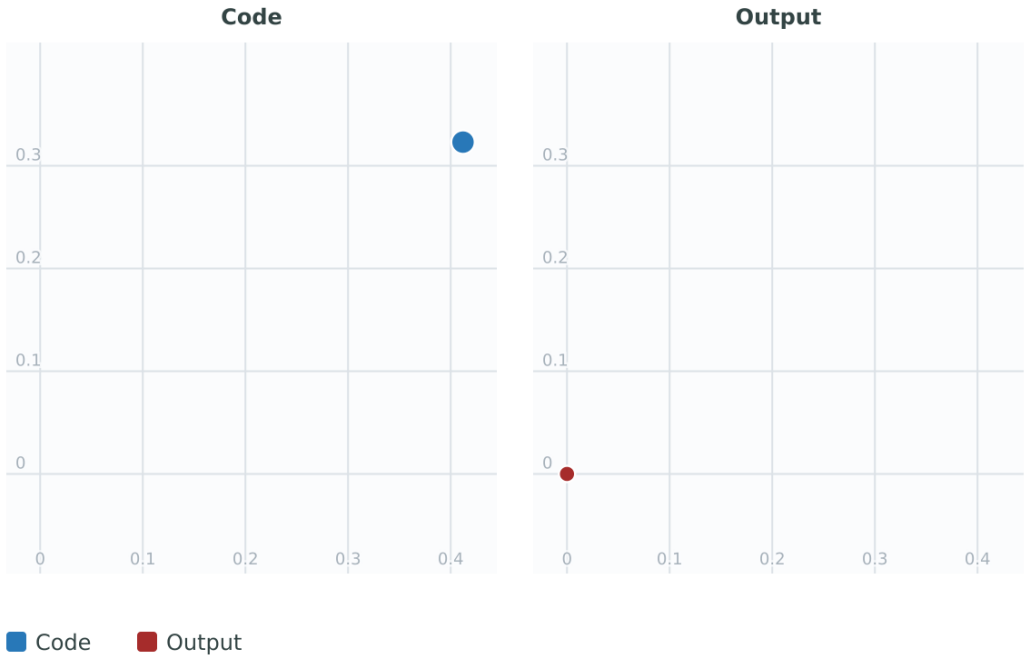
Code

```
SELECT ST_ReducePrecision('POINT(1.412 19.323)', 1.0);
```

Output

```
POINT(1 19)
```

Figure



Code

```
SELECT ST_ReducePrecision('POINT(1.412 19.323)', 10);
```

Output

```
POINT(0 20)
```

Figure



Precision reduction can reduce number of vertices

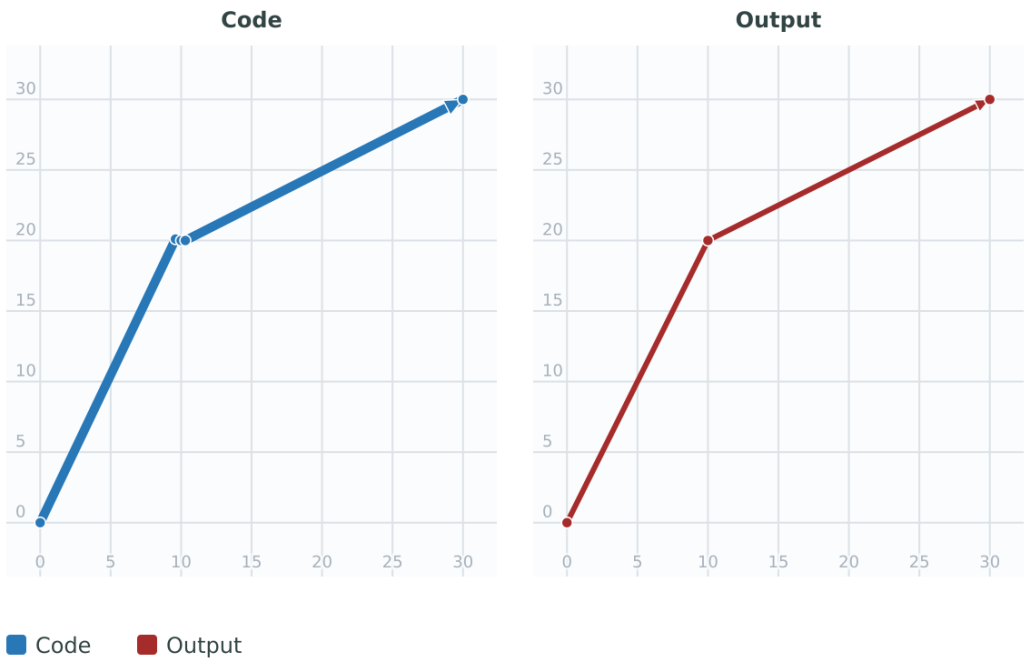
Code

```
SELECT ST_ReducePrecision('LINESTRING (10 10,19.6 30.1,20 30,20.3 30,40 40)', 1);
```

Output

```
LINESTRING (10 10, 20 30, 40 40)
```

Figure



Precision reduction splits polygons if needed to ensure validity

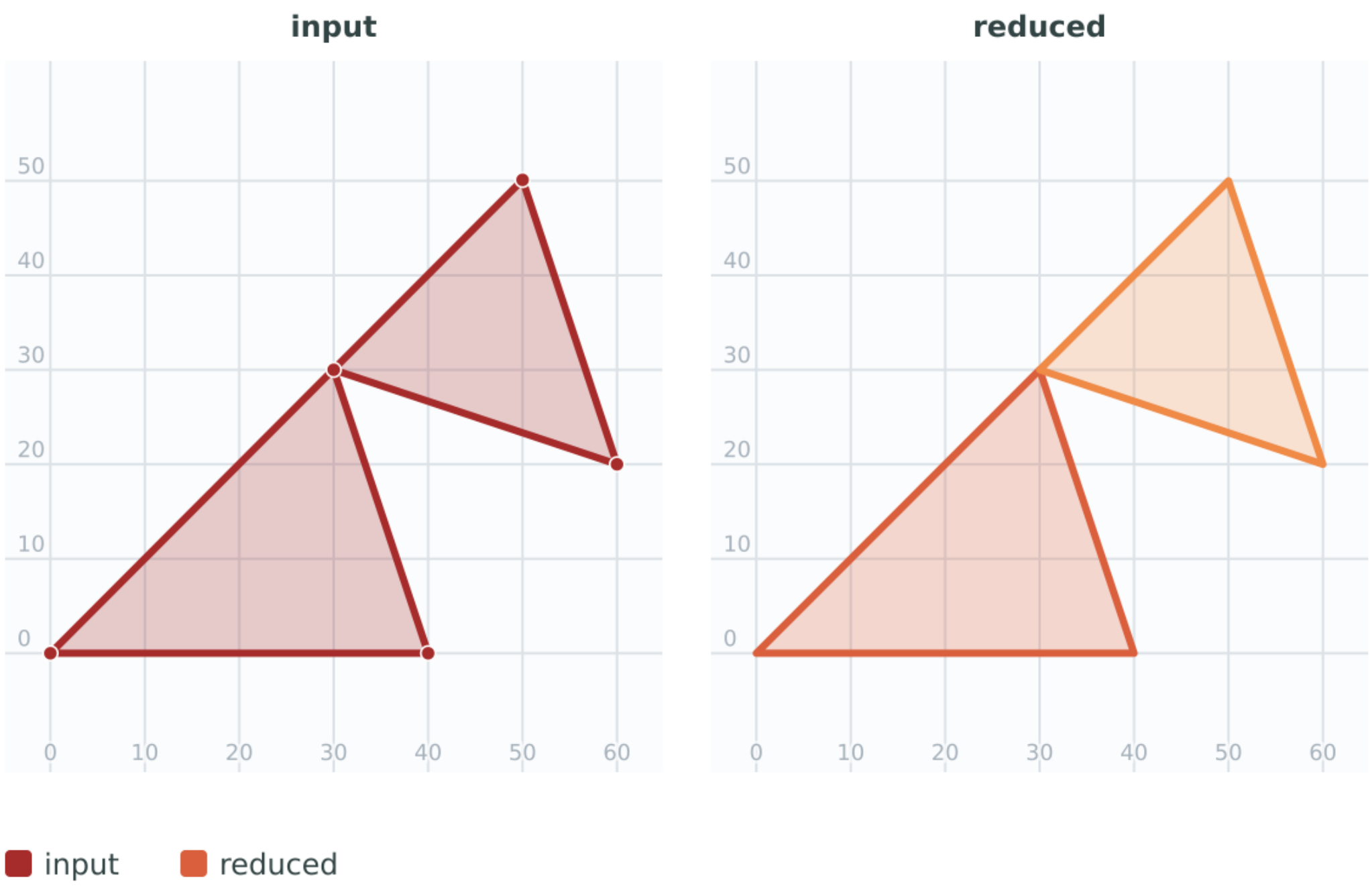
Code

```
WITH data(geom) AS (
  VALUES ('POLYGON ((10 10,60 60.1,70 30,40 40,50 10,10 10))'::geometry)
)
SELECT geom AS input,
       ST_ReducePrecision(geom, 10) AS reduced
FROM data;
```

Output

```
-[ RECORD 1 ]-----------------------------------------------
input   | POLYGON((10 10,60 60.1,70 30,40 40,50 10,10 10))
reduced | MULTIPOLYGON(((10 10,40 40,50 10,10 10)),((40 40,60 60,70 30,40 40)))
```

Figure



□□

ST_SnapToGrid, ST_Simplify, ST_SimplifyVW

7.14.22 ST_SharedPaths

ST_SharedPaths — □□□□□□□□/□□□□□□□□□□□□□□□□□□□□□□□□□□□.

□□

geometry **ST_SharedPaths**(geometry lineal1, geometry lineal2);

□□

□□□□□□□□□□□□□□□□□□□□□□□□□. □□□□□□□□□□□□□□□□□□□□□□□, □ □□□□□□□□□□□□□□□□□□. □□□□□□□□□□□□□□□□□□□.

GEOS □□□□□

2.0.0 □□□□□□□□□□□□.

□□: □□□□□□□

Adjacent shared path segments may be returned as separate or merged LineStrings depending on the GEOS version.

Find the shared paths of a MultiLineString and a LineString.

Code

```
SELECT ST_SharedPaths(ST_GeomFromText('MULTILINESTRING((26 125,26 200,126 200,126 125,26 ↩
    125),
       (51 150,101 150,76 175,51 150))'),
   ST_GeomFromText('LINESTRING(151 100,126 156.25,126 125,101 150,76 175)')
   )
```

Output

```
GEOMETRYCOLLECTION(MULTILINESTRING((126 156.25,126 125),
(101 150,76 175)),MULTILINESTRING EMPTY)
```

Figure



Flip the LineString orientation to return the same paths in the opposite-direction member.

Code

```
SELECT ST_SharedPaths(ST_GeomFromText('LINESTRING(76 175,101 150,126 125,126 156.25,151 ↩
    100)'),
   ST_GeomFromText('MULTILINESTRING((26 125,26 200,126 200,126 125,26 125),
       (51 150,101 150,76 175,51 150))')
    )
```

Output

```
GEOMETRYCOLLECTION(MULTILINESTRING EMPTY,
MULTILINESTRING((76 175,101 150),(126 125,126 156.25)))
```

Figure



☒☒

ST_Dump, ST_GeometryN, ST_NumGeometries

7.14.23 ST_Simplify

ST_Simplify — Returns a simplified representation of a geometry, using the Douglas-Peucker algorithm.

☒☒

geometry **ST_Simplify**(geometry geom, float tolerance);
geometry **ST_Simplify**(geometry geom, float tolerance, boolean preserveCollapsed);

☒☒

Computes a simplified representation of a geometry using the Douglas-Peucker algorithm. The simplification `tolerance` is a distance value, in the units of the input SRS. Simplification removes vertices which are within the tolerance distance of the simplified linework. The result may not be valid even if the input is.

The function can be called with any kind of geometry (including GeometryCollections), but only line and polygon elements are simplified. Endpoints of linear geometry are preserved.

The `preserveCollapsed` flag retains small geometries that would otherwise be removed at the given tolerance. For example, if a 1m long line is simplified with a 10m tolerance, when `preserveCollapsed` is true the line will not disappear. This flag is useful for rendering purposes, to prevent very small features disappearing from a map.



☒☒
The returned geometry may lose its simplicity (see ST_IsSimple), topology may not be preserved, and polygonal results may be invalid (see ST_IsValid). Use ST_SimplifyPreserveTopology to preserve topology and ensure validity.



☒☒
This function does not preserve boundaries shared between polygons. Use ST_CoverageSimplify if this is required.

1.2.2 ☒☒☒☒☒☒☒☒☒☒☒☒.

☒☒

A circle simplified with a medium tolerance becomes an octagon, and with a large tolerance disappears.

Code

```
SELECT ST_NPoints(geom) AS np_before,
     ST_NPoints(ST_Simplify(geom, 0.1)) AS np01_notbadcircle,
     ST_NPoints(ST_Simplify(geom, 0.5)) AS np05_notquitecircle,
     ST_NPoints(ST_Simplify(geom, 1)) AS np1_octagon,
     ST_NPoints(ST_Simplify(geom, 10)) AS np10_gone,
     (ST_Simplify(geom, 100) is null) AS  np100_geometrygoesaway
FROM (SELECT ST_Buffer('POINT(1 3)', 10, 12) As geom) AS t;
```

Output

```
-[ RECORD 1 ]------------
np_before                | 49
np01_notbadcircle        | 33
np05_notquitecircle      | 17
np1_octagon              | 9
np10_gone                | null
np100_geometrygoesaway   | t
```

Simplifying a set of lines. Lines may intersect after simplification.

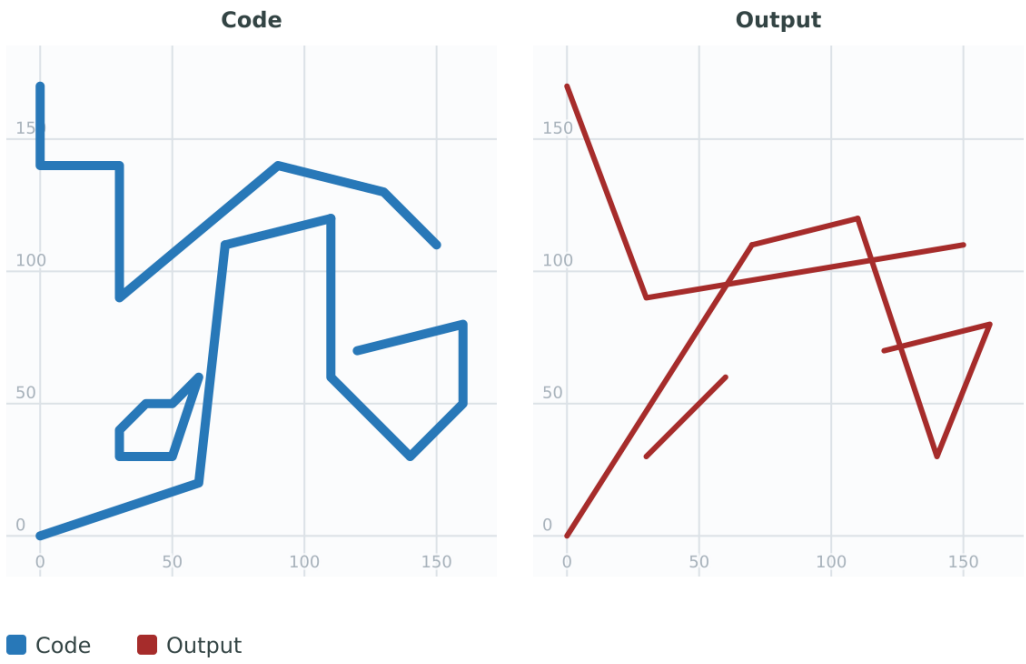
Code

```
SELECT ST_Simplify('MULTILINESTRING ((20 180,20 150,50 150,50 100,110 150,150 140,170 120) ↩
    ,(20 10,80 30,90 120),(90 120,130 130),(130 130,130 70,160 40,180 60,180 90,140 80),(50 ↩
    40,70 40,80 70,70 60,60 60,50 50,50 40))'::geometry,
40);
```

Output

```
MULTILINESTRING((20 180,50 100,170 120),(20 10,90 120),(90 120,130 130),(130 130,160 40,180 90,140 80),(50 40,80 70,50 ↩
     40))
```

Figure



Simplifying a MultiPolygon. Polygonal results may be invalid.

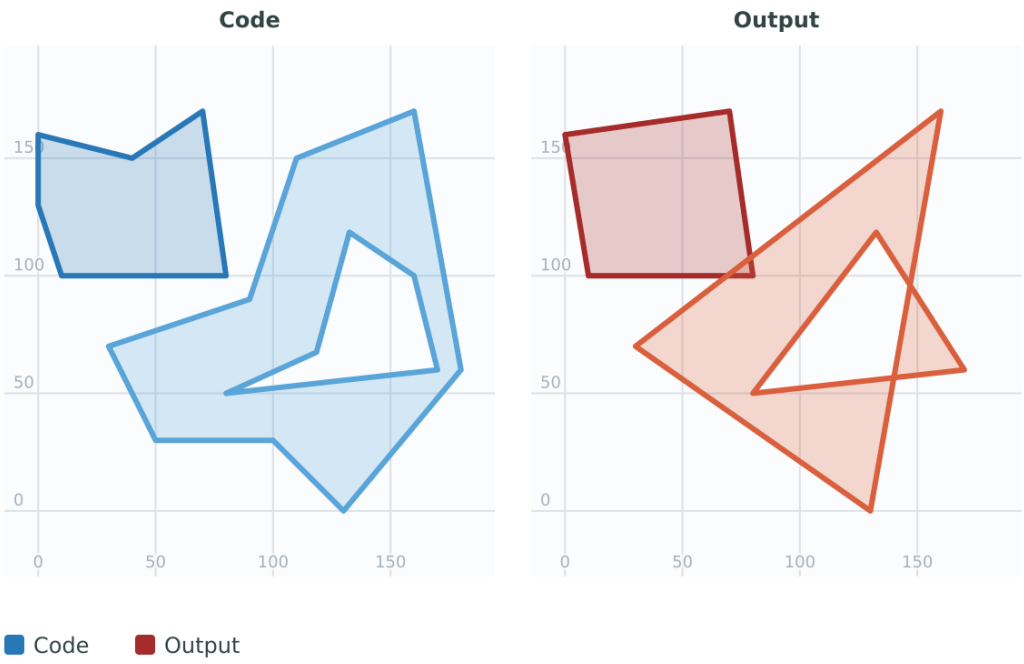
Code

```
SELECT ST_Simplify('MULTIPOLYGON (((90 110,80 180,50 160,10 170,10 140,20 110,90 110)),((40 ↩
    80,100 100,120 160,170 180,190 70,140 10,110 40,60 40,40 80),(180 70,170 110,142.5 ↩
   128.5,128.5 77.5,90 60,180 70)))'::geometry,
40);
```

Output

```
MULTIPOLYGON(((90 110,80 180,10 170,20 110,90 110)),((40 80,170 180,140 10,40 80),(180 70,142.5 128.5,90 60,180 70)))
```

Figure



☒☒

ST_IsSimple, ST_SimplifyPreserveTopology, ST_SimplifyVW, ST_CoverageSimplify, Topology ST_Simplify

7.14.24 ST_SimplifyPreserveTopology

ST_SimplifyPreserveTopology — Returns a simplified and valid representation of a geometry, using the Douglas-Peucker algorithm.

☒☒

geometry **ST_SimplifyPreserveTopology**(geometry geom, float tolerance);

☒☒

Computes a simplified representation of a geometry using a variant of the Douglas-Peucker algorithm which limits simplification to ensure the result has the same topology as the input. The simplification `tolerance` is a distance value, in the units of the input SRS. Simplification removes vertices which are within the tolerance distance of the simplified linework, as long as topology is preserved. The result will be valid and simple if the input is.

The function can be called with any kind of geometry (including GeometryCollections), but only line and polygon elements are simplified. For polygonal inputs, the result will have the same number of rings (shells and holes), and the rings will not cross. Ring endpoints may be simplified. For linear inputs, the result will have the same number of lines, and lines will not intersect if they did not do so in the original geometry. Endpoints of linear geometry are preserved.



☒☒
This function does not preserve boundaries shared between polygons. Use ST_CoverageSimplify if this is required.

GEOS ☒☒☒☒☒
1.3.3 ☒☒☒☒☒☒☒☒☒☒☒☒.

☒☒

For the same example as ST_Simplify, ST_SimplifyPreserveTopology prevents the geometry from disappearing.

Code

```
SELECT  ST_NPoints(geom) AS np_before,
        ST_NPoints(ST_SimplifyPreserveTopology(geom, 0.1)) AS np01_notbadcircle,
        ST_NPoints(ST_SimplifyPreserveTopology(geom, 0.5)) AS np05_notquitecircle,
        ST_NPoints(ST_SimplifyPreserveTopology(geom, 1))   AS np1_octagon,
        ST_NPoints(ST_SimplifyPreserveTopology(geom, 10))  AS np10_minimal,
        ST_NPoints(ST_SimplifyPreserveTopology(geom, 100)) AS np100_minimal
FROM (SELECT ST_Buffer('POINT(1 3)', 10, 12) AS geom) AS t;
```

Output

```
np_before | np01_notbadcircle | np05_notquitecircle | np1_octagon | np10_minimal |  ↩
    np100_minimal
-----------+-------------------+---------------------+-------------+--------------+--------------- ←

        49 |                33 |                  17 |           9 |            4 |         ↩
         4
```

Simplifying a set of lines, preserving topology of non-intersecting lines.

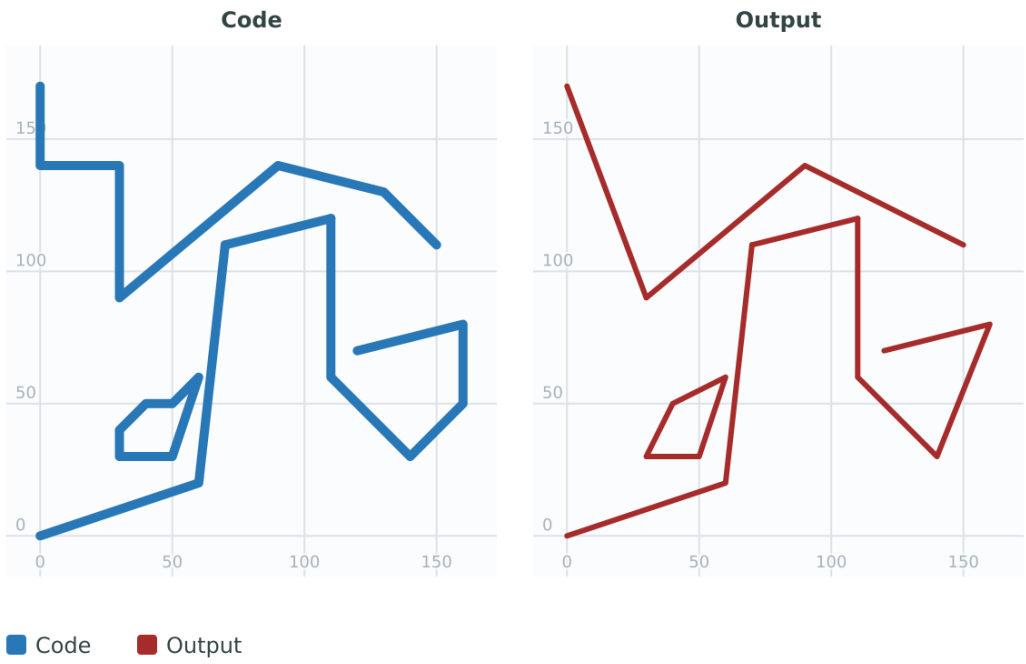
Code

```
SELECT ST_SimplifyPreserveTopology('MULTILINESTRING ((20 180,20 150,50 150,50 100,110  ↩
    150,150 140,170 120),(20 10,80 30,90 120),(90 120,130 130),(130 130,130 70,160 40,180  ↩
    60,180 90,140 80),(50 40,70 40,80 70,70 60,60 60,50 50,50 40))'::geometry,
40);
```

Output

```
MULTILINESTRING((20 180,50 100,110 150,170 120),(20 10,80 30,90 120),(90 120,130 130),(130 130,130 70,160 40,180  ↩
    90,140 80),(50 40,70 40,80 70,60 60,50 40))
```

Figure



Simplifying a MultiPolygon, preserving topology of shells and holes.

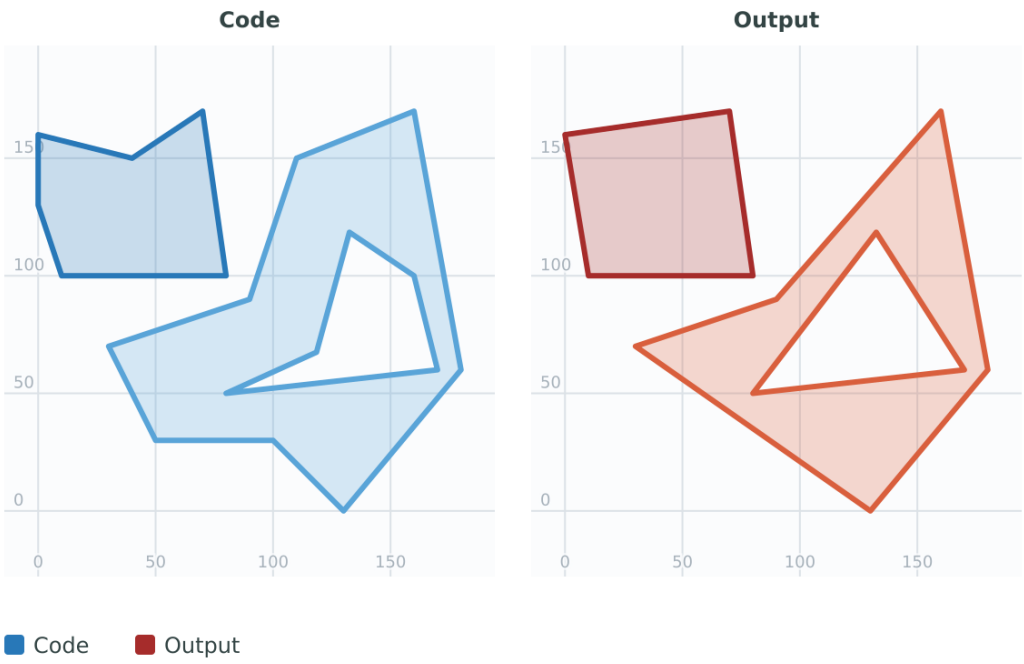
Code

```
SELECT ST_SimplifyPreserveTopology('MULTIPOLYGON (((90 110,80 180,50 160,10 170,10 140,20 ↩
    110,90 110)),((40 80,100 100,120 160,170 180,190 70,140 10,110 40,60 40,40 80),(180 ↩
    70,170 110,142.5 128.5,128.5 77.5,90 60,180 70)))'::geometry,
40);
```

Output

```
MULTIPOLYGON(((90 110,80 180,10 170,20 110,90 110)),((40 80,100 100,170 180,190 70,140 10,40 80),(180 70,142.5 ↩
    128.5,90 60,180 70)))
```

Figure



☒☒

ST_Simplify, ST_SimplifyVW, ST_CoverageSimplify

7.14.25 ST_SimplifyPolygonHull

ST_SimplifyPolygonHull — Computes a simplified topology-preserving outer or inner hull of a polygonal geometry.

☒☒

geometry **ST_SimplifyPolygonHull**(geometry param_geom, float vertex_fraction, boolean is_outer = true);

☒☒

Computes a simplified topology-preserving outer or inner hull of a polygonal geometry. An outer hull completely covers the input geometry. An inner hull is completely covered by the input geometry. The result is a polygonal geometry formed by a subset of the input vertices. MultiPolygons and holes are handled and produce a result with the same structure as the input.

The reduction in vertex count is controlled by the `vertex_fraction` parameter, which is a number in the range 0 to 1. Lower values produce simpler results, with smaller vertex count and less concaveness. For both outer and inner hulls a vertex fraction of 1.0 produces the original geometry. For outer hulls a value of 0.0 produces the convex hull (for a single polygon); for inner hulls it produces a triangle.

The simplification process operates by progressively removing concave corners that contain the least amount of area, until the vertex count target is reached. It prevents edges from crossing, so the result is always a valid polygonal geometry.

To get better results with geometries that contain relatively long line segments, it might be necessary to ”segmentize” the input, as shown below.

GEOS □□□□□

Availability: 3.3.0.

Requires GEOS >= 3.11.0.

□□

Outer hull of a Polygon.

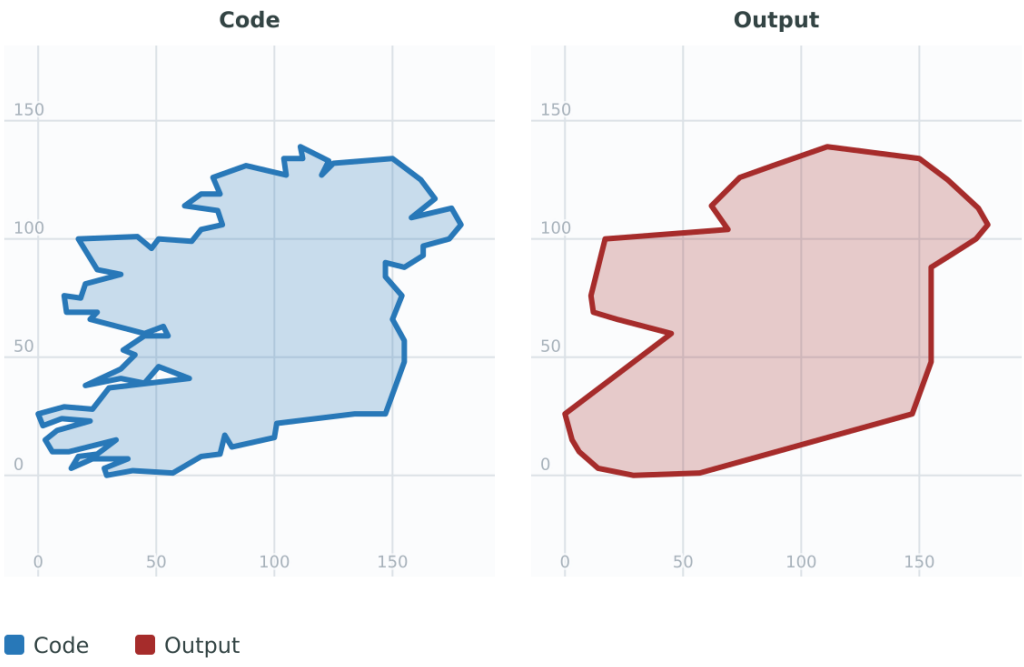
Code

```
SELECT ST_SimplifyPolygonHull(
  'POLYGON ((131 158,136 163,161 165,173 156,179 148,169 140,186 144,190 137,185 131,174
    128,174 124,166 119,158 121,158 115,165 107,161 97,166 88,166 79,158 57,145 57,112
    53,111 47,93 43,90 48,88 40,80 39,68 32,51 33,40 31,39 34,49 38,34 38,25 34,28 39,36
    40,44 46,24 41,17 41,14 46,19 50,33 54,21 55,13 52,11 57,22 60,34 59,41 68,75 72,62
    77,56 70,46 72,31 69,46 76,52 82,47 84,56 90,66 90,64 94,56 91,33 97,36 100,23 100,22
    107,29 106,31 112,46 116,36 118,28 131,53 132,59 127,62 131,76 130,80 135,89 137,87
    143,73 145,80 150,88 150,85 157,99 162,116 158,115 165,123 165,122 170,134 164,131 158))
    '::geometry,
  0.3
);
```

Output

```
POLYGON((161 165,173 156,186 144,190 137,185 131,174 124,166 119,166 79,158 57,68 32,40 31,25 34,17 41,14 46,11 57,56
    91,33 97,23 100,22 107,28 131,80 135,73 145,85 157,99 162,122 170,161 165))
```

Figure



Inner hull of a Polygon.

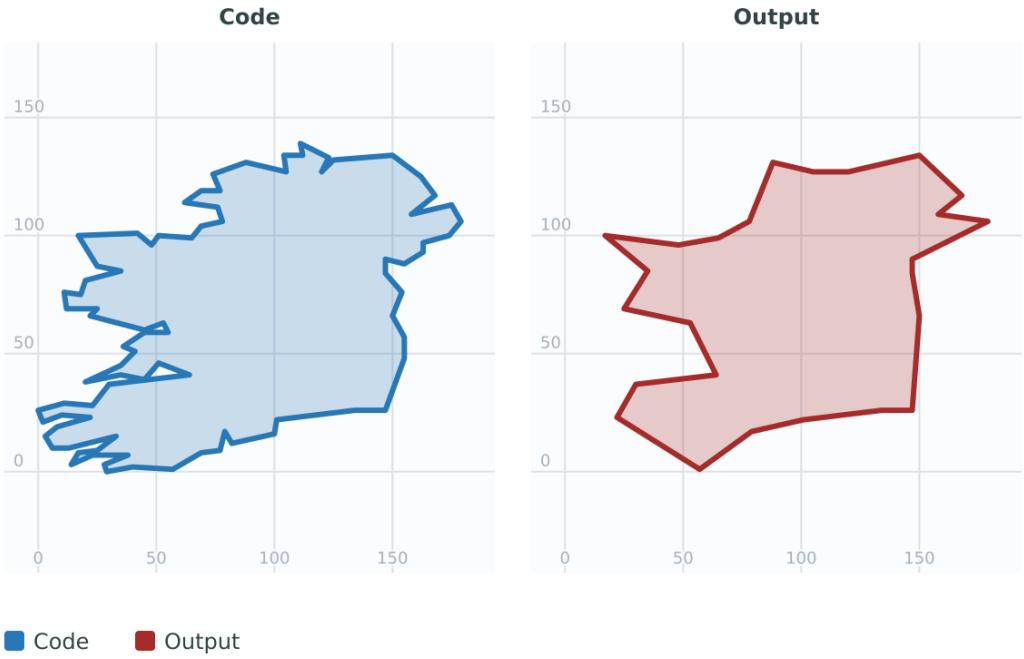
Code

```
SELECT ST_SimplifyPolygonHull(
  'POLYGON ((131 158,136 163,161 165,173 156,179 148,169 140,186 144,190 137,185 131,174 ↩
    128,174 124,166 119,158 121,158 115,165 107,161 97,166 88,166 79,158 57,145 57,112 ↩
    53,111 47,93 43,90 48,88 40,80 39,68 32,51 33,40 31,39 34,49 38,34 38,25 34,28 39,36 ↩
    40,44 46,24 41,17 41,14 46,19 50,33 54,21 55,13 52,11 57,22 60,34 59,41 68,75 72,62 ↩
    77,56 70,46 72,31 69,46 76,52 82,47 84,56 90,66 90,64 94,56 91,33 97,36 100,23 100,22 ↩
    107,29 106,31 112,46 116,36 118,28 131,53 132,59 127,62 131,76 130,80 135,89 137,87 ↩
    143,73 145,80 150,88 150,85 157,99 162,116 158,115 165,123 165,122 170,134 164,131 158)) ↩
    '::geometry,
  0.3, false
);
```

Output

```
POLYGON((131 158,116 158,99 162,89 137,76 130,59 127,28 131,46 116,36 100,64 94,75 72,41 68,33 54,68 32,90 48,112 ↩
    53,145 57,158 57,161 97,158 115,158 121,190 137,169 140,179 148,161 165,131 158))
```

Figure

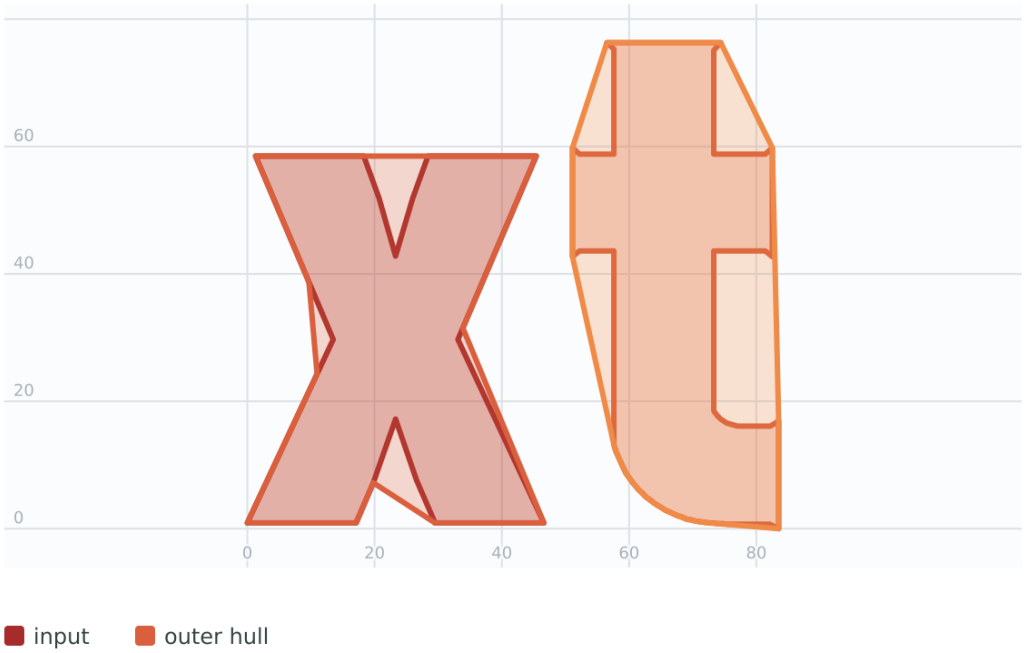


Outer hull simplification of a MultiPolygon, with segmentization.

Code

```
WITH letters AS (
  SELECT ST_Letters('xt') AS geom
)
SELECT geom AS input,
  ST_SimplifyPolygonHull(ST_Segmentize(geom, 2.0), 0.1) AS outer_hull
FROM letters;
```

Figure



□□

ST_ConvexHull, ST_SimplifyVW, ST_ConcaveHull, ST_Segmentize

7.14.26 ST_SimplifyVW

ST_SimplifyVW — Returns a simplified representation of a geometry, using the Visvalingam-Whyatt algorithm

□□

geometry **ST_SimplifyVW**(geometry geom, float tolerance);

□□

Returns a simplified representation of a geometry using the Visvalingam-Whyatt algorithm. The simplification `tolerance` is an area value, in squared units of the input SRS. Simplification removes vertices which form ”corners” with area less than the tolerance. The result may not be valid even if the input is.

The function can be called with any kind of geometry (including GeometryCollections), but only line and polygon elements are simplified. Endpoints of linear geometry are preserved.



□□
The returned geometry may lose its simplicity (see ST_IsSimple), topology may not be preserved, and polygonal results may be invalid (see ST_IsValid). Use ST_SimplifyPreserveTopology to preserve topology and ensure validity. ST_CoverageSimplify also preserves topology and validity.

This is a general-purpose simplification function. It may remove narrow polygon spikes when their effective area is below the tolerance, but it is not a dedicated spike-removal or polygon-repair operation. Use ST_IsValid to check the result and ST_MakeValid to repair invalid polygonal output if needed.

This function does not preserve boundaries shared between polygons. Use ST_CoverageSimplify if this is required.

□□□□ 3 □□□□□□□□, □□□□□□□□□□□□□□□□□□□□□.

2.2.0 □□□□□□□□□□□□.

□□

This polygon has a narrow spike whose effective area is small enough to be removed by a tolerance of 20 square units.

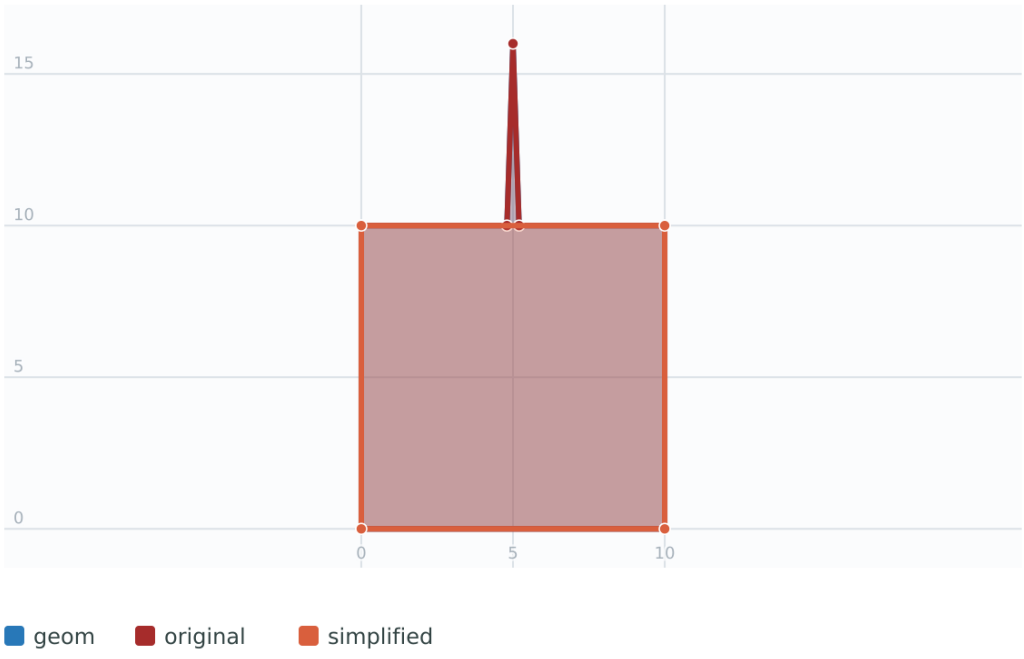
Code

```
SELECT geom AS original, ST_SimplifyVW(geom, 20) AS simplified
FROM (SELECT 'POLYGON((0 0,10 0,10 10,5.2 10,5 16,4.8 10,0 10,0 0))'::geometry AS geom) AS ↩
    t;
```

Output

```
POLYGON((0 0,10 0,10 10,5.2 10,5 16,4.8 10,0 10,0 0)) | POLYGON((0 0,10 0,10 10,0 10,0 0))
```

Figure



A LineString is simplified with a minimum-area tolerance of 30. For a linear simplification using ST_Simplify, a comparable tolerance would be a length in input-SRS units, so the value here is roughly that linear tolerance squared.

Code

```
SELECT ST_SimplifyVW(geom, 30) simplified
  FROM (SELECT 'LINESTRING(5 2,3 8,6 20,7 25,10 10)'::geometry AS geom) AS t;
```

Output

```
LINESTRING(5 2,7 25,10 10)
```

Figure



Simplifying a line.

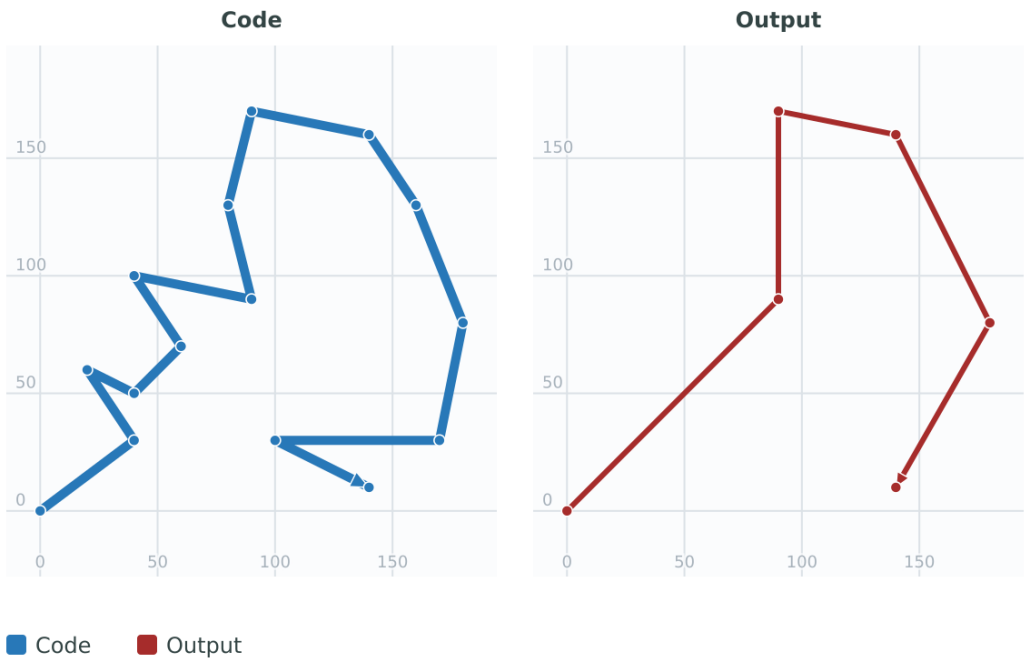
Code

```
SELECT ST_SimplifyVW('LINESTRING (10 10,50 40,30 70,50 60,70 80,50 110,100 100,90 140,100 ↩
    180,150 170,170 140,190 90,180 40,110 40,150 20)'::geometry,
1600);
```

Output

```
LINESTRING(10 10,100 100,100 180,150 170,190 90,150 20)
```

Figure



Simplifying a polygon.

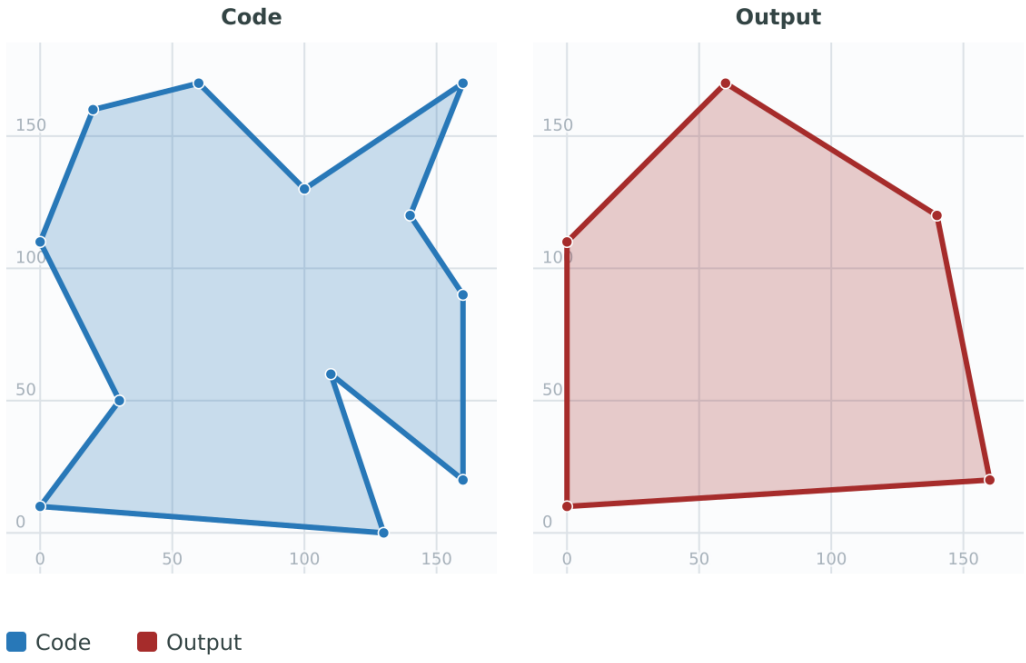
Code

```
SELECT ST_SimplifyVW('POLYGON ((20 20,50 60,20 120,40 170,80 180,120 140,180 180,160 ↩
    130,180 100,180 30,130 70,150 10,20 20))'::geometry,
1600);
```

Output

```
POLYGON((20 20,20 120,80 180,160 130,180 30,20 20))
```

Figure



⊠⊠

ST_SetEffectiveArea, ST_Simplify, ST_SimplifyPreserveTopology, ST_CoverageSimplify, Topology ST_Simpl

7.14.27 ST_SetEffectiveArea

ST_SetEffectiveArea — Sets the effective area for each vertex, using the Visvalingam-Whyatt algorithm.

⊠⊠

geometry **ST_SetEffectiveArea**(geometry geom, float threshold = 0, integer set_area = 1);

⊠⊠

⊠⊠⊠⊠⊠⊠-⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠ M ⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠” ⊠⊠” ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ 0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠ M ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠⊠⊠ [⊠⊠] ⊠⊠, [⊠⊠] ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

> **Note!**
> ⊠⊠
> ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (ST_IsSimple ⊠⊠).

> **Note!**
> ⊠⊠
> ⊠⊠ (topology) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠ ST_SimplifyPreserveTopology ⊠⊠⊠⊠⊠⊠⊠⊠⊠.

> **Note!**
> ⊠⊠
> ⊠⊠⊠⊠⊠ M ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

> **Note!**
> ⊠⊠
> ⊠⊠⊠⊠ 3 ⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

2.2.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠ 0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

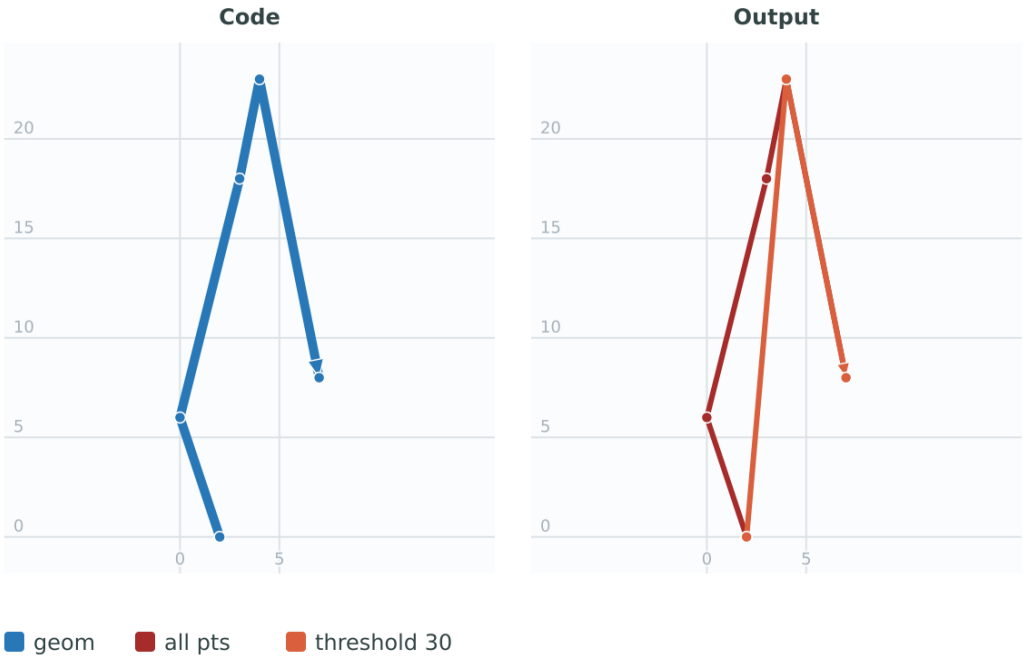
Code

```
SELECT ST_SetEffectiveArea(geom) AS all_pts,
       ST_SetEffectiveArea(geom, 30) AS threshold_30
FROM (SELECT  'LINESTRING(5 2,3 8,6 20,7 25,10 10)'::geometry geom) As foo;
```

Output

```
-[ RECORD 1 ]----------------
all_pts      | LINESTRING M (5 2 3.4e+38,3 8 29,6 20 1.5,7 25 49.5,10 10 3.4e+38)
threshold_30 | LINESTRING M (5 2 3.4e+38,7 25 49.5,10 10 3.4e+38)
```

Figure



☒☒

ST_SimplifyVW

7.14.28 ST_TriangulatePolygon

ST_TriangulatePolygon — Computes the constrained Delaunay triangulation of polygons

☒☒

geometry **ST_TriangulatePolygon**(geometry geom);

☒☒

Computes the constrained Delaunay triangulation of polygons. Holes and Multipolygons are supported.

The ”constrained Delaunay triangulation” of a polygon is a set of triangles formed from the vertices of the polygon, and covering it exactly, with the maximum total interior angle over all possible triangulations. It provides the ”best quality” triangulation of the polygon.

Availability: 3.3.0.

Requires GEOS >= 3.11.0.

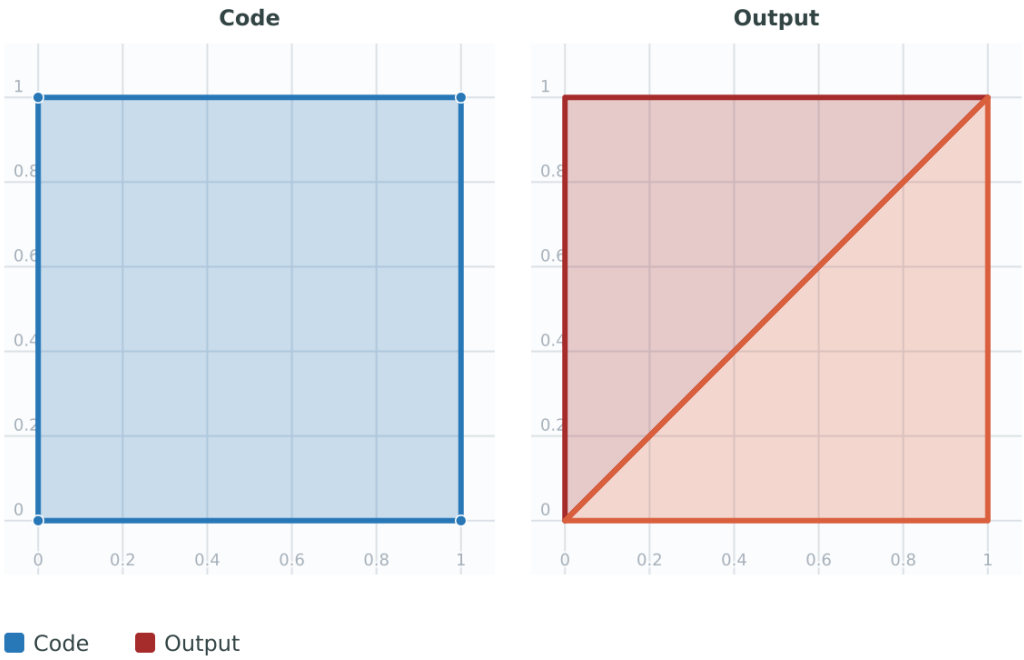
☒☒

Triangulation of a square.

Code

```
SELECT ST_TriangulatePolygon('POLYGON((0 0,0 1,1 1,1 0,0 0))');
```

Figure

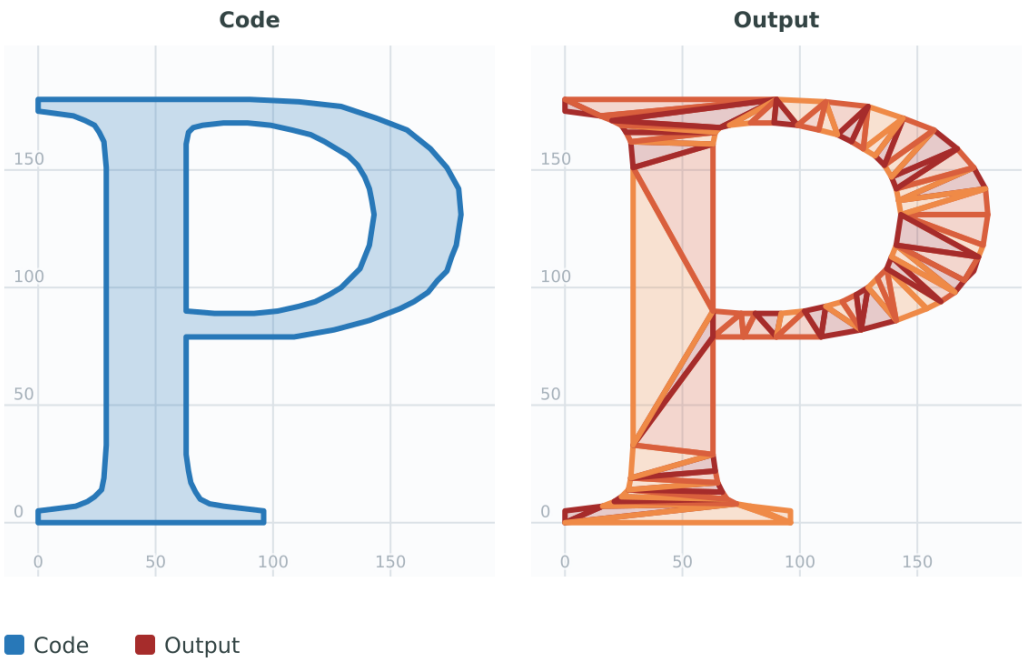


Triangulation of the letter P.

Code

```
SELECT ST_TriangulatePolygon('POLYGON ((26 17,31 19,34 21,37 24,38 29,39 43,39 161,38 ↩
    172,36 176,34 179,30 181,25 183,10 185,10 190,100 190,121 189,139 187,154 182,167 ↩
    177,177 169,184 161,189 152,190 141,188 128,186 123,184 117,180 113,176 108,170 104,164 ↩
    101,151 96,136 92,119 89,100 89,86 89,73 89,73 39,74 32,75 27,77 23,79 20,83 18,89 ↩
    17,106 15,106 10,10 10,10 15,26 17),(152 147,151 152,149 157,146 162,142 166,137 169,132 ↩
     172,126 175,118 177,109 179,99 180,89 180,80 179,76 178,74 176,73 171,73 100,85 99,91 ↩
    99,102 99,112 100,121 102,128 104,134 107,139 110,143 114,147 118,149 123,151 128,153 ↩
    141,152 147))'
);
```

Figure

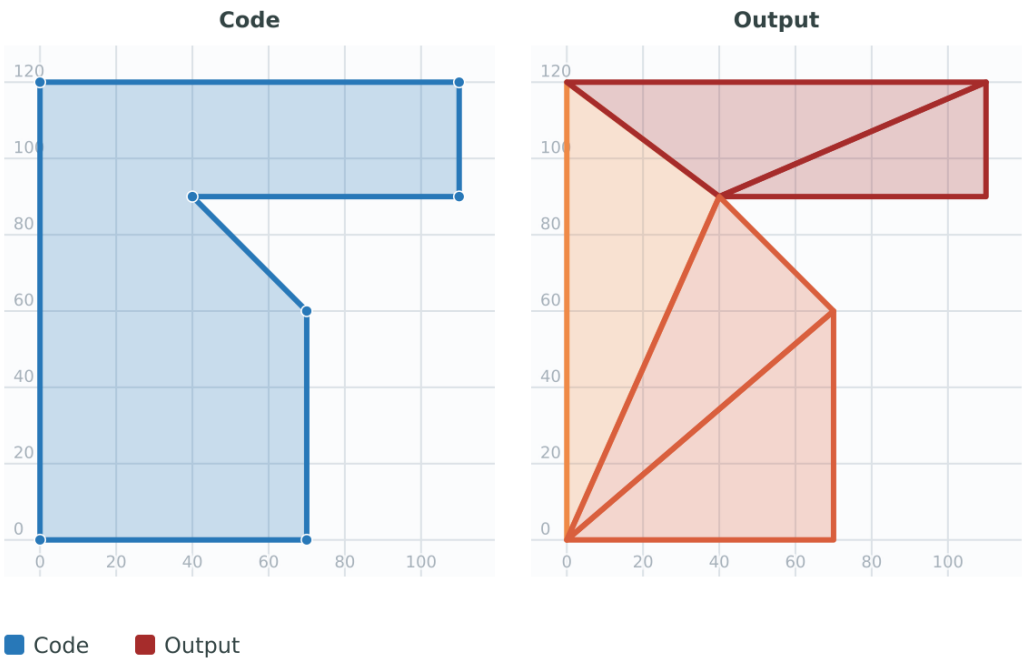


Triangulation of the polygon used in the CG_Tesselate example.

Code

```
SELECT ST_TriangulatePolygon('POLYGON (( 10 190,10 70,80 70,80 130,50 160,120 160,120 ↩
    190,10 190 ))'::geometry
);
```

Figure



□□

CG_ConstrainedDelaunayTriangles, ST_DelaunayTriangles, CG_Tesselate

7.14.29 ST_VoronoiLines

ST_VoronoiLines — Returns the boundaries of the Voronoi diagram of the vertices of a geometry.

□□

geometry **ST_VoronoiLines**(geometry geom , float8 tolerance = 0.0 , geometry extend_to = NULL);

□□

Computes a two-dimensional Voronoi diagram from the vertices of the supplied geometry and returns the boundaries between cells in the diagram as a MultiLineString. Returns null if input geometry is null. Returns an empty geometry collection if the input geometry contains only one vertex. Returns an empty geometry collection if the `extend_to` envelope has zero area.

□□□□□□□□□□:

- `tolerance`: The distance within which vertices will be considered equivalent. Robustness of the algorithm can be improved by supplying a nonzero tolerance distance. (default = 0.0)
- `extend_to`: If present, the diagram is extended to cover the envelope of the supplied geometry, unless smaller than the default envelope (default = NULL, default envelope is the bounding box of the input expanded by about 50%).

GEOS □□□□□

2.3.0 □□□□□□□□□□□.

□□

Voronoi diagram lines with a tolerance of 30 units.

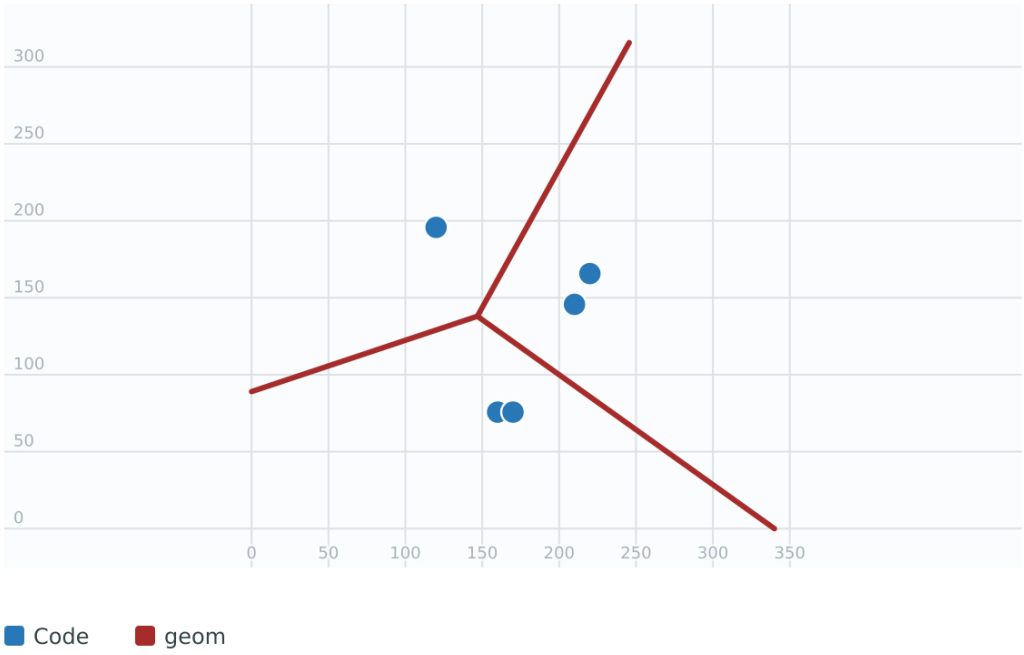
Code

```
SELECT ST_VoronoiLines('MULTIPOINT (50 30,60 30,100 100,10 150,110 120)'::geometry,
            30) AS geom;
```

Output

```
MULTILINESTRING((135.555555555556 270,36.8181818181818 92.2727272727273),(36.8181818181818 92.2727272727273,-110 ↩
    43.3333333333333),(230 -45.7142857142858,36.8181818181818 92.2727272727273))
```

Figure



□□

ST_DelaunayTriangles, ST_VoronoiPolygons

7.14.30 ST_VoronoiPolygons

ST_VoronoiPolygons — Returns the cells of the Voronoi diagram of the vertices of a geometry.

□□

geometry **ST_VoronoiPolygons**(geometry geom , float8 tolerance = 0.0 , geometry extend_to = NULL);

□□

Computes a two-dimensional Voronoi diagram from the vertices of the supplied geometry. The result is a GEOMETRYCOLLECTION of POLYGONs that covers an envelope larger than the extent of the input vertices. Returns null if input geometry is null. Returns an empty geometry collection if the input geometry contains only one vertex. Returns an empty geometry collection if the `extend_to` envelope has zero area.

□□□□□□□□□□:

- `tolerance`: The distance within which vertices will be considered equivalent. Robustness of the algorithm can be improved by supplying a nonzero tolerance distance. (default = 0.0)
- `extend_to`: If present, the diagram is extended to cover the envelope of the supplied geometry, unless smaller than the default envelope (default = NULL, default envelope is the bounding box of the input expanded by about 50%).

GEOS □□□□□
2.3.0 □□□□□□□□□□□□.

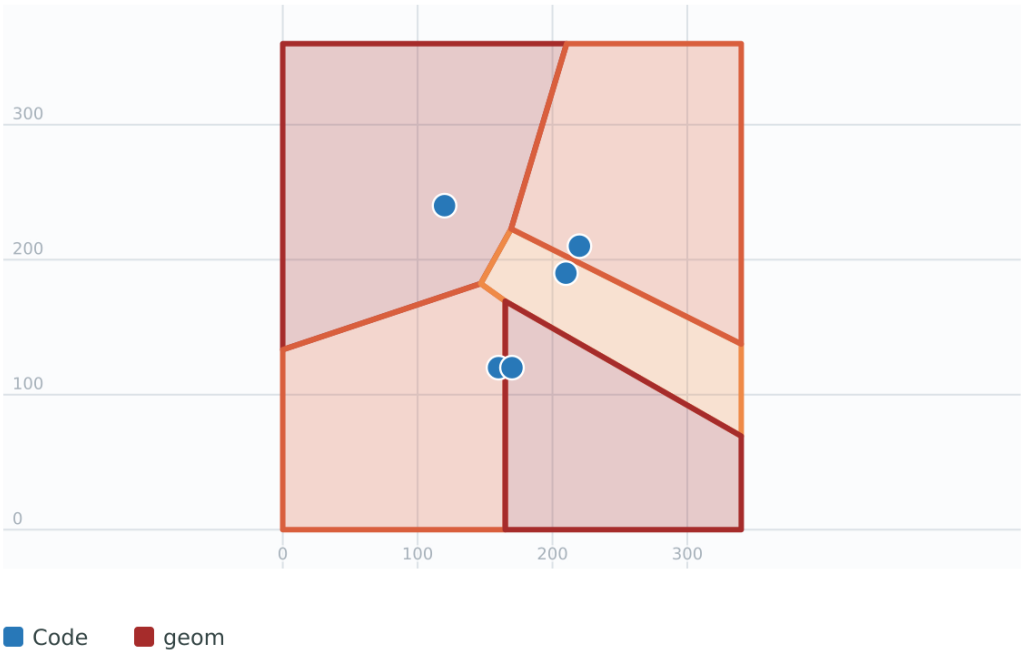
□□

Points overlaid on top of the Voronoi diagram.

Code

```
SELECT ST_VoronoiPolygons('MULTIPOINT (50 30,60 30,100 100,10 150,110 120)'::geometry
            ) AS geom;
```

Figure

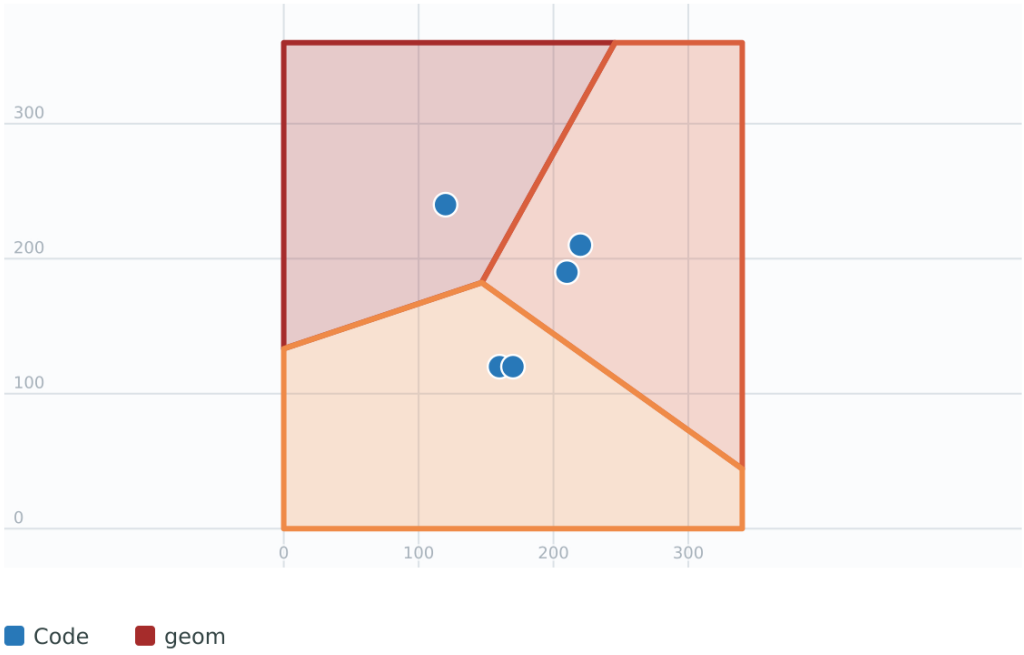


Voronoi diagram with a tolerance of 30 units.

Code

```
SELECT ST_VoronoiPolygons('MULTIPOINT (50 30,60 30,100 100,10 150,110 120)'::geometry,
            30) AS geom;
```

Figure



□□

ST_DelaunayTriangles, ST_VoronoiLines

7.15 Coverages

7.15.1 ST_CoverageInvalidEdges

ST_CoverageInvalidEdges — Window function that finds locations where polygons fail to form a valid coverage.

□□

geometry **ST_CoverageInvalidEdges**(geometry winset geom, float8 tolerance = 0);

□□

A window function which checks if the polygons in the window partition form a valid polygonal coverage. It returns linear indicators showing the location of invalid edges (if any) in each polygon.

A set of valid polygons is a valid coverage if the following conditions hold:

- **Non-overlapping** - polygons do not overlap (their interiors do not intersect)
- **Edge-Matched** - vertices along shared edges are identical

As a window function a value is returned for every input polygon. For polygons which violate one or more of the validity conditions the return value is a MULTILINESTRING containing the problematic edges. Coverage-valid polygons return the value NULL. Non-polygonal or empty geometries also produce NULL values.

The conditions allow a valid coverage to contain holes (gaps between polygons), as long as the surrounding polygons are edge-matched. However, very narrow gaps are often undesirable. If the *tolerance* parameter is specified with a non-zero distance, edges forming narrower gaps will also be returned as invalid.

The polygons being checked for coverage validity must also be valid geometries. This can be checked with ST_IsValid.

Availability: 3.4.0

Requires GEOS >= 3.12.0

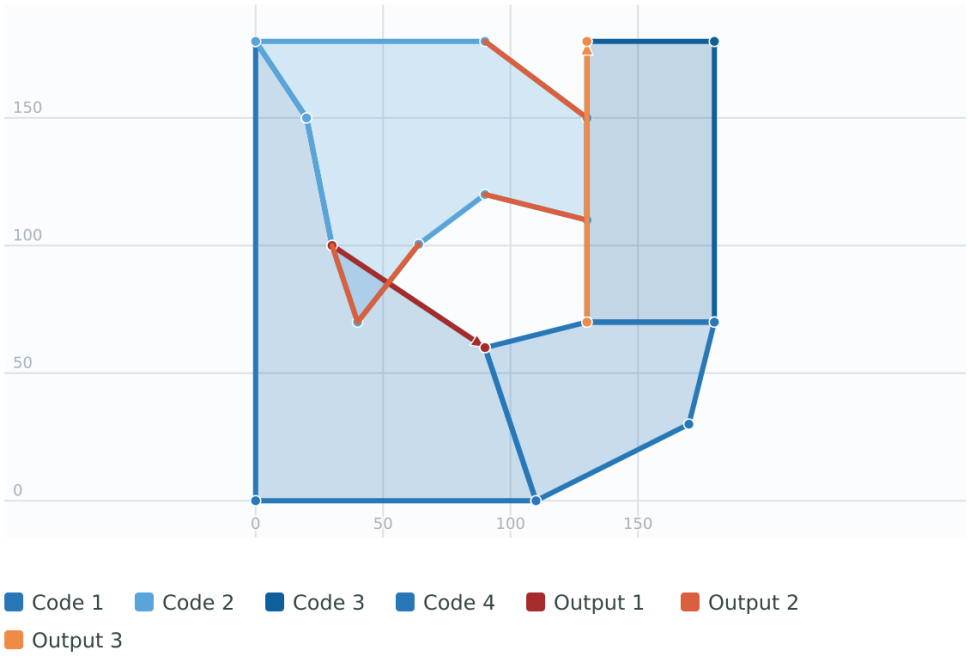
☒☒

Invalid edges caused by overlap and non-matching vertices.

Code

```
WITH coverage(id, geom) AS (VALUES
  (1, 'POLYGON ((10 190,30 160,40 110,100 70,120 10,10 10,10 190))'::geometry),
  (2, 'POLYGON ((100 190,10 190,30 160,40 110,50 80,74 110.5,100 130,140 120,140 160,100 ↩
    190))'::geometry),
  (3, 'POLYGON ((140 190,190 190,190 80,140 80,140 190))'::geometry),
  (4, 'POLYGON ((180 40,120 10,100 70,140 80,190 80,180 40))'::geometry)
)
SELECT id, ST_CoverageInvalidEdges(geom) OVER ()
  FROM coverage;
```

Figure



This example tests the entire table for coverage validity.

Code

```
SELECT true = ALL (
    SELECT ST_CoverageInvalidEdges(geom) OVER () IS NULL
    FROM coverage
    );
```

□□

ST_IsValid, ST_CoverageUnion, ST_CoverageClean, ST_CoverageSimplify

7.15.2 ST_CoverageClean

ST_CoverageClean — Computes a clean (edge matched, non-overlapping, gap-cleared) polygonal coverage, given a non-clean input.

□□

geometry **ST_CoverageClean**(geometry winset geom, float8 gapMaximumWidth = 0, float8 snappingDistance = -1, text overlapMergeStrategy = 'MERGE_LONGEST_BORDER');

□□

A window function which adjusts the edges of a set of valid polygonal geometries to produce a clean coverage. Cleaning involves:

- snapping vertices and edges to remove small discrepancies and ensure common edges are identically noded
- merging overlaps into a parent polygon
- merging narrow gaps into adjacent polygons

gapMaximumWidth controls which gaps between polygons are merged. Gaps with width <= this distance are merged into an adjacent polygon.

snappingDistance controls snapping of vertices and edges. The default (-1) automatically determines a snapping distance based on the input extent. Set to 0.0 to turn off snapping.

overlapMergeStrategy specifies how overlaps are merged into a parent polygon:

- MERGE_LONGEST_BORDER - merges into polygon with longest common border
- MERGE_MAX_AREA - merges into polygon with maximum area
- MERGE_MIN_AREA - merges into polygon with minimum area
- MERGE_MIN_INDEX - merges into polygon with smallest input index (defined by order of input polygons)

The result is a clean polygonal coverage that will pass validation by ST_CoverageInvalidEdges and can be input to coverage processing functions.



☒☒

To aid in determining a maximum gap width, gaps can be computed by cleaning with gapMaximumWidth = > 0 and using ST_CoverageUnion to union the result coverage. Holes in the union correspond to gaps in the original dataset. Gap widths can be measured by extracting the holes as polygons and running ST_MaximumInscribedCircle on them; the gap width is twice the computed radius.

Availability: 3.6.0 - requires GEOS >= 3.14.0

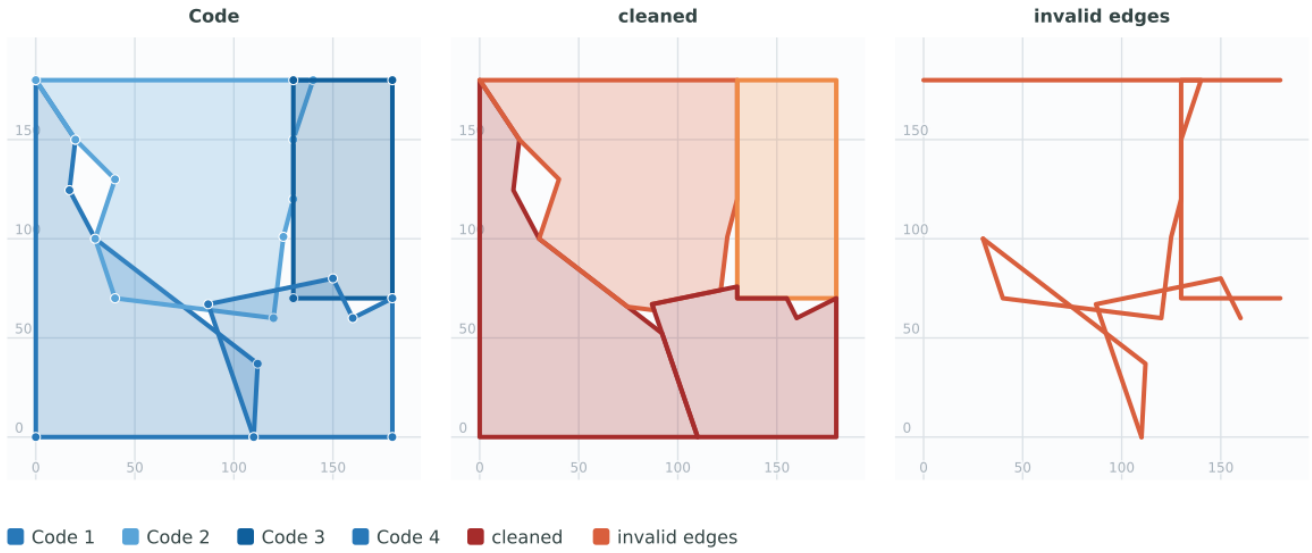
☒☒

Clean a coverage containing overlaps and narrow gaps, and compare the cleaned polygons with the original invalid edges.

Code

```
WITH coverage(id, geom) AS (VALUES
  (1, 'POLYGON ((10 190,30 160,27 134.5,40 110,122 47,120 10,10 10,10 190))'::geometry),
  (2, 'POLYGON ((150 190,10 190,30 160,50 140,40 110,50 80,130 70,135 111,140 130,140 ↩
    160,150 190))'::geometry),
  (3, 'POLYGON ((140 190,190 190,190 80,140 80,140 190))'::geometry),
  (4, 'POLYGON ((190 10,120 10,97 77,160 90,170 70,190 80,190 10))'::geometry)
), processed AS (
  SELECT id,
         ST_CoverageClean(geom, 1) OVER () AS cleaned,
         ST_CoverageInvalidEdges(geom) OVER () AS invalid_edges
  FROM coverage
)
SELECT ST_Collect(cleaned ORDER BY id) AS cleaned,
       ST_Collect(invalid_edges ORDER BY id) AS invalid_edges
FROM processed;
```

Figure



☒☒

ST_CoverageInvalidEdges, ST_CoverageUnion, ST_CoverageSimplify, ST_MaximumInscribedCircle

7.15.3 ST_CoverageSimplify

ST_CoverageSimplify — Window function that simplifies the edges of a polygonal coverage.

□□

geometry **ST_CoverageSimplify**(geometry winset geom, float8 tolerance, boolean simplifyBoundary = true);

□□

A window function which simplifies the edges of polygons in a polygonal coverage. The simplification preserves the coverage topology. This means the simplified output polygons are consistent along shared edges, and still form a valid coverage.

The simplification uses a variant of the Visvalingam-Whyatt algorithm. The *tolerance* parameter has units of distance, and is roughly equal to the square root of triangular areas to be simplified.

To simplify only the "internal" edges of the coverage (those that are shared by two polygons) set the *simplifyBoundary* parameter to false.

Use this function to simplify adjacent polygon layers when shared boundaries must remain edge-matched. For a single geometry where only self-intersection and ring validity matter, see ST_SimplifyPreserv



□□
If the input is not a valid coverage there may be unexpected artifacts in the output (such as boundary intersections, or separated boundaries which appeared to be shared). Use ST_CoverageInvalidEdges to determine if a coverage is valid.

Availability: 3.4.0

Requires GEOS >= 3.12.0

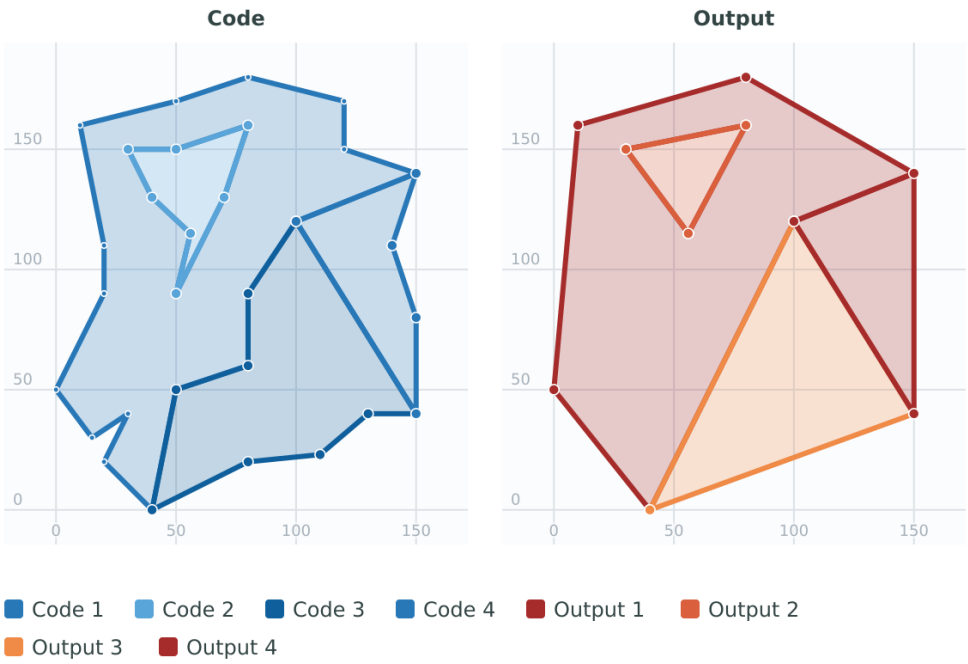
□□

Simplify a polygonal coverage while preserving shared boundaries.

Code

```
WITH coverage(id, geom) AS (VALUES
  (1, 'POLYGON ((160 150,110 130,90 100,90 70,60 60,50 10,30 30,40 50,25 40,10 60,30 100,30 ↩
     120,20 170,60 180,90 190,130 180,130 160,160 150),(40 160,50 140,66 125,60 100,80  ↩
    140,90 170,60 160,40 160))'::geometry),
  (2, 'POLYGON ((40 160,60 160,90 170,80 140,60 100,66 125,50 140,40 160))'::geometry),
  (3, 'POLYGON ((110 130,160 50,140 50,120 33,90 30,50 10,60 60,90 70,90 100,110 130))':: ↩
    geometry),
  (4, 'POLYGON ((160 150,150 120,160 90,160 50,110 130,160 150))'::geometry)
)
SELECT id, ST_CoverageSimplify(geom, 30) OVER ()
  FROM coverage;
```

Figure



□□

ST_CoverageInvalidEdges, ST_CoverageUnion, ST_CoverageClean

7.15.4 ST_CoverageEdges

ST_CoverageEdges — Computes the unique edges of a polygonal coverage.

□□

geometry **ST_CoverageEdges**(geometry geom, integer edgetype = 0);
geometry **ST_CoverageEdges**(geometry[] geoms, integer edgetype = 0);

□□

Returns a MultiLineString representing the unique edges of a polygonal coverage. A polygonal coverage is a set of non-overlapping polygons where adjacent polygons have matching vertices along shared edges.

The *edgetype* parameter can be used to select which edges are returned:

- **0 (ALL)** - all unique edges (default)
- **1 (EXTERIOR)** - only exterior edges (non-shared)
- **2 (INTERIOR)** - only interior edges (shared)

Availability: 3.7.0

Requires GEOS >= 3.15.0

□□

This example returns all unique edges of a coverage.

Code

```
WITH coverage(geom) AS (
  VALUES (
    'GEOMETRYCOLLECTION(
      POLYGON((0 0,10 0,10 10,0 10,0 0)),
      POLYGON((10 0,20 0,20 10,10 10,10 0))
    )'::geometry
  )
)
SELECT geom AS input_coverage,
       ST_CoverageEdges(geom) AS all_edges
FROM coverage;
```

Output

```
-[ RECORD 1 ]--+-----------------------------------------------------------------------------------------------
input_coverage | GEOMETRYCOLLECTION(POLYGON((0 0,10 0,10 10,0 10,0 0)),POLYGON((10 0,20 0,20 10,10 10,10 0)))
all_edges      | MULTILINESTRING((10 0,10 10),(10 10,0 10,0 0,10 0),(10 0,20 0,20 10,10 10))
```

This example returns only the interior shared edges of a coverage.

Code

```
WITH coverage(geom) AS (
  VALUES (
    'GEOMETRYCOLLECTION(
      POLYGON((0 0,10 0,10 10,0 10,0 0)),
      POLYGON((10 0,20 0,20 10,10 10,10 0))
    )'::geometry
  )
)
SELECT geom AS input_coverage,
       ST_CoverageEdges(geom, 2) AS shared_edges
FROM coverage;
```

Output

```
-[ RECORD 1 ]--+-----------------------------------------------------------------------------------------------
input_coverage | GEOMETRYCOLLECTION(POLYGON((0 0,10 0,10 10,0 10,0 0)),POLYGON((10 0,20 0,20 10,10 10,10 0)))
shared_edges   | MULTILINESTRING((10 0,10 10))
```

□□

ST_CoverageInvalidEdges, ST_CoverageSimplify, ST_CoverageUnion

7.15.5 ST_CoverageUnion

ST_CoverageUnion — Computes the union of a set of polygons forming a coverage by removing shared edges.

□□

geometry **ST_CoverageUnion**(geometry set geom);

□□

An aggregate function which unions a set of polygons forming a polygonal coverage. The result is a polygonal geometry covering the same area as the coverage. This function produces the same result as ST_Union, but uses the coverage structure to compute the union much faster.



□□
If the input is not a valid coverage there may be unexpected artifacts in the output (such as unmerged or overlapping polygons). Use ST_CoverageInvalidEdges to determine if a coverage is valid.

Availability: 3.4.0 - requires GEOS >= 3.8.0

□□

Union a polygonal coverage.

Code

```
WITH coverage(id, geom) AS (VALUES
  (1, 'POLYGON ((10 10,10 150,80 190,110 150,90 110,40 110,50 60,10 10))'::geometry),
  (2, 'POLYGON ((120 10,10 10,50 60,100 70,120 10))'::geometry),
  (3, 'POLYGON ((140 80,120 10,100 70,40 110,90 110,110 150,140 80))'::geometry),
  (4, 'POLYGON ((140 190,120 170,140 130,160 150,140 190))'::geometry),
  (5, 'POLYGON ((180 160,170 140,140 130,160 150,140 190,180 160))'::geometry)
)
SELECT ST_CoverageUnion(geom) AS wkt
  FROM coverage;
```

Figure



□□

ST_CoverageInvalidEdges, ST_CoverageSimplify, ST_CoverageClean, ST_Union

7.16 Affine Transformations

7.16.1 ST_Affine

ST_Affine — Apply a 3D affine transformation to a geometry.

□□

geometry **ST_Affine**(geometry geomA, float a, float b, float c, float d, float e, float f, float g, float h, float i, float xoff, float yoff, float zoff);
geometry **ST_Affine**(geometry geomA, float a, float b, float d, float e, float xoff, float yoff);

□□

Applies a 3D affine transformation to the geometry to do things like translate, rotate, scale in one step.

Version 1: The call

Code

```
ST_Affine(geom, a, b, c, d, e, f, g, h, i, xoff, yoff, zoff)
```

represents the transformation matrix

Code

```
/ a  b  c  xoff \
| d  e  f  yoff |
| g  h  i  zoff |
\ 0  0  0     1 /
```

and the vertices are transformed as follows:

Code

```
x' = a*x + b*y + c*z + xoff
y' = d*x + e*y + f*z + yoff
z' = g*x + h*y + i*z + zoff
```

All of the translate / scale functions below are expressed via such an affine transformation.

Version 2: Applies a 2d affine transformation to the geometry. The call

Code

```
ST_Affine(geom, a, b, d, e, xoff, yoff)
```

represents the transformation matrix

Code

```
/  a  b  0  xoff  \       /  a  b  xoff  \
|  d  e  0  yoff  | rsp.  |  d  e  yoff  |
|  0  0  1     0  |       \  0  0     1  /
\  0  0  0     1  /
```

and the vertices are transformed as follows:

Code

```
x' = a*x + b*y + xoff
y' = d*x + e*y + yoff
z' = z
```

This method is a subcase of the 3D method above.
□□□□: 2.0.0 □□□□□□□□□□, □□□□ TIN □□□□□□□□□□□□.
Availability: 1.1.2. Name changed from Affine to ST_Affine in 1.2.2



□□
1.3.4 □□□□□□□□□□□□□□□□ (curve) □□□□□□□□□□□□□□□□□□□□□□□□. 1.3.4 □□□□□□□□□□□□□□□□.

This function supports Polyhedral surfaces.

This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).

This function supports 3d and will not drop the z-index.

This method supports Circular Strings and Curves.

□□

Rotate a 3D line 180 degrees about the z axis. Note this is long-hand for doing ST_Rotate();

Code

```
SELECT ST_Affine(geom, cos(pi()), -sin(pi()), 0, sin(pi()), cos(pi()), 0, 0, 0, 1, 0, 0, 0) ↩
    As using_affine,
   ST_Rotate(geom, pi()) As using_rotate
  FROM (SELECT 'LINESTRING(1 2 3,1 4 3)'::geometry As geom) As foo;
```

Figure

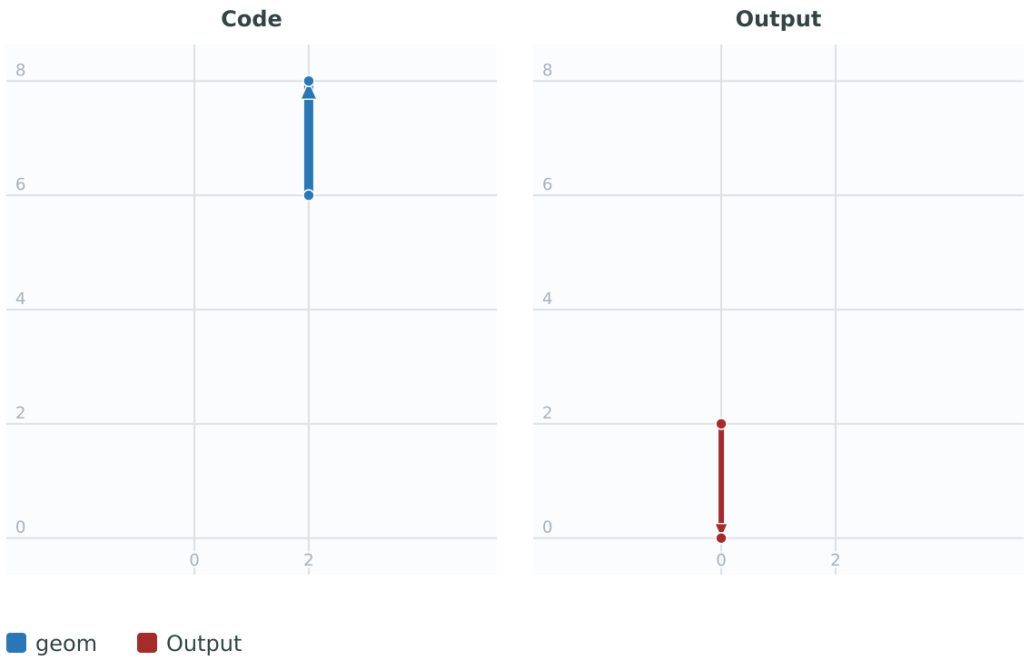


Rotate a 3D line 180 degrees in both the x and z axis.

Code

```
SELECT ST_Affine(geom, cos(pi()), -sin(pi()), 0, sin(pi()), cos(pi()), -sin(pi()), 0, sin( ↩
    pi()), cos(pi()), 0, 0, 0)
    FROM (SELECT 'LINESTRING(1 2 3,1 4 3)'::geometry As geom) As foo;
```

Figure



□□

ST_Rotate, ST_Scale, ST_Translate, ST_TransScale

7.16.2 ST_Rotate

ST_Rotate — Rotates a geometry about an origin point.

□□

geometry **ST_Rotate**(geometry geomA, float rotRadians);
geometry **ST_Rotate**(geometry geomA, float rotRadians, float x0, float y0);
geometry **ST_Rotate**(geometry geomA, float rotRadians, geometry pointOrigin);

□□

Rotates geometry rotRadians counter-clockwise about the origin point. The rotation origin can be specified either as a POINT geometry, or as x and y coordinates. If the origin is not specified, the geometry is rotated about POINT(0 0).

□□□□: 2.0.0 □□□□□□□□□□, □□□□ TIN □□□□□□□□□□□□.

Enhanced: 2.0.0 additional parameters for specifying the origin of rotation were added.

Availability: 1.1.2. Name changed from Rotate to ST_Rotate in 1.2.2

This function supports 3d and will not drop the z-index.

This method supports Circular Strings and Curves.

This function supports Polyhedral surfaces.

This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).

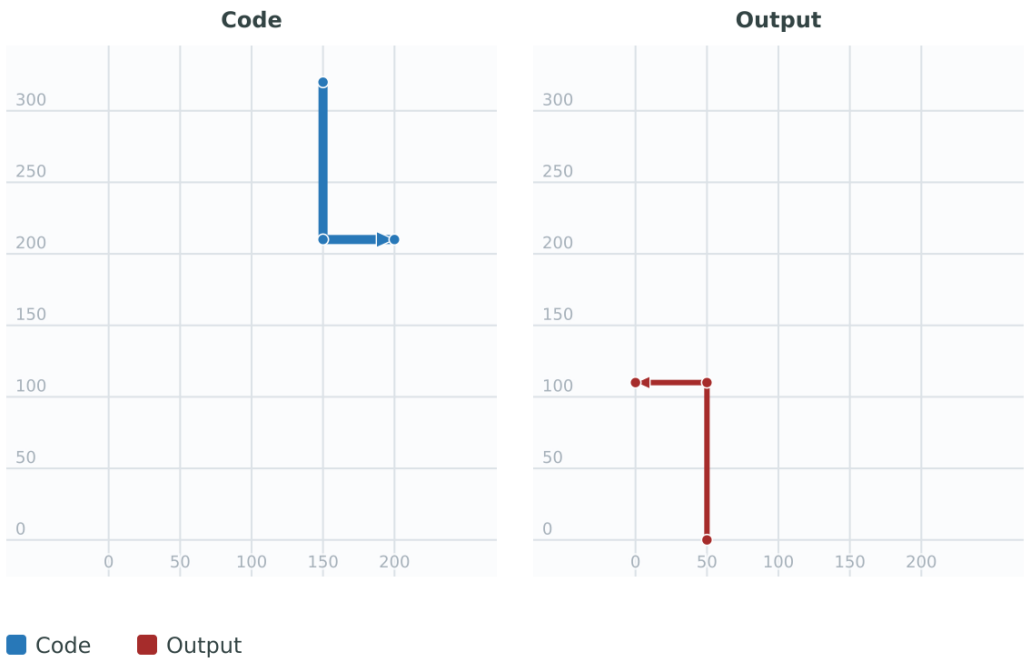
□□

This example rotates a geometry by 180 degrees.

Code

```
SELECT ST_Rotate('LINESTRING (50 160,50 50,100 50)'::geometry, pi());
```

Figure

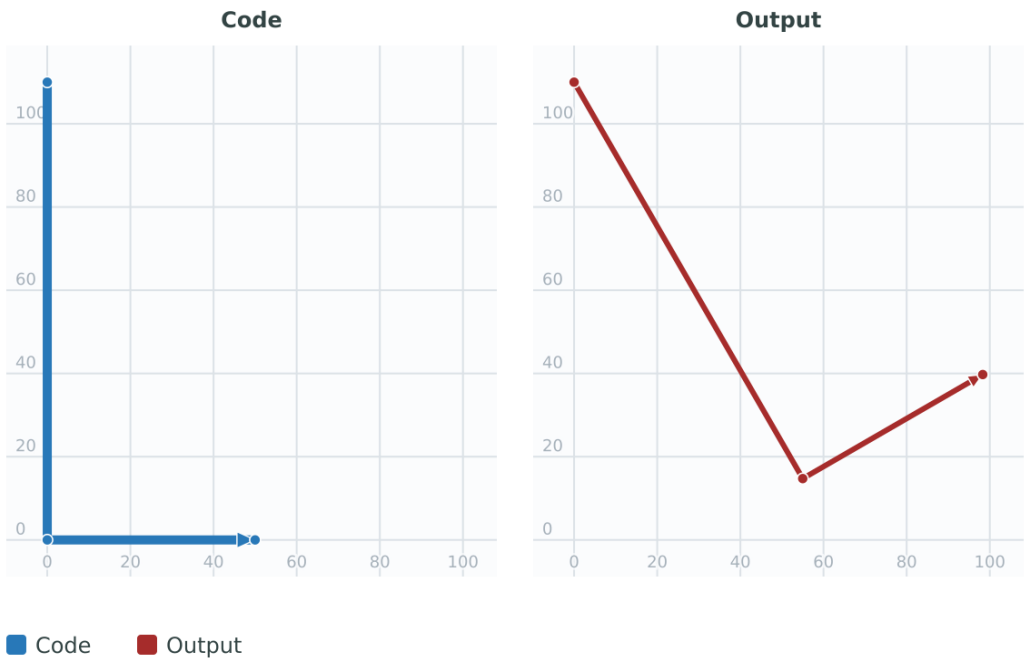


This example rotates a geometry 30 degrees counter-clockwise around x=50, y=160.

Code

```
SELECT ST_Rotate('LINESTRING (50 160,50 50,100 50)'::geometry, pi()/6, 50, 160);
```

Figure

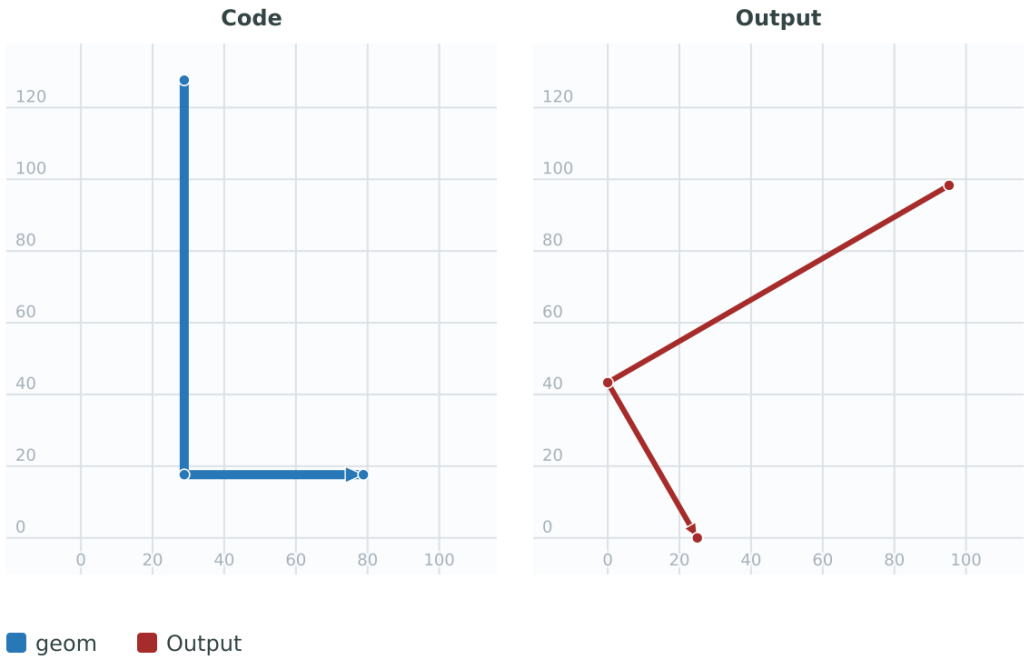


This example rotates a geometry 60 degrees clockwise around its centroid.

Code

```
SELECT ST_Rotate(geom, -pi()/3, ST_Centroid(geom))
FROM (SELECT 'LINESTRING (50 160,50 50,100 50)'::geometry AS geom) AS foo;
```

Figure



□□

ST_Affine, ST_RotateX, ST_RotateY, ST_RotateZ

7.16.3 ST_RotateX

ST_RotateX — Rotates a geometry about the X axis.

□□

geometry **ST_RotateX**(geometry geomA, float rotRadians);

□□

Rotates a geometry geomA - rotRadians about the X axis.



□□
ST_RotateX(geomA, rotRadians) is short-hand for ST_Affine(geomA, 1, 0, 0, 0, cos(rotRadians), -sin(rotRadians), 0, sin(rotRadians), cos(rotRadians), 0, 0, 0).

□□□□: 2.0.0 □□□□□□□□□□, □□□□ TIN □□□□□□□□□□□□.

Availability: 1.1.2. Name changed from RotateX to ST_RotateX in 1.2.2


This function supports Polyhedral surfaces.


This function supports 3d and will not drop the z-index.


This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).

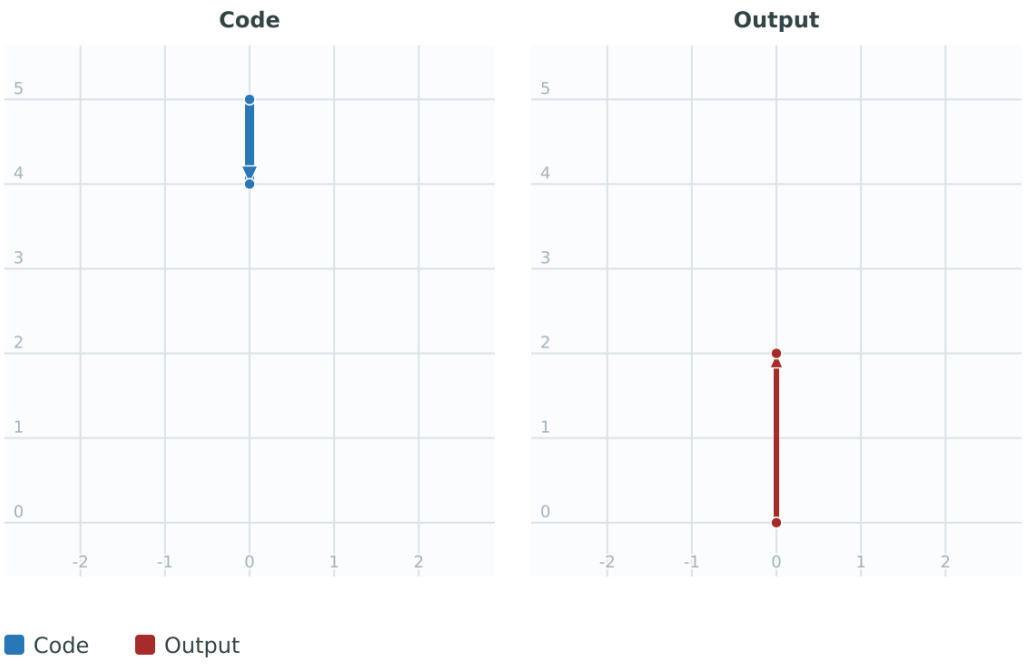
□□

This example rotates a line 90 degrees around the X axis.

Code

```
SELECT ST_RotateX('LINESTRING(1 2 3,1 1 1)', pi()/2);
```

Figure



☒☒

ST_Affine, ST_RotateY, ST_RotateZ

7.16.4 ST_RotateY

ST_RotateY — Rotates a geometry about the Y axis.

☒☒

geometry **ST_RotateY**(geometry geomA, float rotRadians);

☒☒

Rotates a geometry geomA - rotRadians about the y axis.

☒☒
`ST_RotateY(geomA, rotRadians)` is short-hand for `ST_Affine(geomA, cos(rotRadians), 0, sin(rotRadians), 0, 1, 0, -sin(rotRadians), 0, cos(rotRadians), 0, 0, 0)`.

Availability: 1.1.2. Name changed from RotateY to ST_RotateY in 1.2.2
☒☒☒☒: 2.0.0 ☒☒☒☒☒☒☒☒☒, ☒☒☒☒ TIN ☒☒☒☒☒☒☒☒☒☒☒.

This function supports Polyhedral surfaces.

This function supports 3d and will not drop the z-index.

This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).

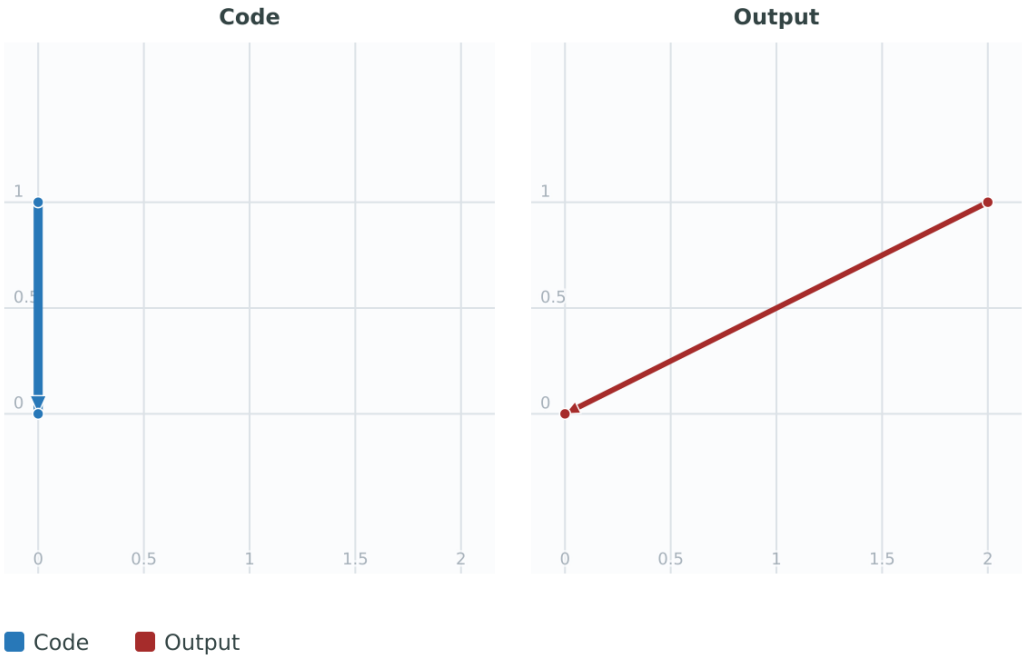
□□

This example rotates a line 90 degrees around the Y axis.

Code

```
SELECT ST_RotateY('LINESTRING(1 2 3,1 1 1)', pi()/2);
```

Figure



□□

ST_Affine, ST_RotateX, ST_RotateZ

7.16.5 ST_RotateZ

ST_RotateZ — Rotates a geometry about the Z axis.

□□

geometry **ST_RotateZ**(geometry geomA, float rotRadians);

□□

Rotates a geometry geomA - rotRadians about the Z axis.



□□
This is a synonym for ST_Rotate



□□
ST_RotateZ(geomA, rotRadians) is short-hand for SELECT ST_Affine(geomA, cos(rotRadians), -sin(rotRadians), 0, sin(rotRadians), cos(rotRadians), 0, 0, 0, 1, 0, 0, 0).

□□□□: 2.0.0 □□□□□□□□□□, □□□□ TIN □□□□□□□□□□□□.
Availability: 1.1.2. Name changed from RotateZ to ST_RotateZ in 1.2.2



□□
1.3.4 □□□□□□□□□□□□□□□□ (curve) □□□□□□□□□□□□□□□□□□□□□□□□. 1.3.4 □□□□□□□□□□□□□□□.


This function supports 3d and will not drop the z-index.


This method supports Circular Strings and Curves.


This function supports Polyhedral surfaces.


This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).

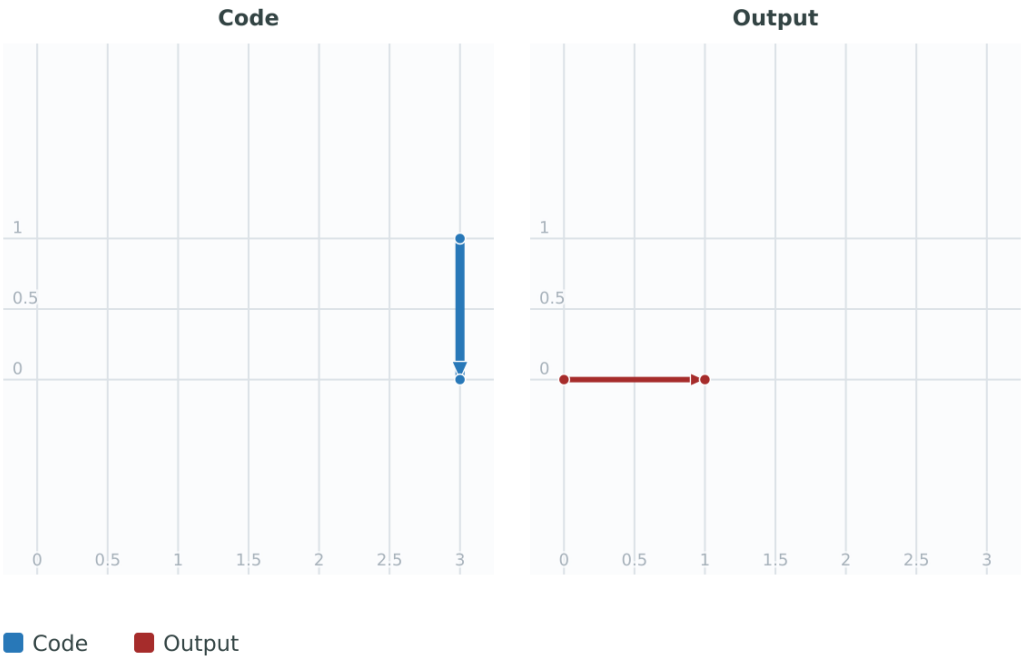
□□

This example rotates a line 90 degrees around the Z axis.

Code

```
SELECT ST_RotateZ('LINESTRING(1 2 3,1 1 1)', pi()/2);
```

Figure

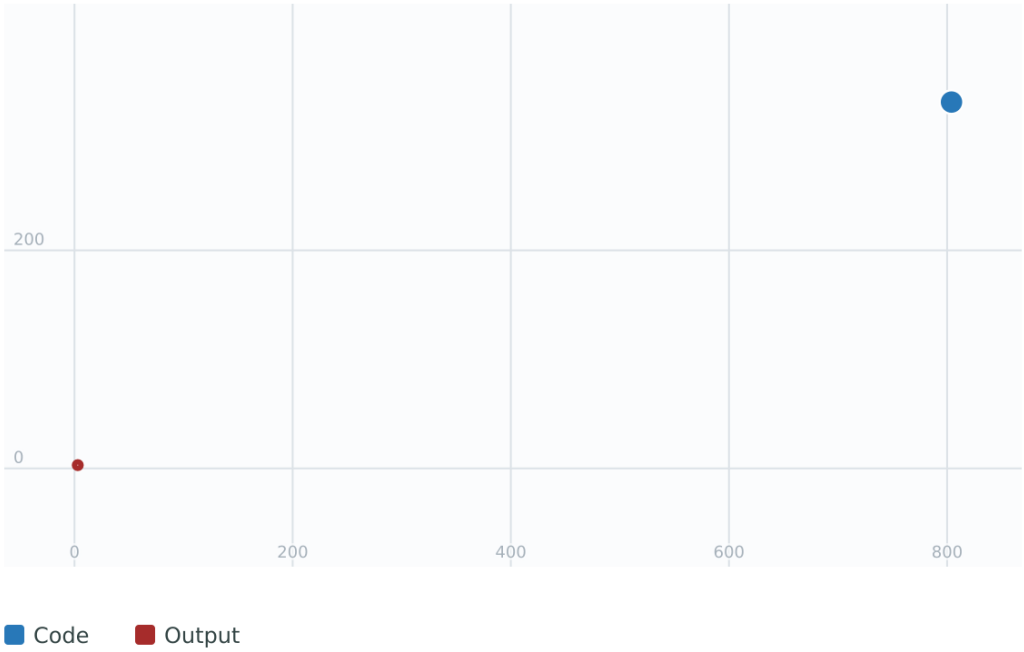


This example rotates a curved circle around the Z axis.

Code

```
SELECT ST_RotateZ(geom, pi()/2)
FROM (SELECT ST_LineToCurve(ST_Buffer('POINT(234 567)'::geometry, 3)) As geom) As foo;
```

Figure



☒☒

ST_Affine, ST_RotateX, ST_RotateY

7.16.6 ST_Scale

ST_Scale — Scales a geometry by given factors.

☒☒

geometry **ST_Scale**(geometry geomA, float XFactor, float YFactor, float ZFactor);
geometry **ST_Scale**(geometry geomA, float XFactor, float YFactor);
geometry **ST_Scale**(geometry geom, geometry factor);
geometry **ST_Scale**(geometry geom, geometry factor, geometry origin);

☒☒

Scales the geometry to a new size by multiplying the ordinates with the corresponding factor parameters.

The version taking a geometry as the `factor` parameter allows passing a 2d, 3dm, 3dz or 4d point to set scaling factor for all supported dimensions. Missing dimensions in the `factor` point are equivalent to no scaling the corresponding dimension.

The three-geometry variant allows a ”false origin” for the scaling to be passed in. This allows ”scaling in place”, for example using the centroid of the geometry as the false origin. Without a false origin, scaling takes place relative to the actual origin, so all coordinates are just multiplied by the scale factor.



□□
1.3.4 □□□□□□□□□□□□□□□□ (curve) □□□□□□□□□□□□□□□□□□□□□□□□. 1.3.4 □□□□□□□□□□□□□□□□.

Availability: 1.1.0.

□□□□: 2.0.0 □□□□□□□□□□, □□□□ TIN □□□□□□□□□□□□.

Enhanced: 2.2.0 support for scaling all dimension (`factor` parameter) was introduced.

Enhanced: 2.5.0 support for scaling relative to a local origin (`origin` parameter) was introduced.

This function supports Polyhedral surfaces.

This function supports 3d and will not drop the z-index.

This method supports Circular Strings and Curves.

This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).

This function supports M coordinates.

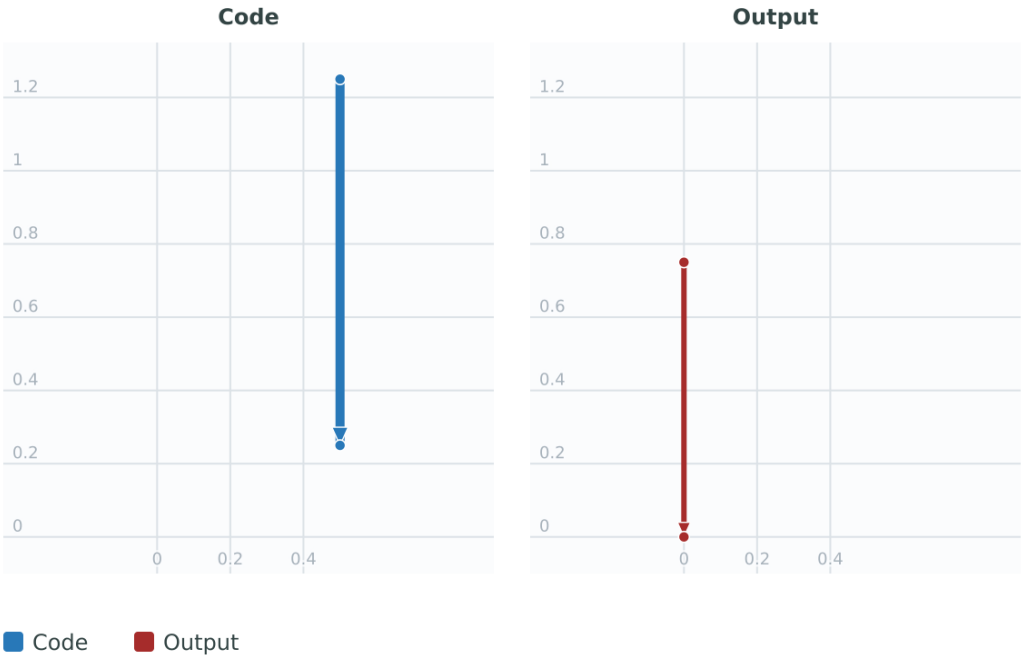
□□

This example scales X, Y, and Z values.

Code

```
SELECT ST_Scale('LINESTRING(1 2 3,1 1 1)', 0.5, 0.75, 0.8);
```

Figure

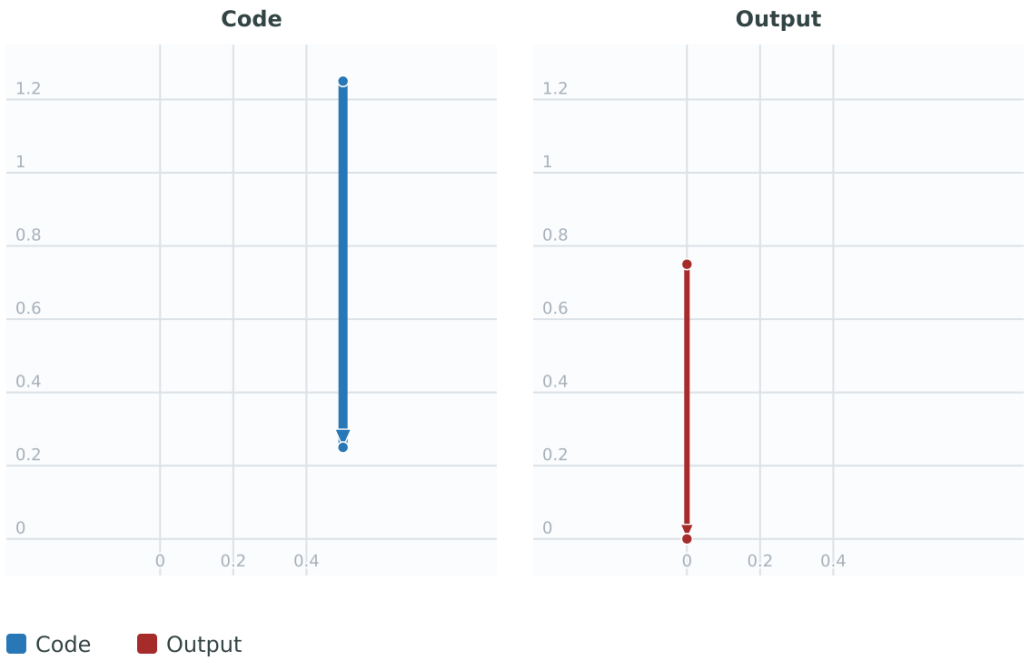


This example scales X and Y values.

Code

```
SELECT ST_Scale('LINESTRING(1 2 3,1 1 1)', 0.5, 0.75);
```

Figure

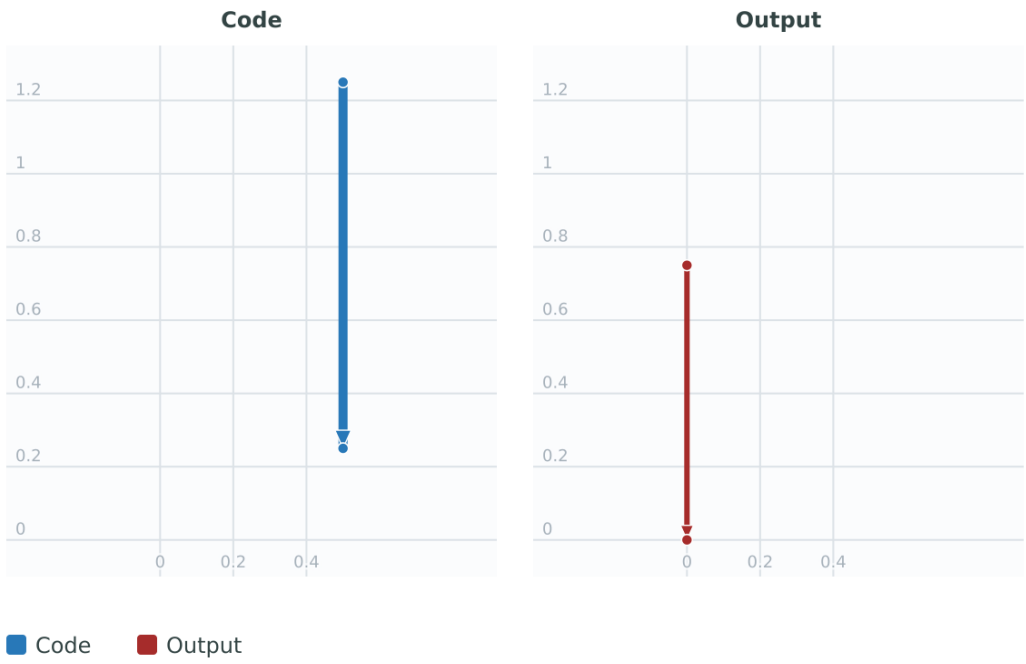


This example scales X, Y, Z, and M values.

Code

```
SELECT ST_Scale('LINESTRING(1 2 3 4,1 1 1 1)',
    ST_MakePoint(0.5, 0.75, 2, -1));
```

Figure

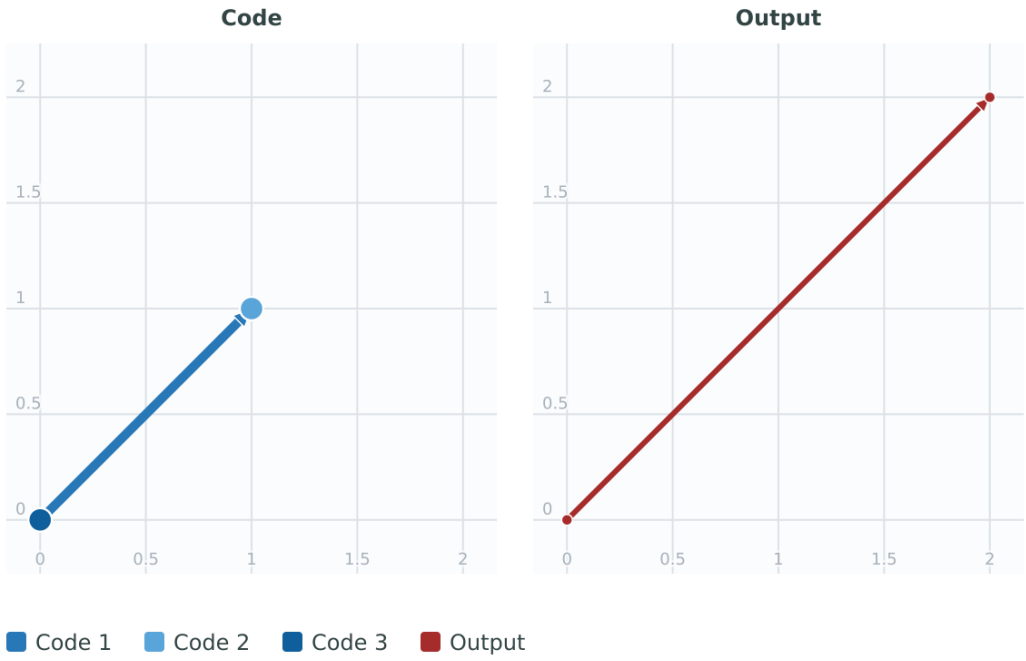


This example scales X and Y values using a false origin.

Code

```
SELECT ST_Scale('LINESTRING(1 1,2 2)', 'POINT(2 2)', 'POINT(1 1)'::geometry);
```

Figure



This example constructs an ellipse centered at (10, 20), with X radius 8, Y radius 4, and rotated 30 degrees counter-clockwise. Scaling and rotation use the center as their origin, so the ellipse keeps both its position and SRID.

Code

```
WITH params AS (
  SELECT
    ST_SetSRID(ST_Point(10, 20), 3857) AS center,
    8.0 AS x_radius,
    4.0 AS y_radius,
    radians(30) AS angle
)
SELECT ST_Rotate(
         ST_Scale(
           ST_Buffer(center, 1, 'quad_segs=32'),
           ST_MakePoint(x_radius, y_radius),
           center
         ),
         angle,
         center
       ) AS ellipse
FROM params;
```

□□

ST_Affine, ST_Buffer, ST_MakePoint, ST_Rotate, ST_TransScale

7.16.7 ST_Translate

ST_Translate — Translates a geometry by given offsets.

□□

geometry **ST_Translate**(geometry g1, float deltax, float deltay);
geometry **ST_Translate**(geometry g1, float deltax, float deltay, float deltaz);

□□

Returns a new geometry whose coordinates are translated delta x,delta y,delta z units. Units are based on the units defined in spatial reference (SRID) for this geometry. M coordinates are preserved.



□□
1.3.4 □□□□□□□□□□□□□□□□ (curve) □□□□□□□□□□□□□□□□□□□□□□□□. 1.3.4 □ □□□□□□□□□□□□□□□.

1.2.2 □□□□□□□□□□□□.

This function supports 3d and will not drop the z-index.

This method supports Circular Strings and Curves.

This function supports M coordinates.

□□

Move a point 1 degree longitude

Code

```
SELECT ST_Translate(ST_GeomFromText('POINT(-71.01 42.37)', 4326), 1, 0) AS wgs_transgeomtxt ↩
    ;
```

Output

```
POINT(-70.01 42.37)
```

Move a linestring 1 degree longitude and 1/2 degree latitude

Code

```
SELECT ST_Translate(ST_GeomFromText('LINESTRING(-71.01 42.37,-71.11 42.38)', 4326), 1, 0.5) ↩
    AS wgs_transgeomtxt;
```

Figure



Move a 3d point

Code

```
SELECT ST_Translate(CAST('POINT(0 0 0)' AS geometry), 5, 12, 3);
```

Output

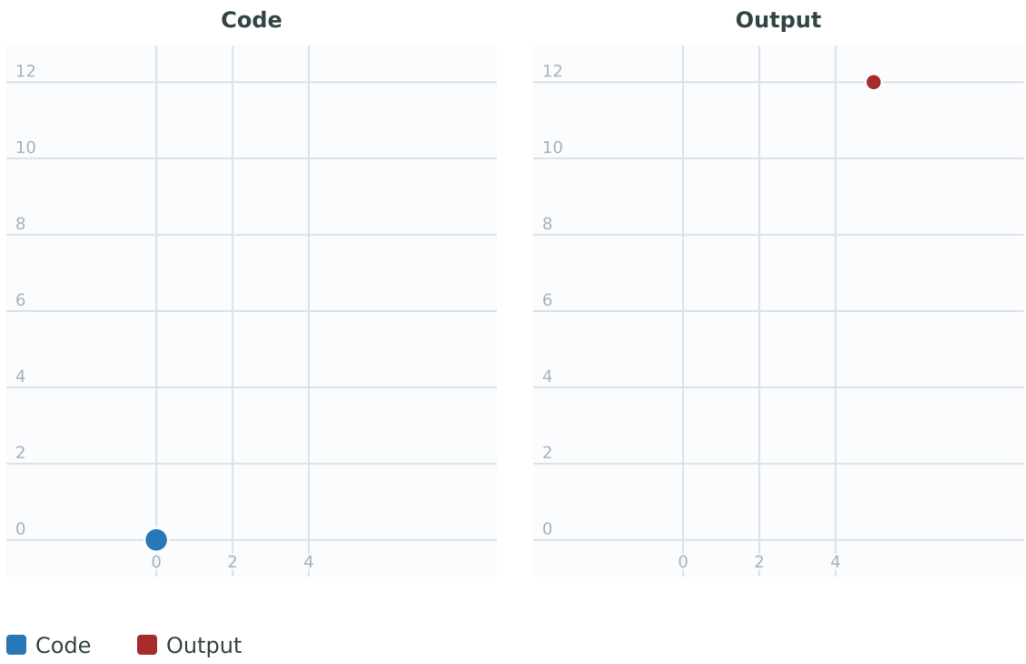
```
POINT(5 12 3)
```

Move points with a measure coordinate

Code

```
SELECT ST_Translate('POINTM(0 0 11)', 5, 12);
```

Figure



Code

```
SELECT ST_Translate('POINT(0 0 7 11)', 5, 12, 3);
```

Output

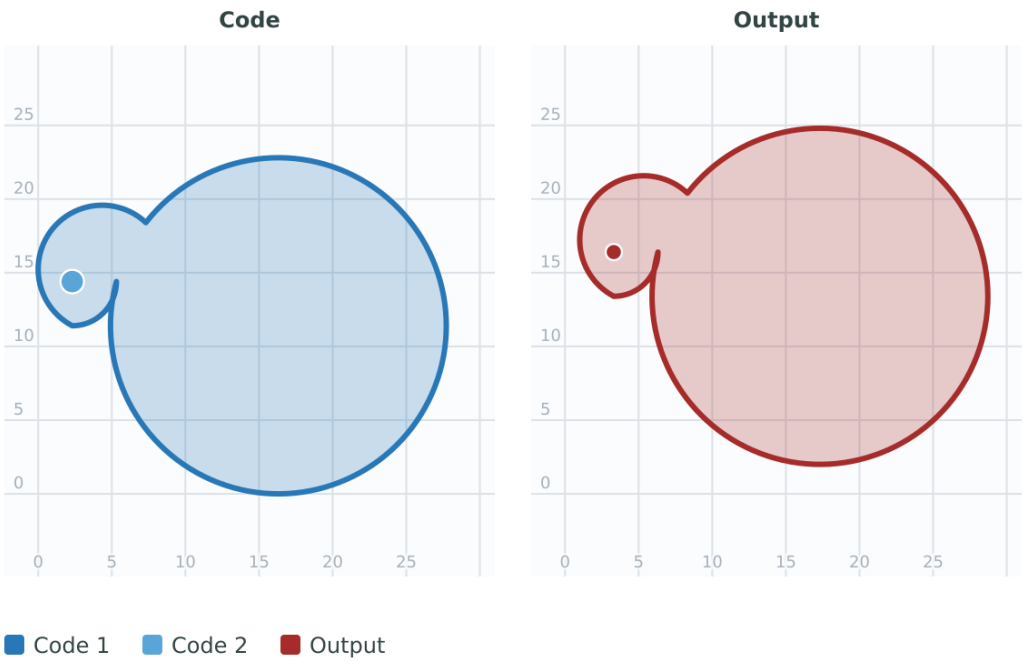
```
POINT(5 12 10 11)
```

Move a curve and a point

Code

```
SELECT ST_Translate(ST_Collect('CURVEPOLYGON(CIRCULARSTRING(4 3,3.12 0.878,1 0,-1.121 ↩
    5.1213,6 7,8 9,4 3))', 'POINT(1 3)'), 1, 2);
```

Figure



☒☒

ST_Affine, ST_AsText, ST_GeomFromText

7.16.8 ST_TransScale

ST_TransScale — Translates and scales a geometry by given offsets and factors.

☒☒

geometry **ST_TransScale**(geometry geomA, float deltaX, float deltaY, float XFactor, float YFactor);

☒☒

Translates the geometry using the deltaX and deltaY args, then scales it using the XFactor, YFactor args, working in 2D only.



☒☒
`ST_TransScale(geomA, deltaX, deltaY, XFactor, YFactor)` is short-hand for `ST_Affine(geomA, XFactor, 0, 0, 0, YFactor, 0, 0, 0, 1, deltaX*XFactor, deltaY*YFactor, 0)`.



☒☒
1.3.4 ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒ (curve) ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒. 1.3.4 ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.

Availability: 1.1.0.

This function supports 3d and will not drop the z-index.

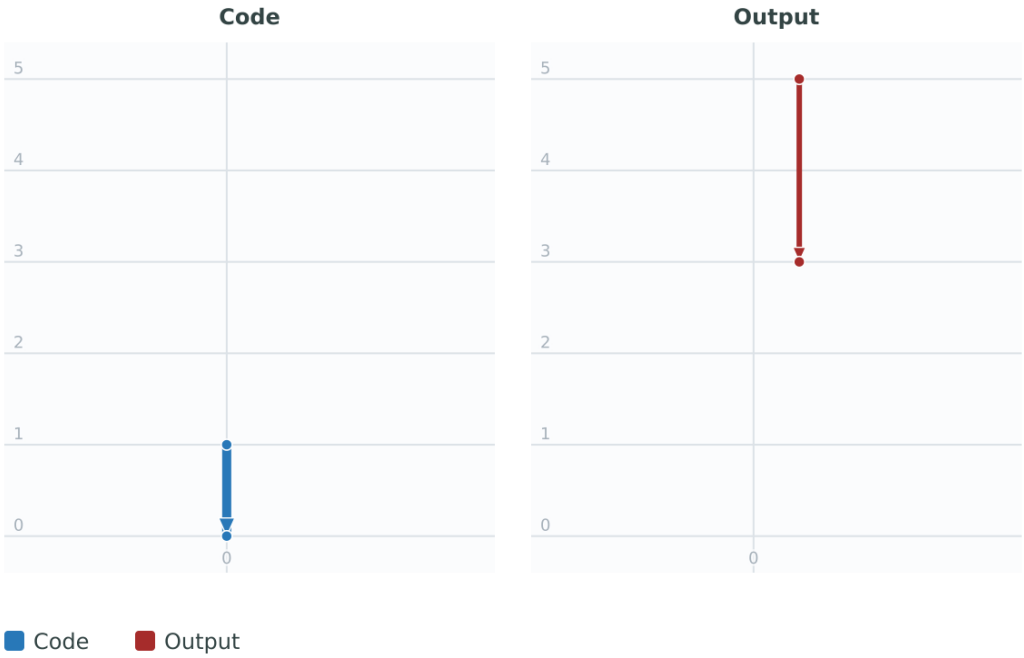
This method supports Circular Strings and Curves.

□□

Code

```
SELECT ST_TransScale('LINESTRING(1 2 3,1 1 1)', 0.5, 1, 1, 2);
```

Figure

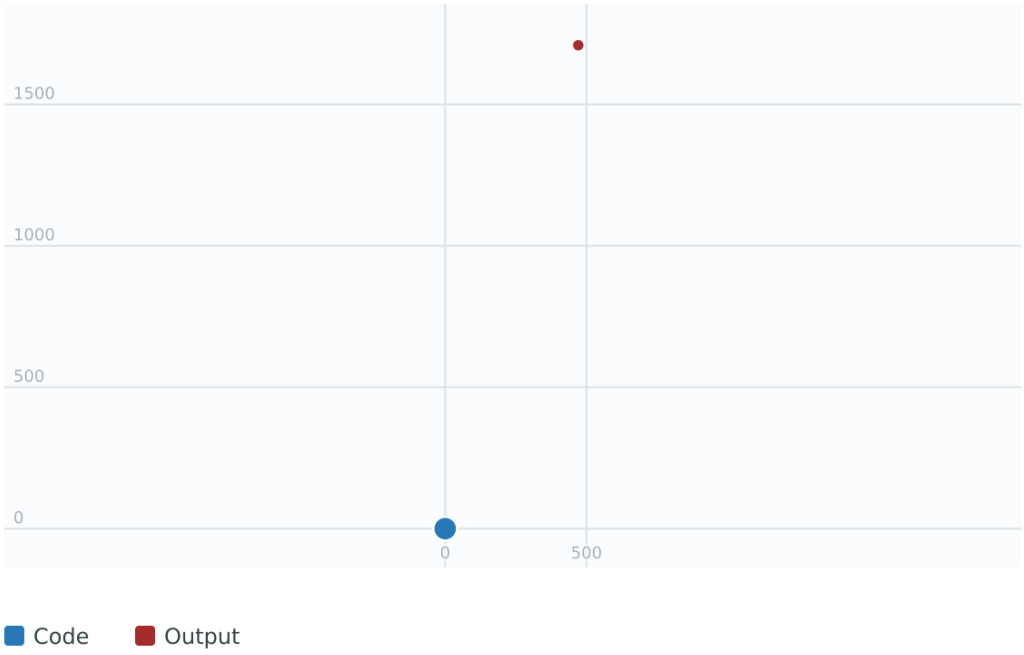


This example buffers a point to approximate a circle, converts it to a curve, translates it by 1,2, and scales it by 3,4.

Code

```
SELECT ST_TransScale(ST_LineToCurve(ST_Buffer('POINT(234 567)', 3)), 1, 2, 3, 4);
```

Figure



□□

ST_Affine, ST_Translate

7.17 Clustering Functions

7.17.1 ST_ClusterDBSCAN

ST_ClusterDBSCAN — Window function that returns a cluster id for each input geometry using the DBSCAN algorithm.

□□

integer **ST_ClusterDBSCAN**(geometry winset geom, float8 eps, integer minpoints);

□□

A window function that returns a cluster number for each input geometry, using the 2D Density-based spatial clustering of applications with noise (DBSCAN) algorithm. Unlike ST_ClusterKMeans, it does not require the number of clusters to be specified, but instead uses the desired distance (`eps`) and density (`minpoints`) parameters to determine each cluster.

An input geometry is added to a cluster if it is either:

- A "core" geometry, that is within `eps` distance of at least `minpoints` input geometries (including itself); or
- A "border" geometry, that is within `eps` distance of a core geometry.

Note that border geometries may be within `eps` distance of core geometries in more than one cluster. Either assignment would be correct, so the border geometry will be arbitrarily assigned to one of the available clusters. In this situation it is possible for a correct cluster to be generated with fewer than `minpoints` geometries. To ensure deterministic assignment of border geometries (so that repeated calls to ST_ClusterDBSCAN will produce identical results) use an `ORDER BY` clause in the window definition. Ambiguous cluster assignments may differ from other DBSCAN implementations.



☒☒
Geometries that do not meet the criteria to join any cluster are assigned a cluster number of NULL.

2.3.0 ☒☒☒☒☒☒☒☒☒☒☒.

 This method supports Circular Strings and Curves.

☒☒

Cluster polygons within 50 units of each other, requiring at least two polygons per cluster. The isolated polygon is reported as noise.

Code

```
WITH input(name, geom) AS (
  VALUES
    ('A1', ST_MakeEnvelope(0, 0, 10, 10)),
    ('A2', ST_MakeEnvelope(30, 0, 40, 10)),
    ('B1', ST_MakeEnvelope(100, 0, 110, 10)),
    ('B2', ST_MakeEnvelope(130, 0, 140, 10)),
    ('noise', ST_MakeEnvelope(250, 0, 260, 10))
), clustered AS (
  SELECT name,
         geom,
         ST_ClusterDBSCAN(geom, eps => 50, minpoints => 2)
           OVER (ORDER BY name) AS cid
  FROM input
)
SELECT COALESCE(cid::text, 'noise') AS cluster,
       string_agg(name, ',' ORDER BY name) AS members,
       ST_Collect(geom ORDER BY name) AS cluster_geom
FROM clustered
GROUP BY cid
ORDER BY cid NULLS LAST;
```

Output

```
0 | A1,A2 | MULTIPOLYGON(((0 0,0 10,10 10,10 0,0 0)),((30 0,30 10,40 10,40 0,30 0)))
1 | B1,B2 | MULTIPOLYGON(((100 0,100 10,110 10,110 0,100 0)),((130 0,130 10,140 10,140 0,130 0)))
noise | noise | MULTIPOLYGON(((250 0,250 10,260 10,260 0,250 0)))
```

Figure



A example showing combining parcels with the same cluster number into geometry collections.

Code

```
SELECT cid, ST_Collect(geom) AS cluster_geom, array_agg(parcel_id) AS ids_in_cluster FROM (
    SELECT parcel_id, ST_ClusterDBSCAN(geom, eps => 0.5, minpoints => 5) over () AS cid,  ↩
    geom
    FROM parcels) sq
GROUP BY cid;
```

☒☒

ST_DWithin, ST_ClusterKMeans, ST_ClusterIntersecting, ST_ClusterIntersectingWin, ST_ClusterWithin, ST_ClusterWithinWin

7.17.2 ST_ClusterIntersecting

ST_ClusterIntersecting — Aggregate function that clusters input geometries into connected sets.

☒☒

geometry[] **ST_ClusterIntersecting**(geometry set g);

☒☒

An aggregate function that returns an array of GeometryCollections partitioning the input geometries into connected clusters that are disjoint. Each geometry in a cluster intersects at least one other geometry in the cluster, and does not intersect any geometry in other clusters.

2.2.0 ☒☒☒☒☒☒☒☒☒☒☒.

□□

Code

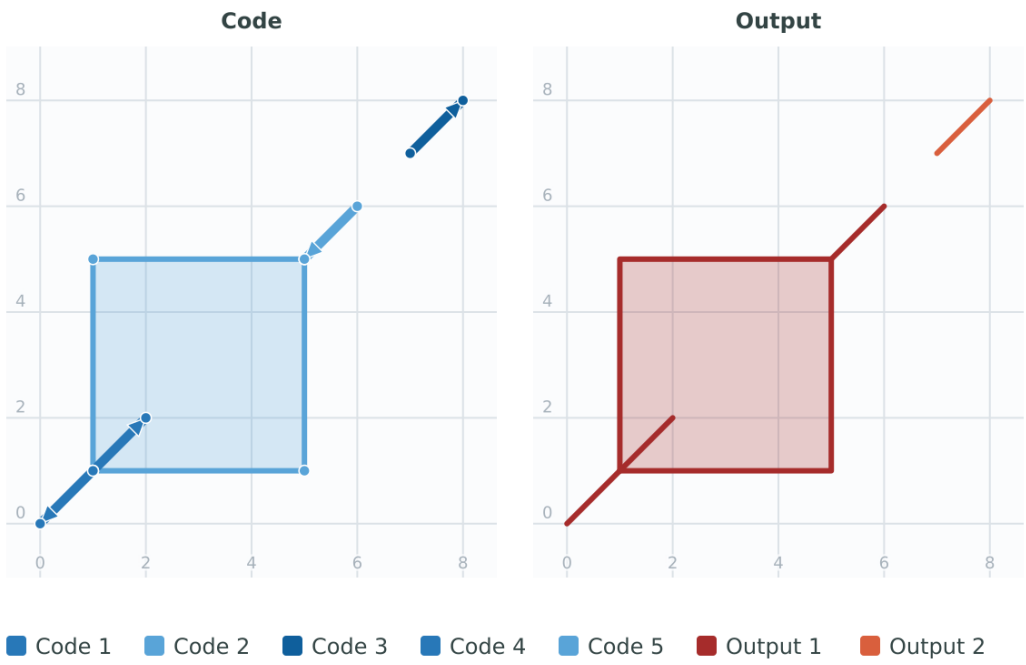
```
WITH testdata AS
  (SELECT unnest(
            ARRAY['LINESTRING (0 0,1 1)'::geometry,
            'LINESTRING (5 5,4 4)'::geometry,
            'LINESTRING (6 6,7 7)'::geometry,
            'LINESTRING (0 0,-1 -1)'::geometry,
            'POLYGON ((0 0,4 0,4 4,0 4,0 0))'::geometry]) AS geom)

SELECT unnest(ST_ClusterIntersecting(geom)) FROM testdata;
```

Output

```
GEOMETRYCOLLECTION(LINESTRING(0 0,1 1),LINESTRING(5 5,4 4),LINESTRING(0 0,-1 -1),POLYGON((0 0,4 0,4 4,0 4,0 0)))
GEOMETRYCOLLECTION(LINESTRING(6 6,7 7))
```

Figure



□□

ST_ClusterIntersectingWin, ST_ClusterWithin, ST_ClusterWithinWin

7.17.3 ST_ClusterIntersectingWin

ST_ClusterIntersectingWin — Window function that returns a cluster id for each input geometry, clustering input geometries into connected sets.

□□

integer **ST_ClusterIntersectingWin**(geometry winset geom);

□□

A window function that builds connected clusters of geometries that intersect. It is possible to traverse all geometries in a cluster without leaving the cluster. The return value is the cluster number that the geometry argument participates in, or null for null inputs.

Availability: 3.4.0

□□

Return an explicit cluster id for each input geometry. Using the conventional `cid` alias keeps the geometry column ready for color-by-cluster rendering.

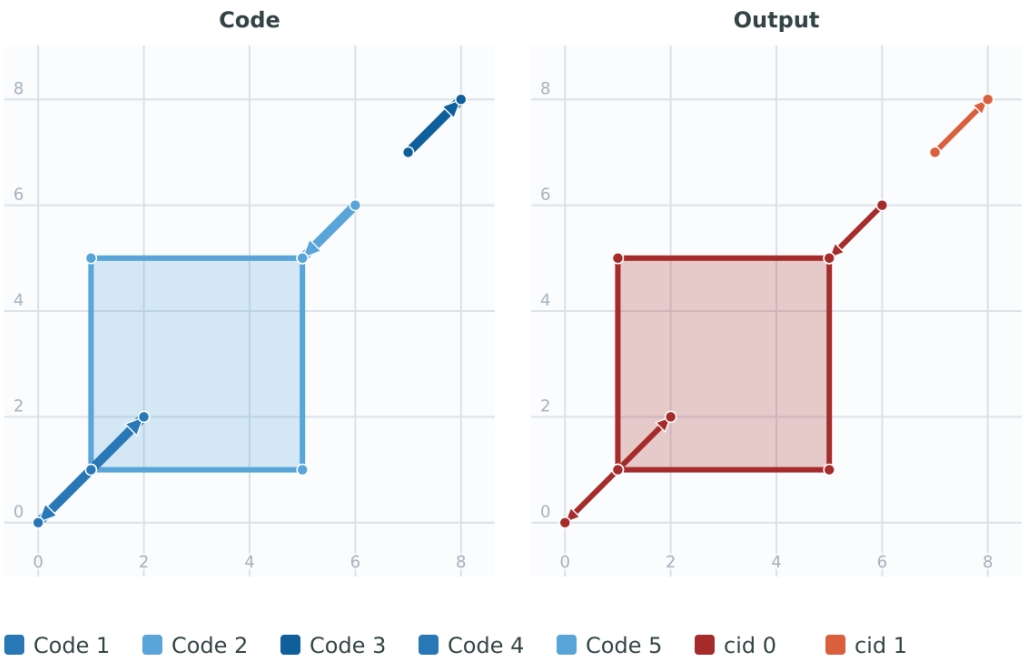
Code

```
WITH testdata(id, geom) AS (
  VALUES (1, 'LINESTRING (0 0,1 1)'::geometry),
         (2, 'LINESTRING (5 5,4 4)'::geometry),
         (3, 'LINESTRING (6 6,7 7)'::geometry),
         (4, 'LINESTRING (0 0,-1 -1)'::geometry),
         (5, 'POLYGON ((0 0,4 0,4 4,0 4,0 0))'::geometry)
), clustered AS (
  SELECT id,
         geom,
         ST_ClusterIntersectingWin(geom) OVER () AS cid
  FROM testdata
)
SELECT id,
       geom AS geom_wkt,
       cid
FROM clustered
ORDER BY id;
```

Output

```
id |            geom_wkt             | cid
----+---------------------------------+-----
  1 | LINESTRING(0 0,1 1)             |   0
  2 | LINESTRING(5 5,4 4)             |   0
  3 | LINESTRING(6 6,7 7)             |   1
  4 | LINESTRING(0 0,-1 -1)           |   0
  5 | POLYGON((0 0,4 0,4 4,0 4,0 0)) |   0
```

Figure



☒☒

ST_ClusterIntersecting, ST_ClusterWithin, ST_ClusterWithinWin

7.17.4 ST_ClusterRelateWin

ST_ClusterRelateWin — Window function that returns a cluster id for each input geometry, clustering input geometries into connected sets using the relate pattern to determine whether the geometries are connected.

☒☒

integer **ST_ClusterRelateWin**(geometry winset geom, text relate_matrix);

☒☒

A window function that builds connected clusters of geometries that intersect. Geometries are added to a cluster if they share a pairwise DE9IM relationship with another member of the cluster. With this function it is possible to build a cluster of all objects that touch at boundaries, but exclude those that merely overlap.

Availability: 3.7.0

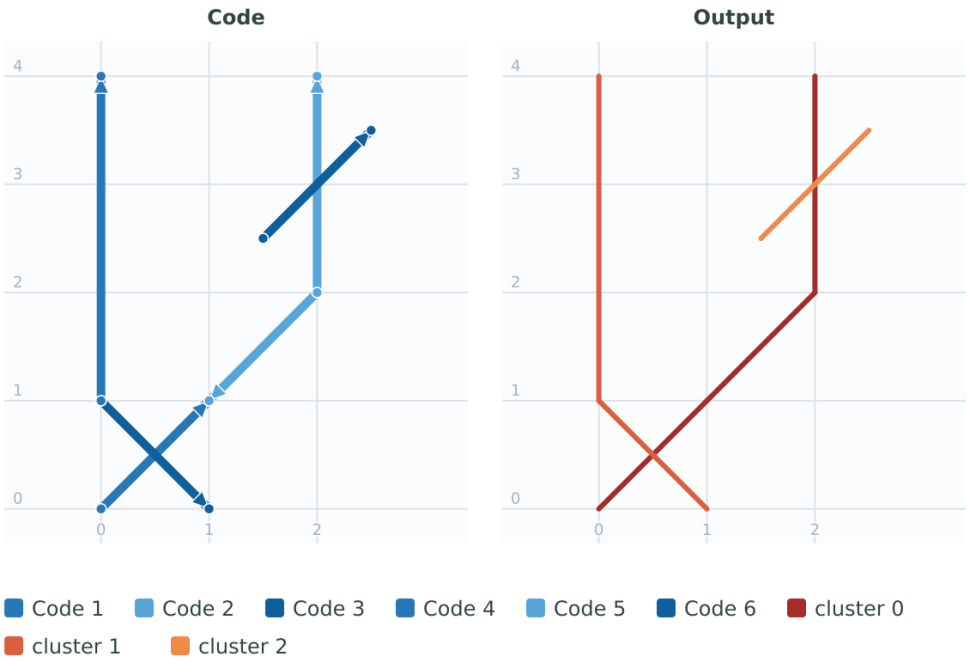
☒☒

This collection of line strings would form a single cluster using ST_ClusterIntersectingWin, but using ST_ClusterRelateWin can be clustered into three groups that connect only via their end points.

Code

```
WITH data(id, geom) AS (
  VALUES (1, 'LINESTRING(0 0,1 1)'::geometry),
         (2, 'LINESTRING(2 2,1 1)'::geometry),
         (3, 'LINESTRING(0 1,1 0)'::geometry),
         (4, 'LINESTRING(0 1,0 4)'::geometry),
         (5, 'LINESTRING(2 2,2 4)'::geometry),
         (6, 'LINESTRING(1.5 2.5,2.5 3.5)'::geometry)
), clustered AS (
  SELECT id, geom,
         ST_ClusterRelateWin(geom, '****0****') OVER () AS cluster
  FROM data
)
SELECT cluster,
       string_agg(id::text, ',' ORDER BY id) AS members,
       ST_Collect(geom ORDER BY id) AS cluster_geom
FROM clustered
GROUP BY cluster
ORDER BY cluster;
```

Figure



☒☒

ST_Relate, ST_ClusterIntersectingWin, ST_ClusterWithinWin

7.17.5 ST_MinimumSpanningTree

ST_MinimumSpanningTree — Window function that returns a tree id for each input geometry, clustering input geometries into connected trees using the Minimum Spanning Tree algorithm.

□□

integer **ST_MinimumSpanningTree**(geometry winset geom);

□□

A window function that builds connected graphs of line strings based on the Minimum Spanning Tree (MST) of the input geometries. The return value is the cluster number that the geometry argument participates in, or zero if it is not part of the minimum tree.

The Minimum Spanning Tree connects all geometries in the window partition such that the total length of the connecting lines is minimized. If the graph is not fully connected (e.g. infinite distance between some geometries), it produces a Minimum Spanning Forest, and each connected component is assigned a unique tree ID.

Availability: 3.7.0

Requires GEOS >= 3.15.0

□□

These six edges describe a square with both diagonals. The minimum spanning tree keeps three edges and excludes the rest.

Code

```
WITH edges(id, geom) AS (
  VALUES
    (1, 'LINESTRING(0 0,1 0)'::geometry),
    (2, 'LINESTRING(0 0,0 1)'::geometry),
    (3, 'LINESTRING(1 1,0 1)'::geometry),
    (4, 'LINESTRING(1 1,1 0)'::geometry),
    (5, 'LINESTRING(0 0,1 1)'::geometry),
    (6, 'LINESTRING(1 0,0 1)'::geometry)
), marked AS (
  SELECT id,
         geom,
         ST_MinimumSpanningTree(geom) OVER (ORDER BY id) AS tree_id
  FROM edges
)
SELECT string_agg(id::text, ',' ORDER BY id)
         FILTER (WHERE tree_id > 0) AS tree_edge_ids,
       ST_Collect(geom ORDER BY id)
         FILTER (WHERE tree_id > 0) AS tree_edges,
       string_agg(id::text, ',' ORDER BY id)
         FILTER (WHERE tree_id = 0) AS excluded_edge_ids,
       ST_Collect(geom ORDER BY id)
         FILTER (WHERE tree_id = 0) AS excluded_edges
FROM marked;
```

Output

```
-[ RECORD 1 ]-----
tree_edge_ids     | 1,2,3
tree_edges        | MULTILINESTRING((0 0,1 0),(0 0,0 1),(1 1,0 1))
excluded_edge_ids | 4,5,6
excluded_edges    | MULTILINESTRING((1 1,1 0),(0 0,1 1),(1 0,0 1))
```

□□

ST_ClusterDBSCAN, ST_ClusterKMeans, ST_ClusterIntersectingWin

7.17.6 ST_ClusterKMeans

ST_ClusterKMeans — Window function that returns a cluster id for each input geometry using the K-means algorithm.

□□

integer **ST_ClusterKMeans**(geometry winset geom , integer k , float8 max_radius);

□□

Returns K-means cluster number for each input geometry. The distance used for clustering is the distance between the centroids for 2D geometries, and distance between bounding box centers for 3D geometries. For POINT inputs, M coordinate will be treated as weight of input and has to be larger than 0.

`max_radius`, if set, will cause ST_ClusterKMeans to generate more clusters than `k` ensuring that no cluster in output has radius larger than `max_radius`. This is useful in reachability analysis.

Enhanced: 3.2.0 Support for `max_radius`

Enhanced: 3.1.0 Support for 3D geometries and weights

2.3.0 □□□□□□□□□□□□.

□□

Define the parcel geometries once in a CTE and reuse them across the executable examples.

Parcels color-coded by cluster number (`cid`).

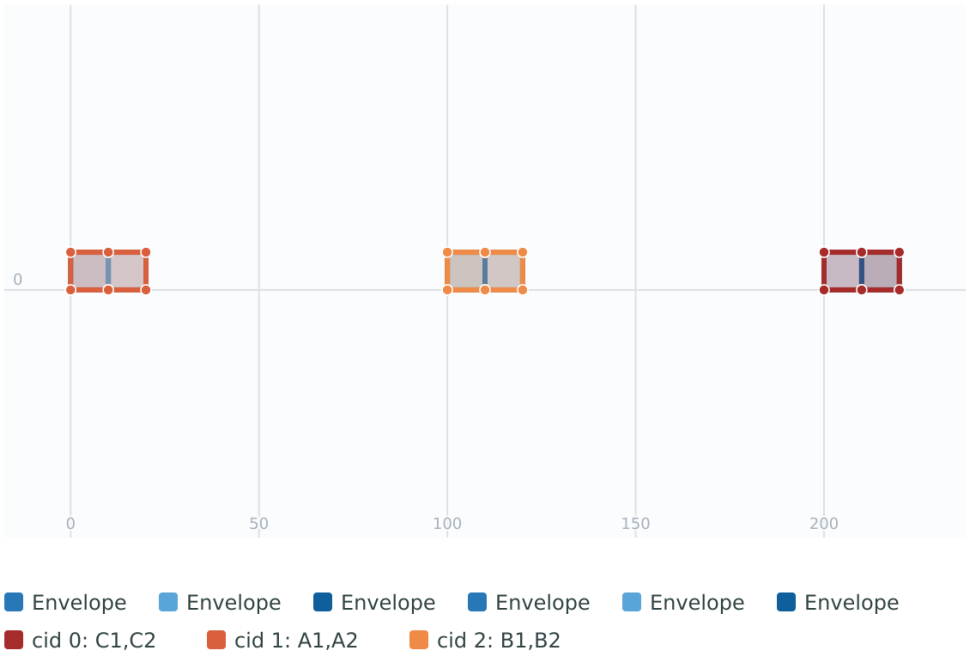
Code

```
WITH parcels(parcel_id, geom) AS (
  VALUES
    ('A1', ST_MakeEnvelope(0, 0, 10, 10)),
    ('A2', ST_MakeEnvelope(10, 0, 20, 10)),
    ('B1', ST_MakeEnvelope(100, 0, 110, 10)),
    ('B2', ST_MakeEnvelope(110, 0, 120, 10)),
    ('C1', ST_MakeEnvelope(200, 0, 210, 10)),
    ('C2', ST_MakeEnvelope(210, 0, 220, 10))
), clustered AS (
  SELECT parcel_id,
         geom,
         ST_ClusterKMeans(geom, 3) OVER (ORDER BY parcel_id) AS cid
  FROM parcels
)
SELECT cid,
       string_agg(parcel_id, ',' ORDER BY parcel_id) AS parcels,
       ST_Union(geom) AS cluster_geom
FROM clustered
GROUP BY cid
ORDER BY cid;
```

Output

```
0 | C1,C2 | POLYGON((200 10,210 10,220 10,220 0,210 0,200 0,200 10))
1 | A1,A2 | POLYGON((0 10,10 10,20 10,20 0,10 0,0 0,0 10))
2 | B1,B2 | POLYGON((100 10,110 10,120 10,120 0,110 0,100 0,100 10))
```

Figure



Partitioning parcel clusters by type:

Code

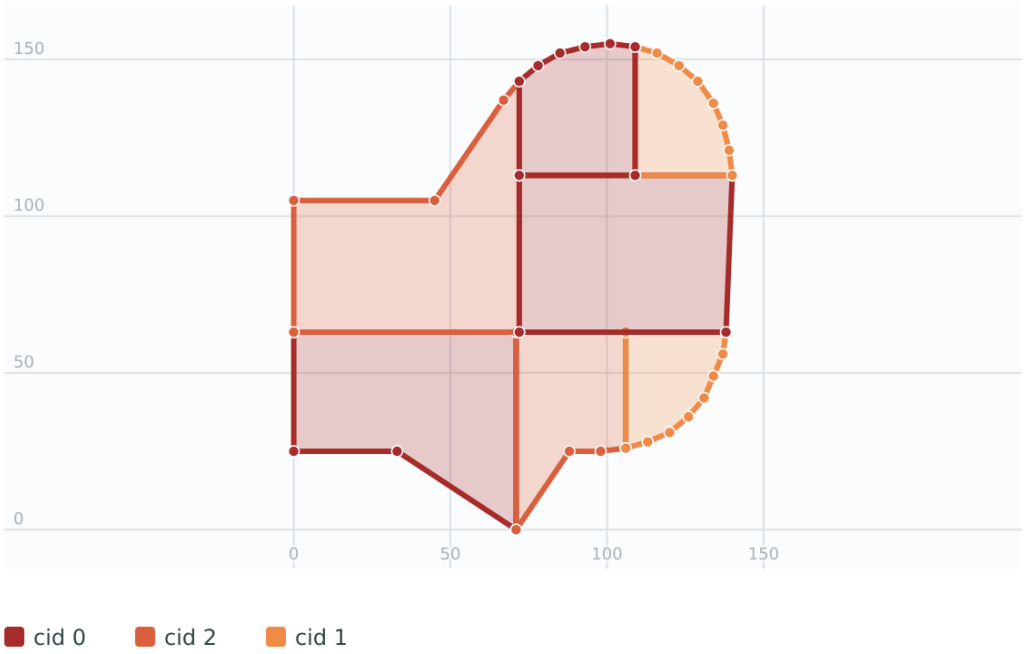
```
WITH parcel_geoms AS (
  SELECT geom
  FROM ST_Subdivide(
    ST_Buffer('SRID=3857;LINESTRING(40 100,98 100,100 150,60 90)'::geometry,
              40, 'endcap=square'),
    12) AS geom
), parcels AS (
  SELECT lpad(row_number() OVER (ORDER BY ST_YMin(geom), ST_XMin(geom))::text, 3, '0') AS ↩
    parcel_id,
         geom,
         ('{residential,commercial}'::text[])[1 + mod(row_number() OVER (ORDER BY ST_YMin( ↩
    geom), ST_XMin(geom)), 2)] AS type
  FROM parcel_geoms
)
SELECT ST_ClusterKMeans(geom, 3) over (PARTITION BY type) AS cid,
       parcel_id,
       type,
       ST_SnapToGrid(geom, 1) AS parcel_geom
FROM parcels
ORDER BY type, parcel_id;
```

Output

```
cid | parcel_id |    type     | parcel_geom
----+-----------+-------------+-----------------------------------------------------------------
   0 | 001       | commercial  | POLYGON((33 60,0 60,0 98,71 98,71 35,33 60))
   2 | 003       | commercial  | POLYGON((106 98,138 98,137 91,134 84,131 77,126 71,120 66,113 63,106 61,106 98))
   2 | 005       | commercial  | POLYGON((140 148,138 98,72 98,72 148,140 148))
```

```
   1 | 007       | commercial  | POLYGON((109 189,116 187,123 183,129 178,134 171,137 164,139 156,140 148,109 148,109 ↩
     189))
   0 | 002       | residential | POLYGON((71 98,106 98,106 61,98 60,88 60,71 35,71 98))
   2 | 004       | residential | POLYGON((0 98,0 140,45 140,67 172,72 178,72 98,0 98))
   1 | 006       | residential | POLYGON((72 178,78 183,85 187,93 189,101 190,109 189,109 148,72 148,72 178))
```

Figure



Example: Clustering a preaggregated planetary-scale data population dataset using 3D clustering and weighting. Identify at least 20 regions based on Kontur Population Data that do not span more than 3000 km from their center:

Code

```
CREATE TABLE kontur_population_3000km_clusters AS
SELECT
    geom,
    ST_ClusterKMeans(
        ST_Force4D(
            ST_Transform(ST_Force3D(geom), 4978),
            mvalue => population
        ),
        20,
        max_radius => 3000000
    ) OVER () AS cid
FROM kontur_population;
```

The clustering produces 46 regions. Clusters are centered at well-populated regions such as New York and Moscow. Greenland forms one cluster, several island clusters span the antimeridian, and cluster edges follow the Earth's curvature.



Kontur population clustered with 3000 km maximum radius.

Cluster weighted city populations on the Earth in geocentric coordinates. The maximum radius is specified in meters, while the returned geometries stay in WGS 84 for display.

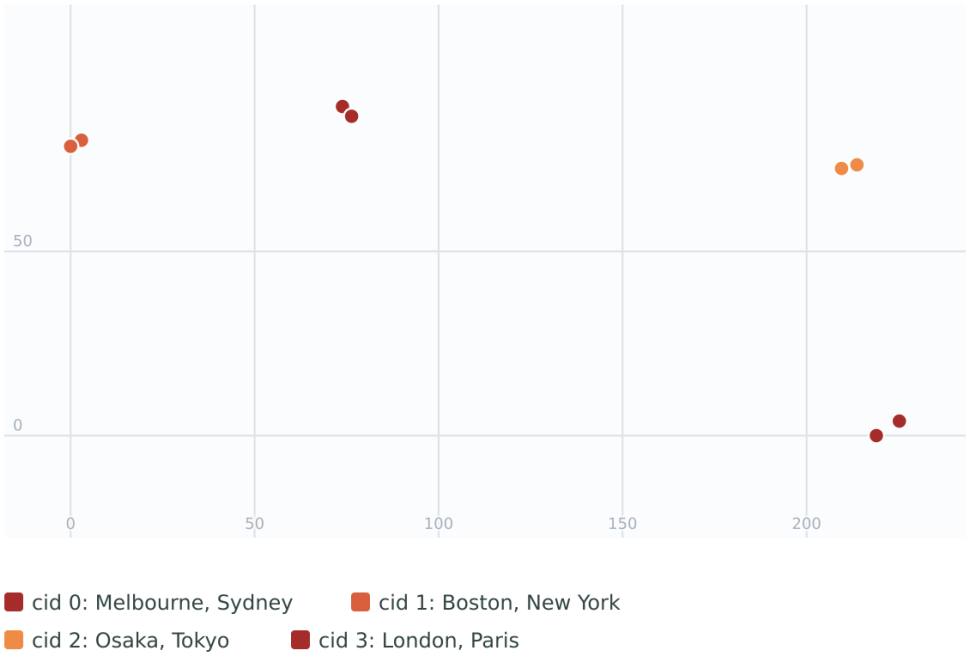
Code

```
WITH population(place, population, geom) AS (
  VALUES
    ('Boston', 5.0, ST_SetSRID(ST_Point(-71.06, 42.36), 4326)),
    ('New York', 20.0, ST_SetSRID(ST_Point(-74.01, 40.71), 4326)),
    ('London', 15.0, ST_SetSRID(ST_Point(-0.13, 51.51), 4326)),
    ('Paris', 11.0, ST_SetSRID(ST_Point(2.35, 48.86), 4326)),
    ('Osaka', 19.0, ST_SetSRID(ST_Point(135.50, 34.69), 4326)),
    ('Tokyo', 37.0, ST_SetSRID(ST_Point(139.69, 35.68), 4326)),
    ('Melbourne', 5.0, ST_SetSRID(ST_Point(144.96, -37.81), 4326)),
    ('Sydney', 5.0, ST_SetSRID(ST_Point(151.21, -33.87), 4326))
), clustered AS (
  SELECT place,
         geom,
         ST_ClusterKMeans(
           ST_Force4D(
             ST_Transform(ST_Force3D(geom), 4978),
             mvalue => population
           ),
           4,
           max_radius => 3000000
         ) OVER (ORDER BY place) AS cid
  FROM population
)
SELECT cid,
       string_agg(place, ', ' ORDER BY place) AS places,
       ST_Collect(geom ORDER BY place) AS cluster_geom
FROM clustered
GROUP BY cid
ORDER BY cid;
```

Output

```
0 | Melbourne, Sydney | MULTIPOINT((144.96 -37.81),(151.21 -33.87))
1 | Boston, New York | MULTIPOINT((-71.06 42.36),(-74.01 40.71))
2 | Osaka, Tokyo | MULTIPOINT((135.5 34.69),(139.69 35.68))
3 | London, Paris | MULTIPOINT((-0.13 51.51),(2.35 48.86))
```

Figure



□□

ST_ClusterDBSCAN, ST_ClusterIntersectingWin, ST_ClusterWithinWin, ST_ClusterIntersecting, ST_Cluster
ST_Subdivide, ST_Force3D, ST_Force4D,

7.17.7 ST_ClusterWithin

ST_ClusterWithin — Aggregate function that clusters geometries by separation distance.

□□

geometry[] **ST_ClusterWithin**(geometry set g, float8 distance);

□□

An aggregate function that returns an array of GeometryCollections, where each collection is a cluster containing some input geometries. Clustering partitions the input geometries into sets in which each geometry is within the specified *`distance`* of at least one other geometry in the same cluster. Distances are Cartesian distances in the units of the SRID.

ST_ClusterWithin is equivalent to running ST_ClusterDBSCAN with `minpoints => 0`.

2.2.0 □□□□□□□□□□□□.

This method supports Circular Strings and Curves.

□□

Code

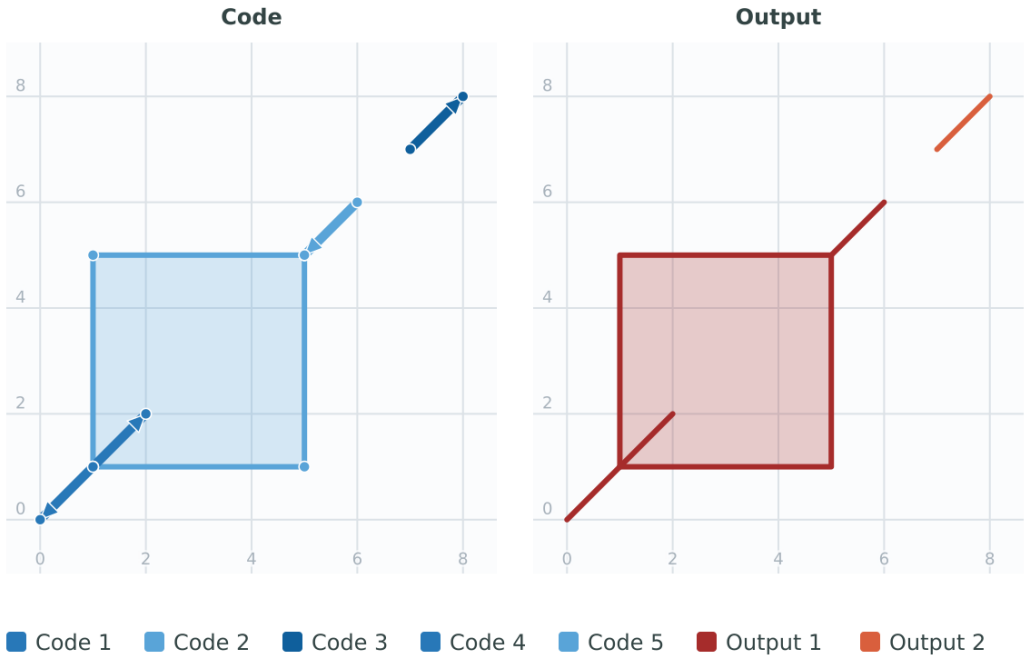
```
WITH testdata AS
  (SELECT unnest(ARRAY['LINESTRING (0 0,1 1)'::geometry,
                'LINESTRING (5 5,4 4)'::geometry,
                'LINESTRING (6 6,7 7)'::geometry,
                'LINESTRING (0 0,-1 -1)'::geometry,
                'POLYGON ((0 0,4 0,4 4,0 4,0 0))'::geometry]) AS geom)

SELECT unnest(ST_ClusterWithin(geom, 1.4)) FROM testdata;
```

Output

```
GEOMETRYCOLLECTION(LINESTRING(0 0,1 1),LINESTRING(5 5,4 4),LINESTRING(0 0,-1 -1),POLYGON((0 0,4 0,4 4,0 4,0 0)))
GEOMETRYCOLLECTION(LINESTRING(6 6,7 7))
```

Figure



☒☒

ST_ClusterWithinWin, ST_ClusterDBSCAN, ST_ClusterIntersecting, ST_ClusterIntersectingWin

7.17.8 ST_ClusterWithinWin

ST_ClusterWithinWin — Window function that returns a cluster id for each input geometry, clustering using separation distance.

☒☒

integer **ST_ClusterWithinWin**(geometry winset geom, float8 distance);

□□

A window function that returns a cluster number for each input geometry. Clustering partitions the geometries into sets in which each geometry is within the specified `distance` of at least one other geometry in the same cluster. Distances are Cartesian distances in the units of the SRID.

ST_ClusterWithinWin is equivalent to running ST_ClusterDBSCAN with `minpoints => 0`.

Availability: 3.4.0

 This method supports Circular Strings and Curves.

□□

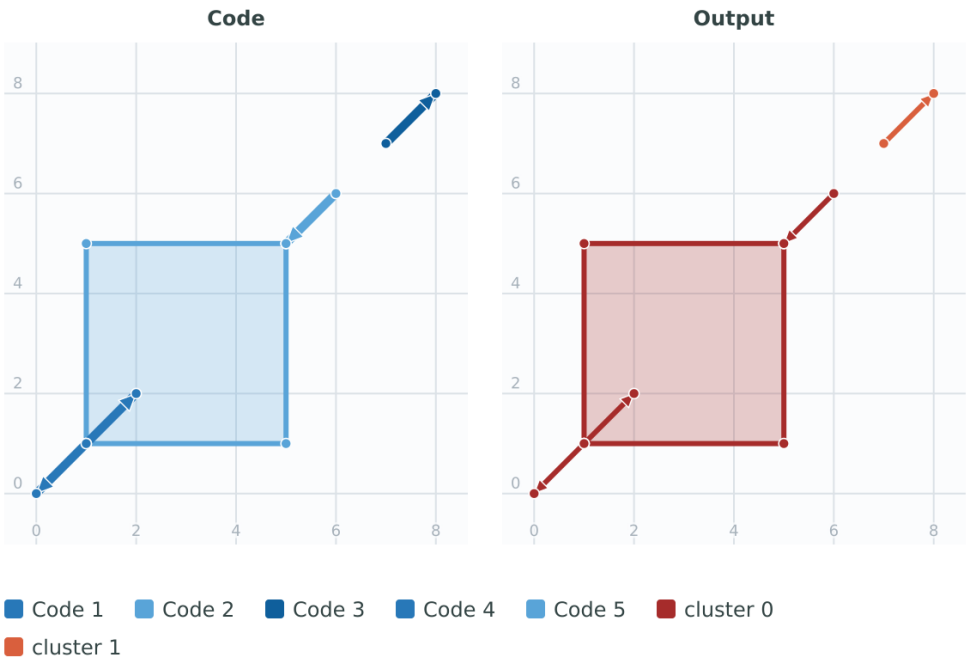
Code

```
WITH testdata AS (
  SELECT id, geom::geometry FROM (
  VALUES  (1, 'LINESTRING (0 0,1 1)'),
          (2, 'LINESTRING (5 5,4 4)'),
          (3, 'LINESTRING (6 6,7 7)'),
          (4, 'LINESTRING (0 0,-1 -1)'),
          (5, 'POLYGON ((0 0,4 0,4 4,0 4,0 0))')) AS t(id, geom)
)
SELECT id,
  geom,
  ST_ClusterWithinWin(geom, 1.4) OVER () AS cluster
FROM testdata;
```

Output

```
id |          st_astext            | cluster
----+-------------------------------+---------
  1 | LINESTRING(0 0,1 1)           |       0
  2 | LINESTRING(5 5,4 4)           |       0
  3 | LINESTRING(6 6,7 7)           |       1
  4 | LINESTRING(0 0,-1 -1)         |       0
  5 | POLYGON((0 0,4 0,4 4,0 4,0 0)) |       0
```

Figure



□□

ST_ClusterWithin, ST_ClusterDBSCAN, ST_ClusterIntersecting, ST_ClusterIntersectingWin,

7.18 Bounding Box Functions

7.18.1 Box2D

Box2D — Returns a BOX2D representing the 2D extent of a geometry.

□□

box2d **Box2D**(geometry geom);

□□

Returns a box2d representing the 2D extent of the geometry.
□□□□: 2.0.0 □□□□□□□□□□, □□□□ TIN □□□□□□□□□□□.

This method supports Circular Strings and Curves.

This function supports Polyhedral surfaces.

This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).

□□

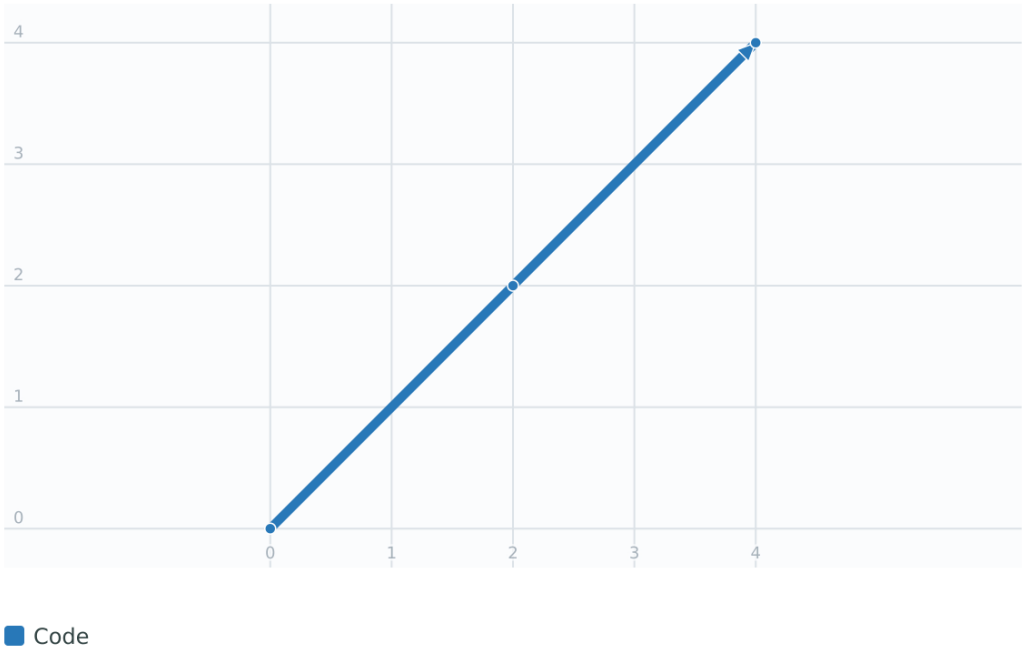
Code

```
SELECT Box2D('LINESTRING(1 2,3 4,5 6)'::geometry);
```

Output

```
BOX(1 2,5 6)
```

Figure



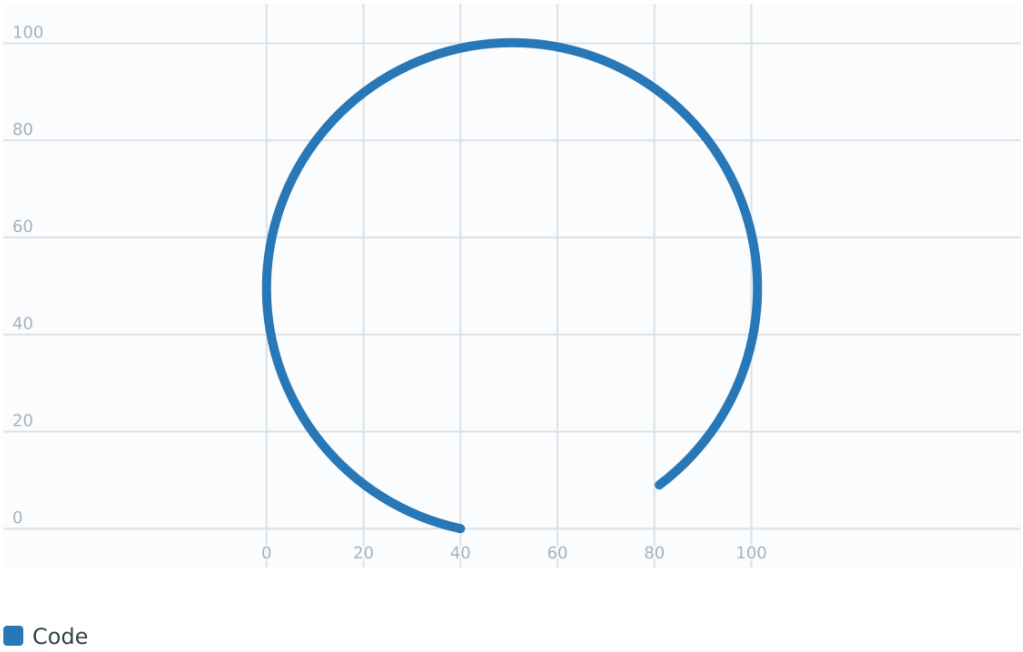
Code

```
SELECT Box2D('CIRCULARSTRING(220268 150415,220227 150505,220227 150406)'::geometry);
```

Output

```
BOX(220186.99512189245 150406,220288.24878054656 150506.12682932706)
```

Figure



□□

Box3D, ST_GeomFromText

7.18.2 Box3D

Box3D — Returns a BOX3D representing the 3D extent of a geometry.

□□

box3d **Box3D**(geometry geom);

□□

Returns a box3d representing the 3D extent of the geometry.
□□□□: 2.0.0 □□□□□□□□□□, □□□□ TIN □□□□□□□□□□□□.

This method supports Circular Strings and Curves.

This function supports Polyhedral surfaces.

This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).

This function supports 3d and will not drop the z-index.

□□

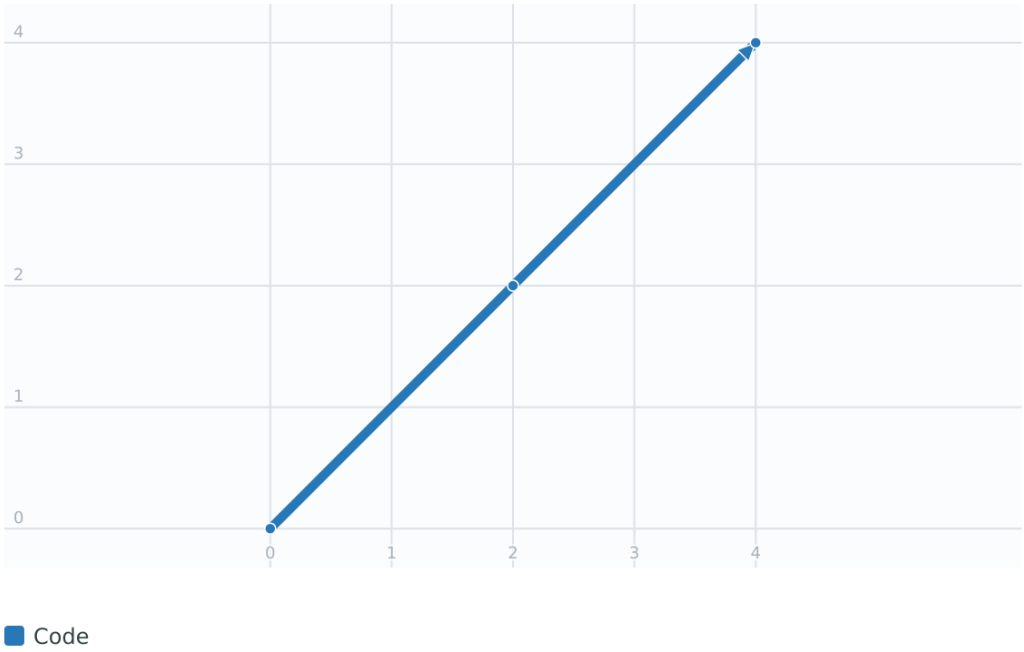
Code

```
SELECT Box3D('LINESTRING(1 2 3,3 4 5,5 6 5)'::geometry);
```

Output

```
BOX3D(1 2 3,5 6 5)
```

Figure



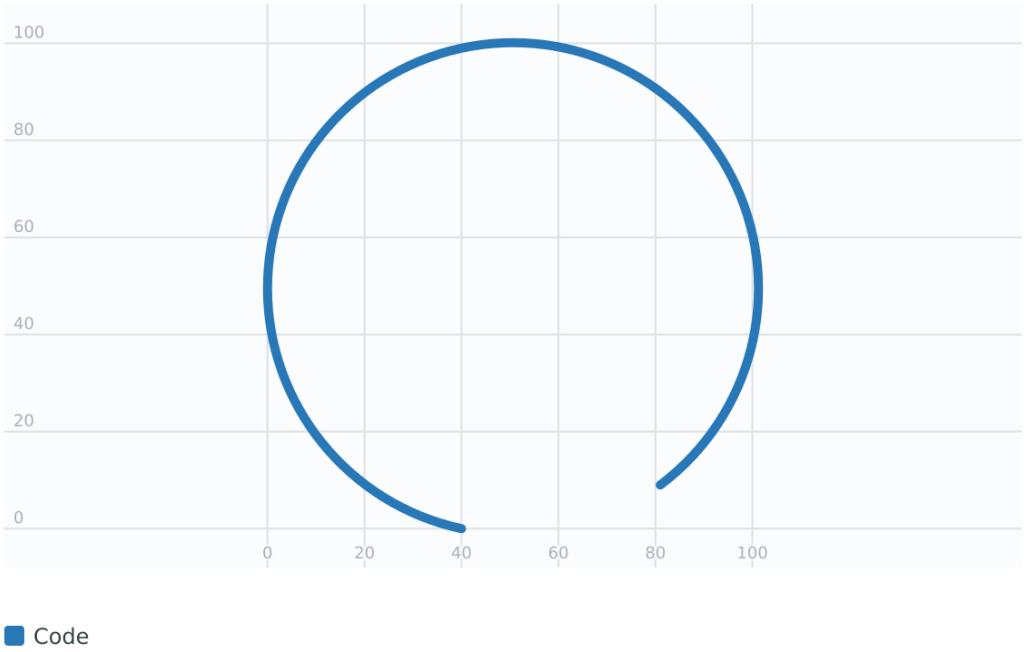
Code

```
SELECT Box3D('CIRCULARSTRING(220268 150415 1,220227 150505 1,220227 150406 1)'::geometry);
```

Output

```
BOX3D(220186.99512189245 150406 1,220288.24878054656 150506.12682932706 1)
```

Figure



□□

Box2D, ST_GeomFromEWKT

7.18.3 ST_EstimatedExtent

ST_EstimatedExtent — Returns the estimated extent of a spatial table.

□□

box2d **ST_EstimatedExtent**(text schema_name, text table_name, text geocolumn_name, boolean parent_only);
box2d **ST_EstimatedExtent**(text schema_name, text table_name, text geocolumn_name);
box2d **ST_EstimatedExtent**(text table_name, text geocolumn_name);

□□

Returns the estimated extent of a spatial table as a box2d. The current schema is used if not specified. The estimated extent is taken from the geometry column's statistics. This is usually much faster than computing the exact extent of the table using ST_Extent or ST_3DExtent.

The default behavior is to also use statistics collected from child tables (tables with INHERITS) if available. If `parent_only` is set to TRUE, only statistics for the given table are used and child tables are ignored.

For PostgreSQL >= 8.0.0 statistics are gathered by VACUUM ANALYZE and the result extent will be about 95% of the actual one. For PostgreSQL < 8.0.0 statistics are gathered by running `update_geometry_st` and the result extent is exact.



□□
In the absence of statistics (empty table or no ANALYZE called) this function returns NULL. Prior to version 1.5.4 an exception was thrown instead.



□□
Escaping names for tables and/or namespaces that include special characters and quotes may require special handling. A user notes: ”For schemas and tables, use identifier escaping rules to produce a double-quoted string, and afterwards remove the first and last double-quote character. For geometry column pass as is.”

1.0.0 □□□□□□□□□□□□□.

Changed: 2.1.0. Up to 2.0.x this was called ST_Estimated_Extent.

This method supports Circular Strings and Curves.

□□

Code

```
SELECT ST_EstimatedExtent('ny', 'edges', 'geom');
```

Output

```
st_estimatedextent
--------------------------------------------------
 BOX(-8877653 4912316,-8010225.5 5589284)
(1 row)
```

Code

```
SELECT ST_EstimatedExtent('feature_poly', 'geom');
```

Output

```
st_estimatedextent
----------------------------------------------------------------------
 BOX(-124.659652709961 24.6830825805664,-67.7798080444336 49.0012092590332)
(1 row)
```

□□

ST_Extent, ST_3DExtent

7.18.4 ST_Expand

ST_Expand — Returns a bounding box expanded from another bounding box or a geometry.

□□

geometry **ST_Expand**(geometry geom, float units_to_expand);
geometry **ST_Expand**(geometry geom, float dx, float dy, float dz=0, float dm=0);
box2d **ST_Expand**(box2d box, float units_to_expand);
box2d **ST_Expand**(box2d box, float dx, float dy);
box3d **ST_Expand**(box3d box, float units_to_expand);
box3d **ST_Expand**(box3d box, float dx, float dy, float dz=0);

□□

Returns a bounding box expanded from the bounding box of the input, either by specifying a single distance with which the box should be expanded on both axes, or by specifying an expansion distance for each axis. Uses double-precision. Can be used for distance queries, or to add a bounding box filter to a query to take advantage of a spatial index.

In addition to the version of ST_Expand accepting and returning a geometry, variants are provided that accept and return box2d and box3d data types.

Distances are in the units of the spatial reference system of the input.

ST_Expand is similar to ST_Buffer, except while buffering expands a geometry in all directions, ST_Expand expands the bounding box along each axis.



□□
Pre version 1.3, ST_Expand was used in conjunction with ST_Distance to do indexable distance queries. For example, `geom && ST_Expand('POINT(10 20)', 10) AND ST_Distance(geom, 'POINT(10 20)') < 10`. This has been replaced by the simpler and more efficient ST_DWithin function.

Availability: 1.5.0 behavior changed to output double precision instead of float4 coordinates.

□□□□: 2.0.0 □□□□□□□□□□, □□□□ TIN □□□□□□□□□□□□.

Enhanced: 2.3.0 support was added to expand a box by different amounts in different dimensions.

This function supports Polyhedral surfaces.

This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).

□□



□□
Examples below use US National Atlas Equal Area (SRID=2163) which is a meter projection

10-meter expanded box around bbox of a linestring.

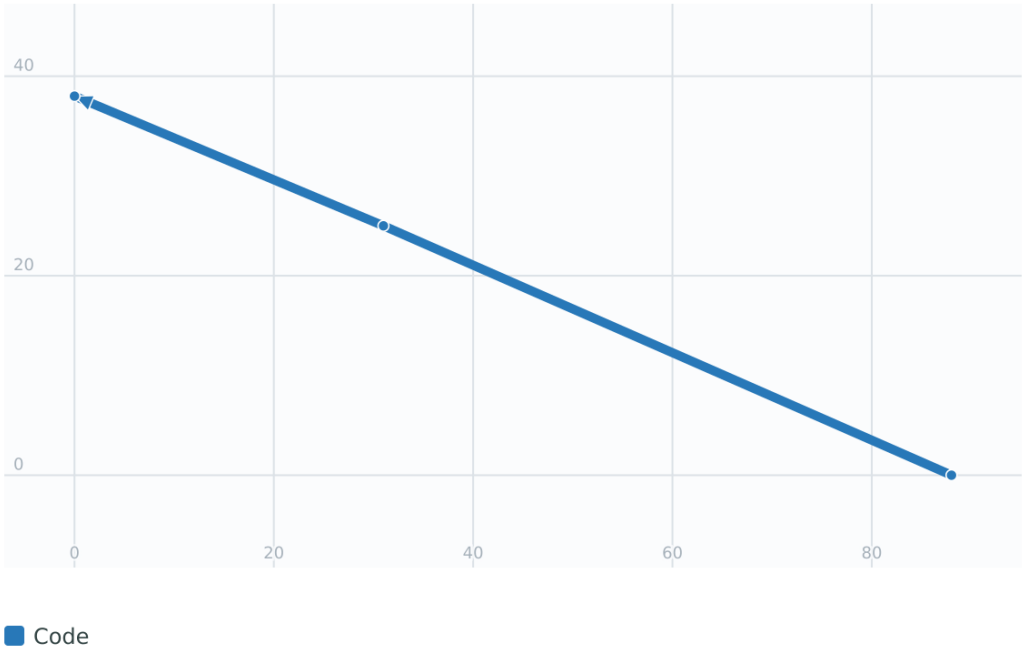
Code

```
SELECT CAST(ST_Expand(ST_GeomFromText('LINESTRING(2312980 110676,2312923 110701,2312892 ↩
    110714)', 2163), 10) As box2d);
```

Output

```
BOX(2312882 110666,2312990 110724)
```

Figure



This example expands a 3D box by 10 meters.

Code

```
SELECT ST_Expand(CAST('BOX3D(778783 2951741 1,794875 2970042.61545891 10)' AS box3d), 10)
```

Output

```
BOX3D(778773 2951731 -9,794885 2970052.61545891 20)
```

This example renders the text representation of a 10-meter expanded box around a point geometry.

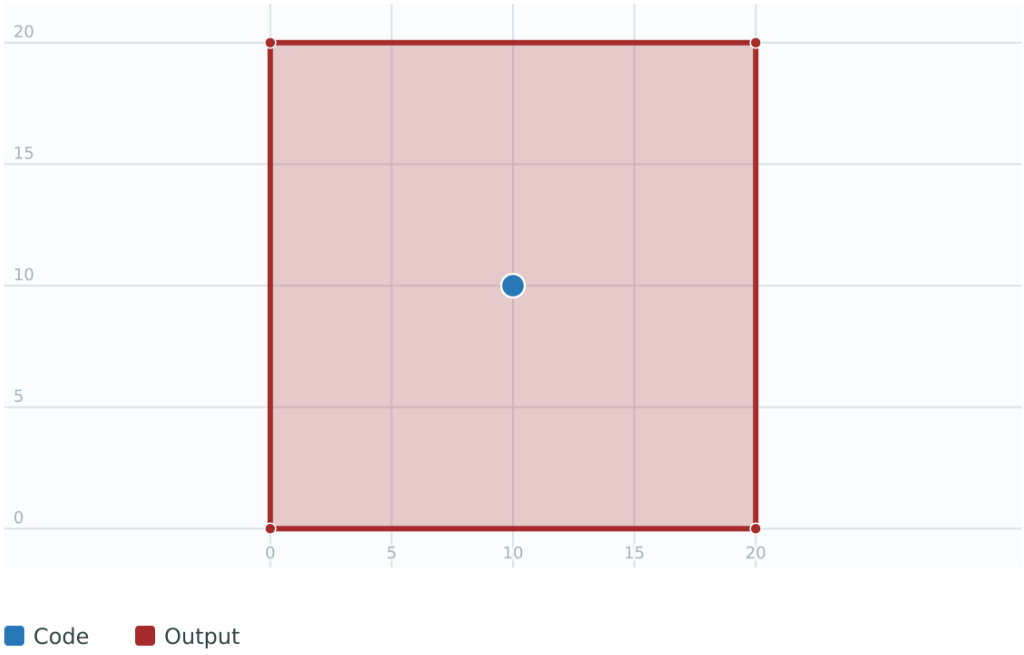
Code

```
SELECT ST_Expand('SRID=2163;POINT(2312980 110676)'::geometry, 10);
```

Output

```
SRID=2163;POLYGON((2312970 110666,2312970 110686,2312990 110686,2312990 110666,2312970 110666))
```

Figure



□□

ST_Buffer, ST_DWithin, ST_SRID

7.18.5 ST_Extent

ST_Extent — Aggregate function that returns the bounding box of geometries.

□□

box2d **ST_Extent**(geometry set geomfield);

□□

An aggregate function that returns a box2d bounding box that bounds a set of geometries.
The bounding box coordinates are in the spatial reference system of the input geometries.
ST_Extent is similar in concept to Oracle Spatial/Locator's SDO_AGGR_MBR.



□□
ST_Extent returns boxes with only X and Y ordinates even with 3D geometries. To return XYZ ordinates use ST_3DExtent.



□□
The returned box3d value does not include a SRID. Use ST_SetSRID to convert it into a geometry with SRID metadata. The SRID is the same as the input geometries.

□□□□: 2.0.0 □□□□□□□□□□, □□□□ TIN □□□□□□□□□□□□.

This function supports Polyhedral surfaces.

This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).

□□

Note! □□
Examples below use Massachusetts State Plane ft (SRID=2249)

Code

```
SELECT ST_Extent(geom) AS bextent FROM sometable;
```

Output

```
BOX(739651.875 2908247.25,794875.8125 2970042.75)
```

Return extent of each geometry category.

Code

```
SELECT ST_Extent(geom) as bextent
FROM sometable
GROUP BY category ORDER BY category;
```

Output

```
bextent                       |        name
------------------------------------------------+----------------
 BOX(778783.5625 2951741.25,794875.8125 2970042.75) | A
 BOX(751315.8125 2919164.75,765202.6875 2935417.25) | B
 BOX(739651.875 2917394.75,756688.375 2935866)      | C
```

This example converts the extent back into a geometry and renders its extended text representation.

Code

```
SELECT ST_SetSRID(ST_Extent(geom), 2249) AS bextent FROM sometable;
```

Output

```
SRID=2249;POLYGON((739651.875 2908247.25,739651.875 2970042.75,794875.8125 2970042.75,
794875.8125 2908247.25,739651.875 2908247.25))
```

□□

ST_EstimatedExtent, ST_3DExtent, ST_SetSRID

7.18.6 ST_3DExtent

ST_3DExtent — Aggregate function that returns the 3D bounding box of geometries.

□□

box3d **ST_3DExtent**(geometry set geomfield);

□□

An aggregate function that returns a box3d (includes Z ordinate) bounding box that bounds a set of geometries.

The bounding box coordinates are in the spatial reference system of the input geometries.



□□
The returned box3d value does not include a SRID. Use ST_SetSRID to convert it into a geometry with SRID metadata. The SRID is the same as the input geometries.

□□□□: 2.0.0 □□□□□□□□□□, □□□□ TIN □□□□□□□□□□□□.

Changed: 2.0.0 In prior versions this used to be called ST_Extent3D

This function supports 3d and will not drop the z-index.

This method supports Circular Strings and Curves.

This function supports Polyhedral surfaces.

This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).

□□

Code

```
SELECT ST_3DExtent(foo.geom) As b3extent
FROM (SELECT ST_MakePoint(x, y, z) As geom
    FROM generate_series(1, 3) As x
        CROSS JOIN generate_series(1, 2) As y
        CROSS JOIN generate_series(0, 2) As Z) As foo;
```

Output

```
BOX3D(1 1 0,3 2 2)
```

This example returns the extent of several elevated CircularStrings.

Code

```
SELECT ST_3DExtent(foo.geom) As b3extent
FROM (SELECT ST_Translate(ST_Force3DZ(ST_LineToCurve(ST_Buffer(ST_Point(x, y), 1))), 0, 0,  ↩
    z) As geom
    FROM generate_series(1, 3) As x
        CROSS JOIN generate_series(1, 2) As y
        CROSS JOIN generate_series(0, 2) As Z) As foo;
```

Output

```
BOX3D(0 0 0,4 3 2)
```

□□

ST_Extent, ST_Force3DZ, ST_SetSRID

7.18.7 ST_MakeBox2D

ST_MakeBox2D — Creates a BOX2D defined by two 2D point geometries.

□□

box2d **ST_MakeBox2D**(geometry pointLowLeft, geometry pointUpRight);

□□

Creates a box2d defined by two Point geometries. This is useful for doing range queries.

□□

Return all features that fall within or partly reside in a US National Atlas coordinate bounding box. It is assumed here that the geometries are stored with SRID=2163 (US National Atlas equal area).

Code

```
SELECT feature_id, feature_name, geom
FROM features
WHERE geom && ST_SetSRID(
  ST_MakeBox2D(
    ST_Point(-989502.1875, 528439.5625),
    ST_Point(-987121.375, 529933.1875)
  ),
  2163
);
```

□□

ST_Point, ST_SetSRID, ST_SRID

7.18.8 ST_3DMakeBox

ST_3DMakeBox — Creates a BOX3D defined by two 3D point geometries.

□□

box3d **ST_3DMakeBox**(geometry point3DLowLeftBottom, geometry point3DUpRightTop);

□□

Creates a box3d defined by two 3D Point geometries.

This function supports 3D and will not drop the z-index.

Changed: 2.0.0 In prior versions this used to be called ST_MakeBox3D

□□

Code

```
SELECT ST_3DMakeBox(ST_MakePoint(-989502.1875, 528439.5625, 10),
    ST_MakePoint(-987121.375 , 529933.1875, 10))
```

Output

```
BOX3D(-989502.1875 528439.5625 10,-987121.375 529933.1875 10)
```

□□

ST_MakePoint, ST_SetSRID, ST_SRID

7.18.9 ST_XMax

ST_XMax — Returns the X maxima of a 2D or 3D bounding box or a geometry.

□□

float **ST_XMax**(box3d aGeomorBox2DorBox3D);

□□

Returns the X maxima of a 2D or 3D bounding box or a geometry.



□□
Although this function is only defined for box3d, it also works for box2d and geometry values due to automatic casting. However, it will not accept a geometry or box2d text representation, since those do not auto-cast.

This function supports 3d and will not drop the z-index.

This method supports Circular Strings and Curves.

□□

Code

```
SELECT ST_XMax('BOX3D(1 2 3,4 5 6)');
```

Output

```
4
```

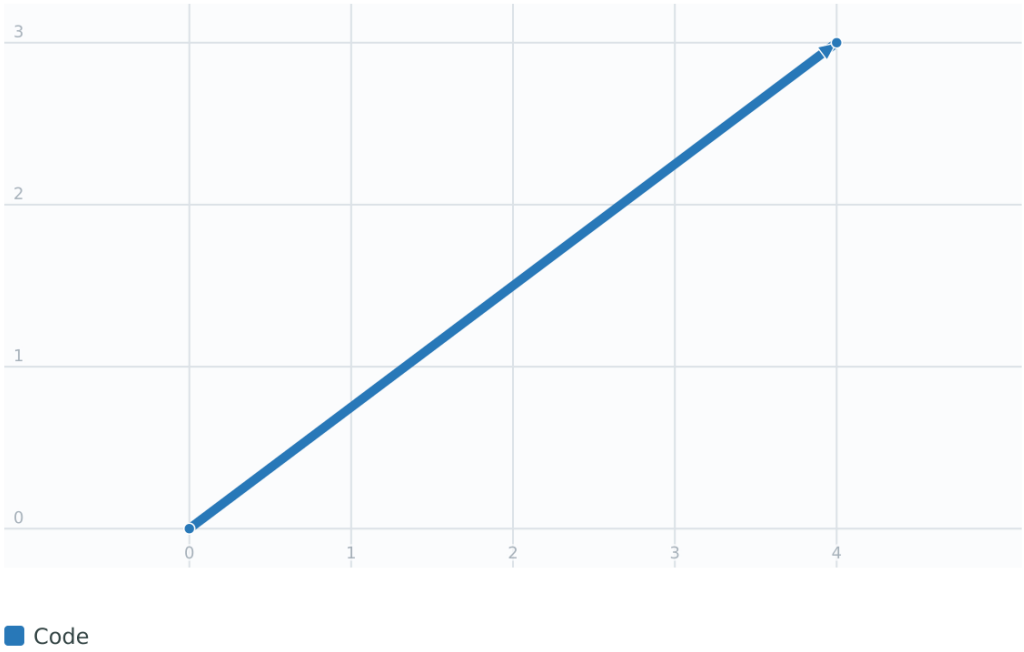
Code

```
SELECT ST_XMax('LINESTRING(1 3 4,5 6 7)'::geometry);
```

Output

```
5
```

Figure



Code

```
SELECT ST_XMax(CAST('BOX(-3 2,3 4)' AS box2d));
```

Output

```
3
```

Observe THIS DOES NOT WORK because it will try to auto-cast the string representation to a BOX3D.

Code

```
SELECT ST_XMax('LINESTRING(1 3,5 6)');
```

Output

```
ERROR: BOX3D parser - doesn't start with BOX3D(
```

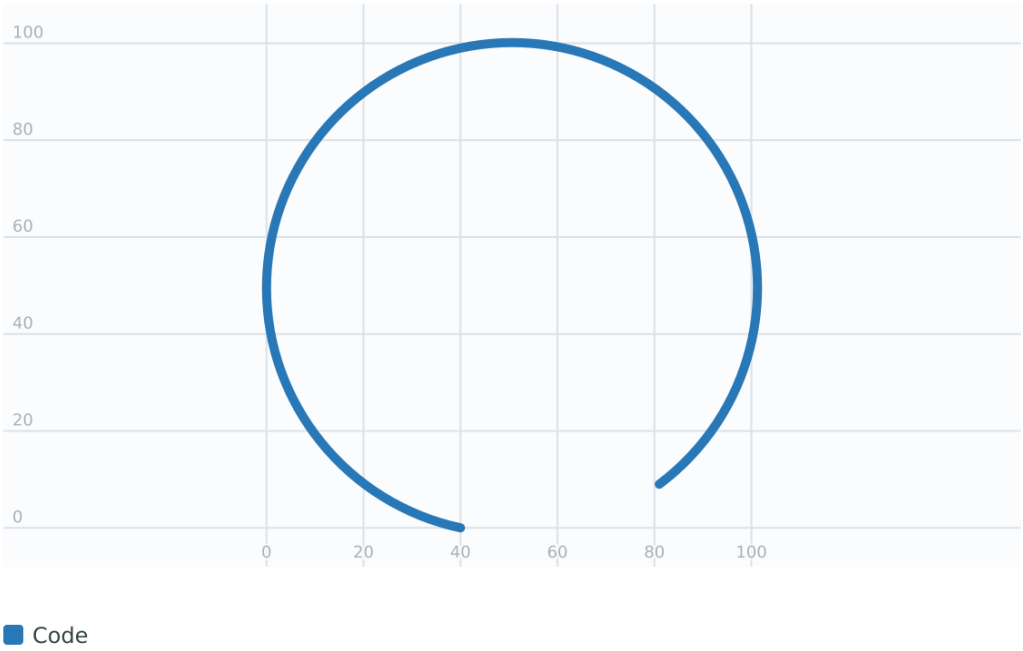
Code

```
SELECT ST_XMax('CIRCULARSTRING(220268 150415 1,220227 150505 2,220227 150406 3)'::geometry) ↩
    ;
```

Output

```
220288.248780547
```

Figure



☒☒

ST_XMin, ST_YMin, ST_YMax, ST_ZMin, ST_ZMax, ST_MMin, ST_MMax

7.18.10 ST_XMin

ST_XMin — Returns the X minima of a 2D or 3D bounding box or a geometry.

☒☒

float **ST_XMin**(box3d aGeomorBox2DorBox3D);

☒☒

Returns the X minima of a 2D or 3D bounding box or a geometry.



□□
Although this function is only defined for box3d, it also works for box2d and geometry values due to automatic casting. However it will not accept a geometry or box2d text representation, since those do not auto-cast.

 This function supports 3d and will not drop the z-index.

 This method supports Circular Strings and Curves.

□□

Code

```
SELECT ST_XMin('BOX3D(1 2 3,4 5 6)');
```

Output

```
1
```

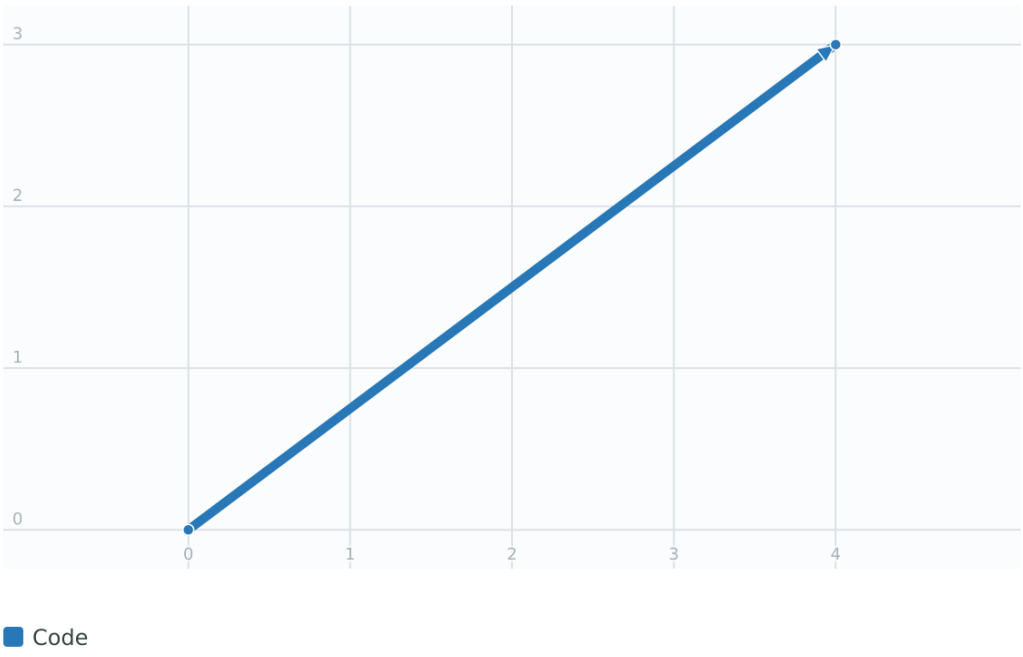
Code

```
SELECT ST_XMin('LINESTRING(1 3 4,5 6 7)'::geometry);
```

Output

```
1
```

Figure



Code

```
SELECT ST_XMin(CAST('BOX(-3 2,3 4)' AS box2d));
```

Output

```
-3
```

Observe THIS DOES NOT WORK because it will try to auto-cast the string representation to a BOX3D.

Code

```
SELECT ST_XMin('LINESTRING(1 3,5 6)');
```

Output

```
ERROR: BOX3D parser - doesn't start with BOX3D(
```

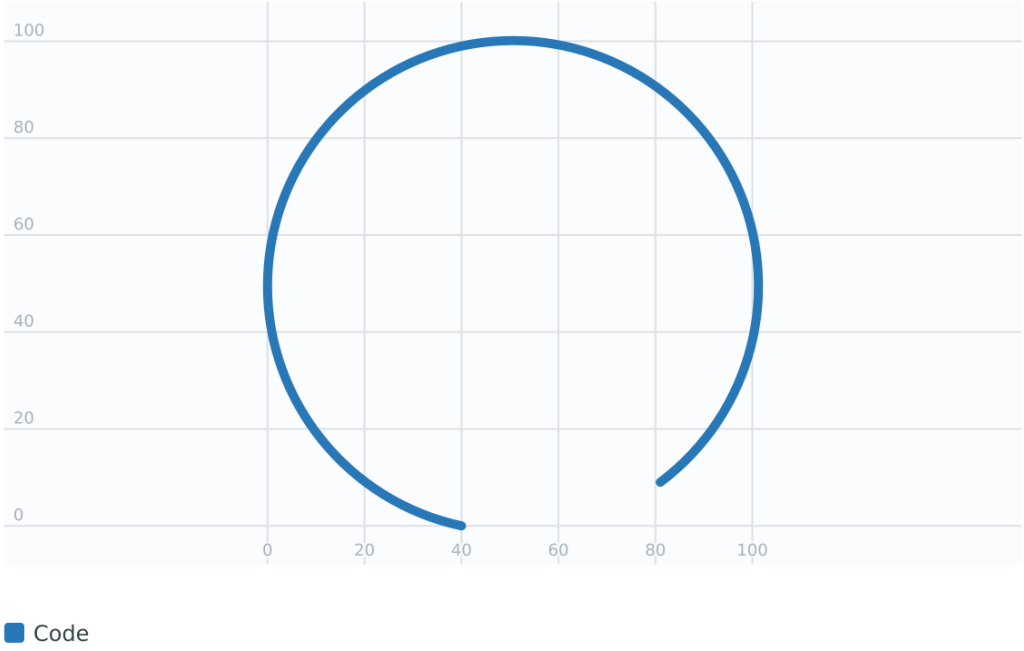
Code

```
SELECT ST_XMin('CIRCULARSTRING(220268 150415 1,220227 150505 2,220227 150406 3)'::geometry)
    ;
```

Output

```
220186.995121892
```

Figure



☒☒

ST_XMax, ST_YMin, ST_YMax, ST_ZMin, ST_ZMax, ST_MMin, ST_MMax

7.18.11 ST_YMax

ST_YMax — Returns the Y maxima of a 2D or 3D bounding box or a geometry.

□□

float **ST_YMax**(box3d aGeomorBox2DorBox3D);

□□

Returns the Y maxima of a 2D or 3D bounding box or a geometry.



□□
Although this function is only defined for box3d, it also works for box2d and geometry values due to automatic casting. However it will not accept a geometry or box2d text representation, since those do not auto-cast.

 This function supports 3d and will not drop the z-index.

 This method supports Circular Strings and Curves.

□□

Code

```
SELECT ST_YMax('BOX3D(1 2 3,4 5 6)');
```

Output

```
5
```

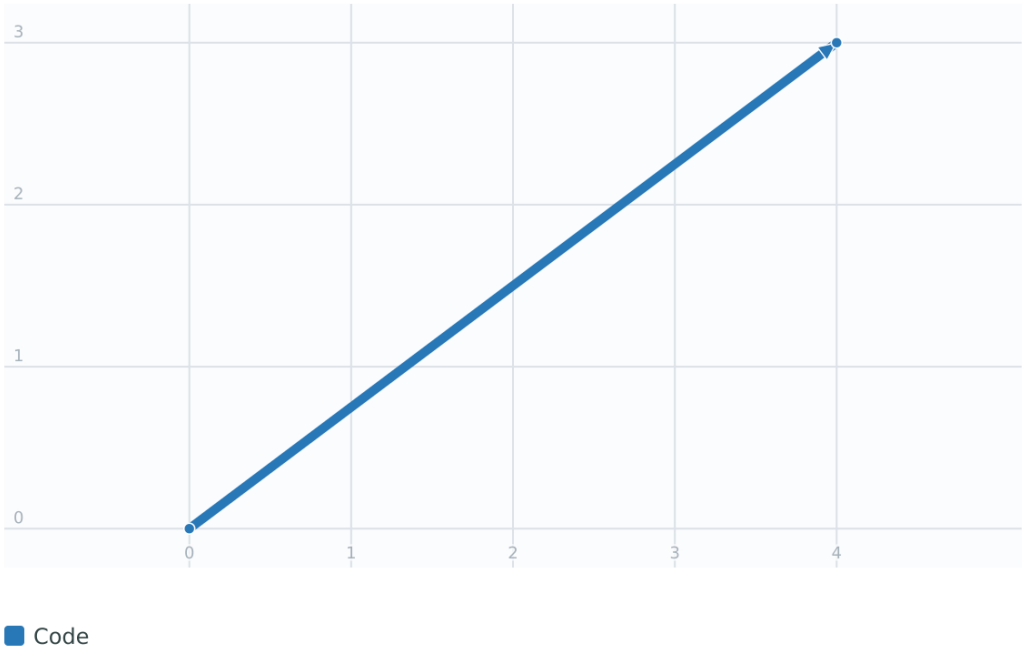
Code

```
SELECT ST_YMax('LINESTRING(1 3 4,5 6 7)'::geometry);
```

Output

```
6
```

Figure



Code

```
SELECT ST_YMax(CAST('BOX(-3 2,3 4)' AS box2d));
```

Output

```
4
```

Observe THIS DOES NOT WORK because it will try to auto-cast the string representation to a BOX3D.

Code

```
SELECT ST_YMax('LINESTRING(1 3,5 6)');
```

Output

```
ERROR: BOX3D parser - doesn't start with BOX3D(
```

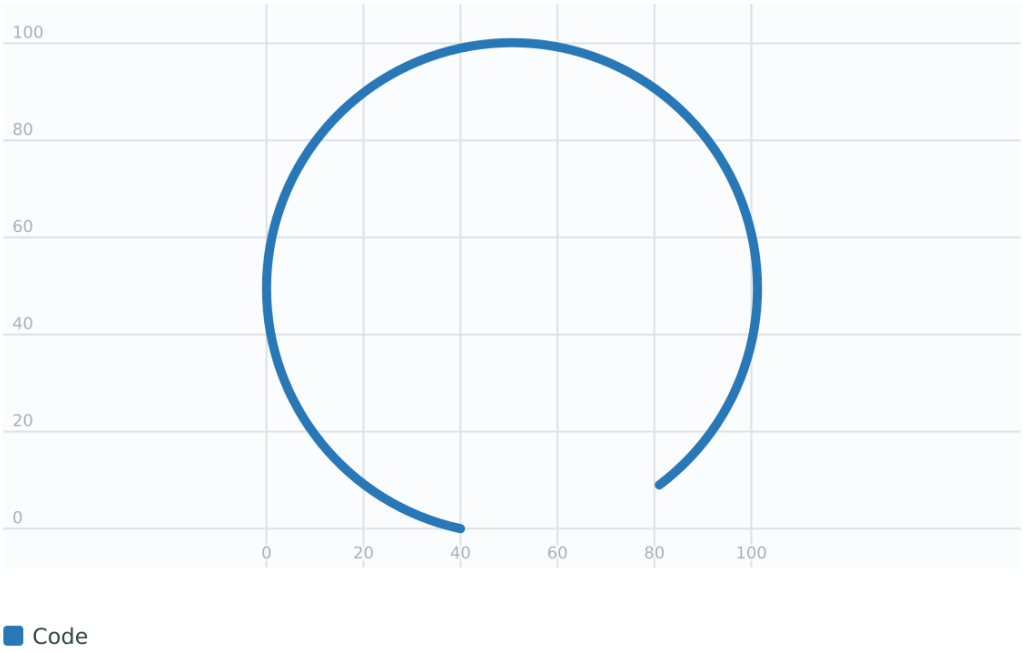
Code

```
SELECT ST_YMax('CIRCULARSTRING(220268 150415 1,220227 150505 2,220227 150406 3)'::geometry) ↩
    ;
```

Output

```
150506.126829327
```

Figure



□□

ST_XMin, ST_XMax, ST_YMin, ST_ZMin, ST_ZMax, ST_MMin, ST_MMax

7.18.12 ST_YMin

ST_YMin — Returns the Y minima of a 2D or 3D bounding box or a geometry.

□□

float **ST_YMin**(box3d aGeomorBox2DorBox3D);

□□

Returns the Y minima of a 2D or 3D bounding box or a geometry.



□□
Although this function is only defined for box3d, it also works for box2d and geometry values due to automatic casting. However it will not accept a geometry or box2d text representation, since those do not auto-cast.

 This function supports 3d and will not drop the z-index.

 This method supports Circular Strings and Curves.

□□

Code

```
SELECT ST_YMin('BOX3D(1 2 3,4 5 6)');
```

Output

```
2
```

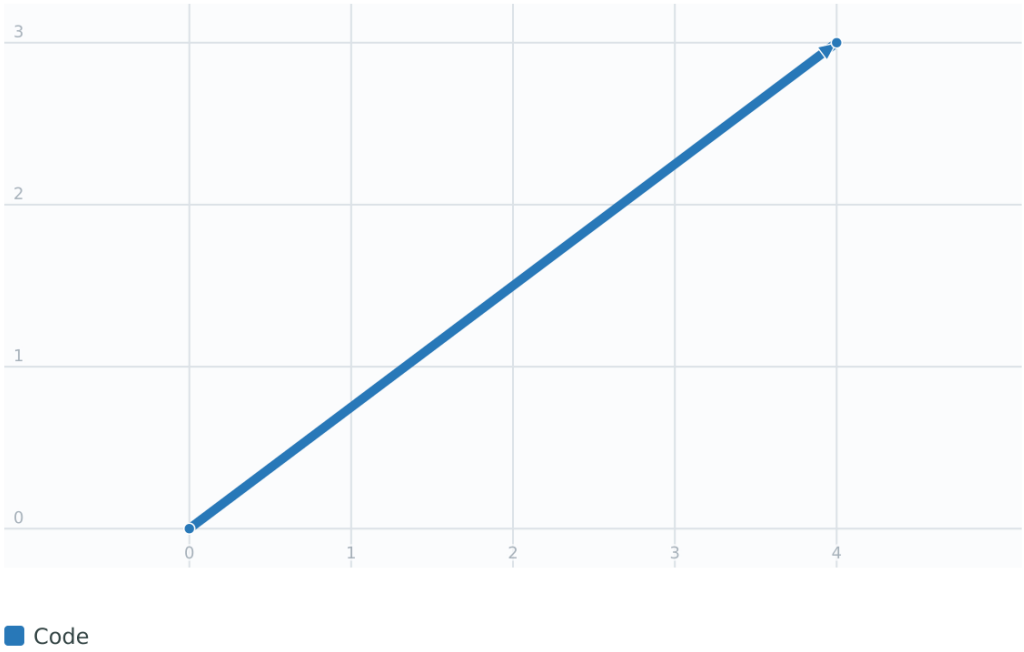
Code

```
SELECT ST_YMin('LINESTRING(1 3 4,5 6 7)'::geometry);
```

Output

```
3
```

Figure



Code

```
SELECT ST_YMin(CAST('BOX(-3 2,3 4)' AS box2d));
```

Output

```
2
```

Observe THIS DOES NOT WORK because it will try to auto-cast the string representation to a BOX3D.

Code

```
SELECT ST_YMin('LINESTRING(1 3,5 6)');
```

Output

```
ERROR: BOX3D parser - doesn't start with BOX3D(
```

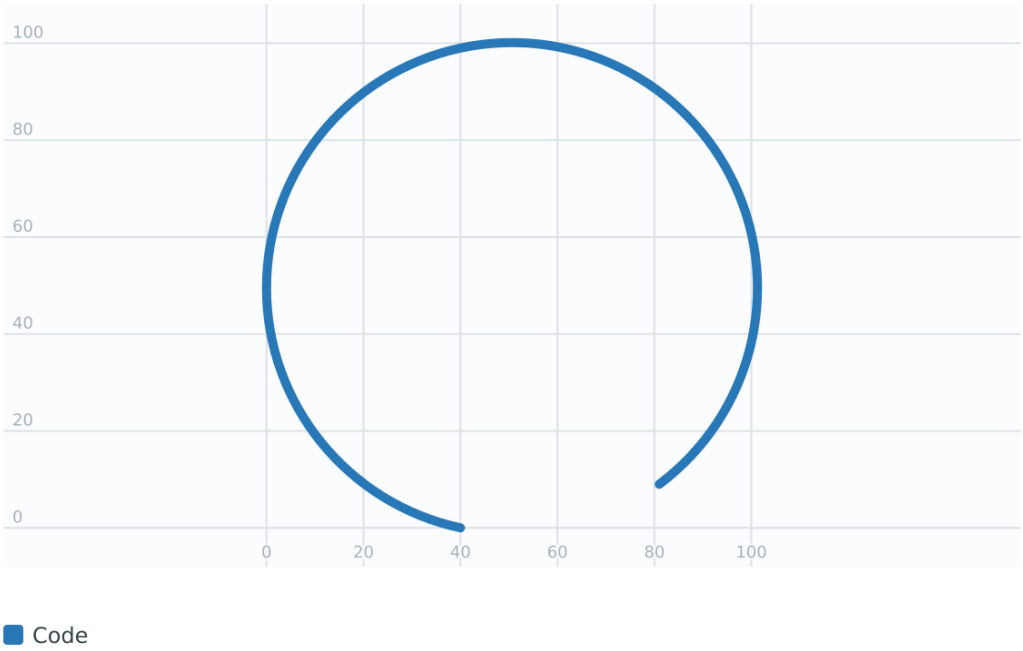
Code

```
SELECT ST_YMin('CIRCULARSTRING(220268 150415 1,220227 150505 2,220227 150406 3)'::geometry) ↩
    ;
```

Output

```
150406
```

Figure



☒☒

ST_GeomFromEWKT, ST_XMin, ST_XMax, ST_YMax, ST_ZMin, ST_ZMax, ST_MMin, ST_MMax

7.18.13 ST_ZMax

ST_ZMax — Returns the Z maxima of a 2D or 3D bounding box or a geometry.

☒☒

float **ST_ZMax**(box3d aGeomorBox2DorBox3D);

☒☒

Returns the Z maxima of a 2D or 3D bounding box or a geometry.



□□
Although this function is only defined for box3d, it also works for box2d and geometry values due to automatic casting. However it will not accept a geometry or box2d text representation, since those do not auto-cast.

 This function supports 3d and will not drop the z-index.

 This method supports Circular Strings and Curves.

□□

Code

```
SELECT ST_ZMax('BOX3D(1 2 3,4 5 6)');
```

Output

```
6
```

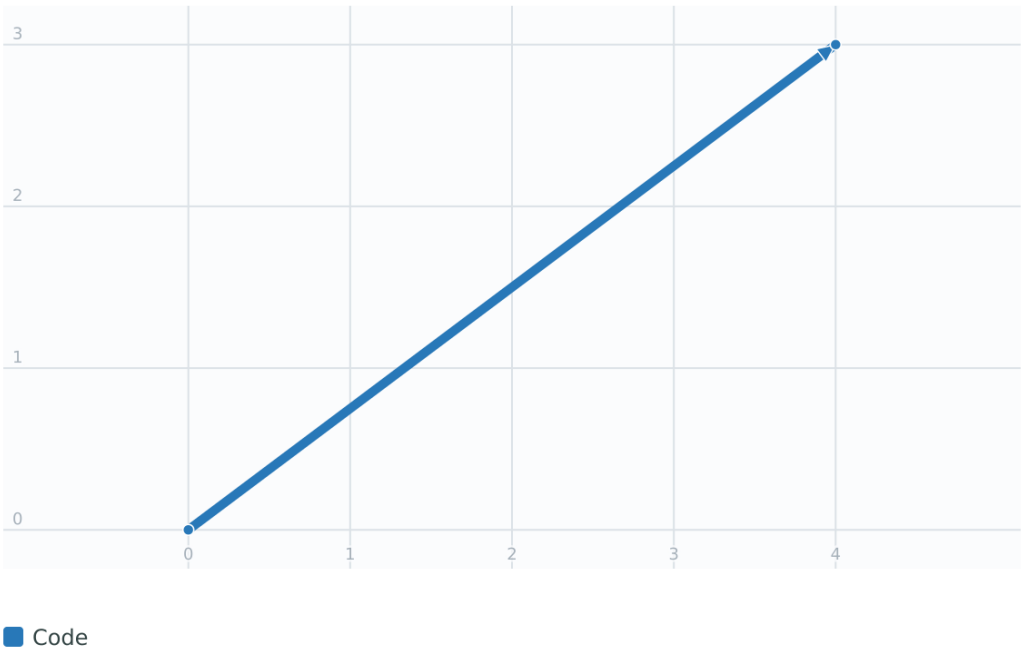
Code

```
SELECT ST_ZMax('LINESTRING(1 3 4,5 6 7)'::geometry);
```

Output

```
7
```

Figure



Code

```
SELECT ST_ZMax('BOX3D(-3 2 1,3 4 1)');
```

Output

```
1
```

Observe THIS DOES NOT WORK because it will try to auto-cast the string representation to a BOX3D.

Code

```
SELECT ST_ZMax('LINESTRING(1 3 4,5 6 7)');
```

Output

```
ERROR: BOX3D parser - doesn't start with BOX3D(
```

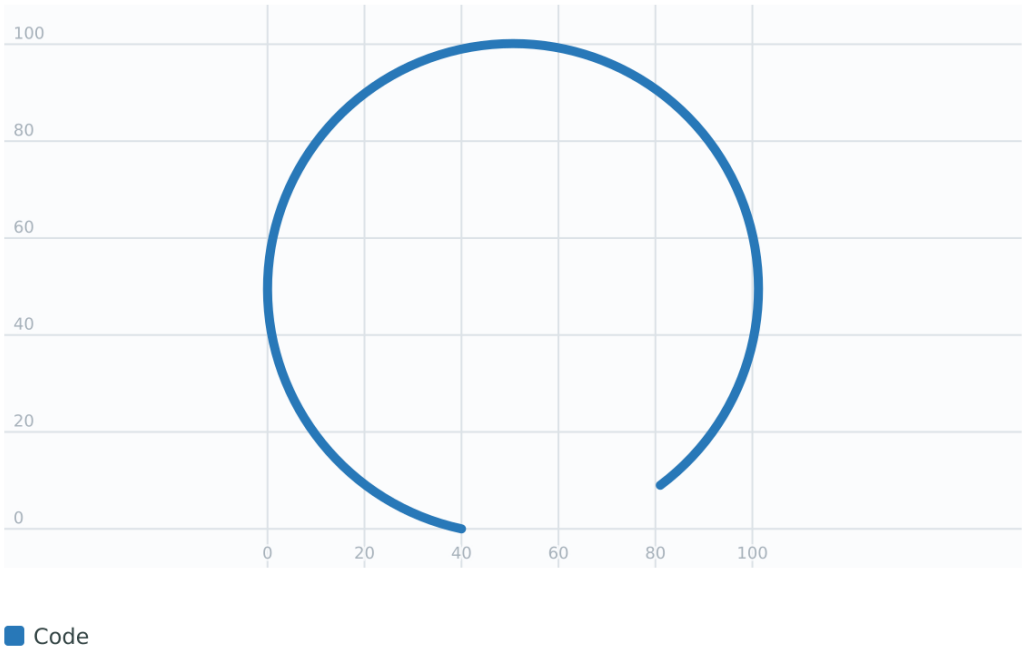
Code

```
SELECT ST_ZMax('CIRCULARSTRING(220268 150415 1,220227 150505 2,220227 150406 3)'::geometry) ↩
    ;
```

Output

```
3
```

Figure



☒☒

ST_GeomFromEWKT, ST_XMin, ST_XMax, ST_YMin, ST_YMax, ST_ZMin, ST_MMin, ST_MMax

7.18.14 ST_ZMin

ST_ZMin — Returns the Z minima of a 2D or 3D bounding box or a geometry.

□□

float **ST_ZMin**(box3d aGeomorBox2DorBox3D);

□□

Returns the Z minima of a 2D or 3D bounding box or a geometry.



□□
Although this function is only defined for box3d, it also works for box2d and geometry values due to automatic casting. However it will not accept a geometry or box2d text representation, since those do not auto-cast.

This function supports 3d and will not drop the z-index.

This method supports Circular Strings and Curves.

□□

Code

```
SELECT ST_ZMin('BOX3D(1 2 3,4 5 6)');
```

Output

```
3
```

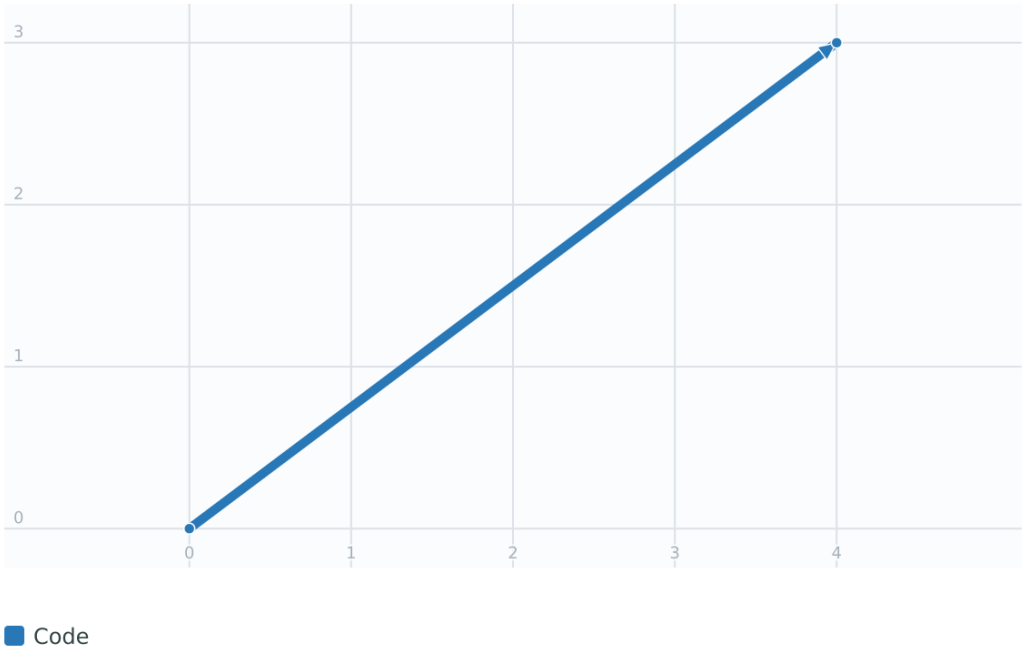
Code

```
SELECT ST_ZMin('LINESTRING(1 3 4,5 6 7)'::geometry);
```

Output

```
4
```

Figure



Code

```
SELECT ST_ZMin('BOX3D(-3 2 1,3 4 1)');
```

Output

```
1
```

Observe THIS DOES NOT WORK because it will try to auto-cast the string representation to a BOX3D.

Code

```
SELECT ST_ZMin('LINESTRING(1 3 4,5 6 7)');
```

Output

```
ERROR: BOX3D parser - doesn't start with BOX3D(
```

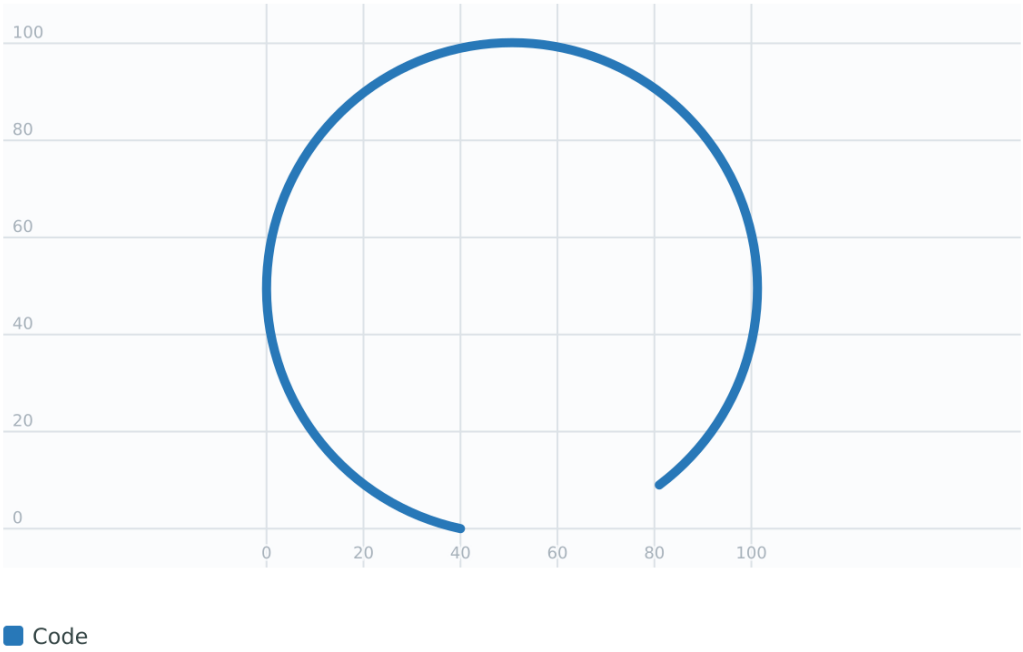
Code

```
SELECT ST_ZMin('CIRCULARSTRING(220268 150415 1,220227 150505 2,220227 150406 3)'::geometry) ↩
    ;
```

Output

```
1
```

Figure



□□

ST_GeomFromEWKT, ST_GeomFromText, ST_XMin, ST_XMax, ST_YMin, ST_YMax, ST_ZMax, ST_MMin, ST_MMax

7.18.15 ST_XSize

ST_XSize — Returns the X size of a 2D or 3D bounding box or a geometry.

□□

float **ST_XSize**(box3d aGeomorBox2DorBox3D);

□□

Returns the X size of a 2D or 3D bounding box or a geometry, computed as `ST_XMax(aGeomorBox2DorBox3D) - ST_XMin(aGeomorBox2DorBox3D)`.



□□
Although this function is only defined for box3d, it also works for box2d and geometry values due to automatic casting. However it will not accept a geometry or box2d text representation, since those do not auto-cast.

 This function supports 3d and will not drop the z-index.

 This method supports Circular Strings and Curves.

Availability: 3.7.0

□□

Code

```
SELECT ST_XSize('BOX3D(1 2 3,4 5 6)');
```

Output

```
3
```

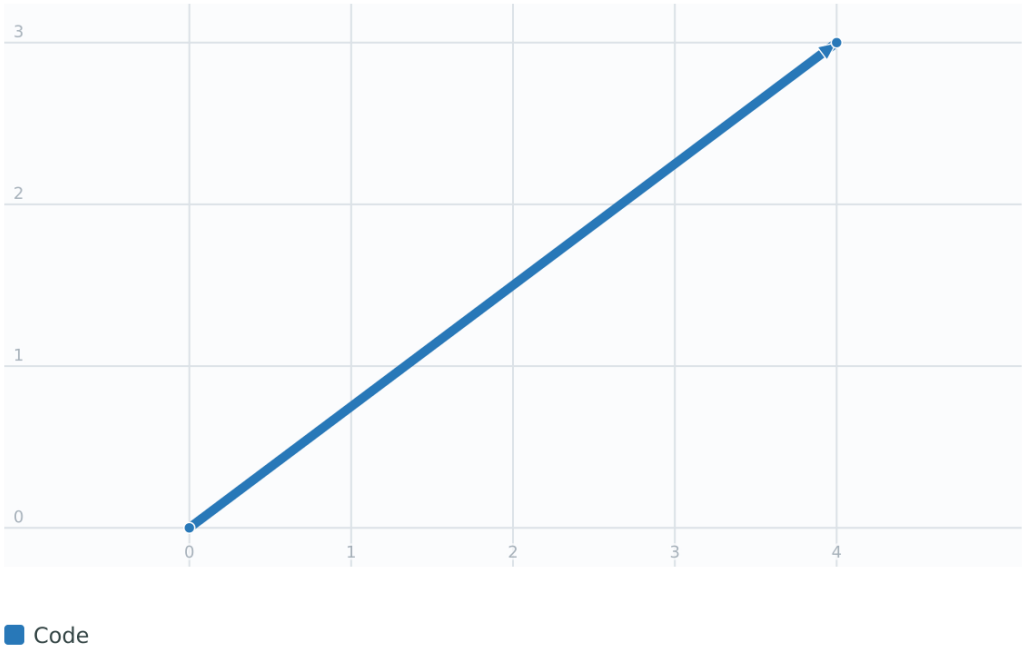
Code

```
SELECT ST_XSize('LINESTRING(1 3,5 6)'::geometry);
```

Output

```
4
```

Figure



□□

ST_XMin, ST_XMax, ST_YSize, ST_ZSize, ST_MSize

7.18.16 ST_YSize

ST_YSize — Returns the Y size of a 2D or 3D bounding box or a geometry.

□□

float **ST_YSize**(box3d aGeomorBox2DorBox3D);

□□

Returns the Y size of a 2D or 3D bounding box or a geometry, computed as `ST_YMax(aGeomorBox2DorBox3D) - ST_YMin(aGeomorBox2DorBox3D)`.



□□
Although this function is only defined for box3d, it also works for box2d and geometry values due to automatic casting. However it will not accept a geometry or box2d text representation, since those do not auto-cast.

This function supports 3d and will not drop the z-index.

This method supports Circular Strings and Curves.

Availability: 3.7.0

□□

Code

```
SELECT ST_YSize('BOX3D(1 2 3,4 5 6)');
```

Output

```
3
```

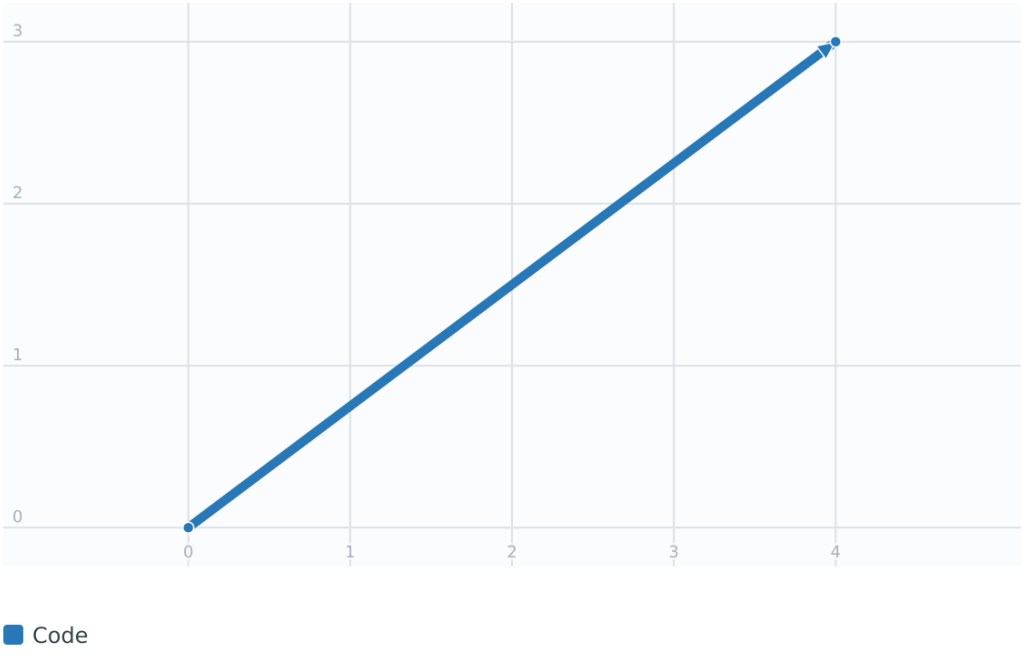
Code

```
SELECT ST_YSize('LINESTRING(1 3,5 6)'::geometry);
```

Output

```
3
```

Figure



□□

ST_YMin, ST_YMax, ST_XSize, ST_ZSize, ST_MSize

7.18.17 ST_ZSize

ST_ZSize — Returns the Z size of a 2D or 3D bounding box or a geometry.

□□

float **ST_ZSize**(box3d aGeomorBox2DorBox3D);

□□

Returns the Z size of a 2D or 3D bounding box or a geometry, computed as `ST_ZMax(aGeomorBox2DorBox3D) - ST_ZMin(aGeomorBox2DorBox3D)`.



□□
Although this function is only defined for box3d, it also works for box2d and geometry values due to automatic casting. However it will not accept a geometry or box2d text representation, since those do not auto-cast.

This function supports 3d and will not drop the z-index.

This method supports Circular Strings and Curves.

Availability: 3.7.0

□□

Code

```
SELECT ST_ZSize('BOX3D(1 2 3,4 5 6)');
```

Output

```
3
```

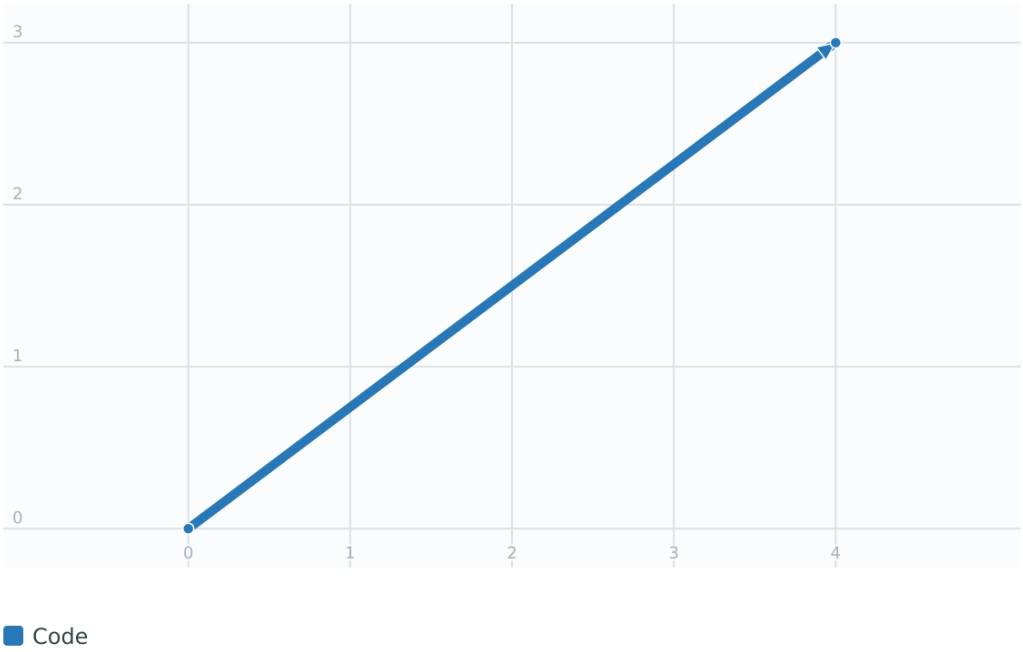
Code

```
SELECT ST_ZSize('LINESTRING(1 3 4,5 6 7)'::geometry);
```

Output

```
3
```

Figure



□□

ST_ZMin, ST_ZMax, ST_XSize, ST_YSize, ST_MSize

7.18.18 ST_MMin

ST_MMin — Returns the M minima of a geometry.

□□

float **ST_MMin**(geometry geom);

□□

Returns the M minima of a geometry, or null if the geometry lacks M values.

This function supports 3d and will not drop the z-index.

This method supports Circular Strings and Curves.

□□

Code

```
SELECT ST_MMin('POINT M (1 2 3)');
```

Output

```
3
```

□□

ST_XMin, ST_XMax, ST_YMin, ST_YMax, ST_ZMin, ST_ZMax, ST_MMax, ST_MSize

7.18.19 ST_MMax

ST_MMax — Returns the M maxima of a geometry.

□□

float **ST_MMax**(geometry geom);

□□

Returns the M maxima of a geometry, or null if the geometry lacks M values.

This function supports 3d and will not drop the z-index.

This method supports Circular Strings and Curves.

□□

Code

```
SELECT ST_MMax('POINT M (1 2 3)');
```

Output

```
3
```

□□

ST_XMin, ST_XMax, ST_YMin, ST_YMax, ST_ZMin, ST_ZMax, ST_MMin, ST_MSize

7.18.20 ST_MSize

ST_MSize — Returns the M size of a geometry.

□□

float **ST_MSize**(geometry geom);

□□

Returns the M size of a geometry, computed as `ST_MMax(geom) - ST_MMin(geom)`, or null if the geometry lacks M values.

This function supports 3d and will not drop the z-index.

This method supports Circular Strings and Curves.

Availability: 3.7.0

□□

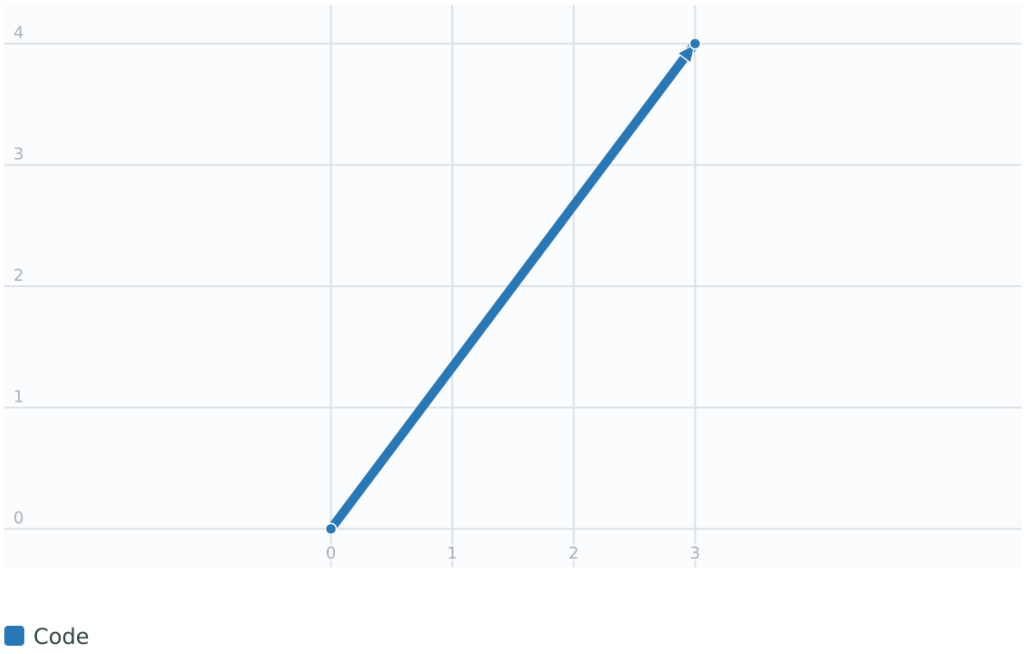
Code

```
SELECT ST_MSize('LINESTRING M (1 2 3,4 6 8)');
```

Output

```
5
```

Figure



□□

ST_MMin, ST_MMax, ST_XSize, ST_YSize, ST_ZSize

7.19 □□□□ (Linear Referencing)

7.19.1 ST_LineInterpolatePoint

ST_LineInterpolatePoint — Returns a point interpolated along a line at a fractional location.

⊠⊠

geometry **ST_LineInterpolatePoint**(geometry a_linestring, float8 a_fraction);
geography **ST_LineInterpolatePoint**(geography a_linestring, float8 a_fraction, boolean use_spheroid = true);

⊠⊠

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠ 0 ⊠ 1 ⊠⊠⊠ Float8 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ST_LineLocatePoint ⊠⊠⊠⊠⊠⊠⊠.

> **⊠⊠**
> This function computes points in 2D and then interpolates values for Z and M, while ST_3DLineInterpolatePoint computes points in 3D and only interpolates the M value.

> **⊠⊠**
> 1.1.1 ⊠⊠⊠⊠⊠⊠⊠⊠⊠ M ⊠ Z ⊠ (⊠⊠⊠⊠⊠) ⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ 0.0 ⊠⊠⊠⊠ ⊠⊠⊠⊠⊠.

0.8.2 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. 1.1.1 ⊠⊠⊠⊠ Z ⊠ M ⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠⊠⊠: 2.1.0 ⊠⊠⊠⊠, ⊠ 2.0.x ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ST_Line_Interpolate_Point ⊠⊠⊠⊠.

This function supports 3d and will not drop the z-index.

⊠⊠

This example returns the point 20 percent along a line.

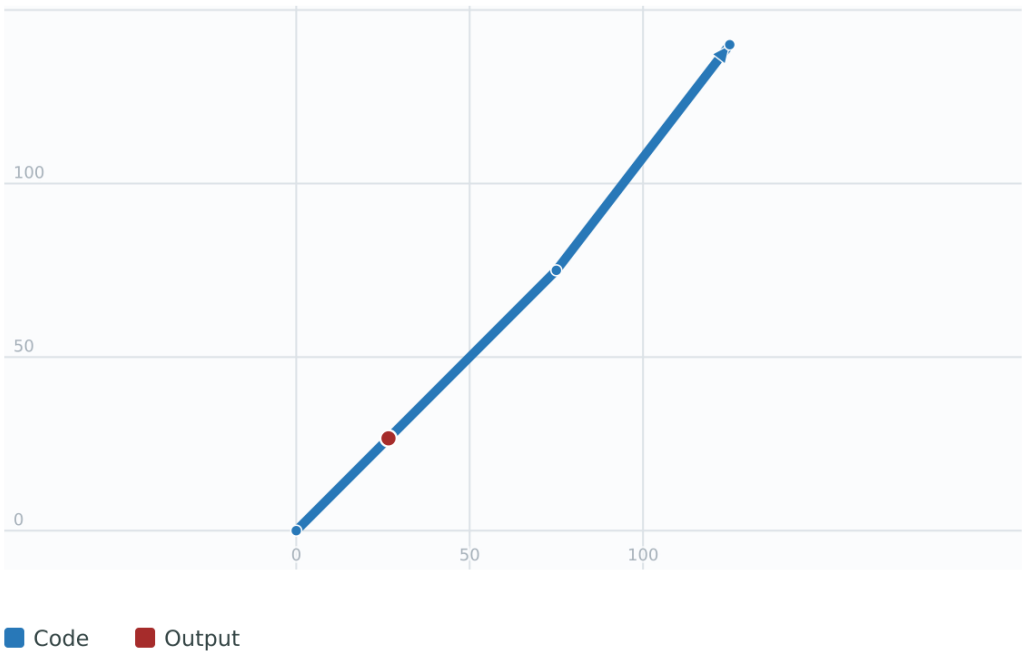
Code

```
SELECT ST_LineInterpolatePoint(
'LINESTRING(25 50,100 125,150 190)',
0.2);
```

Output

```
POINT(51.5974135047432 76.5974135047432)
```

Figure



The mid-point of a 3D line:

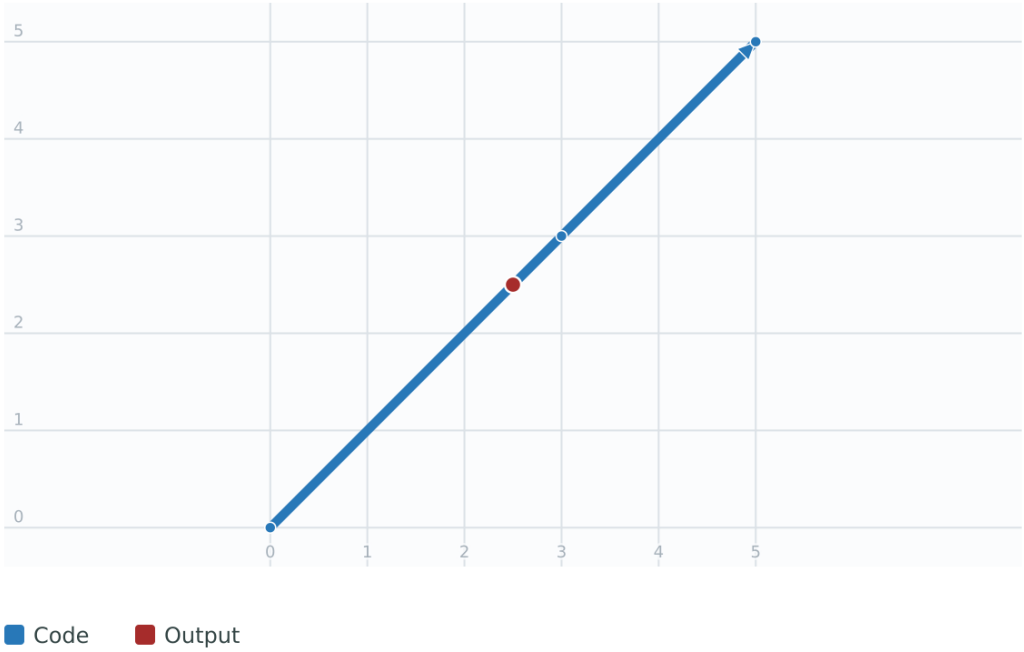
Code

```
SELECT ST_LineInterpolatePoint(
'LINESTRING(1 2 3,4 5 6,6 7 8)',
0.5);
```

Output

```
POINT(3.5 4.5 5.5)
```

Figure



The closest point on a line to a point:

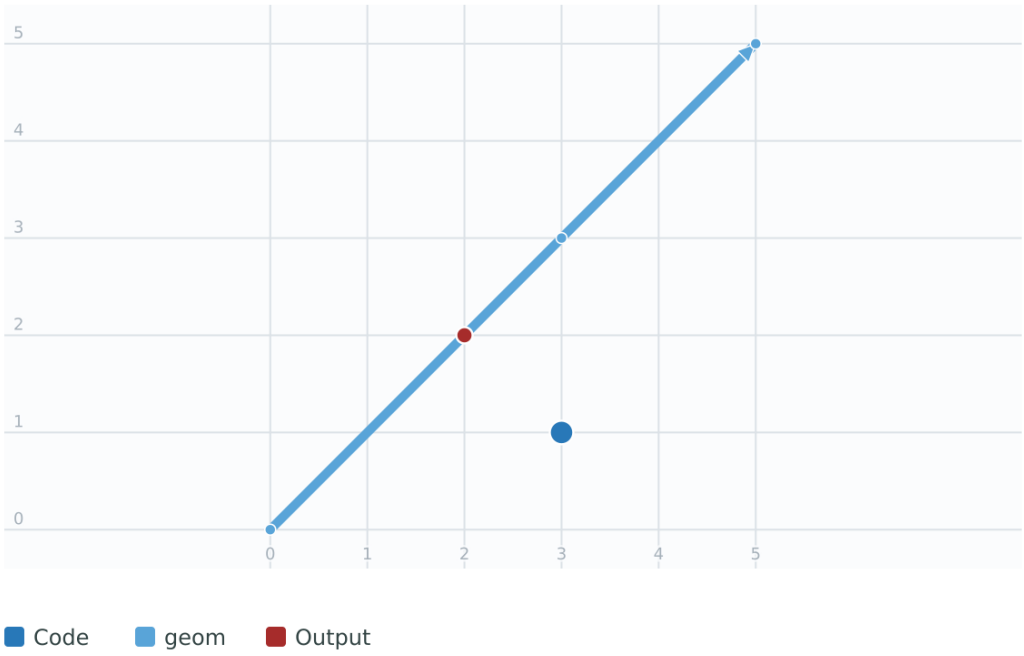
Code

```
SELECT ST_LineInterpolatePoint(
    line.geom,
    ST_LineLocatePoint(line.geom, 'POINT(4 3)'))
FROM (SELECT 'LINESTRING(1 2,4 5,6 7)'::geometry As geom) AS line;
```

Output

```
POINT(3 4)
```

Figure

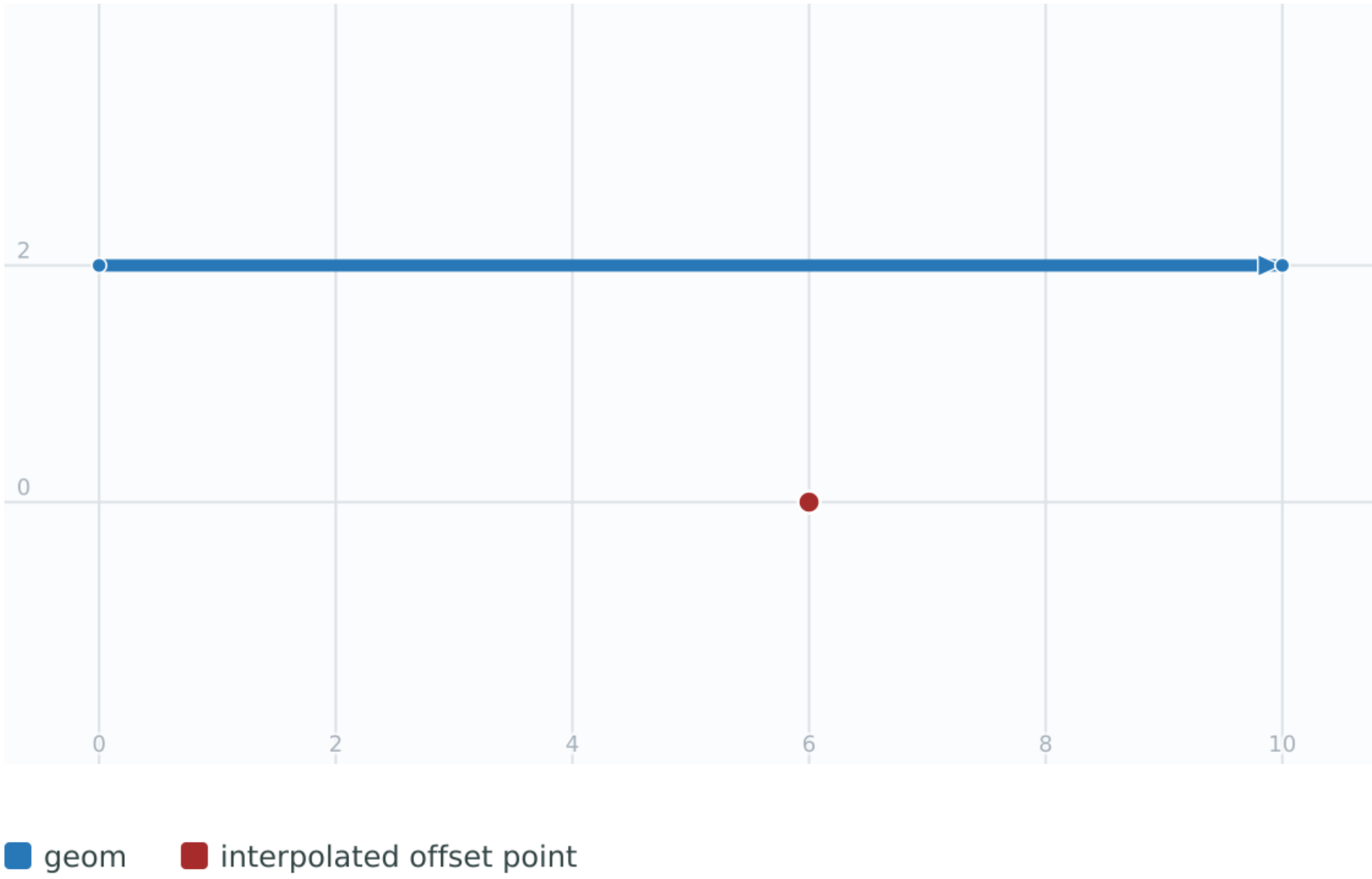


An interpolated point can be offset to the left or right of a line by using ST_OffsetCurve before interpolation. This is useful for locating an address point beside a street centerline.

Code

```
WITH street AS (
    SELECT 'LINESTRING(0 0, 10 0)'::geometry AS geom,
           100 AS from_num,
           200 AS to_num
)
SELECT
    ST_LineInterpolatePoint(
        ST_OffsetCurve(geom, -2),
        (160 - from_num)::float8 / (to_num - from_num)
    ) AS interpolated_offset_point
FROM street;
```

Figure



⊠⊠

ST_LineInterpolatePoints, ST_3DLineInterpolatePoint, ST_LineLocatePoint, ST_OffsetCurve

7.19.2 ST_3DLineInterpolatePoint

ST_3DLineInterpolatePoint — Returns a point interpolated along a 3D line at a fractional location.

⊠⊠

geometry **ST_3DLineInterpolatePoint**(geometry a_linestring, float8 a_fraction);

⊠⊠

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠ 0 ⊠ 1 ⊠⊠⊠ Float8 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

> **Note!** ⊠⊠
> ST_LineInterpolatePoint computes points in 2D and then interpolates the values for Z and M, while this function computes points in 3D and only interpolates the M value.

2.1.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

This function supports 3d and will not drop the z-index.

⊠⊠

Return point 20% along 3D line

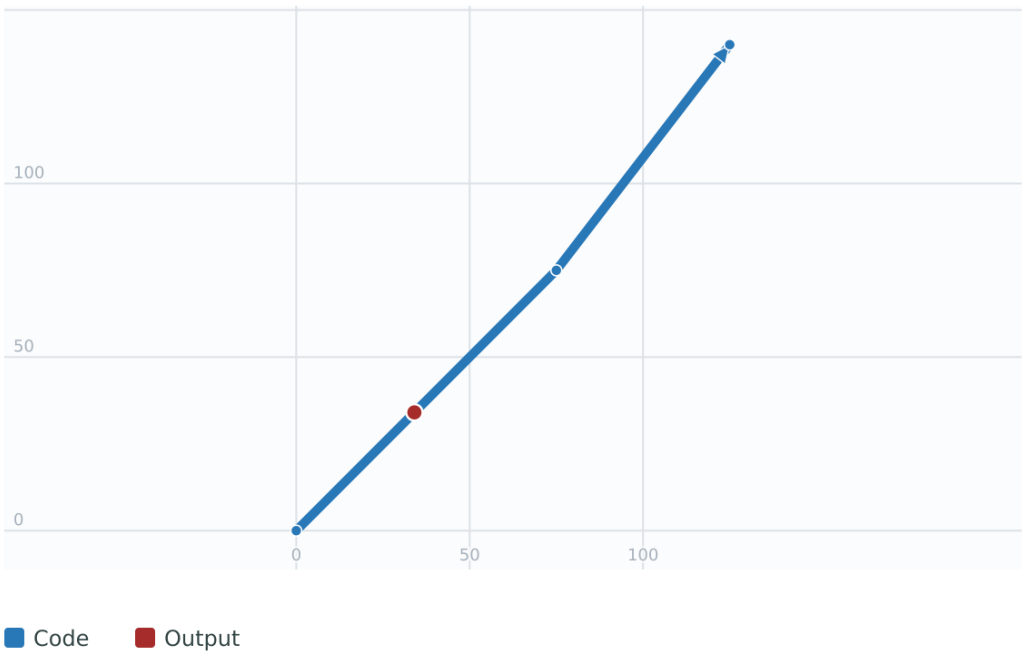
Code

```
SELECT ST_3DLineInterpolatePoint(
'LINESTRING(25 50 70,100 125 90,150 190 200)',
0.20);
```

Output

```
POINT Z (59.06758929108215 84.06758929108216 79.0846904776219)
```

Figure



□□

ST_LineInterpolatePoint, ST_LineInterpolatePoint, ST_LineMerge

7.19.3 ST_LineInterpolatePoints

ST_LineInterpolatePoints — Returns points interpolated along a line at a fractional interval.

□□

geometry **ST_LineInterpolatePoints**(geometry a_linestring, float8 a_fraction, boolean repeat);
geography **ST_LineInterpolatePoints**(geography a_linestring, float8 a_fraction, boolean use_spheroid = true, boolean repeat = true);

□□

Returns one or more points interpolated along a line at a fractional interval. The first argument must be a LINESTRING. The second argument is a float8 between 0 and 1 representing the spacing

between the points as a fraction of line length. If the third argument is false, at most one point will be constructed (which is equivalent to ST_LineInterpolatePoint.)

If the result has zero or one points, it is returned as a POINT. If it has two or more points, it is returned as a MULTIPOINT.

Availability: 2.5.0

This function supports 3d and will not drop the z-index.

This function supports M coordinates.

□□

This example returns points every 20 percent along a two-dimensional line.

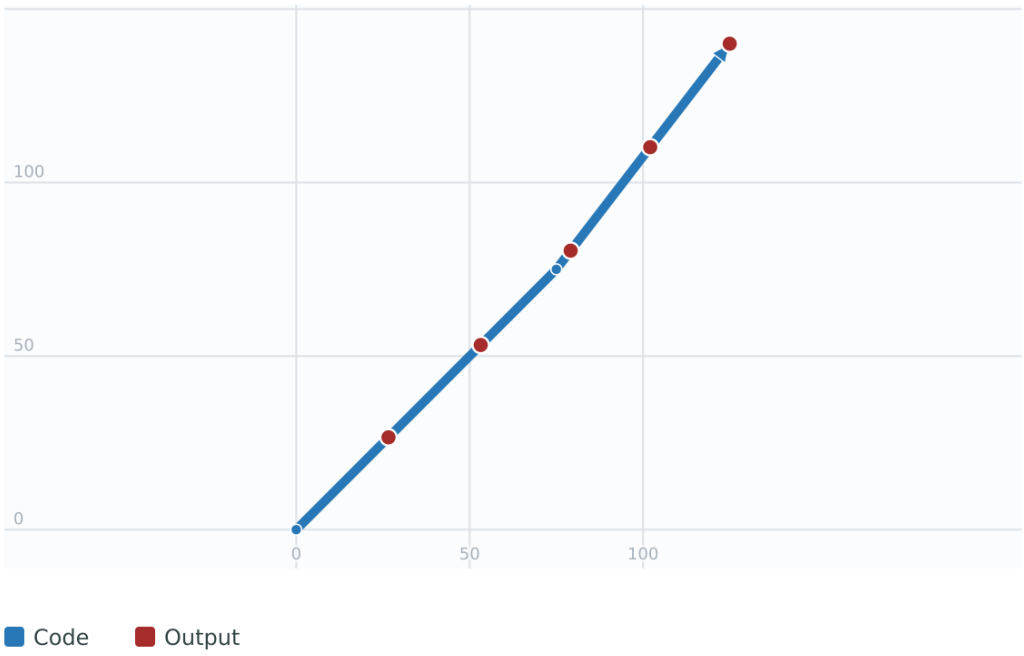
Code

```
SELECT ST_LineInterpolatePoints(
'LINESTRING(25 50,100 125,150 190)',
0.20);
```

Output

```
MULTIPOINT((51.5974135047432 76.5974135047432),(78.1948270094864 103.194827009486),(104.132163186446 130.37181214238) ↩
    ,(127.066081593223 160.18590607119),(150 190))
```

Figure



□□

ST_LineInterpolatePoint, ST_LineLocatePoint

7.19.4 ST_LineLocatePoint

ST_LineLocatePoint — Returns the fractional location of the closest point on a line to a point.

⊠⊠

float8 **ST_LineLocatePoint**(geometry a_linestring, geometry a_point);
float8 **ST_LineLocatePoint**(geography a_linestring, geography a_point, boolean use_spheroid = true);

⊠⊠

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ 2 ⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠ 0 ⊠⊠ 1 ⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (float) ⊠⊠⊠⊠⊠⊠⊠.

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (ST_LineInterpolatePoint) ⊠⊠⊠⊠⊠⊠⊠ (ST_LineSubstring) ⊠⊠⊠⊠⊠⊠⊠ ⊠⊠.

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

1.1.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠⊠⊠: 2.1.0 ⊠⊠⊠⊠, ⊠ 2.0.x ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ST_Line_Locate_Point ⊠⊠⊠⊠.

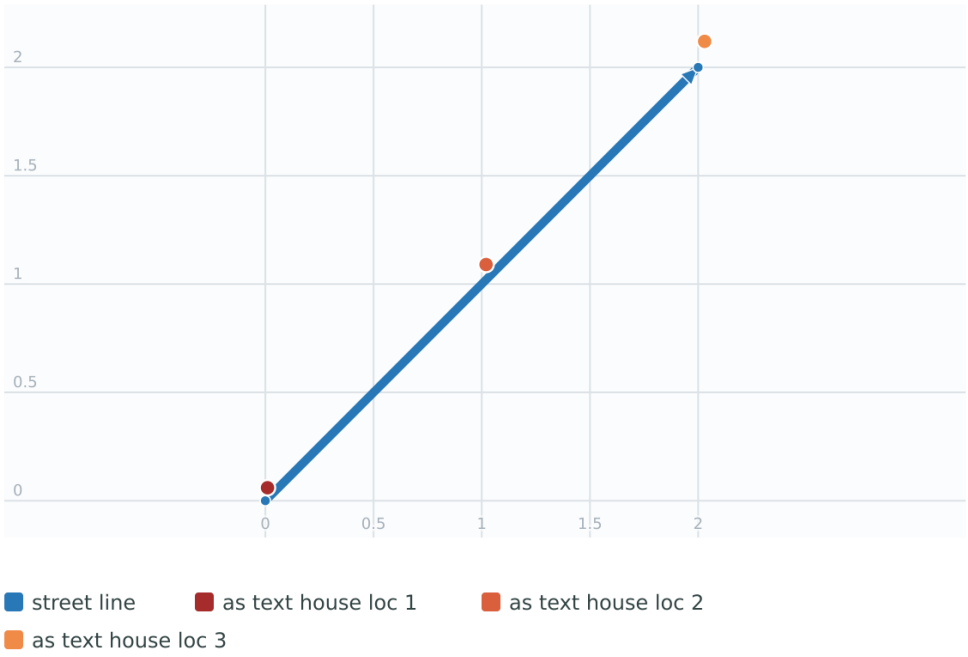
⊠⊠

This example approximates the street number of a house by locating the closest point on a street line. The subquery generates sample house and street data, and `ST_DWithin` excludes houses that are too far from the street.

Code

```
SELECT house_loc As as_text_house_loc,
    start_house_num +
        CAST( (end_house_num - start_house_num)
            * ST_LineLocatePoint(street_line, house_loc) As integer) As house_num
FROM
(SELECT 'LINESTRING(1 2,3 4)'::geometry As street_line,
    ST_Point(x*1.01, y*1.03) As house_loc, 10 As start_house_num,
        20 As end_house_num
FROM generate_series(1, 3) x CROSS JOIN generate_series(2, 4) As y)
As foo
WHERE ST_DWithin(street_line, house_loc, 0.2);
```

Figure



This example finds the closest point on a line to a point or other geometry.

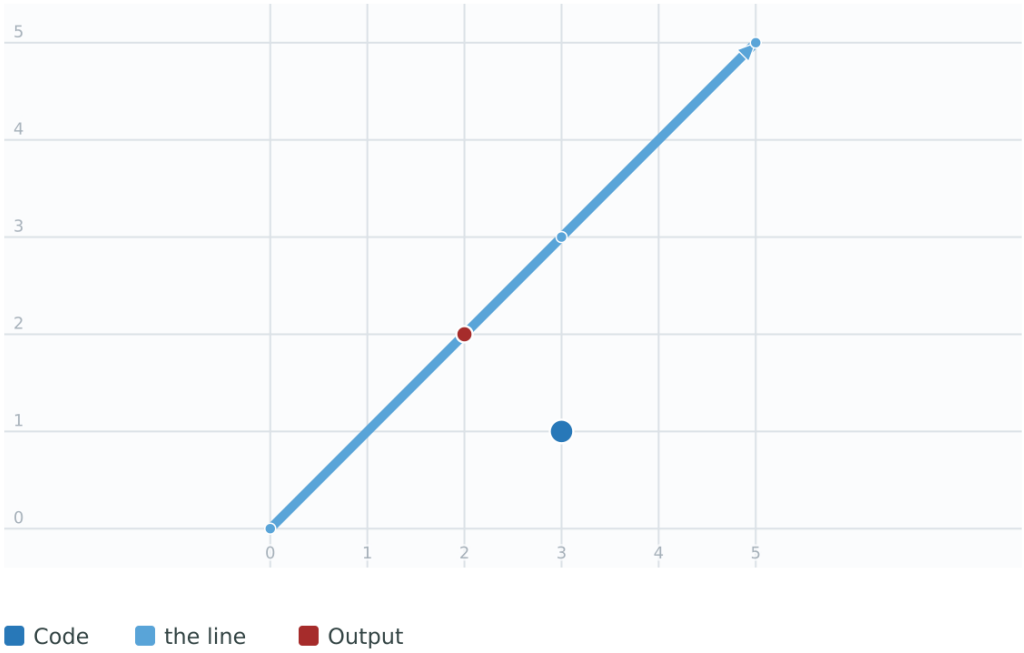
Code

```
SELECT ST_LineInterpolatePoint(
    foo.the_line,
    ST_LineLocatePoint(foo.the_line, 'POINT(4 3)'::geometry))
FROM (SELECT 'LINESTRING(1 2,4 5,6 7)'::geometry As the_line) As foo;
```

Output

```
POINT(3 4)
```

Figure

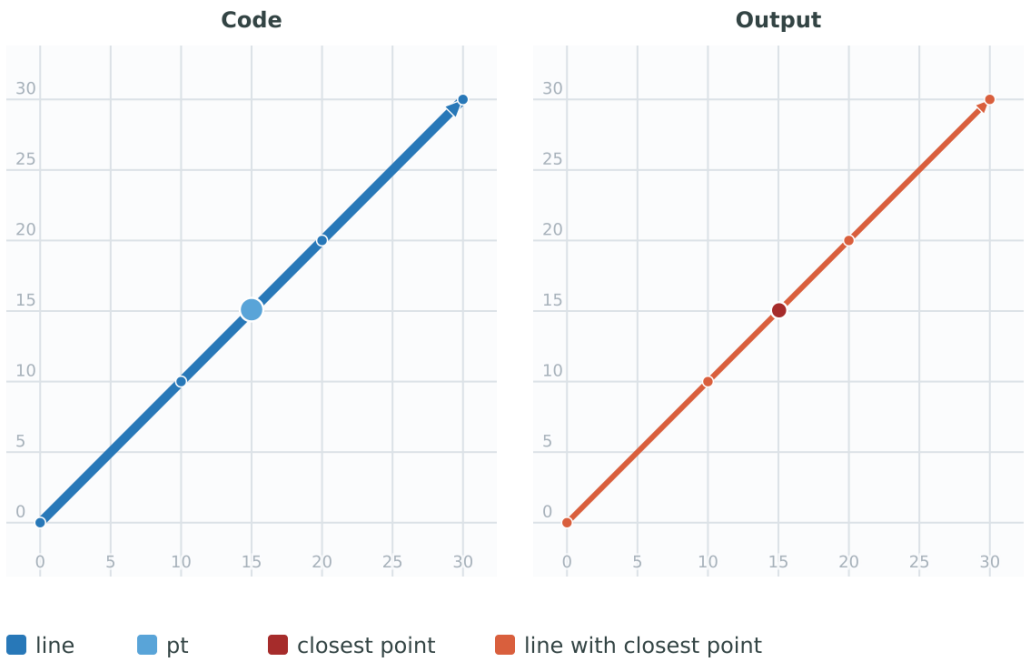


This example locates the closest segment and point on a LineString.

Code

```
WITH data AS (
  SELECT
    'LINESTRING (0 0, 10 10, 20 20, 30 30)'::geometry AS line,
    'POINT(15 15.1)'::geometry AS pt
),
segments AS (
  SELECT
    s.i,
    d.line,
    d.pt,
    ST_MakeLine(ST_PointN(d.line, s.i), ST_PointN(d.line, s.i + 1)) AS segment
  FROM data AS d
  CROSS JOIN LATERAL generate_series(1, ST_NumPoints(d.line) - 1) AS s(i)
)
SELECT
  i AS segment_number,
  round(ST_Distance(segment, pt)::numeric, 3) AS distance,
  ST_ClosestPoint(segment, pt) AS closest_point,
  ST_AddPoint(line, ST_ClosestPoint(segment, pt), i) AS line_with_closest_point
FROM segments
ORDER BY ST_Distance(segment, pt)
LIMIT 1;
```

Figure



☒☒

ST_AddPoint, ST_ClosestPoint, ST_DWithin, ST_Length2D, ST_LineInterpolatePoint, ST_LineSubstring

7.19.5 ST_LineSubstring

ST_LineSubstring — Returns the part of a line between two fractional locations.

☒☒

geometry **ST_LineSubstring**(geometry a_linestring, float8 startfraction, float8 endfraction);
geography **ST_LineSubstring**(geography a_linestring, float8 startfraction, float8 endfraction);

☒☒

Computes the line which is the section of the input line starting and ending at the given fractional locations. The first argument must be a LINESTRING. The second and third arguments are values in the range [0, 1] representing the start and end locations. For geometry inputs, the fractions are measured in 2D line length. The Z and M values are interpolated for added endpoints if present.

’ ☒☒’ ☒’ ☒’ ☒☒☒☒☒☒☒☒☒☒☒☒ ST_LineInterpolatePoint ☒☒☒☒☒☒☒.



☒☒
This only works with LINESTRINGs. To use on contiguous MULTILINESTRINGs first join them with ST_LineMerge.



□□
1.1.1 □□□□□□□□□□ M □ Z □ (□□□□□) □□□□□□. □□□□□□□□□□□□□□□□□□□□□□□□.

Enhanced: 3.4.0 - Support for geography was introduced.

□□□□: 2.1.0 □□□□, □ 2.0.x □□□□□□□□□□□ ST_Line_Substring □□□□□.

1.1.0 □□□□□□□□□□□□. 1.1.1 □□□□ Z □ M □□□□□□.

 This function supports 3d and will not drop the z-index.

□□

Extract the middle third of a LineString, from 0.333 to 0.666.

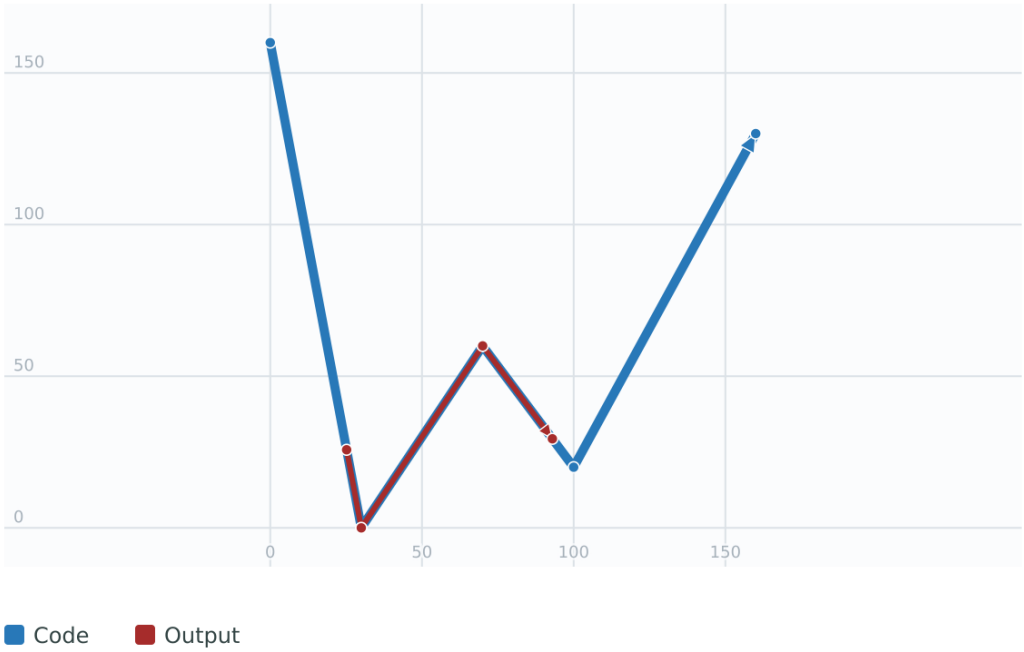
Code

```
SELECT ST_LineSubstring(
'LINESTRING (20 180,50 20,90 80,120 40,180 150)',
0.333,
0.666);
```

Output

```
LINESTRING (45.17311810399485 45.74337011202746, 50 20, 90 80, 112.97593050157862 49.36542599789519)
```

Figure



If start and end locations are the same, the result is a POINT.

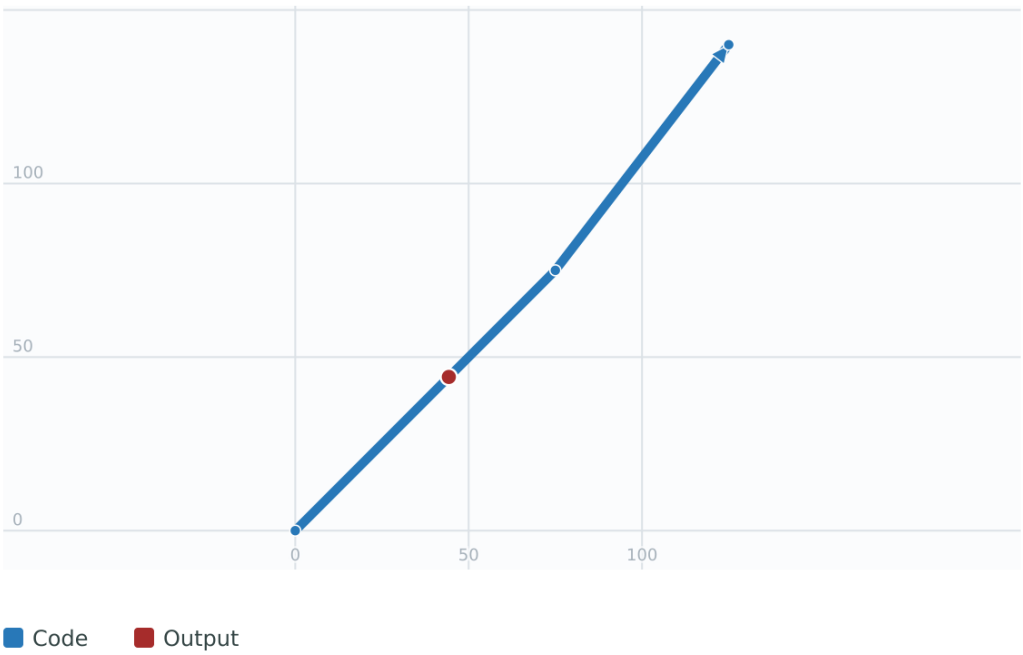
Code

```
SELECT ST_LineSubstring(
'LINESTRING(25 50,100 125,150 190)',
0.333,
0.333);
```

Output

```
POINT(69.28469348539744 94.28469348539744)
```

Figure



A query to cut a LineString into sections of length 100 or shorter. It uses `generate_series()` with a CROSS JOIN LATERAL to produce the equivalent of a FOR loop.

The final generated step is filtered out when the line length is an exact multiple of the requested section length.

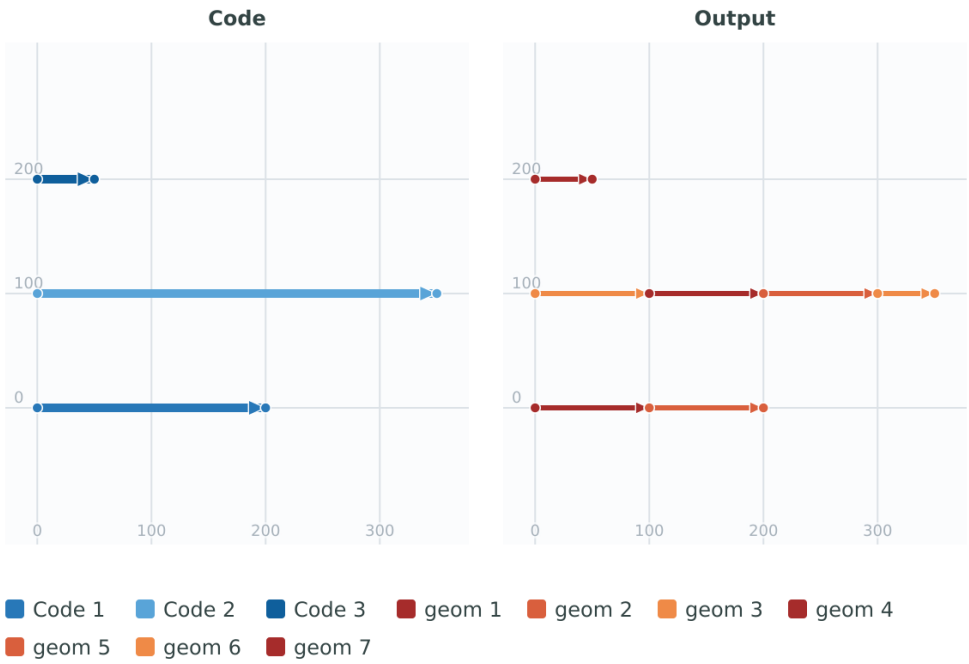
Code

```
WITH data(id, geom) AS (VALUES
        ( 'A', 'LINESTRING( 0 0, 200 0)'::geometry ),
        ( 'B', 'LINESTRING( 0 100, 350 100)'::geometry ),
        ( 'C', 'LINESTRING( 0 200, 50 200)'::geometry )
    )
SELECT id, i,
       ST_LineSubstring(geom, startfrac, LEAST( endfrac, 1 )) AS geom
FROM (
    SELECT id, geom, ST_Length(geom) len, 100 sublen FROM data
    ) AS d
CROSS JOIN LATERAL (
    SELECT i, (sublen * i) / len AS startfrac,
              (sublen * (i+1)) / len AS endfrac
    FROM generate_series(0, floor( len / sublen )::integer ) AS t(i)
    WHERE (sublen * i) / len <
> 1.0
    ) AS d2;
```

Output

```
id | i |            geom
----+---+----------------------------
 A  | 0 | LINESTRING(0 0,100 0)
 A  | 1 | LINESTRING(100 0,200 0)
 B  | 0 | LINESTRING(0 100,100 100)
 B  | 1 | LINESTRING(100 100,200 100)
 B  | 2 | LINESTRING(200 100,300 100)
 B  | 3 | LINESTRING(300 100,350 100)
 C  | 0 | LINESTRING(0 200,50 200)
```

Figure



Split a LineString by points by locating the points along the line, adding the 0 and 1 endpoints, and extracting substrings between adjacent locations.

Code

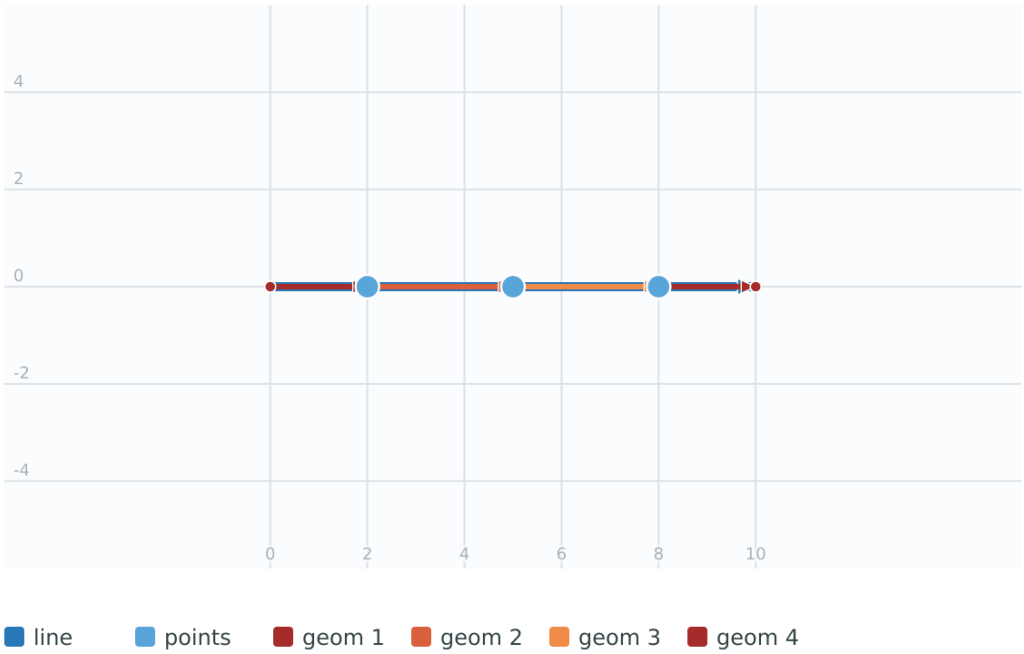
```
WITH data AS (
  SELECT 'LINESTRING(0 0, 10 0)'::geometry AS line,
         'MULTIPOINT((2 0),(5 0),(8 0))'::geometry AS points
),
fractions AS (
  SELECT 0.0 AS fraction
  UNION ALL
  SELECT 1.0
  UNION ALL
  SELECT ST_LineLocatePoint(line, (dp).geom)
  FROM data
  CROSS JOIN LATERAL ST_Dump(points) AS dp
),
ordered AS (
  SELECT DISTINCT fraction
  FROM fractions
),
segments AS (
  SELECT fraction AS start_fraction,
         lead(fraction) OVER (ORDER BY fraction) AS end_fraction
  FROM ordered
)
```

```
SELECT row_number() OVER (ORDER BY start_fraction) AS segment_id,
       ST_LineSubstring(line, start_fraction, end_fraction) AS geom
FROM data
CROSS JOIN segments
WHERE end_fraction IS NOT NULL
  AND start_fraction <
> end_fraction
ORDER BY start_fraction;
```

Output

```
segment_id |         geom
-----------+----------------------
         1 | LINESTRING(0 0,2 0)
         2 | LINESTRING(2 0,5 0)
         3 | LINESTRING(5 0,8 0)
         4 | LINESTRING(8 0,10 0)
```

Figure



Geography implementation measures along a spheroid, geometry along a line

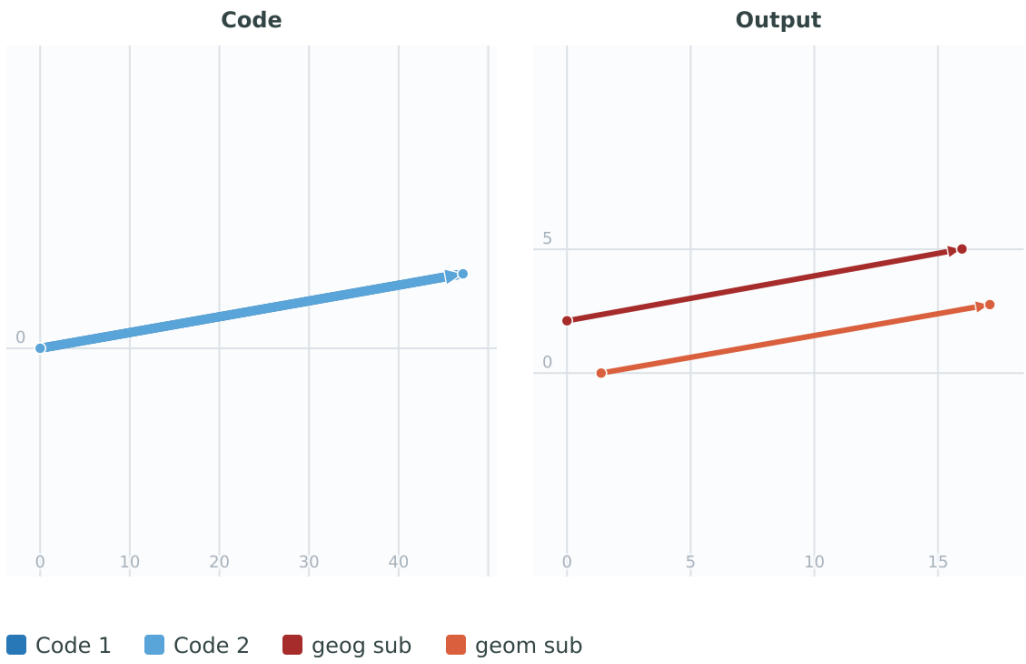
Code

```
SELECT ST_LineSubstring(
'LINESTRING(-118.2436 34.0522,-71.0570 42.3611)'::geography,
0.333,
0.666) AS geog_sub,
ST_LineSubstring(
    'LINESTRING(-118.2436 34.0522,-71.0570 42.3611)'::geometry,
    0.333,
    0.666) AS geom_sub;
```

Output

```
-[ RECORD 1 ]----------
geog_sub | LINESTRING(-103.911641 38.931128,-87.941787 41.831072)
geom_sub | LINESTRING(-102.530462 36.819064,-86.817324 39.585927)
```

Figure



For geometry inputs, the fractional locations are based on the 2D length of the line, even when the input has Z values. The resulting Z values are interpolated along the selected 2D location.

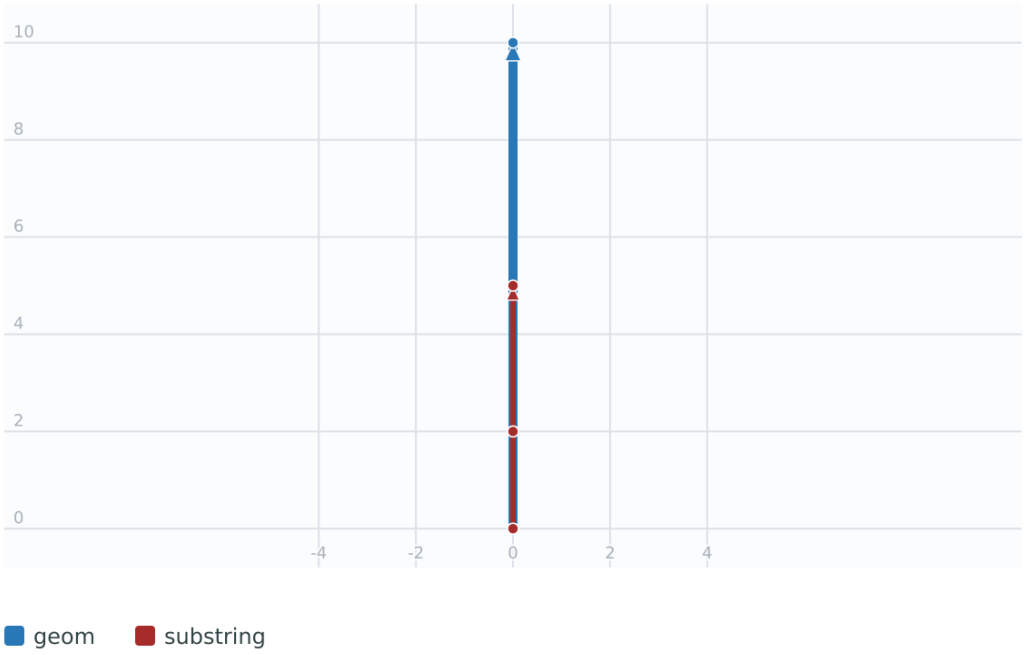
Code

```
WITH data AS (
  SELECT 'LINESTRING Z (0 0 0,0 2 5,0 10 10)'::geometry AS geom
)
SELECT ST_Length(geom) AS length_2d,
       ST_3DLength(geom) AS length_3d,
       ST_LineSubstring(geom, 0, 0.5) AS substring
FROM data;
```

Output

```
length_2d |    length_3d     |              substring
----------+------------------+-------------------------------------
        10 | 14.819145939191106 | LINESTRING Z (0 0 0,0 2 5,0 5 6.875)
```

Figure



☒☒

ST_Length, ST_LineExtend, ST_LineInterpolatePoint, ST_LineLocatePoint, ST_LineMerge, ST_Split

7.19.6 ST_LocateAlong

ST_LocateAlong — Returns the point(s) on a geometry that match a measure value.

☒☒

geometry **ST_LocateAlong**(geometry geom_with_measure, float8 measure, float8 offset = 0);

☒☒

Returns the location(s) along a measured geometry that have the given measure values. The result is a Point or MultiPoint. Polygonal inputs are not supported.

If `offset` is provided, the result is offset to the left or right of the input line by the specified distance. A positive offset will be to the left, and a negative one to the right.



☒☒
Use this function only for linear geometries with an M component

The semantic is specified by the *ISO/IEC 13249-3 SQL/MM Spatial* standard.

1.1.0 ☒☒☒☒☒☒☒☒☒ ST_Locate_Along_Measure ☒☒☒☒☒☒☒☒.

□□□□: 2.0.0 □□□□□□□□ ST_Locate_Along_Measure □□□□□□□□□□. □□□□□□□□□□□□□□ □□□□□□□□□□□□, □□□□□□□□□□□□□□.

This function supports M coordinates.

This method implements the SQL/MM specification. SQL-MM IEC 13249-3: 5.1.13

□□

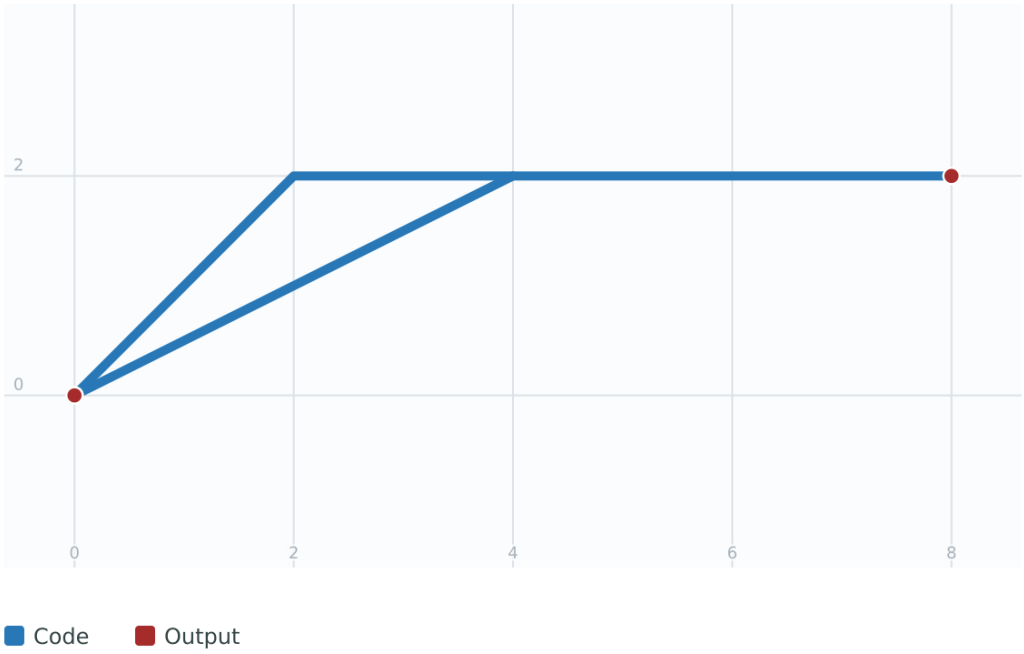
Code

```
SELECT ST_LocateAlong(
'MULTILINESTRINGM((1 2 3, 3 4 2, 9 4 3),(1 2 3, 5 4 5))'::geometry,
3);
```

Output

```
MULTIPOINT M ((1 2 3),(9 4 3),(1 2 3))
```

Figure



□□

ST_LocateBetween, ST_LocateBetweenElevations, ST_InterpolatePoint

7.19.7 ST_LocateBetween

ST_LocateBetween — Returns the portions of a geometry that match a measure range.

⊠⊠

geometry **ST_LocateBetween**(geometry geom, float8 measure_start, float8 measure_end, float8 offset = 0);

⊠⊠

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

Clipping a non-convex POLYGON may produce invalid geometry.

The semantic is specified by the *ISO/IEC 13249-3 SQL/MM Spatial* standard.

1.1.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠ ST_Locate_Between_Measures ⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠⊠⊠: 2.0.0 ⊠⊠⊠⊠⊠⊠⊠ ST_Locate_Along_Measure ⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

Enhanced: 3.0.0 - added support for POLYGON, TIN, TRIANGLE.

This function supports M coordinates.

This method implements the SQL/MM specification. SQL-MM IEC 13249-3: 5.1

⊠⊠

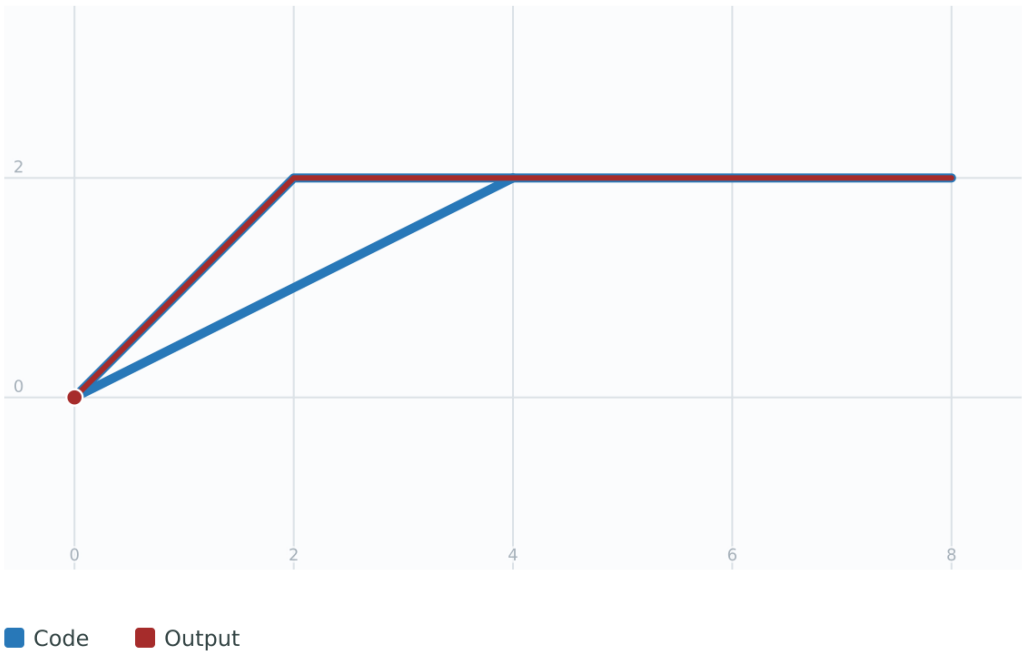
Code

```
SELECT ST_LocateBetween(
'MULTILINESTRING M ((1 2 3,3 4 2,9 4 3),(1 2 3,5 4 5))'::geometry,
1.5,
3);
```

Output

```
GEOMETRYCOLLECTION M (LINESTRING M (1 2 3,3 4 2,9 4 3),POINT M (1 2 3))
```

Figure



Extract the section between measures 2 and 8, offset to the left. Because the clipped line and the offset result share the same coordinate space, this example is intended for a single overlay panel rather than separate side-by-side figures.

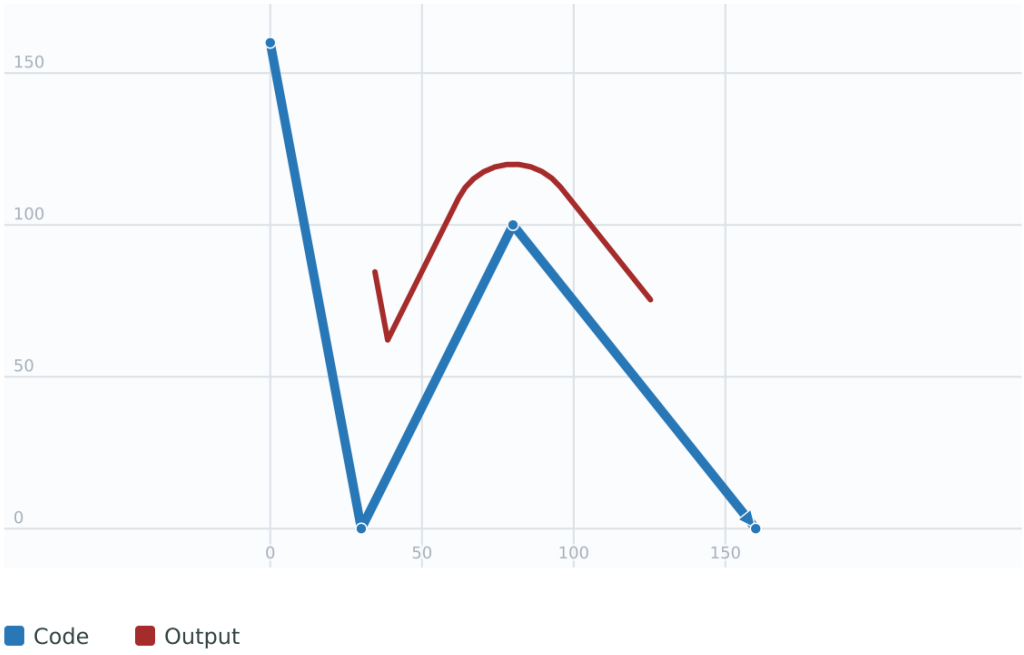
Code

```
SELECT ST_LocateBetween(
ST_AddMeasure(
    'LINESTRING (20 180,50 20,100 120,180 20)',
    0,
    10),
2,
8,
20);
```

Output

```
MULTILINESTRING((54.49835 104.53427,58.700561 82.122481,82.111456 128.944272,84.248937 132.324934,87.01637 ↩
    135.21267,90.303073 137.491987,93.977598 139.071724,97.892984 139.888702,101.892639 139.910247,105.816599 ↩
    139.135495,109.507928 137.595434,112.818995 135.351657,115.617376 132.493901,145.310173 95.377905))
```

Figure



□□

ST_LocateAlong, ST_LocateAlong, ST_LocateBetween

7.19.8 ST_LocateBetweenElevations

ST_LocateBetweenElevations — Returns the portions of a geometry that lie in an elevation (Z) range.

□□

geometry **ST_LocateBetweenElevations**(geometry geom, float8 elevation_start, float8 elevation_end);

□□

Returns a geometry (collection) with the portions of a geometry that lie in an elevation (Z) range.
Clipping a non-convex POLYGON may produce invalid geometry.
1.4.0 □□□□□□□□□□□□.
Enhanced: 3.0.0 - added support for POLYGON, TIN, TRIANGLE.

This function supports 3d and will not drop the z-index.

□□

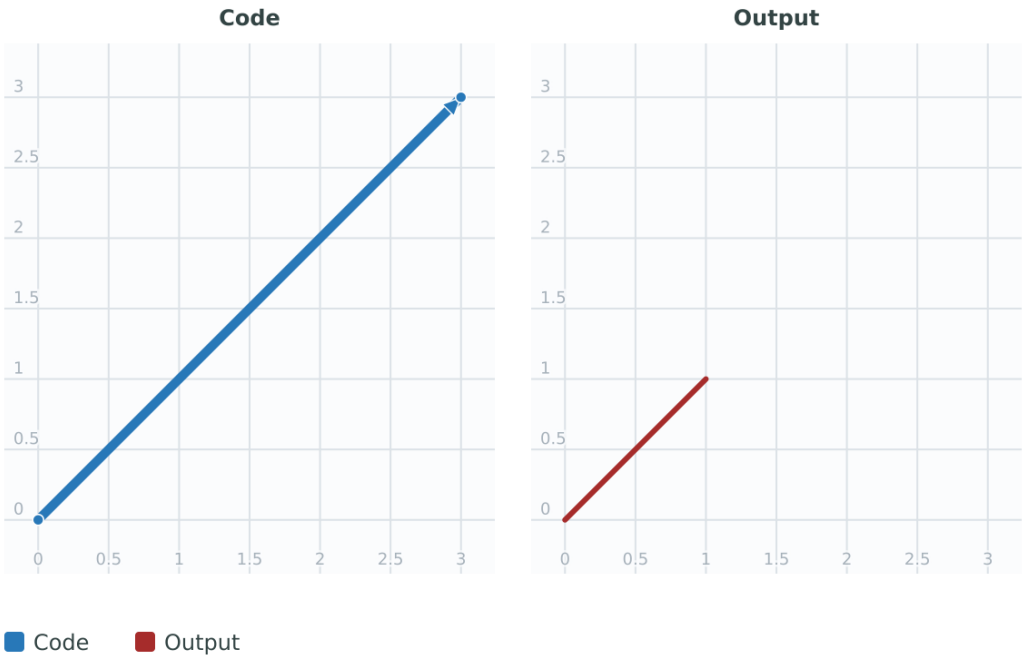
Code

```
SELECT ST_LocateBetweenElevations(
'LINESTRING(1 2 3,4 5 6)'::geometry,
2,
4);
```

Output

```
MULTILINESTRING Z ((1 2 3,2 3 4))
```

Figure



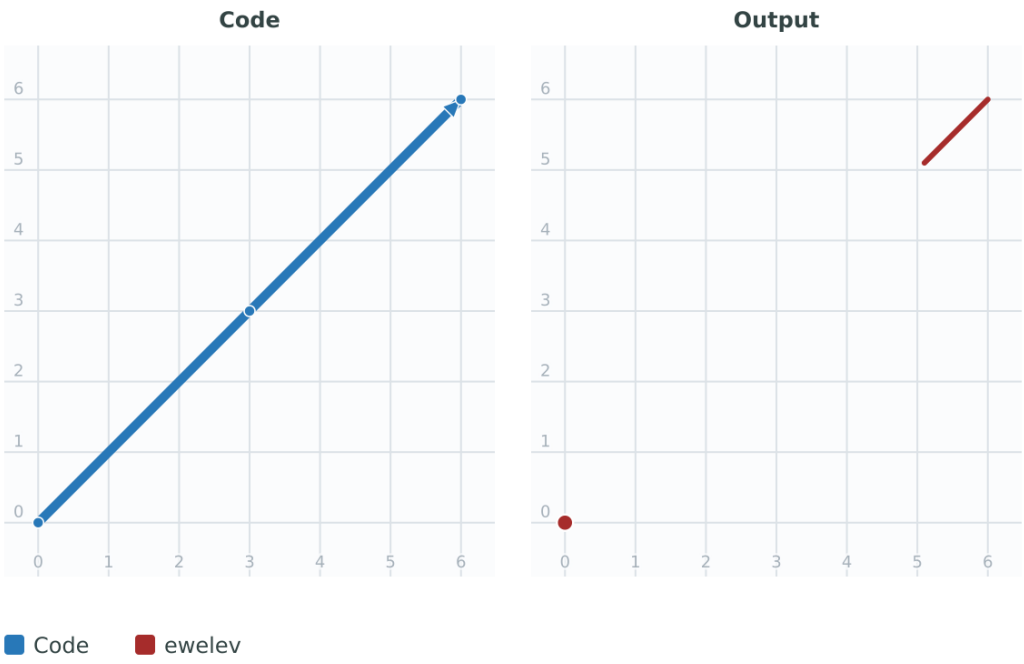
Code

```
SELECT ST_LocateBetweenElevations(
'LINESTRING(1 2 6,4 5 -1,7 8 9)',
6,
9) As ewelev;
```

Output

```
GEOMETRYCOLLECTION Z (POINT Z (1 2 6),LINESTRING Z (6.1 7.1 6,7 8 9))
```

Figure



□□

ST_Dump, ST_LocateAlong, ST_LocateBetween

7.19.9 ST_InterpolatePoint

ST_InterpolatePoint — □□□□□□□□□□□□□□□□□□□□□□□□ (M □□) □□□□□□□□.

□□

float8 **ST_InterpolatePoint**(geometry linear_geom_with_measure, geometry point);

□□

Returns an interpolated measure value of a linear measured geometry at the location closest to the given point.

□□
Use this function only for linear geometries with an M component

---

2.0.0 □□□□□□□□□□□□.

This function supports 3d and will not drop the z-index.

□□

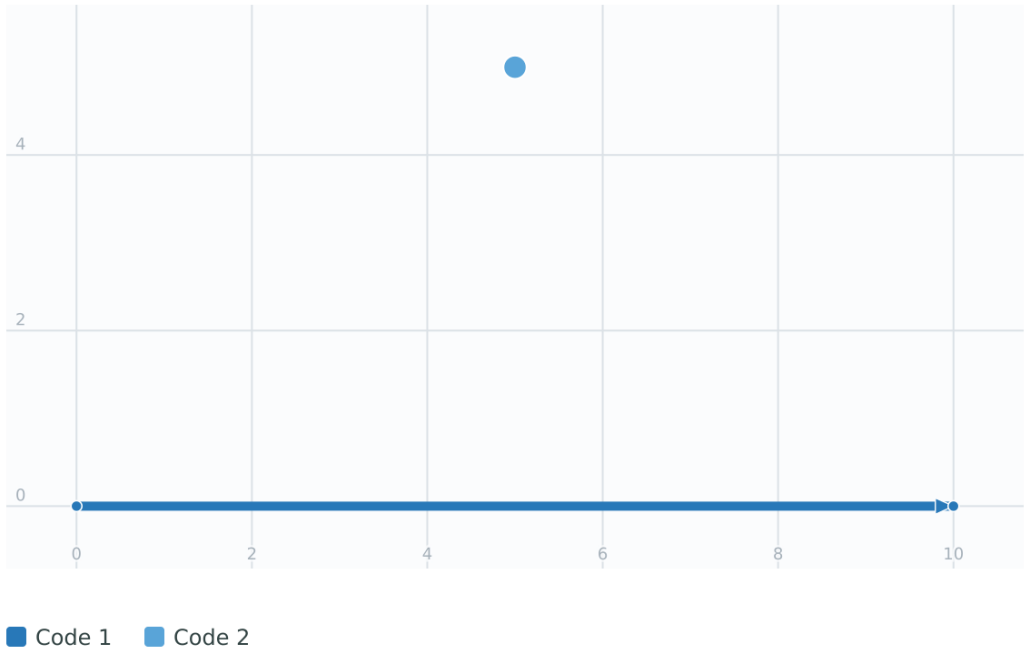
Code

```
SELECT ST_InterpolatePoint(
'LINESTRING M (0 0 0,10 0 20)',
'POINT(5 5)');
```

Output

```
10
```

Figure



☒☒

ST_AddMeasure, ST_LocateAlong, ST_LocateBetween

7.19.10 ST_3DInterpolatePoint

ST_3DInterpolatePoint — Returns the interpolated measure of a geometry closest to a point in 3D.

☒☒

float8 **ST_3DInterpolatePoint**(geometry linear_geom_with_measure, geometry point);

☒☒

Returns the interpolated measure value of a linear ZM geometry at the location closest to the given point, using 3D (XYZ) distance for the projection. Use this function when the geometry has significant Z variation, such as flight trajectories, where ST_InterpolatePoint would give incorrect results by ignoring the Z dimension.



□□
The line must have both Z and M dimensions. The point should have a Z dimension.

Availability: 3.7.0



This function supports 3d and will not drop the z-index.

□□

This example interpolates on a line rising diagonally in Z, using a point at the 3D midpoint.

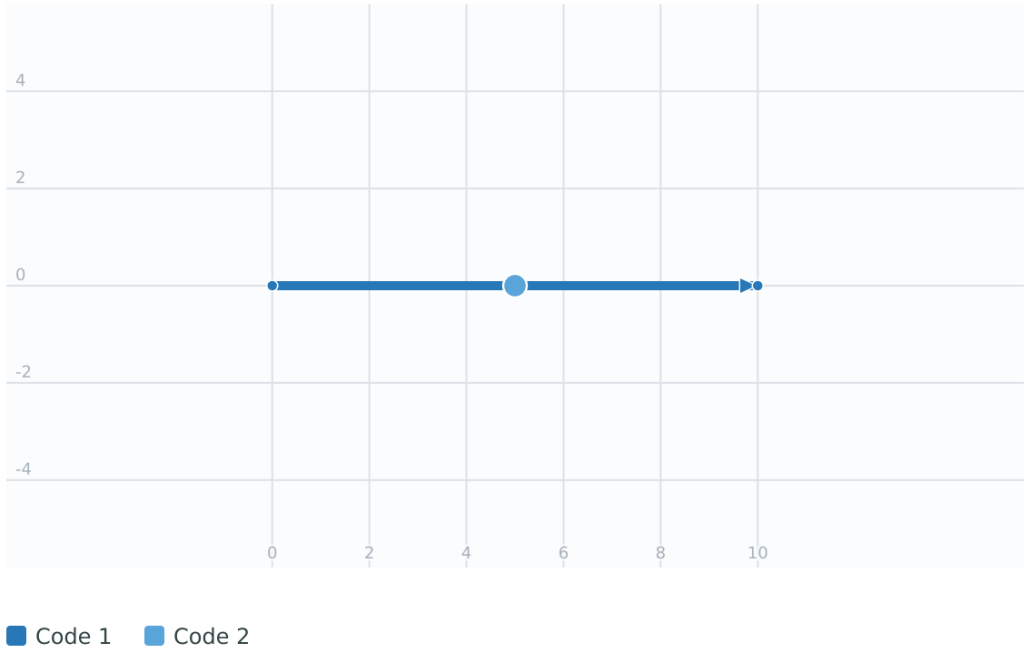
Code

```
SELECT ST_3DInterpolatePoint(
'LINESTRING ZM (0 0 0 0,10 0 10 20)',
'POINT Z (5 0 5)');
```

Output

```
10
```

Figure



This example uses a vertical line with zero XY extent, where 3D projection works but 2D projection does not.

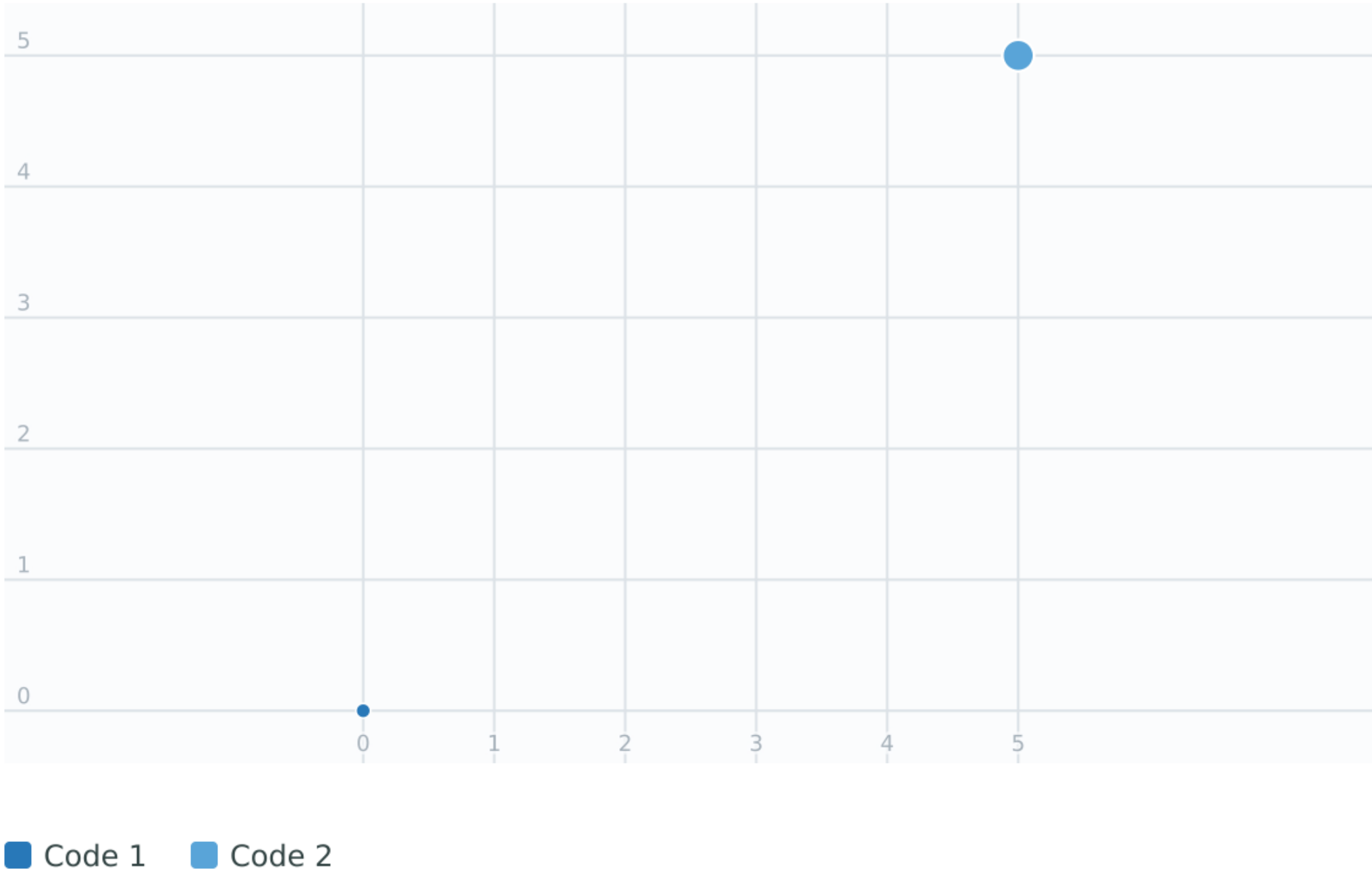
Code

```
SELECT ST_3DInterpolatePoint(
'LINESTRING ZM (0 0 0 0,0 0 10 100)',
'POINT Z (5 5 5)');
```

Output

```
50
```

Figure



⊠⊠

ST_InterpolatePoint, ST_AddMeasure, ST_LocateAlong, ST_LocateBetween

7.19.11 ST_AddMeasure

ST_AddMeasure — Interpolates measures along a linear geometry.

⊠⊠

geometry **ST_AddMeasure**(geometry geom_mline, float8 measure_start, float8 measure_end);

⊠⊠

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

1.5.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

This function supports 3d and will not drop the z-index.

⊠⊠

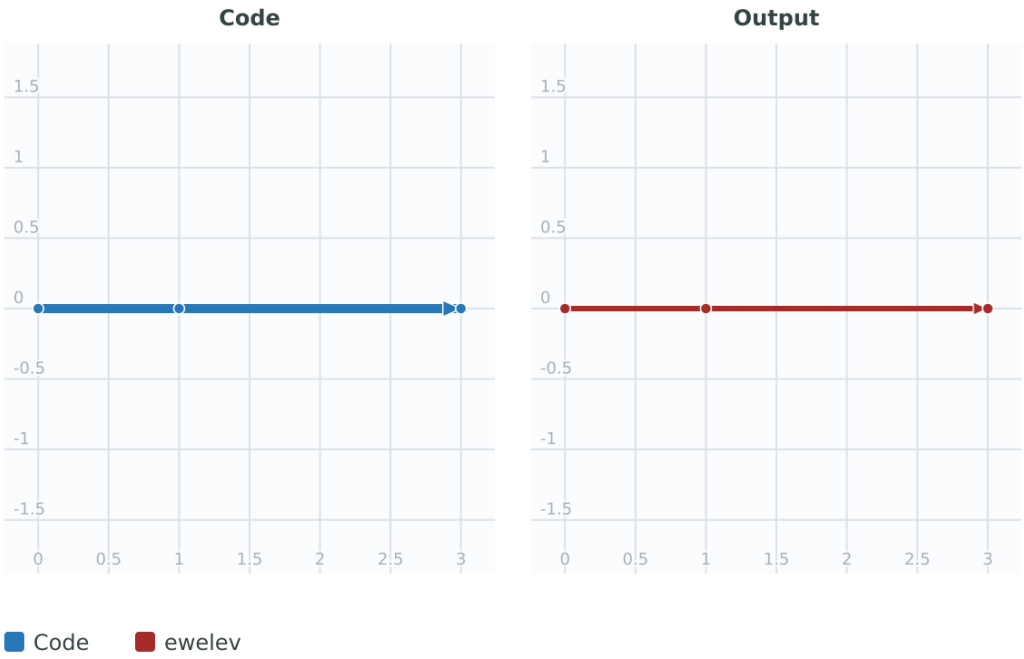
Code

```
SELECT ST_AddMeasure(
'LINESTRING(1 0,2 0,4 0)',
1,
4) As ewelev;
```

Output

```
LINESTRING M (1 0 1,2 0 2,4 0 4)
```

Figure



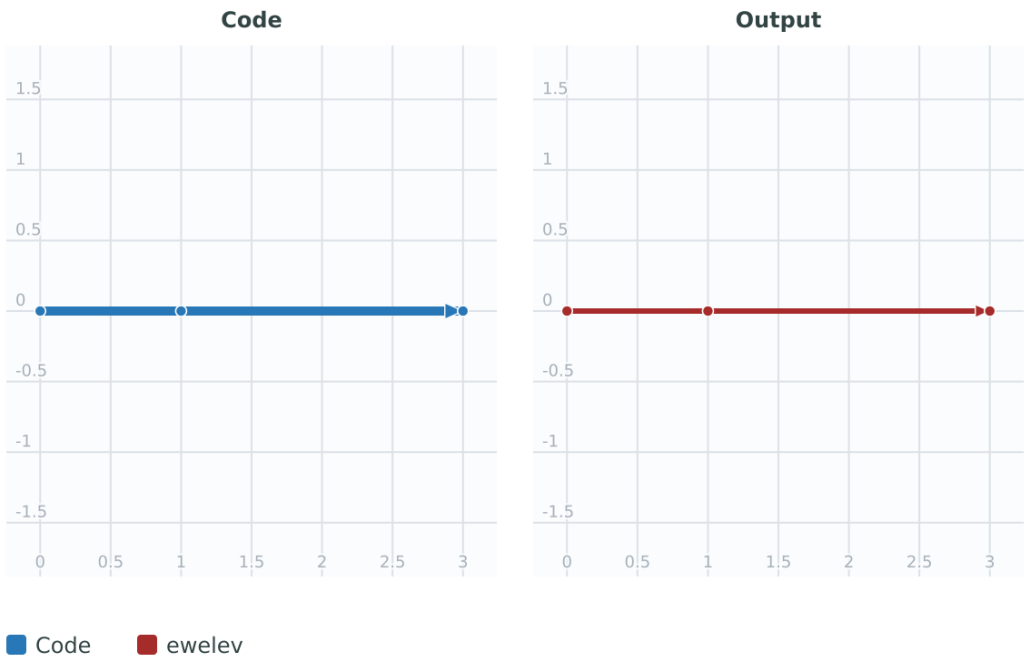
Code

```
SELECT ST_AddMeasure(
'LINESTRING(1 0 4,2 0 4,4 0 4)',
10,
40) As ewelev;
```

Output

```
LINESTRING ZM (1 0 4 10,2 0 4 20,4 0 4 40)
```

Figure



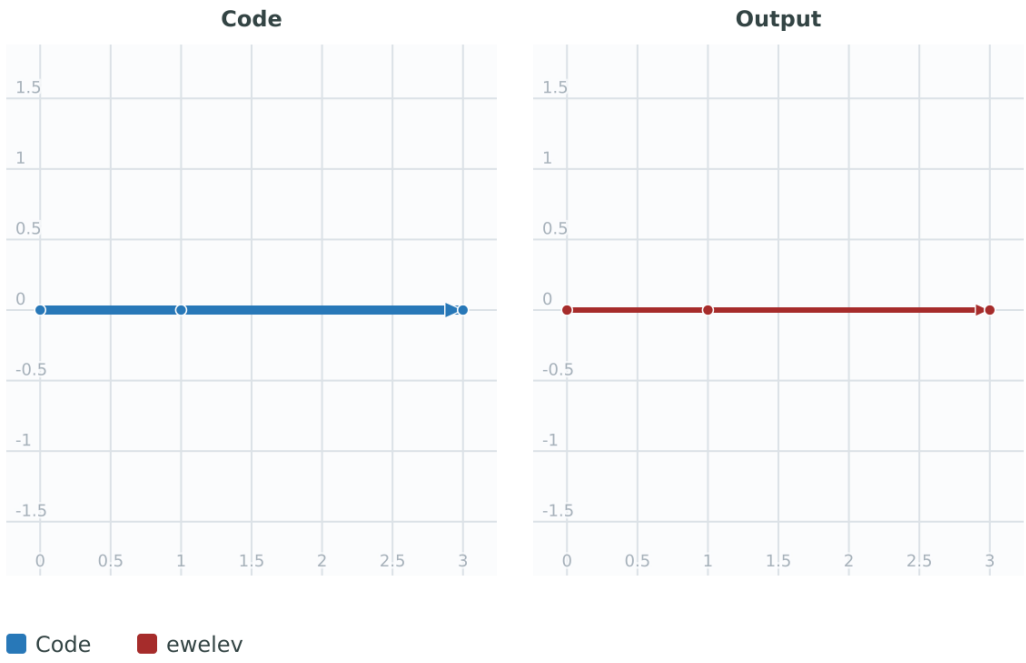
Code

```
SELECT ST_AddMeasure(
'LINESTRINGM(1 0 4, 2 0 4, 4 0 4)',
10,
40) As ewelev;
```

Output

```
LINESTRING M (1 0 10,2 0 20,4 0 40)
```

Figure



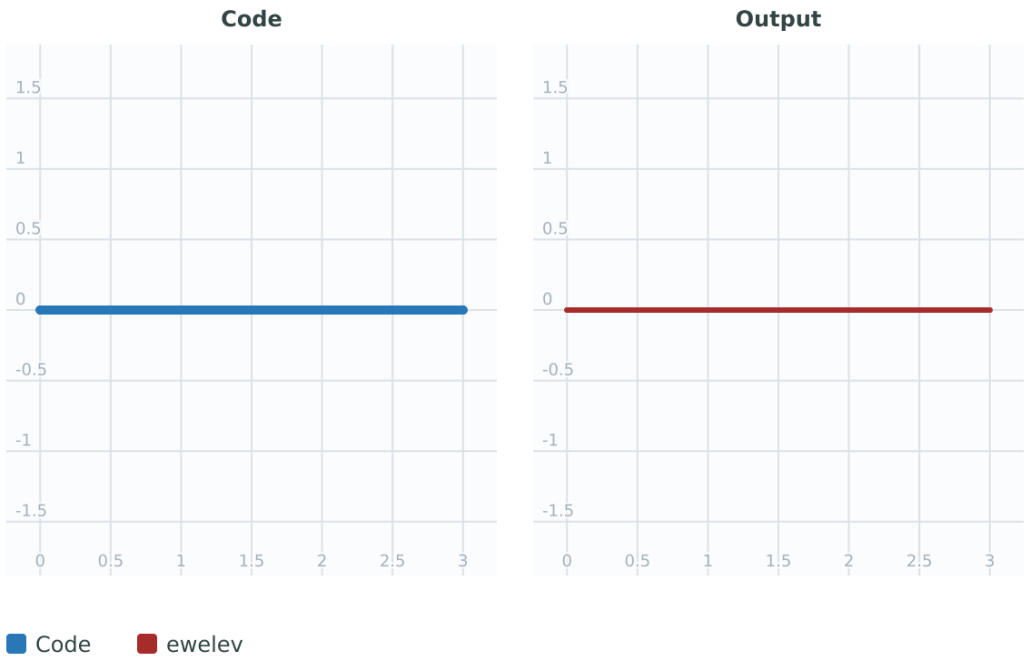
Code

```
SELECT ST_AddMeasure(
'MULTILINESTRINGM((1 0 4, 2 0 4, 4 0 4),(1 0 4, 2 0 4, 4 0 4))',
10,
70) As ewelev;
```

Output

```
MULTILINESTRING M ((1 0 10,2 0 20,4 0 40),(1 0 40,2 0 50,4 0 70))
```

Figure



7.20 Trajectory Functions

7.20.1 ST_IsValidTrajectory

ST_IsValidTrajectory — Tests if the geometry is a valid trajectory.

☒☒

boolean **ST_IsValidTrajectory**(geometry line);

☒☒

Tests if a geometry encodes a valid trajectory. A valid trajectory is represented as a `LINESTRING` with measures (M values). The measure values must increase from each vertex to the next.

Valid trajectories are expected as input to spatio-temporal functions like ST_ClosestPointOfApproach

2.2.0 ☒☒☒☒☒☒☒☒☒☒.

This function supports 3d and will not drop the z-index.

□□

This example constructs a valid trajectory.

Code

```
SELECT ST_IsValidTrajectory(ST_MakeLine(ST_MakePointM(0, 0, 1),
ST_MakePointM(0, 1, 2)));
```

Output

```
t
```

This example constructs an invalid trajectory.

Code

```
SELECT ST_IsValidTrajectory(ST_MakeLine(ST_MakePointM(0, 0, 1), ST_MakePointM(0, 1, 0)));
```

Output

```
NOTICE:  Measure of vertex 1 (0) not bigger than measure of vertex 0 (1)
f
```

□□

ST_ClosestPointOfApproach

7.20.2 ST_ClosestPointOfApproach

ST_ClosestPointOfApproach — Returns a measure at the closest point of approach of two trajectories.

□□

float8 **ST_ClosestPointOfApproach**(geometry track1, geometry track2);

□□

Returns the smallest measure at which points interpolated along the given trajectories are the least distance apart.

Inputs must be valid trajectories as checked by ST_IsValidTrajectory. Null is returned if the trajectories do not overlap in their M ranges.

To obtain the actual points at the computed measure use ST_LocateAlong .

2.2.0 □□□□□□□□□□□□.

This function supports 3d and will not drop the z-index.

□□

Return the time in which two objects moving between 10:00 and 11:00 are closest to each other and their distance at that point.

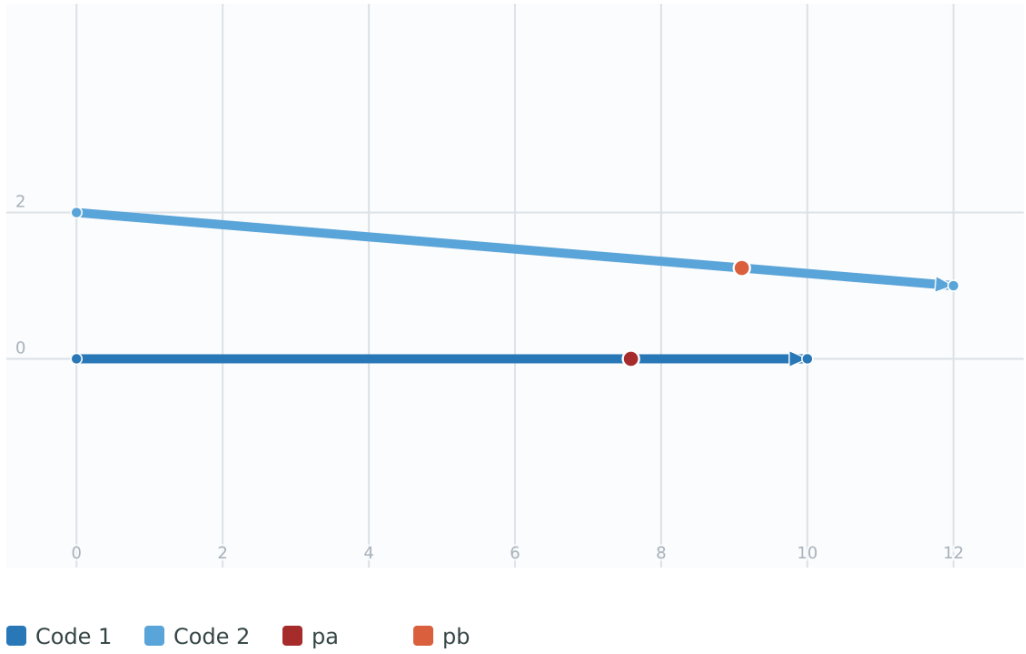
Code

```
WITH inp AS ( SELECT
  ST_AddMeasure('LINESTRING Z (0 0 0,10 0 5)'::geometry,
    extract(epoch from timestamp '2015-05-26 10:00' AT TIME ZONE 'UTC'),
    extract(epoch from timestamp '2015-05-26 11:00' AT TIME ZONE 'UTC')
  ) a,
  ST_AddMeasure('LINESTRING Z (0 2 10,12 1 2)'::geometry,
    extract(epoch from timestamp '2015-05-26 10:00' AT TIME ZONE 'UTC'),
    extract(epoch from timestamp '2015-05-26 11:00' AT TIME ZONE 'UTC')
  ) b
), cpa AS (
  SELECT ST_ClosestPointOfApproach(a, b) m FROM inp
), points AS (
  SELECT ST_GeometryN(ST_LocateAlong(a, m), 1) pa,
         ST_GeometryN(ST_LocateAlong(b, m), 1) pb
  FROM inp, cpa
)
SELECT to_timestamp(m) AT TIME ZONE 'UTC' t,
       ST_3DDistance(pa, pb) distance,
       pa AS pa, pb AS pb
FROM points, cpa;
```

Output

```
-[ RECORD 1 ]---------------------------
t        | 2015-05-26 10:45:31.034483
distance | 1.9652147377620688
pa       | POINT ZM (7.59 0 3.79 1432637131.03)
pb       | POINT ZM (9.1 1.24 3.93 1432637131.03)
```

Figure



☒☒

ST_IsValidTrajectory, ST_DistanceCPA, ST_LocateAlong, ST_AddMeasure

7.20.3 ST_DistanceCPA

ST_DistanceCPA — Returns the distance between the closest point of approach of two trajectories.

□□

float8 **ST_DistanceCPA**(geometry track1, geometry track2);

□□

Returns the distance (in 2D) between two trajectories at their closest point of approach.

Inputs must be valid trajectories as checked by ST_IsValidTrajectory. Null is returned if the trajectories do not overlap in their M ranges.

2.2.0 □□□□□□□□□□□□.

This function supports 3d and will not drop the z-index.

□□

Return the minimum distance of two objects moving between 10:00 and 11:00.

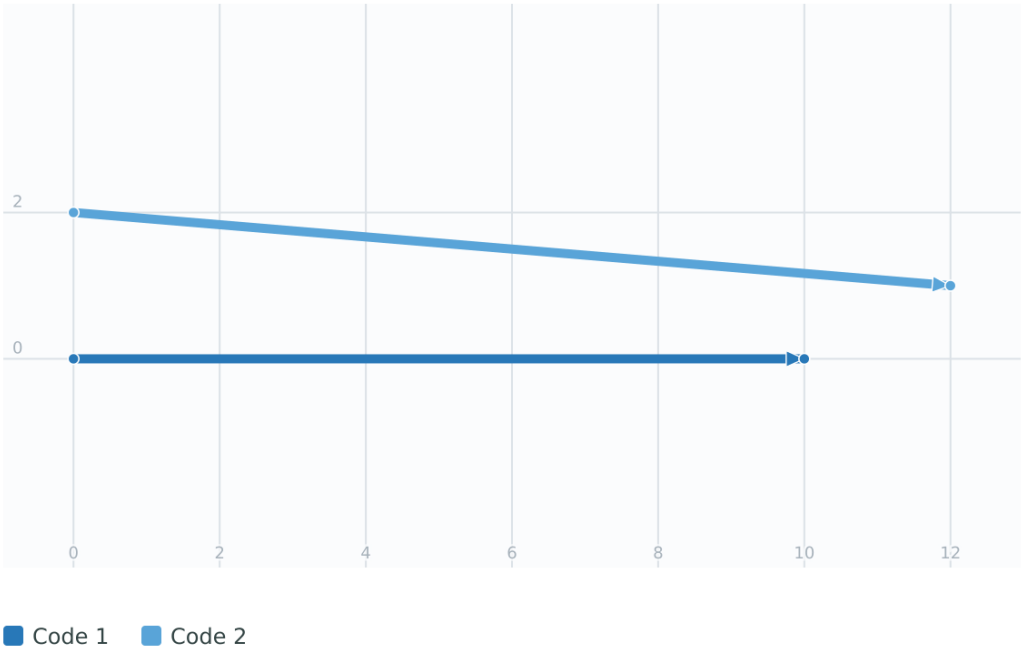
Code

```
WITH inp AS ( SELECT
  ST_AddMeasure('LINESTRING Z (0 0 0,10 0 5)'::geometry,
    extract(epoch from timestamp '2015-05-26 10:00' AT TIME ZONE 'UTC'),
    extract(epoch from timestamp '2015-05-26 11:00' AT TIME ZONE 'UTC')
  ) a,
  ST_AddMeasure('LINESTRING Z (0 2 10,12 1 2)'::geometry,
    extract(epoch from timestamp '2015-05-26 10:00' AT TIME ZONE 'UTC'),
    extract(epoch from timestamp '2015-05-26 11:00' AT TIME ZONE 'UTC')
  ) b
)
SELECT ST_DistanceCPA(a, b) distance FROM inp;
```

Output

```
1.9652147377620688
```

Figure



□□

ST_IsValidTrajectory, ST_ClosestPointOfApproach, ST_AddMeasure, |=|

7.20.4 ST_CPAWithin

ST_CPAWithin — Tests if the closest point of approach of two trajectories is within the specified distance.

□□

boolean **ST_CPAWithin**(geometry track1, geometry track2, float8 dist);

□□

Tests whether two moving objects have ever been closer than the specified distance.

Inputs must be valid trajectories as checked by ST_IsValidTrajectory. False is returned if the trajectories do not overlap in their M ranges.

2.2.0 □□□□□□□□□□□□.

This function supports 3d and will not drop the z-index.

□□

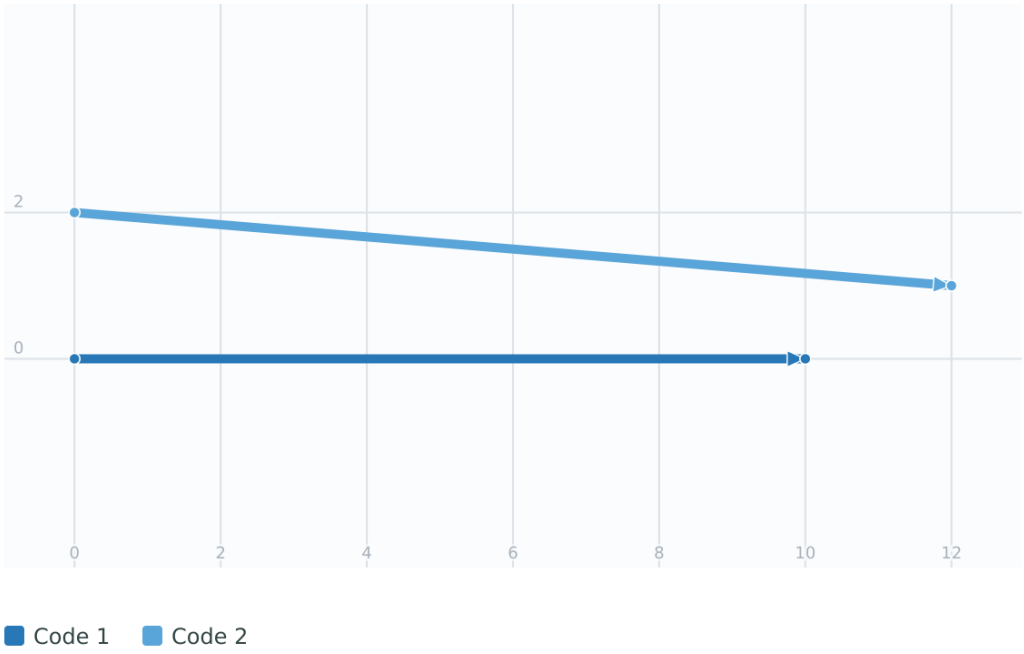
Code

```
WITH inp AS ( SELECT
  ST_AddMeasure('LINESTRING Z (0 0 0,10 0 5)'::geometry,
    extract(epoch from timestamp '2015-05-26 10:00' AT TIME ZONE 'UTC'),
    extract(epoch from timestamp '2015-05-26 11:00' AT TIME ZONE 'UTC')
  ) a,
  ST_AddMeasure('LINESTRING Z (0 2 10,12 1 2)'::geometry,
    extract(epoch from timestamp '2015-05-26 10:00' AT TIME ZONE 'UTC'),
    extract(epoch from timestamp '2015-05-26 11:00' AT TIME ZONE 'UTC')
  ) b
)
SELECT ST_CPAWithin(a, b, 2), ST_DistanceCPA(a, b) distance FROM inp;
```

Output

```
st_cpawithin |     distance
-------------+------------------
 t           | 1.9652147377620688
```

Figure



☒☒

ST_IsValidTrajectory, ST_ClosestPointOfApproach, ST_DistanceCPA, |=|

7.21 Version Functions

7.21.1 PostGIS_Extensions_Upgrade

PostGIS_Extensions_Upgrade — Packages and upgrades PostGIS extensions (e.g. postgis_raster, postgis_topology, postgis_sfcgal) to given or latest version.

□□

text **PostGIS_Extensions_Upgrade**(text target_version=null);

□□

Packages and upgrades PostGIS extensions to given or latest version. Only extensions you have installed in the database will be packaged and upgraded if needed. Reports full PostGIS version and build configuration infos after. This is short-hand for doing multiple CREATE EXTENSION .. FROM unpackaged and ALTER EXTENSION .. UPDATE for each PostGIS extension. Currently only tries to upgrade extensions postgis, postgis_raster, postgis_sfcgal, and postgis_topology.

Availability: 2.5.0

Changed: 3.4.0 to add target_version argument.

Changed: 3.3.0 support for upgrades from any PostGIS version. Does not work on all systems.

Changed: 3.0.0 to repackage loose extensions and support postgis_raster.

□□

Code

```
SELECT PostGIS_Extensions_Upgrade();
```

Output

```
NOTICE:  Packaging extension postgis
NOTICE:  Packaging extension postgis_raster
NOTICE:  Packaging extension postgis_sfcgal
NOTICE:  Extension postgis_topology is not available or not packagable for some reason
```

□□

3.4□, PostGIS_GEOS_Version, PostGIS_Lib_Version, PostGIS_LibXML_Version, PostGIS_PROJ_Version, PostGIS_Version

7.21.2 PostGIS_Full_Version

PostGIS_Full_Version — Reports full PostGIS version and build configuration infos.

□□

text **PostGIS_Full_Version**();

□□

Reports full PostGIS version and build configuration infos. Also informs about synchronization between libraries and scripts suggesting upgrades as needed.

Enhanced: 3.4.0 now includes extra PROJ configurations NETWORK_ENABLED, URL_ENDPOINT and DATABASE_PATH of proj.db location

□□

Code

```
SELECT PostGIS_Full_Version();
```

Output

```
POSTGIS="3.7.0dev 3.7.0alpha1-477-ga9004e778" [EXTENSION] PGSQL="180" GEOS="3.15.0beta1" SFCGAL="2.3.0" PROJ="9.8.1 ↩
     NETWORK_ENABLED=OFF URL_ENDPOINT=https://cdn.proj.org DATABASE_PATH=/usr/share/proj/proj.db" GDAL="GDAL 3.13.1, ↩
     released 2026/06/01" LIBXML="2.15.3"
```

□□

3.4□, PostGIS_GEOS_Version, PostGIS_Lib_Version, PostGIS_LibXML_Version, PostGIS_PROJ_Version, PostGIS_Wagyu_Version, PostGIS_Version

7.21.3 PostGIS_GEOS_Version

PostGIS_GEOS_Version — Returns the version number of the GEOS library.

□□

text **PostGIS_GEOS_Version**();

□□

Returns the version number of the GEOS library, or NULL if GEOS support is not enabled.

□□

Code

```
SELECT PostGIS_GEOS_Version();
```

Output

```
3.12.0dev-CAPI-1.18.0
```

□□

PostGIS_Full_Version, PostGIS_Lib_Version, PostGIS_LibXML_Version, PostGIS_PROJ_Version, PostGIS_Version

7.21.4 PostGIS_GEOS_Compiled_Version

PostGIS_GEOS_Compiled_Version — Returns the version number of the GEOS library against which PostGIS was built.

□□

text **PostGIS_GEOS_Compiled_Version**();

□□

Returns the version number of the GEOS library, or against which PostGIS was built.

Availability: 3.4.0

□□

Code

```
SELECT PostGIS_GEOS_Compiled_Version();
```

Output

```
3.12.0
```

□□

PostGIS_GEOS_Version, PostGIS_Full_Version

7.21.5 PostGIS_Liblwgeom_Version

PostGIS_Liblwgeom_Version — Returns the version number of the liblwgeom library. This should match the version of PostGIS.

□□

text **PostGIS_Liblwgeom_Version**();

□□

Returns the version number of the liblwgeom library/

□□

Code

```
SELECT PostGIS_Liblwgeom_Version();
```

Output

```
3.4.0dev 3.3.0rc2-993-g61bdf43a7
```

□□

PostGIS_Full_Version, PostGIS_Lib_Version, PostGIS_LibXML_Version, PostGIS_PROJ_Version, PostGIS_Version

7.21.6 PostGIS_LibXML_Version

PostGIS_LibXML_Version — Returns the version number of the libxml2 library.

□□

text **PostGIS_LibXML_Version**();

□□

Returns the version number of the LibXML2 library.
1.5 □□□□□□□□□□□□.

□□

Code

```
SELECT PostGIS_LibXML_Version();
```

Output

```
2.9.10
```

□□

PostGIS_Full_Version, PostGIS_Lib_Version, PostGIS_PROJ_Version, PostGIS_GEOS_Version, PostGIS_Versio

7.21.7 PostGIS_LibJSON_Version

PostGIS_LibJSON_Version — Returns the version number of the libjson-c library.

□□

text **PostGIS_LibJSON_Version**();

□□

Returns the version number of the LibJSON-C library.

□□

Code

```
SELECT PostGIS_LibJSON_Version();
```

Output

```
0.17
```

□□

PostGIS_Full_Version, PostGIS_Lib_Version, PostGIS_PROJ_Version, PostGIS_GEOS_Version, PostGIS_Versio

7.21.8 PostGIS_Lib_Build_Date

PostGIS_Lib_Build_Date — Returns build date of the PostGIS library.

□□

text **PostGIS_Lib_Build_Date**();

□□

Returns build date of the PostGIS library.

□□

Code

```
SELECT PostGIS_Lib_Build_Date();
```

Output

```
2023-06-22 03:56:11
```

7.21.9 PostGIS_Lib_Version

PostGIS_Lib_Version — Returns the version number of the PostGIS library.

□□

text **PostGIS_Lib_Version**();

□□

Returns the version number of the PostGIS library.

□□

Code

```
SELECT PostGIS_Lib_Version();
```

Output

```
3.4.0dev
```

□□

PostGIS_Full_Version, PostGIS_GEOS_Version, PostGIS_LibXML_Version, PostGIS_PROJ_Version, PostGIS_Version

7.21.10 PostGIS_PROJ_Version

PostGIS_PROJ_Version — Returns the version number of the PROJ4 library.

□□

text **PostGIS_PROJ_Version**();

□□

Returns the version number of the PROJ library and some configuration options of proj.

Enhanced: 3.4.0 now includes NETWORK_ENABLED, URL_ENDPOINT and DATABASE_PATH of proj.db location

□□

Code

```
SELECT PostGIS_PROJ_Version();
```

Output

```
7.2.1 NETWORK_ENABLED=OFF URL_ENDPOINT=https://cdn.proj.org USER_WRITABLE_DIRECTORY=/tmp/ ↩
    proj DATABASE_PATH=/usr/share/proj/proj.db
```

□□

PostGIS_PROJ_Compiled_Version, PostGIS_Full_Version, PostGIS_GEOS_Version, PostGIS_Lib_Version, PostGIS_LibXML_Version, PostGIS_Version

7.21.11 PostGIS_PROJ_Compiled_Version

PostGIS_PROJ_Compiled_Version — Returns the version number of the PROJ library against which PostGIS was built.

□□

text **PostGIS_PROJ_Compiled_Version**();

□□

Returns the version number of the PROJ library, or against which PostGIS was built.
Availability: 3.5.0

□□

Code

```
SELECT PostGIS_PROJ_Compiled_Version();
```

Output

```
9.1.1
```

□□

PostGIS_PROJ_Version, PostGIS_Full_Version

7.21.12 PostGIS_Wagyu_Version

PostGIS_Wagyu_Version — Returns the version number of the internal Wagyu library.

□□

text **PostGIS_Wagyu_Version**();

□□

Returns the version number of the internal Wagyu library, or `NULL` if Wagyu support is not enabled.

□□

Code

```
SELECT PostGIS_Wagyu_Version();
```

Output

```
0.5.0 (Internal)
```

□□

PostGIS_Full_Version, PostGIS_GEOS_Version, PostGIS_PROJ_Version, PostGIS_Lib_Version, PostGIS_LibXM
PostGIS_Version

7.21.13 PostGIS_Scripts_Build_Date

PostGIS_Scripts_Build_Date — Returns build date of the PostGIS scripts.

□□

text **PostGIS_Scripts_Build_Date**();

□□

Returns build date of the PostGIS scripts.
1.0.0RC1 □□□□□□□□□□□□.

□□

Code

```
SELECT PostGIS_Scripts_Build_Date();
```

Output

```
2023-06-22 03:56:11
```

□□

PostGIS_Full_Version, PostGIS_GEOS_Version, PostGIS_Lib_Version, PostGIS_LibXML_Version, PostGIS_Version

7.21.14 PostGIS_Scripts_Installed

PostGIS_Scripts_Installed — Returns version of the PostGIS scripts installed in this database.

□□

text **PostGIS_Scripts_Installed**();

□□

Returns version of the PostGIS scripts installed in this database.



□□
If the output of this function doesn't match the output of PostGIS_Scripts_Released you probably missed to properly upgrade an existing database. See the Upgrading section for more info.

Availability: 0.9.0

□□

Code

```
SELECT PostGIS_Scripts_Installed();
```

Output

```
3.4.0dev 3.3.0rc2-993-g61bdf43a7
```

□□

PostGIS_Full_Version, PostGIS_Scripts_Released, PostGIS_Version

7.21.15 PostGIS_Scripts_Released

PostGIS_Scripts_Released — Returns the version number of the postgis.sql script released with the installed PostGIS lib.

□□

text **PostGIS_Scripts_Released**();

□□

Returns the version number of the postgis.sql script released with the installed PostGIS lib.



□□
Starting with version 1.1.0 this function returns the same value of PostGIS_Lib_Version. Kept for backward compatibility.

Availability: 0.9.0

□□

Code

```
SELECT PostGIS_Scripts_Released();
```

Output

```
3.4.0dev 3.3.0rc2-993-g61bdf43a7
```

□□

PostGIS_Full_Version, PostGIS_Scripts_Installed, PostGIS_Lib_Version

7.21.16 PostGIS_Version

PostGIS_Version — Returns PostGIS version number and compile-time options.

⊠⊠

text **PostGIS_Version**();

⊠⊠

Returns PostGIS version number and compile-time options.

⊠⊠

Code

```
SELECT PostGIS_Version();
```

Output

```
3.4 USE_GEOS=1 USE_PROJ=1 USE_STATS=1
```

⊠⊠

PostGIS_Full_Version, PostGIS_GEOS_Version, PostGIS_Lib_Version, PostGIS_LibXML_Version, PostGIS_PROJ_Version

7.22 PostGIS GUC(Grand Unified Custom Variable)

7.22.1 postgis.gdal_datapath

postgis.gdal_datapath — GDAL ⊠ GDAL_DATA ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠, ⊠ ⊠⊠⊠⊠⊠⊠⊠ GDAL_DATA ⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

GDAL ⊠ GDAL_DATA ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ PostgreSQL GUC ⊠⊠⊠⊠⊠. `postgis.gdal_datapath` ⊠ ⊠ GDAL ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠⊠⊠⊠⊠ GDAL ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (hard-coded) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠. GDAL ⊠⊠⊠⊠⊠⊠⊠ GDAL ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

> **⊠⊠**
> PostgreSQL ⊠⊠⊠⊠⊠ postgresql.conf ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

2.2.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

> **⊠⊠**
> GDAL ⊠ ⊠⊠⊠⊠ ⊠⊠⊠⊠⊠ GDAL_DATA ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

`postgis.gdal_datapath` ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

Code

```
SET postgis.gdal_datapath TO '/usr/local/share/gdal.hidden';
SET postgis.gdal_datapath TO default;
```

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

Code

```
ALTER DATABASE gisdb
SET postgis.gdal_datapath = 'C:/Program Files/PostgreSQL/9.3/gdal-data';
```

⊠⊠

PostGIS_GDAL_Version, ST_Transform

7.22.2 postgis.gdal_enabled_drivers

postgis.gdal_enabled_drivers — PostGIS ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ GDAL ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. GDAL ⊠⊠⊠⊠ GDAL_SKIP ⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

PostGIS ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ GDAL ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. GDAL ⊠⊠⊠⊠ GDAL_SKIP ⊠⊠⊠⊠⊠⊠⊠⊠⊠. PostgreSQL ⊠⊠⊠⊠⊠ postgresql.conf ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

PostgreSQL ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ `POSTGIS_GDAL_ENABLED_DRIVERS` ⊠⊠⊠ (pass) ⊠⊠ `postgis.gdal_enabled_drivers` ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ GDAL ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ GDAL ⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

> **⊠⊠**
>
> `postgis.gdal_enabled_drivers` ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
>
> - `DISABLE_ALL` ⊠⊠⊠ GDAL ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. `DISABLE_ALL` ⊠⊠⊠⊠⊠, `postgis.gdal_enabled_drivers` ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
> - `ENABLE_ALL` ⊠⊠⊠ GDAL ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
> - `VSICURL` ⊠ GDAL ⊠ `/vsicurl/` ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
>
> `postgis.gdal_enabled_drivers` ⊠ DISABLE_ALL ⊠⊠⊠⊠⊠⊠⊠, DB ⊠⊠⊠⊠⊠, ST_FromGDALRaster(), ST_AsGDALRaster(), ST_AsTIFF(), ST_AsJPEG() ⊠⊠⊠ ST_AsPNG() ⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

> **⊠⊠**
> ⊠⊠ PostGIS ⊠⊠⊠, `postgis.gdal_enabled_drivers` ⊠ DISABLE_ALL ⊠⊠⊠⊠⊠⊠.

> **⊠⊠**
> GDAL_SKIP ⊠⊠⊠⊠⊠⊠⊠⊠ GDAL ⊠ Configuration Options ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

2.2.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

Set and reset `postgis.gdal_enabled_drivers` for the current session.

Code

```
SET postgis.gdal_enabled_drivers = 'ENABLE_ALL';
SET postgis.gdal_enabled_drivers = default;
```

Set specific drivers for all new connections to a database.

Code

```
ALTER DATABASE mygisdb SET postgis.gdal_enabled_drivers TO 'GTiff PNG JPEG';
```

Set the default for the whole database cluster to enable all drivers. This requires superuser access. Also note that database, session, and user settings override this.

This example writes the setting to postgres.auto.conf.

Code

```
ALTER SYSTEM SET postgis.gdal_enabled_drivers TO 'ENABLE_ALL';
```

Reload the PostgreSQL configuration so the changed setting is applied.

Code

```
SELECT pg_reload_conf();
```

⊠⊠

ST_FromGDALRaster, ST_AsGDALRaster, ST_AsTIFF, ST_AsPNG, ST_AsJPEG, postgis.enable_outdb_raster

7.22.3 postgis.enable_outdb_rasters

postgis.enable_outdb_rasters — DB ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

☒☒

DB ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒. PostgreSQL ☒☒☒☒☒ postgresql.conf ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.

PostgreSQL ☒☒☒☒☒☒☒☒☒☒ 0 ☒☒☒☒☒☒☒☒☒☒☒☒ `POSTGIS_ENABLE_OUTDB_RASTERS` ☒☒☒ (pass) ☒☒ `postgis.enable_outdb_rasters` ☒☒☒☒☒☒☒☒☒☒☒☒☒☒.

> **☒☒**
> `postgis.enable_outdb_rasters` ☒☒☒☒☒☒☒☒☒, GUC `postgis.enable_outdb_rasters` ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.

> **☒☒**
> ☒☒ PostGIS ☒☒☒, `postgis.enable_outdb_rasters` ☒☒☒☒☒☒☒☒☒☒☒.

2.2.0 ☒☒☒☒☒☒☒☒☒☒☒☒☒.

☒☒

Set and reset `postgis.enable_outdb_rasters` for the current session.

Code

```
SET postgis.enable_outdb_rasters TO True;
SET postgis.enable_outdb_rasters = default;
SET postgis.enable_outdb_rasters = True;
SET postgis.enable_outdb_rasters = False;
```

Set the default for all new connections to a database.

Code

```
ALTER DATABASE gisdb SET postgis.enable_outdb_rasters = true;
```

Set the default for the whole database cluster. This requires superuser access. Also note that database, session, and user settings override this.

This example writes the setting to postgres.auto.conf.

Code

```
ALTER SYSTEM SET postgis.enable_outdb_rasters = true;
```

Reload the PostgreSQL configuration so the changed setting is applied.

Code

```
SELECT pg_reload_conf();
```

☒☒

postgis.gdal_enabled_drivers postgis.gdal_vsi_options

7.22.4 postgis.gdal_vsi_options

postgis.gdal_vsi_options — DB ⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧.

⌧⌧

A string configuration to set options used when working with an out-db raster. Configuration options control things like how much space GDAL allocates to local data cache, whether to read overviews, and what access keys to use for remote out-db data sources.

Availability: 3.2.0

⌧⌧

`postgis.enable_outdb_rasters` ⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧.

Code

```
SET postgis.gdal_vsi_options = 'AWS_ACCESS_KEY_ID=xxxxxxxxxxxxxxx AWS_SECRET_ACCESS_KEY=yyyyyyyyyyyyyyyyyyyyyyyyyyy';
```

Set `postgis.gdal_vsi_options` just for the *current transaction* using the `LOCAL` keyword:

Code

```
SET LOCAL postgis.gdal_vsi_options = 'AWS_ACCESS_KEY_ID=xxxxxxxxxxxxxxx AWS_SECRET_ACCESS_KEY=yyyyyyyyyyyyyyyyyyyyyyyyyyy';
```

⌧⌧

postgis.enable_outdb_rasters postgis.gdal_enabled_drivers

7.22.5 postgis.gdal_cpl_debug

postgis.gdal_cpl_debug — A boolean configuration to turn logging of GDAL debug messages on and off.

⌧⌧

By default, GDAL logging is printed to stderr, and lower level debug messages are not printed at all. Turning this GUC to true will cause detailed GDAL logging to be sent into the PostgreSQL server log. PostgreSQL clients receive only generic GDAL notice, debug, warning, or error messages.

Availability: 3.6.0

⌧⌧

postgis.enable_outdb_rasters postgis.gdal_enabled_drivers

7.23 Troubleshooting Functions

7.23.1 PostGIS_AddBBox

PostGIS_AddBBox — ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

geometry **PostGIS_AddBBox**(geometry geomA);

⊠⊠

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

> **⊠⊠**
> ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠.

This method supports Circular Strings and Curves.

⊠⊠

Code

```
UPDATE sometable
SET geom =  PostGIS_AddBBox(geom)
WHERE PostGIS_HasBBox(geom) = false;
```

⊠⊠

PostGIS_DropBBox, PostGIS_HasBBox

7.23.2 PostGIS_DropBBox

PostGIS_DropBBox — ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

geometry **PostGIS_DropBBox**(geometry geomA);

[illegible]

[illegible]. [illegible], [illegible]. [illegible]. [illegible], [illegible] ST_Intersects [illegible].

[illegible]

[illegible]. [illegible], [illegible], [illegible]. 8.3 [illegible] 8.3.6 [illegible], [illegible]/[illegible]. [illegible], [illegible]/[illegible].

This method supports Circular Strings and Curves.

[illegible]

This example drops cached bounding boxes where the stored box is wrong. Applying `Box2D` to `ST_AsBinary` forces a fresh box calculation, while `Box2D` on the table geometry reads the cached box.

Code

```
UPDATE sometable
SET geom =  PostGIS_DropBBox(geom)
WHERE Not (Box2D(ST_AsBinary(geom)) = Box2D(geom));
```

Output

```
UPDATE ...
```

After dropping corrupt cached boxes, re-add them where needed.

Code

```
UPDATE sometable
SET geom =  PostGIS_AddBBox(geom)
WHERE Not PostGIS_HasBBOX(geom);
```

Output

```
UPDATE ...
```

[illegible]

PostGIS_AddBBox, PostGIS_HasBBox, Box2D

7.23.3 PostGIS_HasBBox

PostGIS_HasBBox — [illegible], [illegible].

□□

boolean **PostGIS_HasBBox**(geometry geomA);

□□

□□□□□□□□□□□□□□□□□□□□□□, □□□□□□□□□□□□□□□. □□□□□□□□□□ PostGIS_AddBBox □ PostGIS_DropBBox □□□□□□□.

This method supports Circular Strings and Curves.

□□

Code

```
SELECT geom
FROM sometable WHERE PostGIS_HasBBox(geom) = false;
```

□□

PostGIS_AddBBox, PostGIS_DropBBox

Chapter 8

SFCGAL Functions Reference

SFCGAL ⊠⊠⊠ 2D ⊠ 3D ⊠⊠⊠⊠⊠⊠⊠ CGAL ⊠⊠⊠⊠ C++ ⊠⊠ (wrapper) ⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠ ⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

SFCGAL ⊠⊠⊠⊠ http://www.sfcgal.org ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠ postgis_sfcgal ⊠⊠⊠⊠⊠⊠⊠.

8.1 SFCGAL Management Functions

8.1.1 postgis_sfcgal_version

postgis_sfcgal_version — ⊠⊠⊠⊠ SFCGAL ⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

text **postgis_sfcgal_version**(void);

⊠⊠

⊠⊠⊠⊠ SFCGAL ⊠⊠⊠⊠⊠⊠⊠⊠⊠.

2.1.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

This method needs SFCGAL backend.

⊠⊠

postgis_sfcgal_full_version

8.1.2 postgis_sfcgal_full_version

postgis_sfcgal_full_version — Returns the full version of SFCGAL in use including CGAL and Boost versions

□□

text **postgis_sfcgal_version**(void);

□□

Returns the full version of SFCGAL in use including CGAL and Boost versions

Availability: 3.3.0

This method needs SFCGAL backend.

□□

postgis_sfcgal_version

8.2 SFCGAL Accessors and Setters

8.2.1 CG_ForceLHR

CG_ForceLHR — LHR(Left Hand Reverse; □□□□) □□□□□□□□□.

□□

geometry **CG_ForceLHR**(geometry geom);

□□

Returns an equivalent geometry whose polygonal components follow the left-hand rule: exterior rings are counter-clockwise and interior rings are clockwise. Rings are reversed as needed; coordinates and dimensionality are preserved. The input must be valid.

Availability: 3.5.0

This method needs SFCGAL backend.

This function supports 3d and will not drop the z-index.

This function supports Polyhedral surfaces.

This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).

□□

Reorient a polygon and its hole to the left-hand rule.

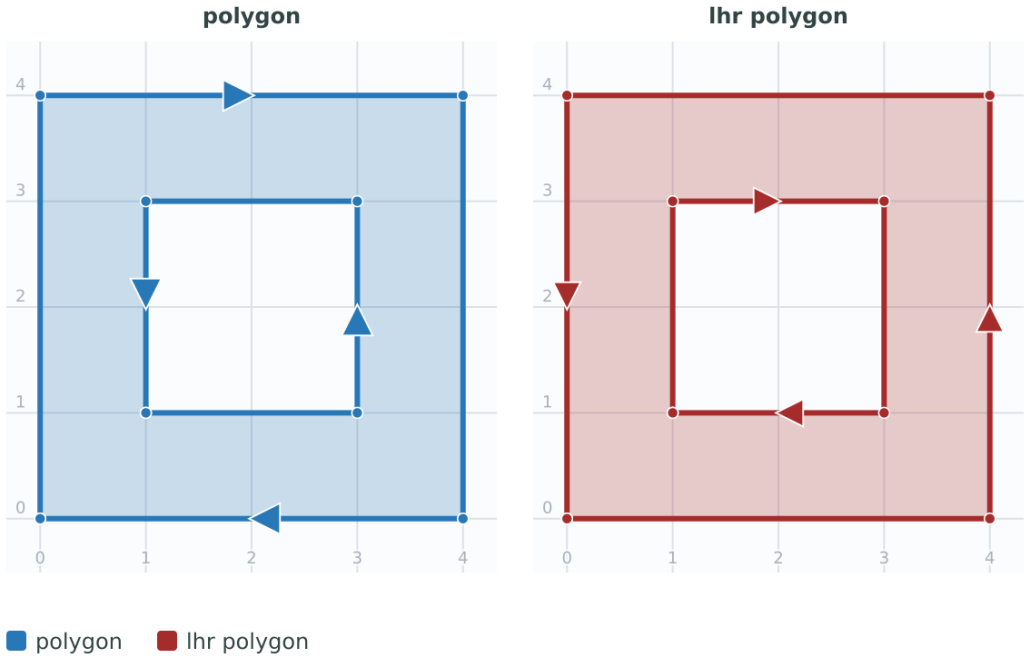
Code

```
WITH data AS (
  SELECT 'POLYGON((0 0,0 4,4 4,4 0,0 0),
                  (1 1,3 1,3 3,1 3,1 1))'::geometry AS geom
)
SELECT geom AS input_polygon,
       CG_ForceLHR(geom) AS lhr_polygon,
       ST_IsPolygonCCW(geom) AS input_is_lhr,
       ST_IsPolygonCCW(CG_ForceLHR(geom)) AS output_is_lhr
FROM data;
```

Output

```
POLYGON((0 0,0 4,4 4,4 0,0 0),(1 1,3 1,3 3,1 3,1 1)) | POLYGON((0 0,4 0,4 4,0 4,0 0),(1 1,1 3,3 3,3 1,1 1)) | f | t
```

Figure



8.2.2 CG_IsPlanar

CG_IsPlanar — ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.

☒☒

boolean **CG_IsPlanar**(geometry geom);

☒☒

Tests whether the input geometry lies in a single plane in 3D.

Availability: 3.5.0

This method needs SFCGAL backend.

This function supports 3d and will not drop the z-index.

This function supports Polyhedral surfaces.

This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).

8.2.3 CG_IsValid

CG_IsValid — Tests whether a geometry is valid according to SFCGAL

☒☒

boolean **CG_IsValid**(geometry geom);

☒☒

Returns true if the input geometry is valid according to SFCGAL.
This differs from ST_IsValid, which checks OGC validity through GEOS.
Availability: 3.8.0

This method needs SFCGAL backend.

This function supports 3d and will not drop the z-index.

This function supports Polyhedral surfaces.

This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).

☒☒

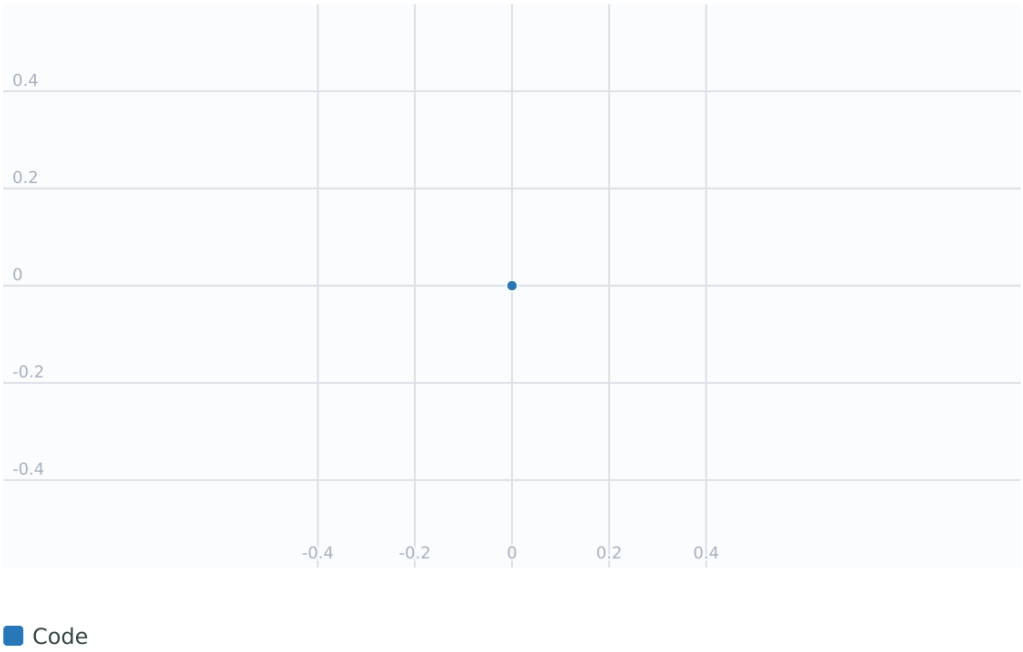
Code

```
SELECT CG_IsValid('LINESTRING Z (0 0 0,0 0 1)'::geometry);
```

Output

```
t
```

Figure



□□

CG_IsValidDetail, CG_IsSimple, ST_IsValid, CG_IsSolid

8.2.4 CG_IsValidDetail

CG_IsValidDetail — Returns a `valid_detail` row stating whether a geometry is valid according to SFCGAL, and if not a reason and a location.

□□

valid_detail **CG_IsValidDetail**(geometry geom);

□□

Returns a `valid_detail` row with a boolean validity flag, a reason for invalidity, and the invalidity location when SFCGAL reports one.

This differs from ST_IsValidDetail, which checks OGC validity through GEOS.

Availability: 3.8.0

This method needs SFCGAL backend.

This function supports 3d and will not drop the z-index.

This function supports Polyhedral surfaces.

This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).

□□

Code

```
SELECT valid, reason, ST_AsText(location) AS location
FROM CG_IsValidDetail('POLYGON((0 0,2 2,2 0,0 2,0 0))'::geometry);
```

Output

```
f | ring 0 self intersects |
```

□□

CG_IsValid, CG_IsSimpleDetail, ST_IsValidDetail

8.2.5 CG_IsSimple

CG_IsSimple — Tests whether a geometry is simple according to SFCGAL

□□

boolean **CG_IsSimple**(geometry geom);

□□

Returns true if the input geometry is simple according to SFCGAL.
This differs from ST_IsSimple, which checks simplicity through liblwgeom.
Availability: 3.8.0 - requires SFCGAL >= 2.3.0.

This method needs SFCGAL backend.

This function supports 3d and will not drop the z-index.

This function supports Polyhedral surfaces.

This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).

□□

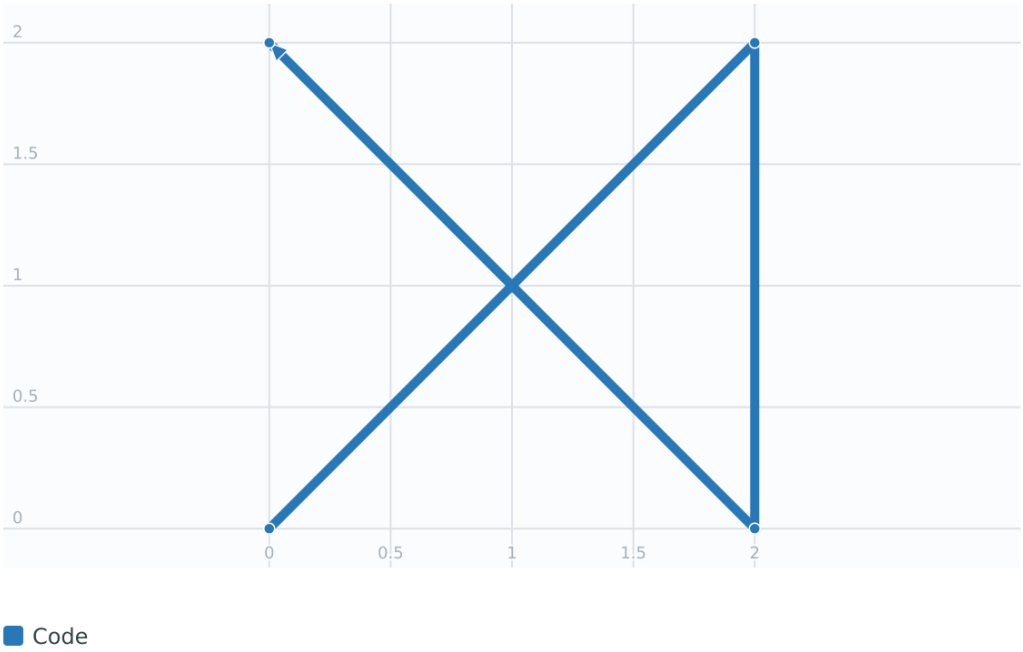
Code

```
SELECT CG_IsSimple('LINESTRING(0 0,2 2,2 0,0 2)'::geometry);
```

Output

```
f
```

Figure



□□

CG_IsSimpleDetail, CG_IsValid, ST_IsSimple

8.2.6 CG_IsSimpleDetail

CG_IsSimpleDetail — Returns a reason when a geometry is not simple according to SFCGAL

□□

text **CG_IsSimpleDetail**(geometry geom);

□□

Returns SFCGAL's reason when the input geometry is not simple. Returns `NULL` for simple geometries.
Availability: 3.8.0 - requires SFCGAL >= 2.3.0.

This method needs SFCGAL backend.

This function supports 3d and will not drop the z-index.

This function supports Polyhedral surfaces.

This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).

□□

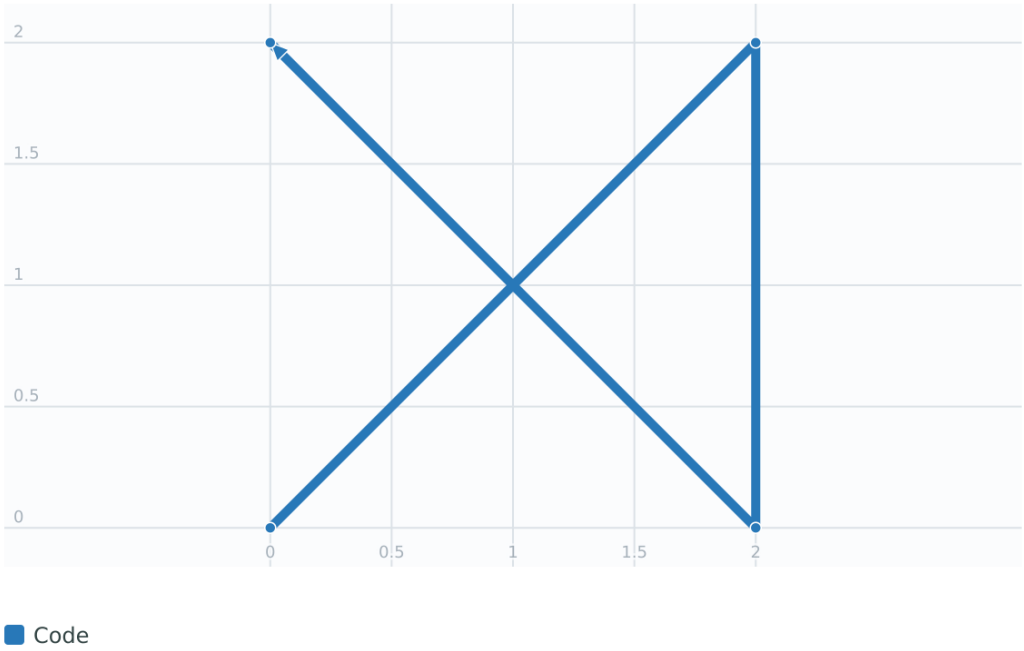
Code

```
SELECT CG_IsSimpleDetail('LINESTRING(0 0,2 2,2 0,0 2)'::geometry);
```

Output

```
linestring self intersects
```

Figure



□□

CG_IsSimple, CG_IsValidDetail, ST_IsSimple

8.2.7 CG_IsSolid

CG_IsSolid — □□□□□□□□□□□□□□. □□□□□□□□□□□□□□□□□□.

□□

boolean **CG_IsSolid**(geometry geom1);

□□

Returns `true` when the input carries the solid flag, such as a geometry produced by CG_MakeSolid. A closed `PolyhedralSurface` remains a surface until it is explicitly converted. This tests the representation, not shell closure or validity; no validity check is performed.

Availability: 3.5.0

This method needs SFCGAL backend.

This function supports 3d and will not drop the z-index.

This function supports Polyhedral surfaces.

This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).

□□

Compare the same closed shell before and after converting it to a solid.

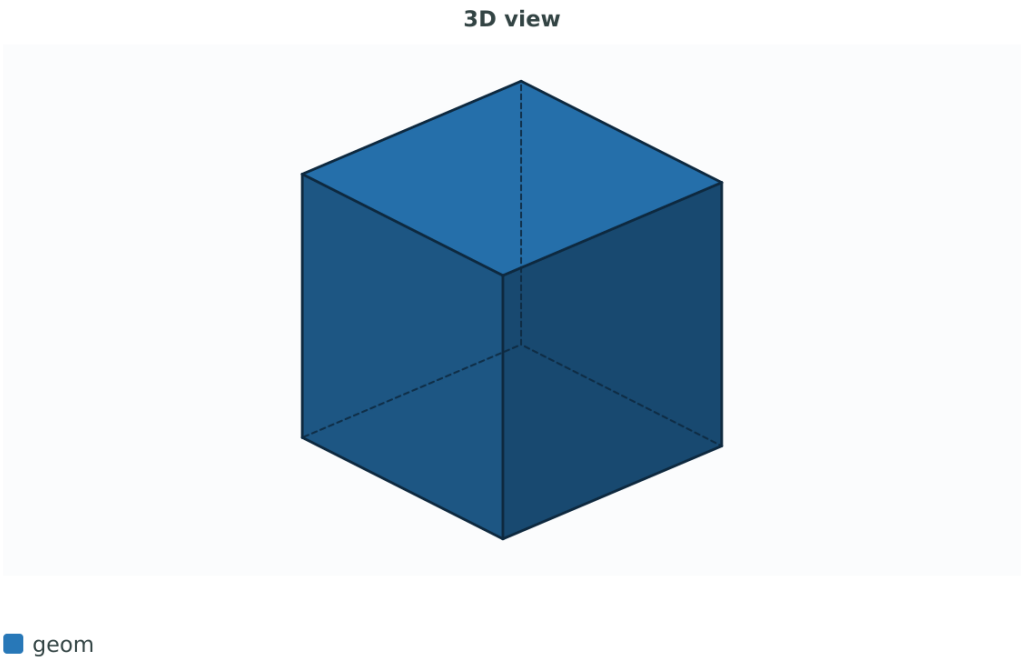
Code

```
WITH data AS (
  SELECT 'POLYHEDRALSURFACE Z (
    ((0 0 0,0 0 1,0 1 1,0 1 0,0 0 0)),
    ((0 0 0,0 1 0,1 1 0,1 0 0,0 0 0)),
    ((0 0 0,1 0 0,1 0 1,0 0 1,0 0 0)),
    ((1 1 0,1 1 1,1 0 1,1 0 0,1 1 0)),
    ((0 1 0,0 1 1,1 1 1,1 1 0,0 1 0)),
    ((0 0 1,1 0 1,1 1 1,0 1 1,0 0 1))
  )'::geometry AS geom
)
SELECT CG_IsSolid(geom) AS shell_is_solid,
       CG_IsSolid(CG_MakeSolid(geom)) AS solid_is_solid
FROM data;
```

Output

```
f | t
```

Figure



8.2.8 CG_MakeSolid

CG_MakeSolid — □□□□□□□□□□□. □□□□□□□□□□□□□□□□. □□□□□□□□□□□, □□□□ □□□□□□□□□□□□□ TIN □□□□□□□□.

□□

geometry **CG_MakeSolid**(geometry geom1);

□□

Marks a PolyhedralSurface or TIN as a solid. The input is expected to describe a closed shell; the function sets the solid representation and does not validate the shell.

Availability: 3.5.0

This method needs SFCGAL backend.

This function supports 3d and will not drop the z-index.

This function supports Polyhedral surfaces.

This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).

8.2.9 CG_Orientation

CG_Orientation — □□□□□ (orientation) □□□□□□.

□□

integer **CG_Orientation**(geometry geom);

□□

□□□□□□□□□□□□□□□. □□□□□□□□□□□□ -1 □, □□□□□□□ 1 □□□□□□□.

Availability: 3.5.0

This method needs SFCGAL backend.

This function supports 3d and will not drop the z-index.

8.2.10 CG_Area

CG_Area — Calculates the area of a geometry

□□

double precision **CG_Area**(geometry geom);

□□

Calculates the area of a geometry.

Performed by the SFCGAL module



□□

NOTE: this function returns a double precision value representing the area.

Availability: 3.5.0



This method needs SFCGAL backend.

□□□□

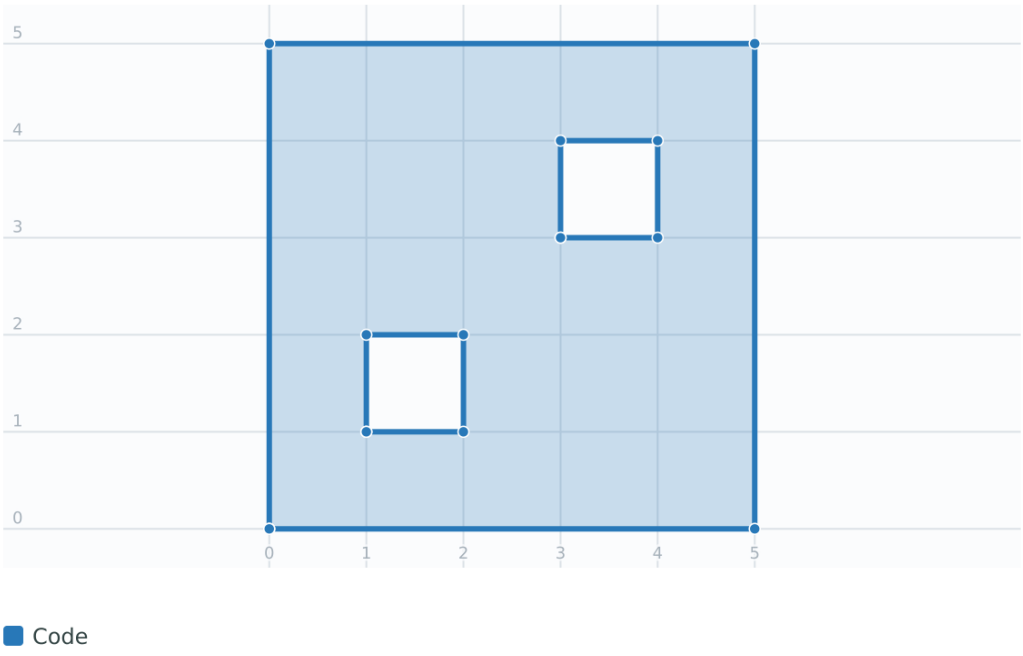
Code

```
SELECT CG_Area('Polygon ((0 0,0 5,5 5,5 0,0 0),(1 1,2 1,2 2,1 2,1 1),(3 3,4 3,4 4,3 4,3 3))
    ');
```

Output

```
23
```

Figure



□□

CG_3DArea, ST_Area

8.2.11 CG_3DArea

CG_3DArea — 3 □□□□□□□□□□□□□□□□. □□□□□ 0 □□□□□□□□□.

□□

float**CG_3DArea**(geometry geom1);

□□

Availability: 3.5.0

This method needs SFCGAL backend.

This method implements the SQL/MM specification. SQL-MM IEC 13249-3: 8.1, 10.5

This function supports 3d and will not drop the z-index.

This function supports Polyhedral surfaces.

This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).

□□

□□: □□□□□ KWT □□□□□□□□□□□□□□□□□□□□□□□□□□□□. □□□□□□□□□□□□□□□□. □□□□□□□□, □□□□□□□□□□□.

Code

```
SELECT CG_3DArea(geom) As cube_surface_area,
CG_3DArea(CG_MakeSolid(geom)) As solid_surface_area
FROM (SELECT 'POLYHEDRALSURFACE( ((0 0 0,0 0 1,0 1 1,0 1 0,0 0 0)),
((0 0 0,0 1 0,1 1 0,1 0 0,0 0 0)),
((0 0 0,1 0 0,1 0 1,0 0 1,0 0 0)),
((1 1 0,1 1 1,1 0 1,1 0 0,1 1 0)),
((0 1 0,0 1 1,1 1 1,1 1 0,0 1 0)),
((0 0 1,1 0 1,1 1 1,0 1 1,0 0 1)) )'::geometry) As f(geom);
```

Output

```
cube_surface_area | solid_surface_area
------------------+--------------------
6                 | 0
```

□□

CG_Area, CG_MakeSolid, CG_IsSolid, CG_Area

8.2.12 CG_Volume

CG_Volume — Computes the volume of a 3D geometry. Closed surfaces can have non-zero volume.

□□

float **CG_Volume**(geometry geom1);

□□

Availability: 3.5.0

This method needs SFCGAL backend.

This function supports 3d and will not drop the z-index.

This function supports Polyhedral surfaces.

This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).

This method implements the SQL/MM specification. SQL-MM IEC 13249-3: 9.1 (same as CG_3DVolume)

Closed surface geometries can have non-zero volume. Use CG_MakeSolid when the geometry value itself must carry the solid flag.

□□

A closed polyhedral surface has volume. CG_MakeSolid makes the solid representation explicit.

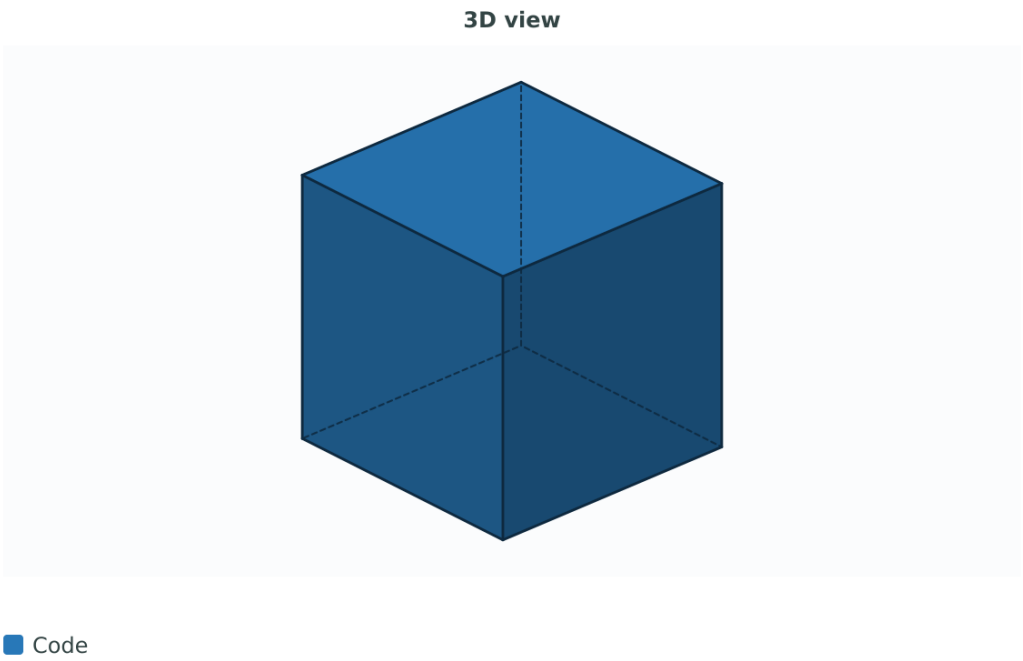
Code

```
SELECT CG_Volume(geom) As cube_surface_vol,
CG_Volume(CG_MakeSolid(geom)) As solid_surface_vol
FROM (SELECT 'POLYHEDRALSURFACE( ((0 0 0,0 0 1,0 1 1,0 1 0,0 0 0)),
((0 0 0,0 1 0,1 1 0,1 0 0,0 0 0)),
((0 0 0,1 0 0,1 0 1,0 0 1,0 0 0)),
((1 1 0,1 1 1,1 0 1,1 0 0,1 1 0)),
((0 1 0,0 1 1,1 1 1,1 1 0,0 1 0)),
((0 0 1,1 0 1,1 1 1,0 1 1,0 0 1)) )'::geometry) As f(geom);
```

Output

```
cube_surface_vol | solid_surface_vol
-----------------+------------------
1                | 1
```

Figure



□□

CG_3DArea, CG_MakeSolid, CG_IsSolid

8.3 SFCGAL Processing and Relationship Functions

8.3.1 CG_Intersection

CG_Intersection — Computes the intersection of two geometries

□□

geometry **CG_Intersection**(geometry geomA , geometry geomB);

□□

Computes the intersection of two geometries.
Performed by the SFCGAL module

□□
NOTE: this function returns a geometry representing the intersection.

Availability: 3.5.0

This method needs SFCGAL backend.

□□□□

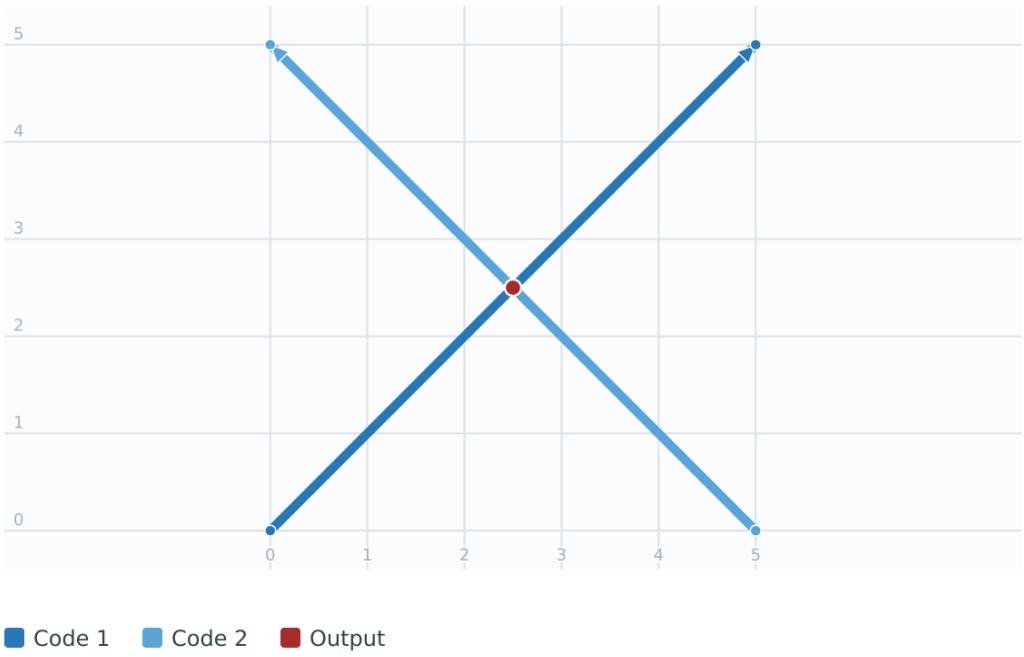
Code

```
SELECT CG_Intersection('LINESTRING(0 0,5 5)', 'LINESTRING(5 0,0 5)');
```

Output

```
POINT(2.5 2.5)
```

Figure



□□

CG_3DIntersection, ST_Intersection

8.3.2 CG_Intersects

CG_Intersects — Tests if two geometries intersect (they have at least one point in common)

□□

boolean **CG_Intersects**(geometry geomA , geometry geomB);

□□

Returns `true` if two geometries intersect. Geometries intersect if they have any point in common. Performed by the SFCGAL module



□□
NOTE: this is the ”allowable” version that returns a boolean, not an integer.

Availability: 3.5.0


This method needs SFCGAL backend.


This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).

□□□□

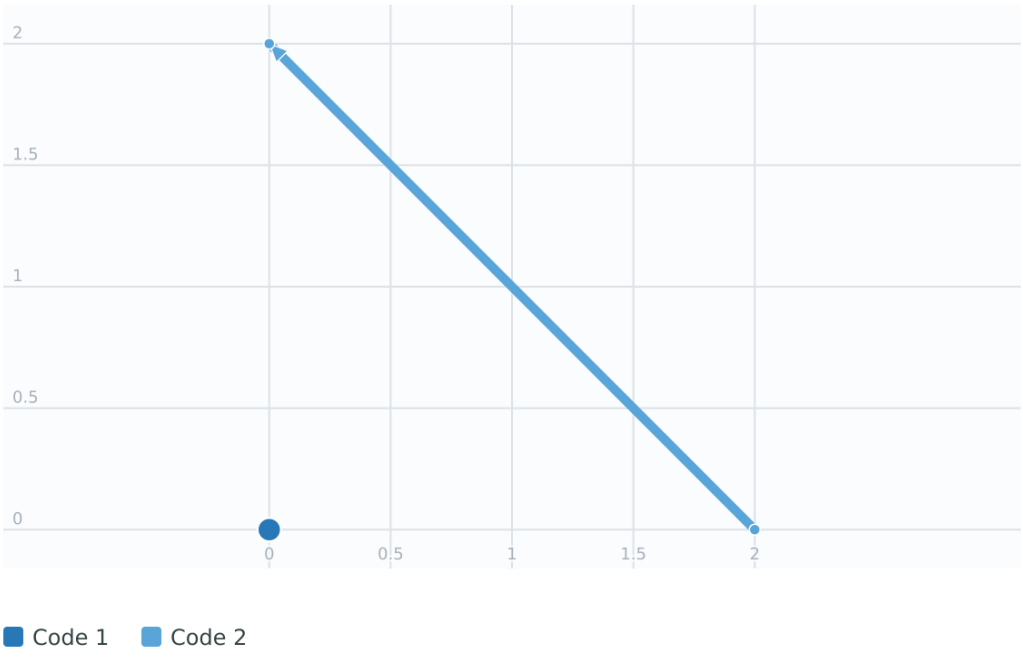
Code

```
SELECT CG_Intersects('POINT(0 0)'::geometry, 'LINESTRING ( 2 0,0 2 )'::geometry);
```

Output

```
f
```

Figure



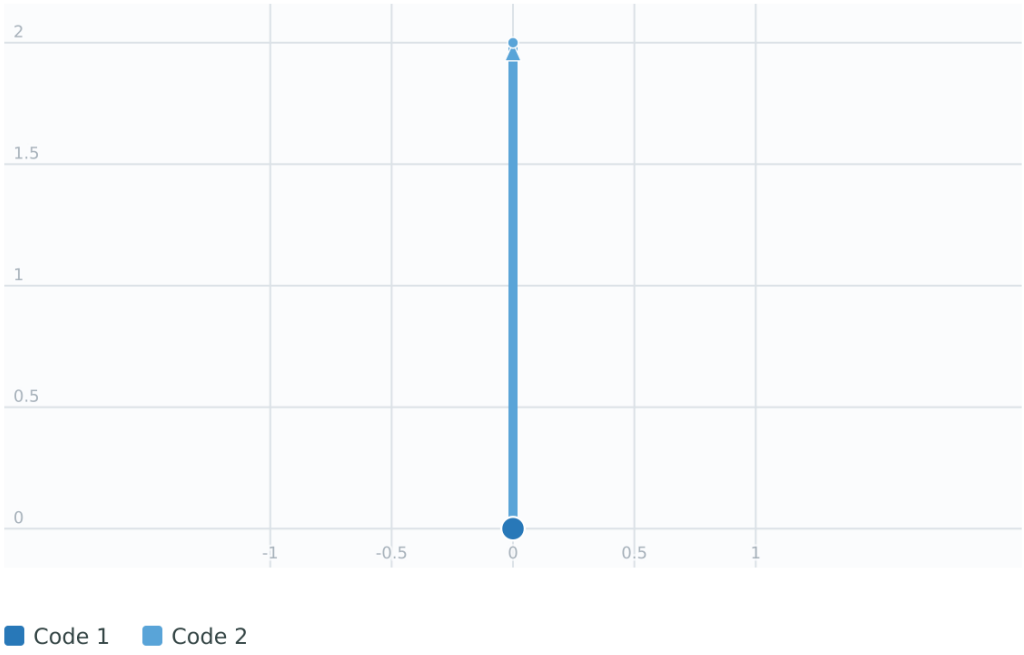
Code

```
SELECT CG_Intersects('POINT(0 0)'::geometry, 'LINESTRING ( 0 0,0 2 )'::geometry);
```

Output

```
t
```

Figure



☒☒

CG_3DIntersects, ST_3DIntersects, ST_Intersects, ST_Disjoint

8.3.3 CG_3DIntersects

CG_3DIntersects — Tests if two 3D geometries intersect

☒☒

boolean **CG_3DIntersects**(geometry geomA , geometry geomB);

☒☒

Tests if two 3D geometries intersect. 3D geometries intersect if they have any point in common in the three-dimensional space.

Performed by the SFCGAL module



☒☒
NOTE: this is the ”allowable” version that returns a boolean, not an integer.

Availability: 3.5.0


This method needs SFCGAL backend.


This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).

□□□□

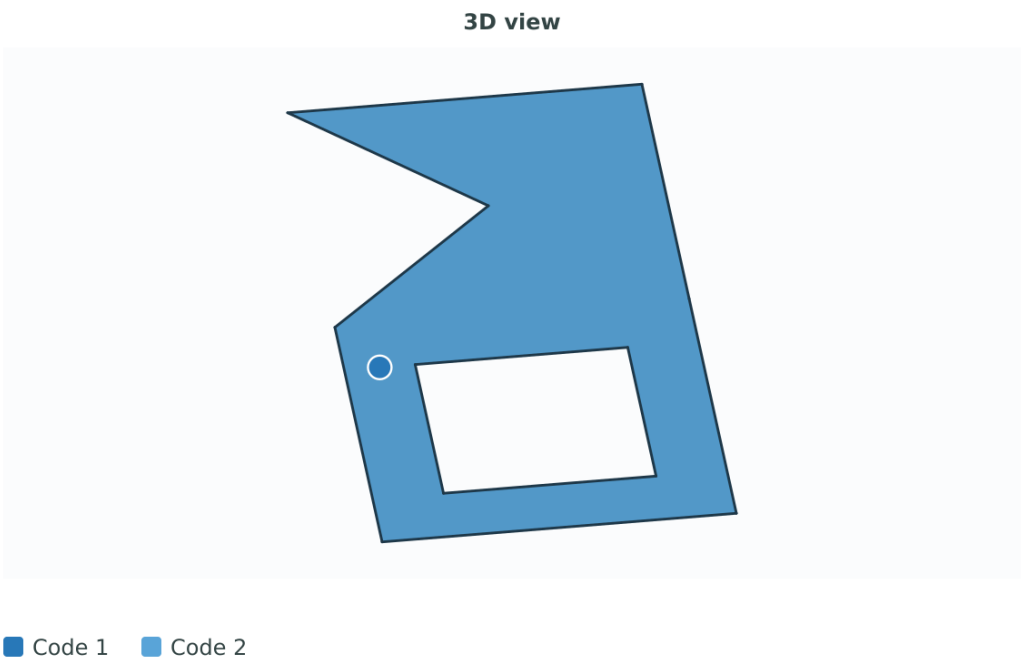
Code

```
SELECT CG_3DIntersects('POINT(1.2 0.1 0)', 'POLYHEDRALSURFACE(((0 0 0,0.5 0.5 0,1 0 0,1 1 0,0 1 0,0 0 0)),((1 0 0,2 0 0,2 1 0,1 1 0,1 0 0),(1.2 0.2 0,1.2 0.8 0,1.8 0.8 0,1.8 0.2 0,1.2 0.2 0)))');
```

Output

```
t
```

Figure



□□

CG_Intersects, ST_3DIntersects, ST_Intersects, ST_Disjoint

8.3.4 CG_Difference

CG_Difference — Computes the geometric difference between two geometries

□□

geometry **CG_Difference**(geometry geomA , geometry geomB);

□□

Computes the geometric difference between two geometries. The resulting geometry is a set of points that are present in geomA but not in geomB.

Performed by the SFCGAL module



□□
NOTE: this function returns a geometry.

Availability: 3.5.0

This method needs SFCGAL backend.

This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).

□□□□

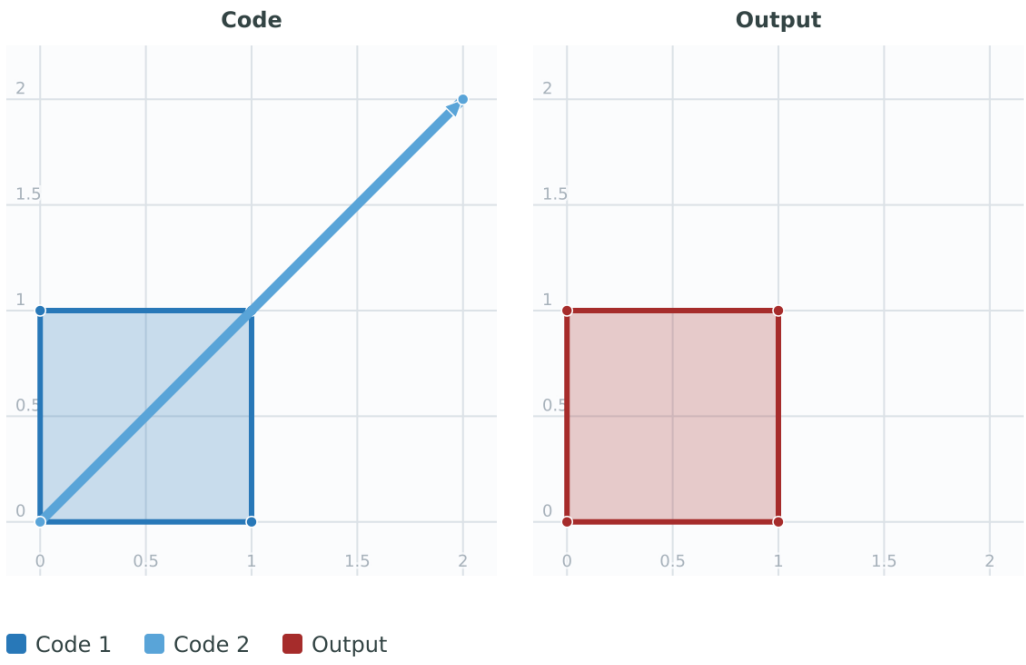
Code

```
SELECT CG_Difference('POLYGON((0 0,0 1,1 1,1 0,0 0))'::geometry, 'LINESTRING(0 0,2 2)':: ↩
    geometry);
```

Output

```
POLYGON((0 0,1 0,1 1,0 1,0 0))
```

Figure



□□

CG_3DDifference, ST_Difference

8.3.5 CG_3DDifference

CG_3DDifference — 3 □□□□□□□□□□.

□□

geometry **CG_3DDifference**(geometry geom1, geometry geom2);

□□

geom2 □□□□□□□ geom1 □□□□□□□□□.

Availability: 3.5.0

This method needs SFCGAL backend.

This method implements the SQL/MM specification. SQL-MM IEC 13249-3: 5.1

This function supports 3d and will not drop the z-index.

This function supports Polyhedral surfaces.

This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).

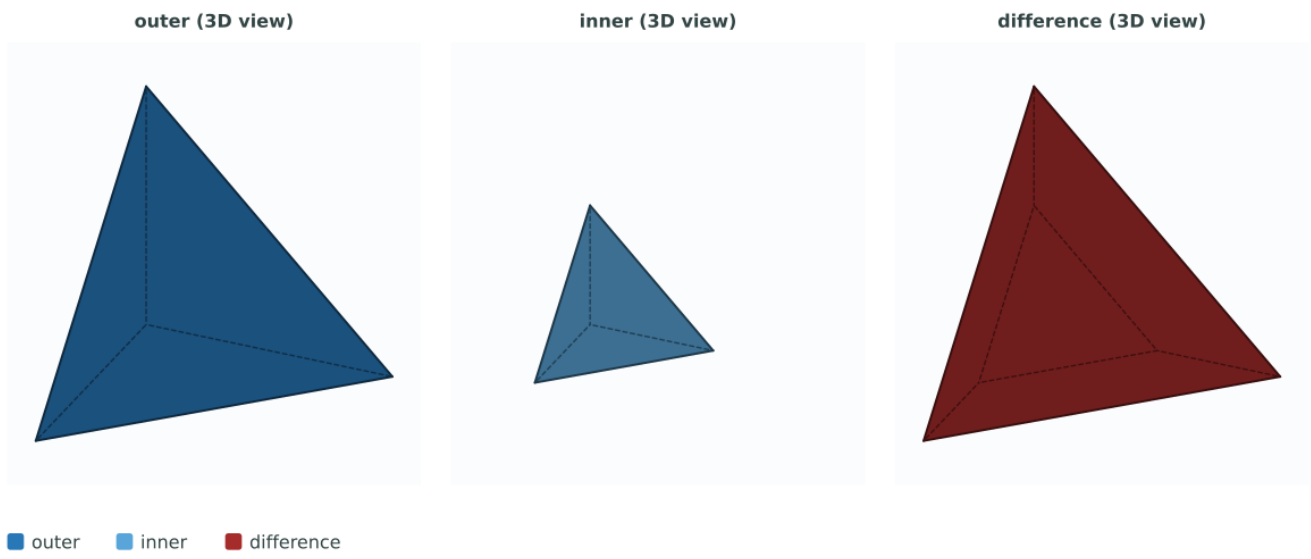
□□

Subtract a smaller tetrahedral solid and render the two inputs and the actual remaining solid separately.

Code

```
WITH data AS (
  SELECT CG_MakeSolid('POLYHEDRALSURFACE Z (
    ((0 0 0,0 4 0,4 0 0,0 0 0)),
    ((0 0 0,4 0 0,0 0 4,0 0 0)),
    ((0 0 0,0 0 4,0 4 0,0 0 0)),
    ((4 0 0,0 4 0,0 0 4,4 0 0))
  )'::geometry) AS outer_solid
), solids AS (
  SELECT outer_solid,
         ST_Scale(outer_solid, 0.5, 0.5, 0.5) AS inner_solid
  FROM data
)
SELECT outer_solid AS input_outer,
       inner_solid AS input_inner,
       CG_3DDifference(outer_solid, inner_solid) AS difference
FROM solids;
```

Figure



□□

CG_Extrude, ST_AsX3D, CG_3DIntersection CG_3DUnion

8.3.6 CG_Distance

CG_Distance — Computes the minimum distance between two geometries

□□

double precision **CG_Distance**(geometry geomA , geometry geomB);

□□

Computes the minimum distance between two geometries.
Performed by the SFCGAL module



□□
NOTE: this function returns a double precision value representing the distance.

Availability: 3.5.0

This method needs SFCGAL backend.

This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).

□□□□

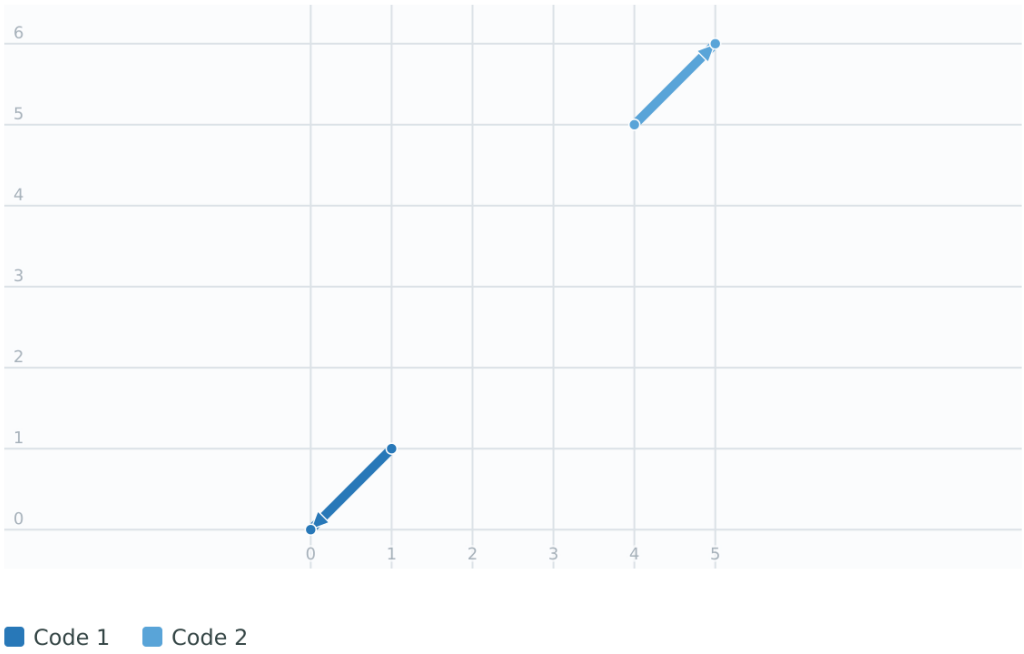
Code

```
SELECT CG_Distance('LINESTRING(0.0 0.0,-1.0 -1.0)', 'LINESTRING(3.0 4.0,4.0 5.0)');
```

Output

```
5
```

Figure



□□

CG_3DDistance, CG_Distance

8.3.7 CG_3DDistance

CG_3DDistance — Computes the minimum 3D distance between two geometries

□□

double precision **CG_3DDistance**(geometry geomA , geometry geomB);

□□

Computes the minimum 3D distance between two geometries.
Performed by the SFCGAL module



□□
NOTE: this function returns a double precision value representing the 3D distance.

Availability: 3.5.0

This method needs SFCGAL backend.

This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).

□□□□

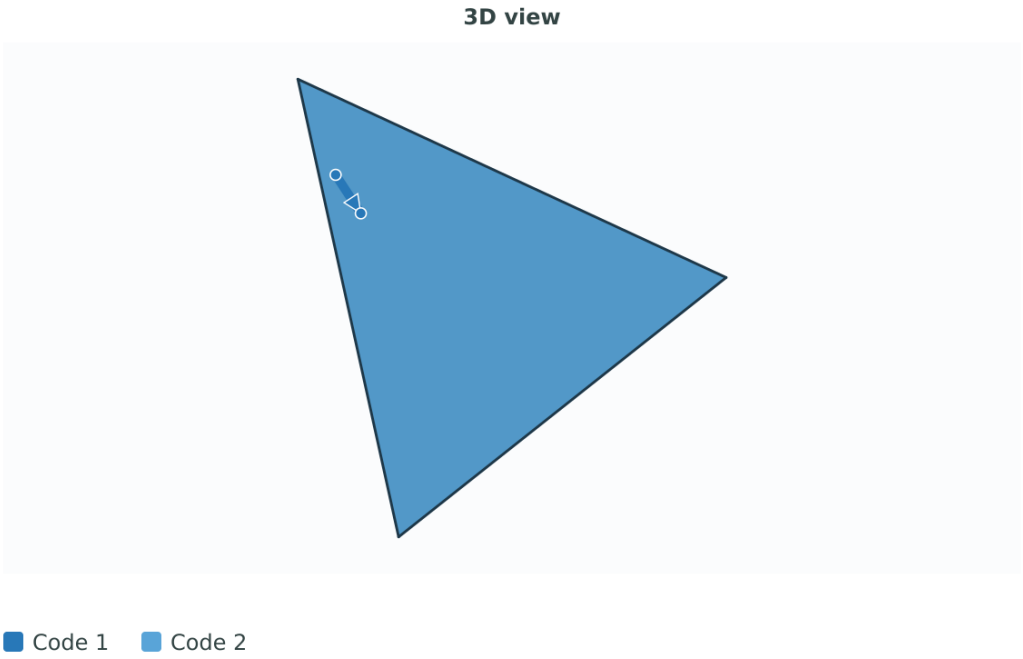
Code

```
SELECT CG_3DDistance('LINESTRING(-1.0 0.0 2.0,1.0 0.0 3.0)', 'TRIANGLE((-4.0 0.0 1.0,4.0 ↩
    0.0 1.0,0.0 4.0 1.0,-4.0 0.0 1.0))');
```

Output

```
1
```

Figure



□□

CG_Distance, ST_3DDistance

8.3.8 CG_3DConvexHull

CG_3DConvexHull — □□□□□□□□□□□□□□□□.

□□

geometry **CG_3DConvexHull**(geometry geom1);

□□

Computes the smallest convex three-dimensional geometry that contains the input. Unlike ST_ConvexHull, this function uses Z coordinates when constructing the hull.

Availability: 3.5.0

This method needs SFCGAL backend.

This function supports 3d and will not drop the z-index.

This function supports Polyhedral surfaces.

This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).

□□

Compute the 3D convex hull of a 3D LineString.

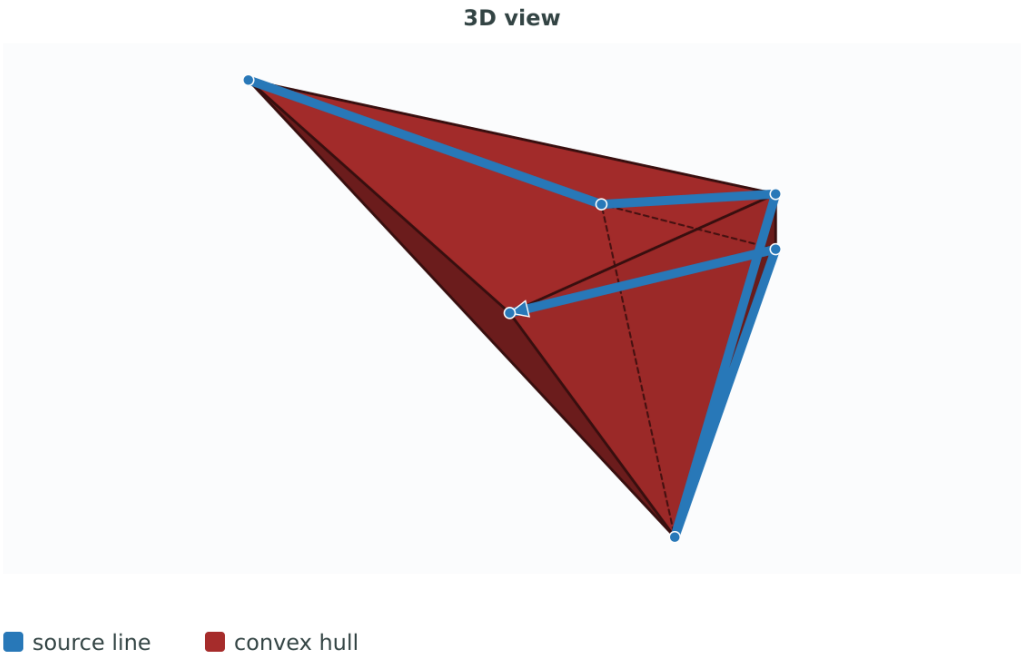
Code

```
WITH data AS (
  SELECT 'LINESTRING Z(0 0 5,1 5 3,5 7 6,9 5 3,5 7 5,6 3 5)'::geometry AS source_line
)
SELECT CG_3DConvexHull(source_line) AS convex_hull
FROM data;
```

Output

```
POLYHEDRALSURFACE Z (((1 5 3,9 5 3,0 0 5,1 5 3)),((1 5 3,0 0 5,5 7 6,1 5 3)),((5 7 6,5 7 5,1 5 3,5 7 6)),((0 0 5,6 3 ↩
    5,5 7 6,0 0 5)),((6 3 5,9 5 3,5 7 6,6 3 5)),((0 0 5,9 5 3,6 3 5,0 0 5)),((9 5 3,5 7 5,5 7 6,9 5 3)),((1 5 3,5 7 ↩
    5,9 5 3,1 5 3)))
```

Figure



Compare the extruded source geometry with its 3D convex hull.

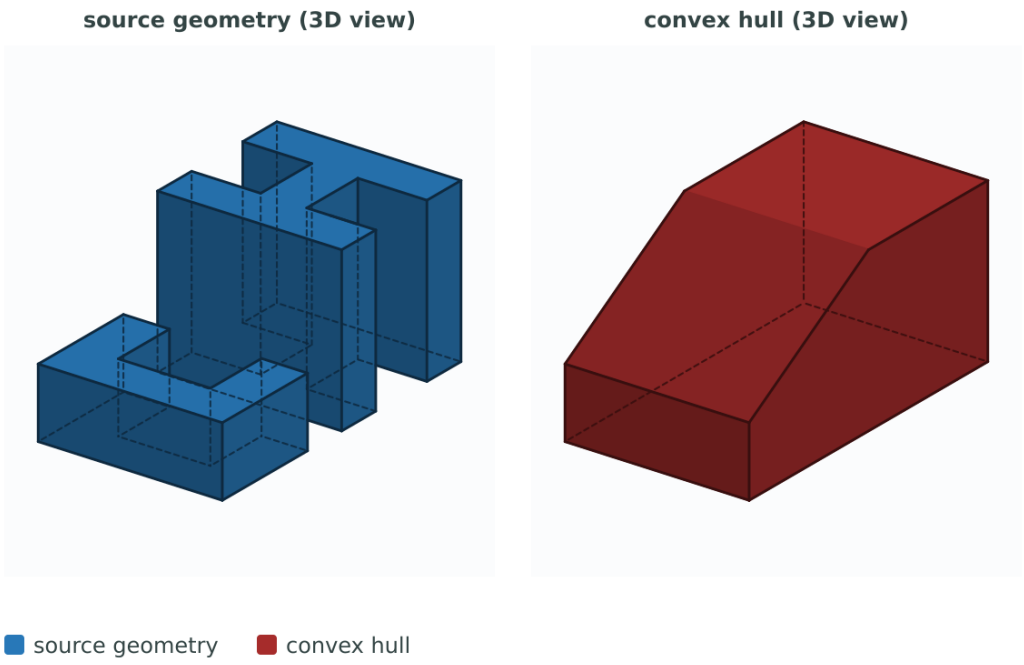
Code

```
WITH footprints(label, geom, height) AS (VALUES
  ('C', 'POLYGON((0 0,0 8,5 8,5 6,2 6,2 2,5 2,5 0,0 0))'::geometry, 3),
  ('H', 'POLYGON((7 0,9 0,9 3,12 3,12 0,14 0,14 8,12 8,12 5,9 5,9 8,7 8,7 0))'::geometry,
    7)
), parts AS (
  SELECT label, CG_Extrude(geom, 0, 0, height) AS geom
  FROM footprints
), shapes AS (
  SELECT ST_Collect(geom ORDER BY label) AS source_geometry
  FROM parts
)
SELECT
  source_geometry AS input_source_geometry,
  CG_3DConvexHull(source_geometry) AS convex_hull
FROM shapes;
```

Output

```
-[ RECORD 1 ]----------
input_source_geometry | GEOMETRYCOLLECTION(POLYHEDRALSURFACE(((0 0 0,0 8 0,5 8 0,5 6 0,2 6 0,2 2 0,5 2 0,5 0 0,0 0 0)) ↩
    ,((0 0 3,5 0 3,5 2 3,2 2 3,2 6 3,5 6 3,5 8 3,0 8 3,0 0 3)),((0 0 0,0 0 3,0 8 3,0 8 0,0 0 0)),((0 8 0,0 8 3,5 8  ↩
    3,5 8 0,0 8 0)),((5 8 0,5 8 3,5 6 3,5 6 0,5 8 0)),((5 6 0,5 6 3,2 6 3,2 6 0,5 6 0)),((2 6 0,2 6 3,2 2 3,2 2 0,2 6 ↩
     0)),((2 2 0,2 2 3,5 2 3,5 2 0,2 2 0)),((5 2 0,5 2 3,5 0 3,5 0 0,5 2 0)),((5 0 0,5 0 3,0 0 3,0 0 0,5 0 0))),  ↩
    POLYHEDRALSURFACE(((7 0 0,7 8 0,9 8 0,9 5 0,12 5 0,12 8 0,14 8 0,14 0 0,12 0 0,12 3 0,9 3 0,9 0 0,7 0 0)),((7 0  ↩
    7,9 0 7,9 3 7,12 3 7,12 0 7,14 0 7,14 8 7,12 8 7,12 5 7,9 5 7,9 8 7,7 8 7,7 0 7)),((7 0 0,7 0 7,7 8 7,7 8 0,7 0  ↩
    0)),((7 8 0,7 8 7,9 8 7,9 8 0,7 8 0)),((9 8 0,9 8 7,9 5 7,9 5 0,9 8 0)),((9 5 0,9 5 7,12 5 7,12 5 0,9 5 0)),((12  ↩
    5 0,12 5 7,12 8 7,12 8 0,12 5 0)),((12 8 0,12 8 7,14 8 7,14 8 0,12 8 0)),((14 8 0,14 8 7,14 0 7,14 0 0,14 8 0)) ↩
    ,((14 0 0,14 0 7,12 0 7,12 0 0,14 0 0)),((12 0 0,12 0 7,12 3 7,12 3 0,12 0 0)),((12 3 0,12 3 7,9 3 7,9 3 0,12 3  ↩
    0)),((9 3 0,9 3 7,9 0 7,9 0 0,9 3 0)),((9 0 0,9 0 7,7 0 7,7 0 0,9 0 0))))
convex_hull           | POLYHEDRALSURFACE(((14 8 7,14 0 0,14 8 0,14 8 7)),((14 0 7,7 8 7,7 0 7,14 0 7)),((0 8 3,7 8  ↩
    7,0 8 0,0 8 3)),((0 0 3,7 8 7,0 8 3,0 0 3)),((0 0 3,0 8 3,0 0 0,0 0 3)),((7 0 7,7 8 7,0 0 3,7 0 7)),((14 0 7,14 0 ↩
     0,14 8 7,14 0 7)),((7 0 7,14 0 0,14 0 7,7 0 7)),((14 8 7,7 8 7,14 0 7,14 8 7)),((0 0 0,0 8 3,0 8 0,0 0 0)),((0 0 ↩
     0,14 0 0,0 0 3,0 0 0)),((0 0 3,14 0 0,7 0 7,0 0 3)),((0 8 0,14 0 0,0 0 0,0 8 0)),((14 8 0,14 0 0,0 8 0,14 8 0)) ↩
    ,((14 8 0,7 8 7,14 8 7,14 8 0)),((0 8 0,7 8 7,14 8 0,0 8 0)))
```

Figure



□□

ST_Letters, ST_AsX3D

8.3.9 CG_3DIntersection

CG_3DIntersection — 3 □□□□□□□□□□.

□□

geometry **CG_3DIntersection**(geometry geom1, geometry geom2);

□□

geom1 □ geom2 □□□□□□□□□□□□□□□□□□□□.

Availability: 3.5.0

This method needs SFCGAL backend.

This method implements the SQL/MM specification. SQL-MM IEC 13249-3: 5.1

This function supports 3d and will not drop the z-index.

This function supports Polyhedral surfaces.

This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).

□□

3D linestrings and polygons can intersect in a 3D segment.

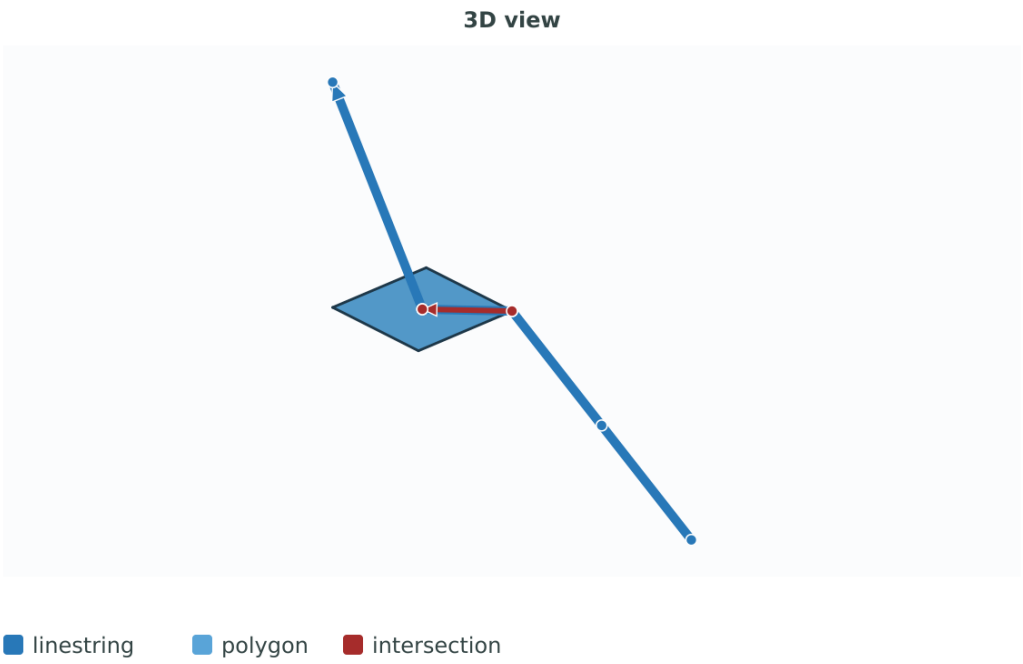
Code

```
WITH data AS (
  SELECT 'LINESTRING Z (2 2 6,1.5 1.5 7,1 1 8,0.5 0.5 8,0 0 10)'::geometry AS  ↩
    input_linestring,
         'POLYGON Z ((0 0 8,0 1 8,1 1 8,1 0 8,0 0 8))'::geometry AS input_polygon
)
SELECT input_linestring AS input_linestring,
       input_polygon AS input_polygon,
       CG_3DIntersection(input_linestring, input_polygon) AS intersection
FROM data;
```

Output

```
LINESTRING Z (2 2 6,1.5 1.5 7,1 1 8,0.5 0.5 8,0 0 10) | POLYGON Z ((0 0 8,0 1 8,1 1 8,1 0 8,0 0 8)) | LINESTRING Z (1  ↩
    1 8,0.5 0.5 8)
```

Figure



Intersect a cube shell with a vertical polygon.

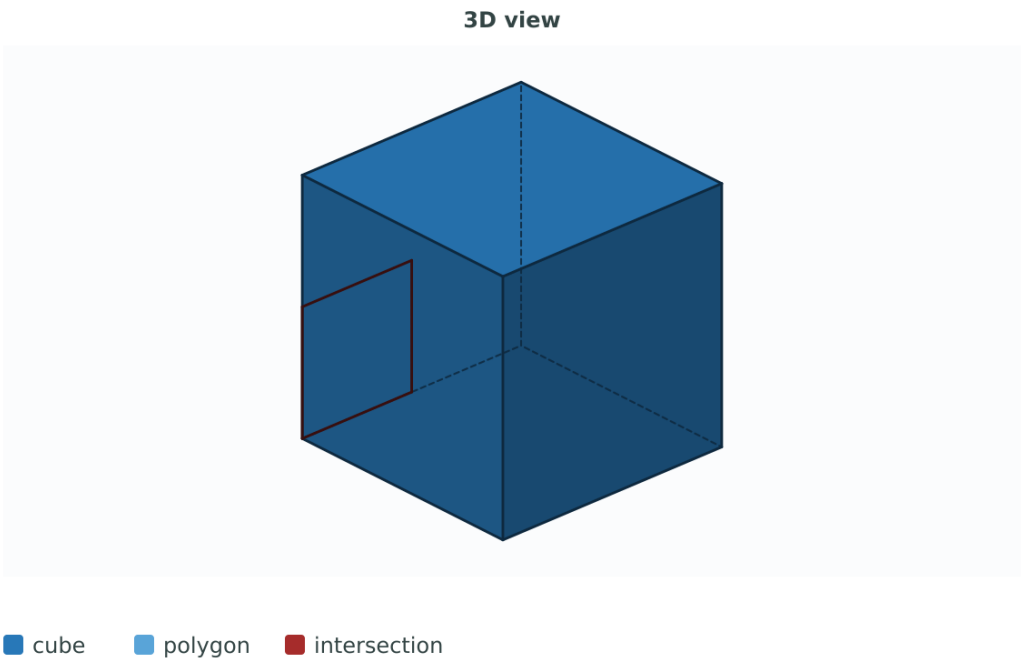
Code

```
WITH data AS (
  SELECT 'POLYHEDRALSURFACE Z(
    ((0 0 0,0 0 1,0 1 1,0 1 0,0 0 0)),
    ((0 0 0,0 1 0,1 1 0,1 0 0,0 0 0)),
    ((0 0 0,1 0 0,1 0 1,0 0 1,0 0 0)),
    ((1 1 0,1 1 1,1 0 1,1 0 0,1 1 0)),
    ((0 1 0,0 1 1,1 1 1,1 1 0,0 1 0)),
    ((0 0 1,1 0 1,1 1 1,0 1 1,0 0 1))
  )'::geometry AS input_cube,
  'POLYGON Z ((0 0 0,0 0 0.5,0 0.5 0.5,0 0.5 0,0 0 0))'::geometry AS input_polygon
)
SELECT input_cube AS input_cube,
       input_polygon AS input_polygon,
       CG_3DIntersection(input_cube, input_polygon) AS intersection
FROM data;
```

Output

```
POLYHEDRALSURFACE Z (((0 0 0,0 0 1,0 1 1,0 1 0,0 0 0)),((0 0 0,0 1 0,1 1 0,1 0 0,0 0 0)),((0 0 0,1 0 0,1 0 1,0 0 1,0 0 ↩
     0)),((1 1 0,1 1 1,1 0 1,1 0 0,1 1 0)),((0 1 0,0 1 1,1 1 1,1 1 0,0 1 0)),((0 0 1,1 0 1,1 1 1,0 1 1,0 0 1))) |  ↩
    POLYGON Z ((0 0 0,0 0 0.5,0 0.5 0.5,0 0.5 0,0 0 0)) | TIN Z (((0 0 0,0 0 0.5,0 0.5 0.5,0 0 0)),((0 0.5 0,0 0 0,0  ↩
    0.5 0.5,0 0.5 0)))
```

Figure



A smaller solid contained by a larger solid remains volumetric after intersection.

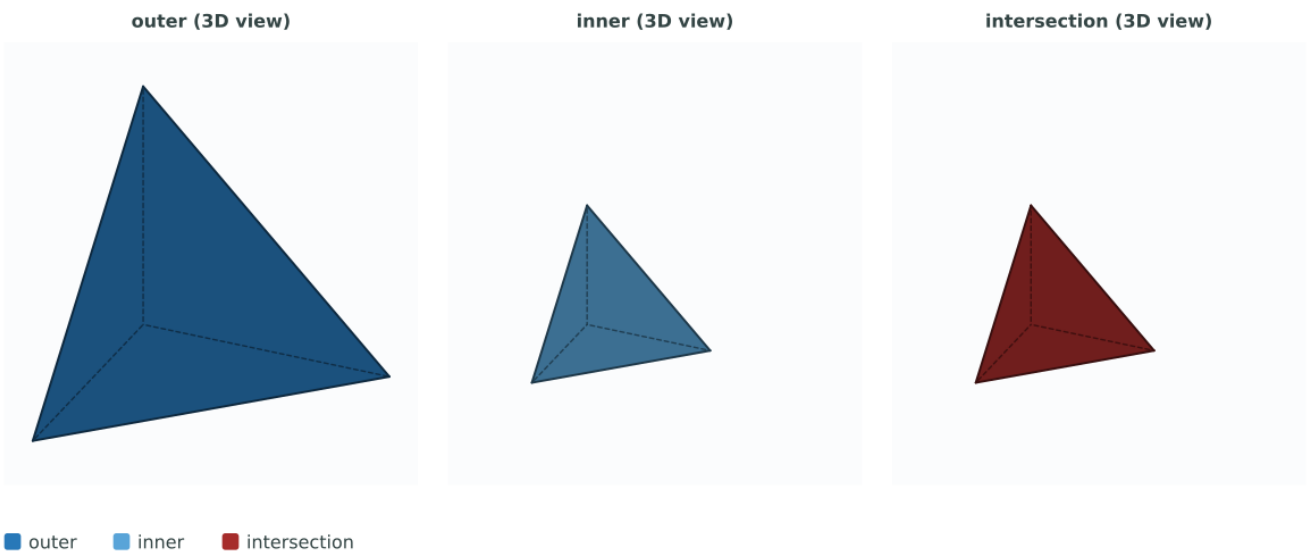
Code

```
WITH data AS (
  SELECT CG_MakeSolid('POLYHEDRALSURFACE Z (
    ((0 0 0,0 4 0,4 0 0,0 0 0)),
    ((0 0 0,4 0 0,0 0 4,0 0 0)),
    ((0 0 0,0 0 4,0 4 0,0 0 0)),
    ((4 0 0,0 4 0,0 0 4,4 0 0))
  )'::geometry) AS outer_solid
), solids AS (
  SELECT outer_solid,
         ST_Scale(outer_solid, 0.5, 0.5, 0.5) AS inner_solid
  FROM data
)
SELECT outer_solid AS input_outer,
       inner_solid AS input_inner,
       CG_3DIntersection(outer_solid, inner_solid) AS intersection
FROM solids;
```

Output

```
POLYHEDRALSURFACE Z (((0 0 0,0 4 0,4 0 0,0 0 0)),((0 0 0,4 0 0,0 0 4,0 0 0)),((0 0 0,0 0 4,0 4 0,0 0 0)),((4 0 0,0 4 ↩
    0,0 0 4,4 0 0))) | POLYHEDRALSURFACE Z (((0 0 0,0 2 0,2 0 0,0 0 0)),((0 0 0,2 0 0,0 0 2,0 0 0)),((0 0 0,0 0 2,0 2 ↩
     0,0 0 0)),((2 0 0,0 2 0,0 0 2,2 0 0))) | POLYHEDRALSURFACE Z (((0 0 0,0 2 0,2 0 0,0 0 0)),((0 0 0,0 0 2,0 2 0,0 ↩
    0 0)),((2 0 0,0 0 2,0 0 0,2 0 0)),((2 0 0,0 2 0,0 0 2,2 0 0)))
```

Figure



8.3.10 CG_Union

CG_Union — Computes the union of two geometries

☒☒

geometry **CG_Union**(geometry geomA , geometry geomB);

☒☒

Computes the union of two geometries.
Performed by the SFCGAL module



☒☒
NOTE: this function returns a geometry representing the union.

Availability: 3.5.0


This method needs SFCGAL backend.

☒☒☒☒

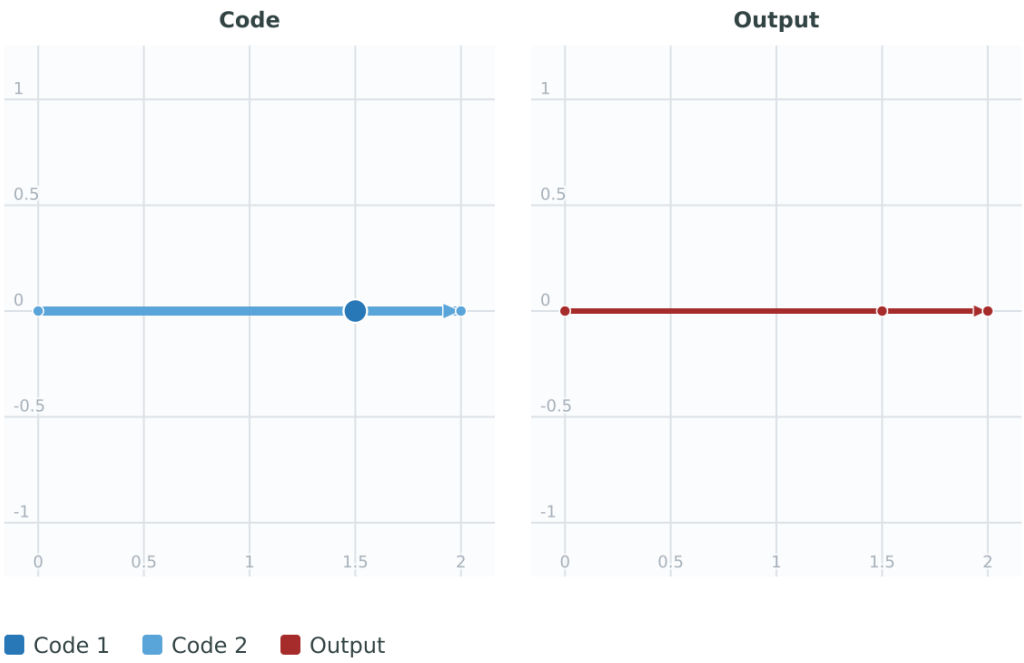
Code

```
SELECT CG_Union('POINT(.5 0)', 'LINESTRING(-1 0,1 0)');
```

Output

```
LINESTRING(-1 0,0.5 0,1 0)
```

Figure



□□

CG_3DUnion, ST_AsBinary

8.3.11 CG_3DUnion

CG_3DUnion — Perform 3D union using postgis_sfcgal.

□□

geometry **CG_3DUnion**(geometry geom1, geometry geom2);
geometry **CG_3DUnion**(geometry set g1field);

□□

Availability: 3.5.0

This method needs SFCGAL backend.

This method implements the SQL/MM specification. SQL-MM IEC 13249-3: 5.1

This function supports 3d and will not drop the z-index.

This function supports Polyhedral surfaces.

This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).

Aggregate variant: returns a geometry that is the 3D union of a rowset of geometries. The CG_3DUnion() function is an ”aggregate” function in the terminology of PostgreSQL. That means that it operates on rows of data, in the same way the SUM() and AVG() functions do and like most aggregates, it also ignores NULL geometries.

☒☒

Union two overlapping tetrahedral solids.

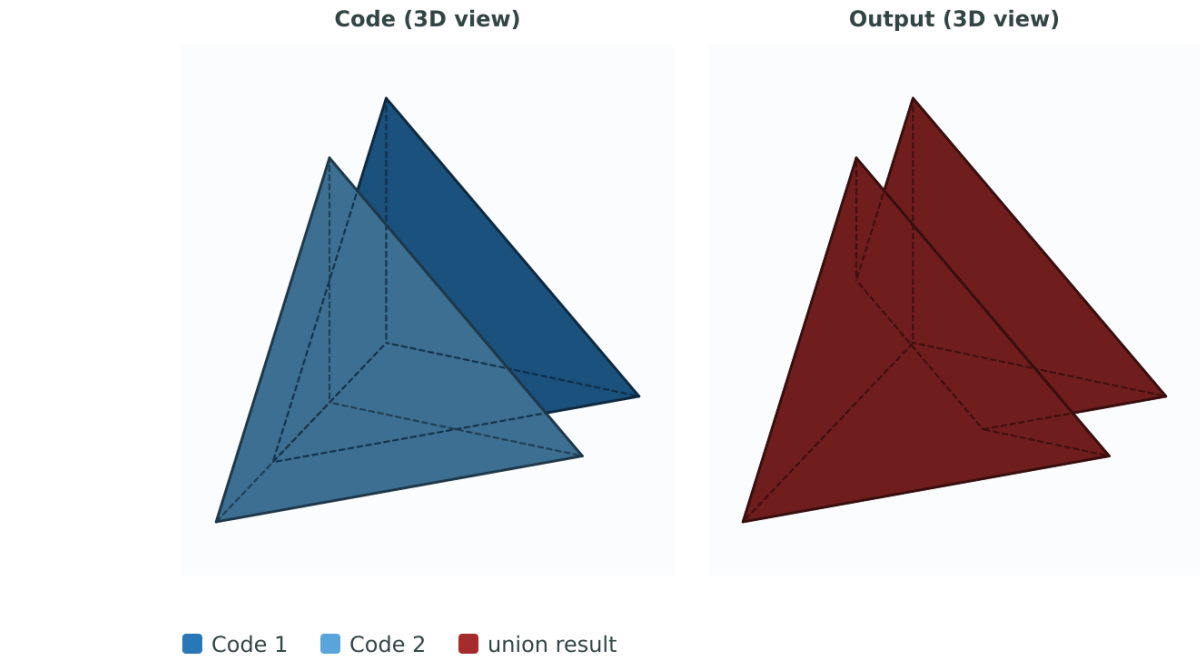
Code

```
WITH solids AS (
  SELECT CG_MakeSolid('POLYHEDRALSURFACE Z (
            ((0 0 0,4 0 0,0 4 0,0 0 0)),
            ((0 0 0,0 0 4,4 0 0,0 0 0)),
            ((0 0 0,0 4 0,0 0 4,0 0 0)),
            ((4 0 0,0 0 4,0 4 0,4 0 0)))'::geometry) AS geom1,
         CG_MakeSolid('POLYHEDRALSURFACE Z (
            ((2 0 0,6 0 0,2 4 0,2 0 0)),
            ((2 0 0,2 0 4,6 0 0,2 0 0)),
            ((2 0 0,2 4 0,2 0 4,2 0 0)),
            ((6 0 0,2 0 4,2 4 0,6 0 0)))'::geometry) AS geom2
)
SELECT CG_3DUnion(geom1, geom2) AS union_result
FROM solids;
```

Output

```
POLYHEDRALSURFACE Z (((2 0 0,0 0 0,2 2 0,2 0 0)),((2 0 2,0 0 0,2 0 0,2 0 2)),((0 0 4,0 4 0,0 0 0,0 0 4)),((2 0 2,0 4 ↩
    0,0 0 4,2 0 2)),((0 4 0,2 2 0,0 0 0,0 4 0)),((2 2 0,0 4 0,2 0 2,2 2 0)),((0 0 0,2 0 2,0 0 4,0 0 0)),((4 0 0,2 0 ↩
    0,2 2 0,4 0 0)),((4 0 0,2 0 2,2 0 0,4 0 0)),((4 0 0,2 2 0,6 0 0,4 0 0)),((2 0 2,4 0 0,2 0 4,2 0 2)),((2 0 2,2 4 ↩
    0,2 2 0,2 0 2)),((2 4 0,2 0 4,6 0 0,2 4 0)),((2 2 0,2 4 0,6 0 0,2 2 0)),((2 4 0,2 0 2,2 0 4,2 4 0)),((4 0 0,6 0 ↩
    0,2 0 4,4 0 0)))
```

Figure



□□

CG_Extrude, ST_AsX3D, CG_3DIntersection CG_3DDifference

8.3.12 CG_AlphaShape

CG_AlphaShape — Computes an Alpha-shape enclosing a geometry

□□

geometry **CG_AlphaShape**(geometry geom, float alpha, boolean allow_holes = false);

□□

Computes the Alpha-Shape of the points in a geometry. An alpha-shape is a (usually) concave polygonal geometry which contains all the vertices of the input, and whose vertices are a subset of the input vertices. An alpha-shape provides a closer fit to the shape of the input than the shape produced by the convex hull.

The "closeness of fit" is controlled by the `alpha` parameter, which can have values from 0 to infinity. Smaller alpha values produce more concave results. Alpha values greater than some data-dependent value produce the convex hull of the input.



□□
Following the CGAL implementation, the alpha value is the *square* of the radius of the disc used in the Alpha-Shape algorithm to "erode" the Delaunay Triangulation of the input points. See CGAL Alpha-Shapes for more information. This is different from the original definition of alpha-shapes, which defines alpha as the radius of the eroding disc.

The computed shape does not contain holes unless the optional `allow_holes` argument is specified as true.

This function effectively computes a concave hull of a geometry in a similar way to ST_ConcaveHull, but uses CGAL and a different algorithm.

Availability: 3.5.0 - requires SFCGAL >= 1.4.1.

 This method needs SFCGAL backend.

□□

The original point cloud and the alpha-shape share one comparison frame.

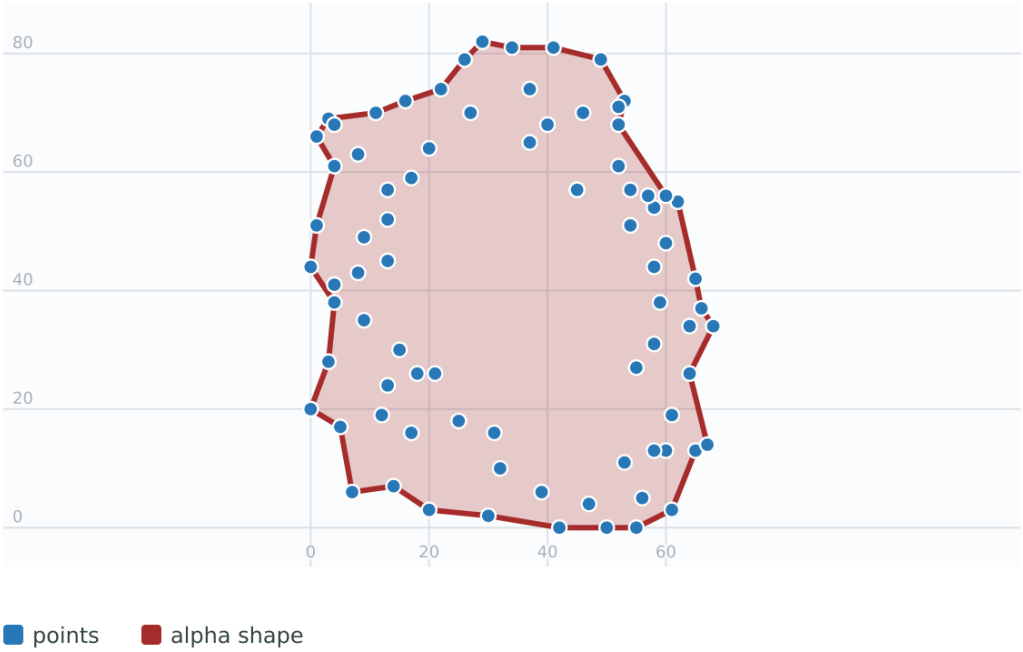
Code

```
WITH data AS (
  SELECT 'MULTIPOINT((63 84),(76 88),(68 73),(53 18),(91 50),(81 70),
(88 29),(24 82),(32 51),(37 23),(27 54),(84 19),(75 87),(44 42),(77 67),(90 30),(36 61),(32 ↩
    65),
(81 47),(88 58),(68 73),(49 95),(81 60),(87 50),
(78 16),(79 21),(30 22),(78 43),(26 85),(48 34),(35 35),(36 40),(31 79),(83 29),(27 84),(52 ↩
    98),(72 95),(85 71),
(75 84),(75 77),(81 29),(77 73),(41 42),(83 72),(23 36),(89 53),(27 57),(57 97),(27 77),(39 ↩
    88),(60 81),
```

```
(80 72),(54 32),(55 26),(62 22),(70 20),(76 27),(84 35),(87 42),(82 54),(83 64),(69 86),(60 ↩
    90),(50 86),(43 80),(36 73),
(36 68),(40 75),(24 67),(23 60),(26 44),(28 33),(40 32),(43 19),(65 16),(73 16),(38 46),(31 ↩
    59),(34 86),(45 90),(64 97))'::geometry AS input_points
)
SELECT input_points AS input_points,

        CG_AlphaShape(input_points, 80.2) AS alpha_shape
FROM data;
```

Figure



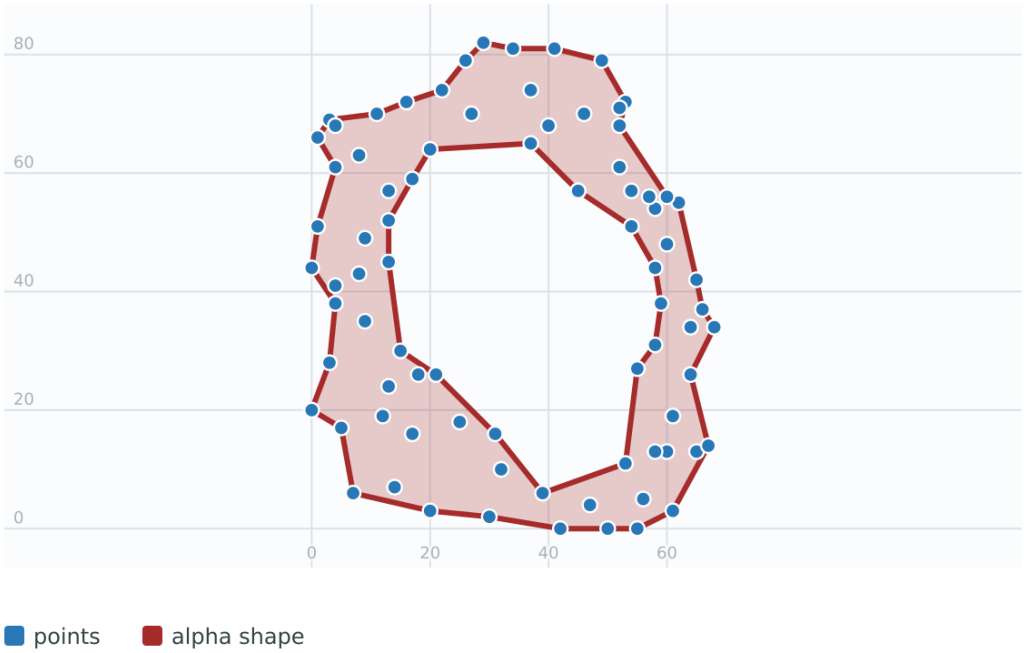
The same point cloud can produce a shape with a hole when holes are allowed.

Code

```
WITH data AS (
  SELECT 'MULTIPOINT((63 84),(76 88),(68 73),(53 18),(91 50),(81 70),(88 29),(24 82),(32  ↩
    51),(37 23),(27 54),(84 19),(75 87),(44 42),(77 67),(90 30),(36 61),(32 65),(81 47),(88  ↩
    58),(68 73),(49 95),(81 60),(87 50),
(78 16),(79 21),(30 22),(78 43),(26 85),(48 34),(35 35),(36 40),(31 79),(83 29),(27 84),(52 ↩
    98),(72 95),(85 71),
(75 84),(75 77),(81 29),(77 73),(41 42),(83 72),(23 36),(89 53),(27 57),(57 97),(27 77),(39 ↩
    88),(60 81),
(80 72),(54 32),(55 26),(62 22),(70 20),(76 27),(84 35),(87 42),(82 54),(83 64),(69 86),(60 ↩
    90),(50 86),(43 80),(36 73),
(36 68),(40 75),(24 67),(23 60),(26 44),(28 33),(40 32),(43 19),(65 16),(73 16),(38 46),(31 ↩
    59),(34 86),(45 90),(64 97))'::geometry AS input_points
)
SELECT input_points AS input_points,

        CG_AlphaShape(input_points, 100.1, true) AS alpha_shape
FROM data;
```

Figure



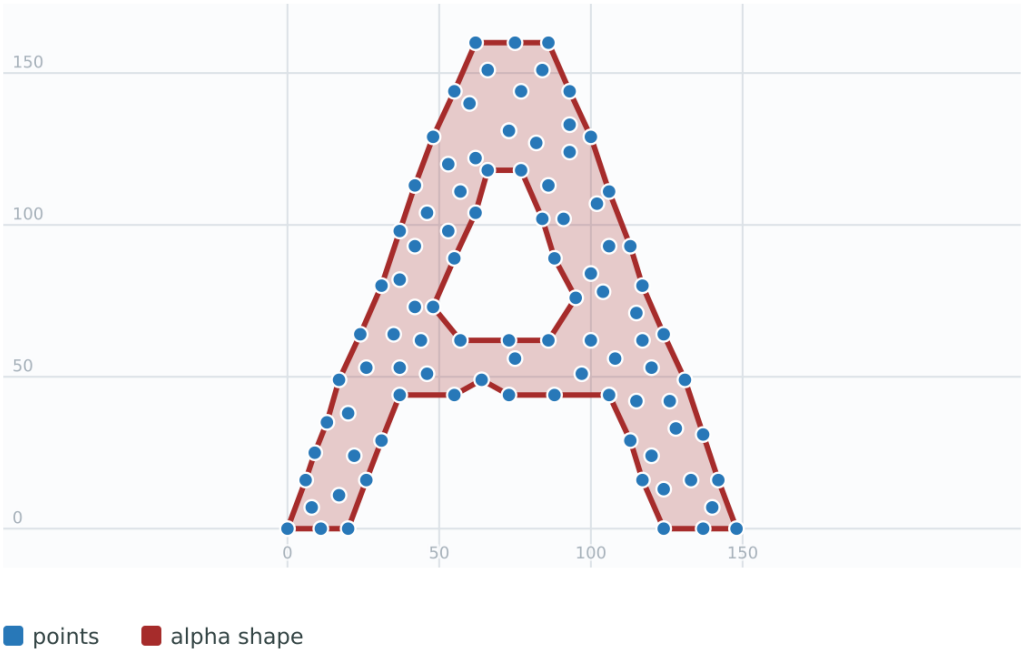
The second documented point cloud also produces a holed alpha-shape.

Code

```
WITH data AS (
  SELECT 'MULTIPOINT ((132 64),(114 64),(99 64),(81 64),(63 64),(57 49),(52 36),(46 20),(37 ↩
     20),(26 20),(32 36),(39 55),(43 69),(50 84),(57 100),(63 118),(68 133),(74 149),(81  ↩
    164),(88 180),(101 180),(112 180),(119 164),(126 149),(132 131),(139 113),(143 100),(150 ↩
     84),(157 69),(163 51),(168 36),(174 20),(163 20),(150 20),(143 36),(139 49),(132 64) ↩
    ,(99 151),(92 138),(88 124),(81 109),(74 93),(70 82),(83 82),(99 82),(112 82),(126 82) ↩
    ,(121 96),(114 109),(110 122),(103 138),(99 151),(34 27),(43 31),(48 44),(46 58),(52 73) ↩
    ,(63 73),(61 84),(72 71),(90 69),(101 76),(123 71),(141 62),(166 27),(150 33),(159 36) ↩
    ,(146 44),(154 53),(152 62),(146 73),(134 76),(143 82),(141 91),(130 98),(126 104),(132  ↩
    113),(128 127),(117 122),(112 133),(119 144),(108 147),(119 153),(110 171),(103 164),(92 ↩
     171),(86 160),(88 142),(79 140),(72 124),(83 131),(79 118),(68 113),(63 102),(68 93) ↩
    ,(35 45))'::geometry AS input_points
)
SELECT input_points AS input_points,

         CG_AlphaShape(input_points, 102.2, true) AS alpha_shape
FROM data;
```

Figure



□□

ST_ConcaveHull, CG_OptimalAlphaShape

8.3.13 CG_ApproxConvexPartition

CG_ApproxConvexPartition — Computes approximal convex partition of the polygon geometry

□□

geometry **CG_ApproxConvexPartition**(geometry geom);

□□

Computes approximal convex partition of the polygon geometry (using a triangulation).



□□
A partition of a polygon P is a set of polygons such that the interiors of the polygons do not intersect and the union of the polygons is equal to the interior of the original polygon P. CG_ApproxConvexPartition and CG_GreeneApproxConvexPartition functions produce approximately optimal convex partitions. Both these functions produce convex decompositions by first decomposing the polygon into simpler polygons; CG_ApproxConvexPartition uses a triangulation and CG_GreeneApproxConvexPartition a monotone partition. These two functions both guarantee that they will produce no more than four times the optimal number of convex pieces but they differ in their runtime complexities. Though the triangulation-based approximation algorithm often results in fewer convex pieces, this is not always the case.

Availability: 3.5.0 - requires SFCGAL >= 1.5.0.

Requires SFCGAL >= 1.5.0

This method needs SFCGAL backend.

□□

Approximate convex partition of the same polygon used by the other partition examples.

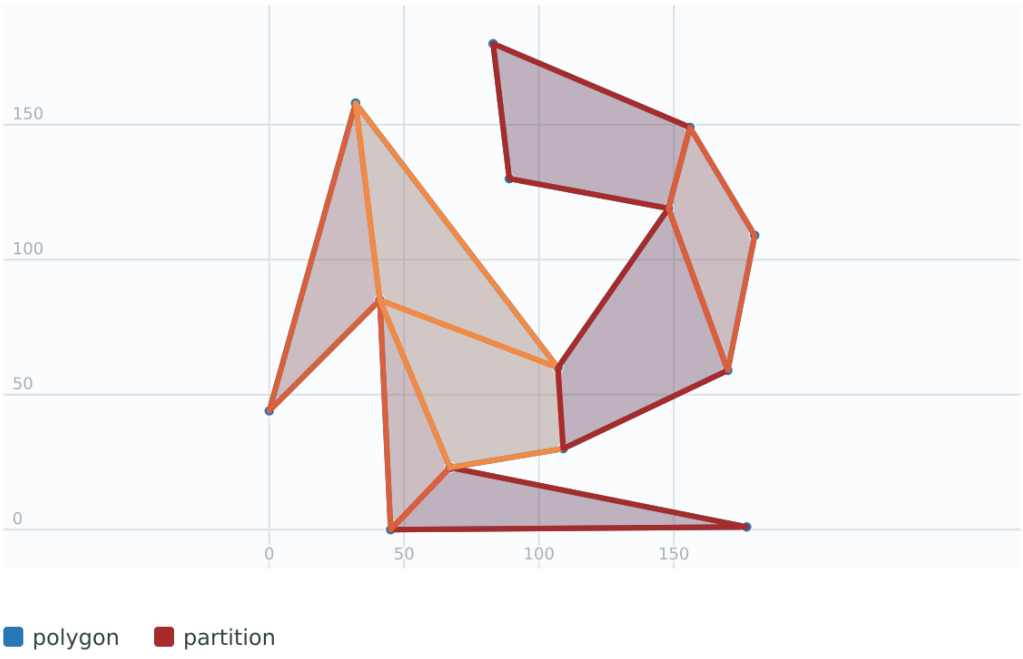
Code

```
WITH data AS (
  SELECT 'POLYGON((156 150,83 181,89 131,148 120,107 61,32 159,0 45,41 86,45 1,177 2,67 ↩
    24,109 31,170 60,180 110,156 150))'::geometry AS input_polygon
)
SELECT input_polygon AS input_polygon,
       CG_ApproxConvexPartition(input_polygon) AS partition
FROM data;
```

Output

```
POLYGON((156 150,83 181,89 131,148 120,107 61,32 159,0 45,41 86,45 1,177 2,67 24,109 31,170 60,180 110,156 150)) | ↩
    GEOMETRYCOLLECTION(POLYGON((156 150,83 181,89 131,148 120,156 150)), POLYGON((32 159,0 45,41 86,32 159)), POLYGON ↩
    ((107 61,32 159,41 86,107 61)), POLYGON((45 1,177 2,67 24,45 1)), POLYGON((41 86,45 1,67 24,41 86)), POLYGON((107 ↩
     61,41 86,67 24,109 31,107 61)), POLYGON((148 120,107 61,109 31,170 60,148 120)), POLYGON((156 150,148 120,170  ↩
    60,180 110,156 150)))
```

Figure



□□

CG_YMonotonePartition, CG_GreeneApproxConvexPartition, CG_OptimalConvexPartition

8.3.14 CG_ApproximateMedialAxis

CG_ApproximateMedialAxis — □□□□□□□□□□□□□□□□.

□□

geometry **CG_ApproximateMedialAxis**(geometry geom);
geometry **CG_ApproximateMedialAxis**(geometry geom, boolean projected);

□□

Return an approximate medial axis for the areal input based on its straight skeleton. Uses an SFCGAL specific API when built against a capable version (1.2.0+). Otherwise the function is just a wrapper around CG_StraightSkeleton (slower case).

When `projected` is `true`, free endpoints of the medial axis are extended to reach the polygon boundary (projected medial axis). Requires SFCGAL 2.3.0+. When built against an older SFCGAL version, a notice is emitted and the non-projected result is returned instead.

Availability: 3.5.0

Availability: 3.7.0 - `projected` parameter. Requires SFCGAL >= 2.3.0 for projected result; falls back to non-projected with a notice on older versions.



□□
This function ignores the Z dimension. It always gives a 2D result even when used on a 3D geometry.

 This method needs SFCGAL backend.

 This function supports Polyhedral surfaces.

 This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).

□□

A polygon and its approximate medial axis.

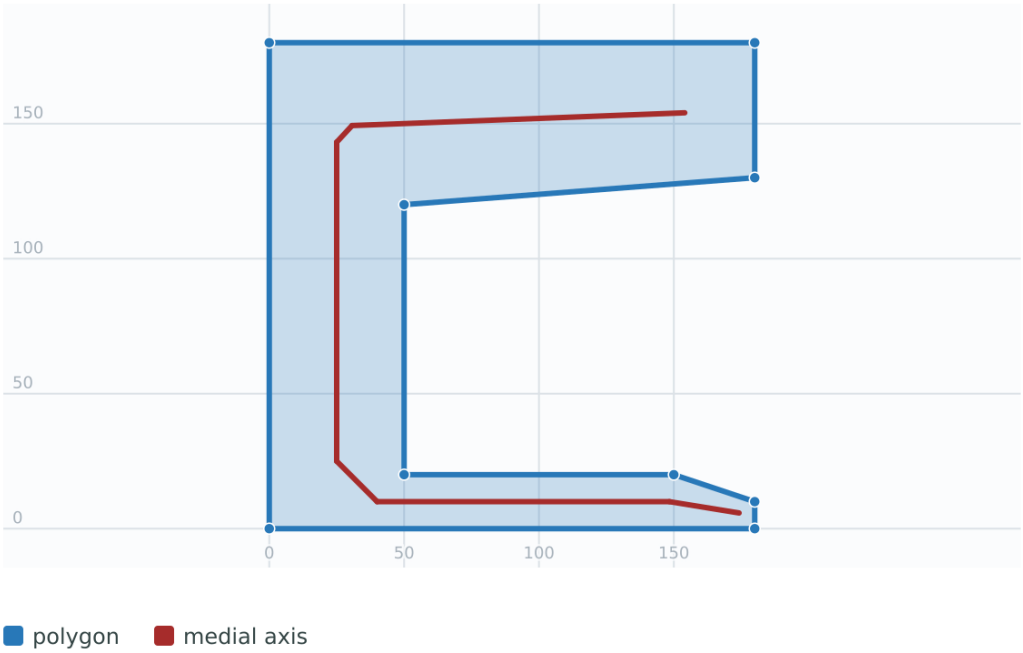
Code

```
WITH data AS (
  SELECT 'POLYGON((190 190,10 190,10 10,190 10,190 20,160 30,60 30,60 130,190 140,190 190))
    '::geometry AS polygon
)
SELECT polygon AS input_polygon,
       CG_ApproximateMedialAxis(polygon) AS medial_axis
FROM data;
```

Output

```
POLYGON((190 190,10 190,10 10,190 10,190 20,160 30,60 30,60 130,190 140,190 190)) | MULTILINESTRING((184.189
    15.811,158.377 20),(50 20,158.377 20),(50 20,35 35),(35 153.151,35 35),(35 153.151,40.697 159.303),(164.04
    164.04,40.697 159.303))
```

Figure



This example computes the projected medial axis with free endpoints extended to the polygon boundary.

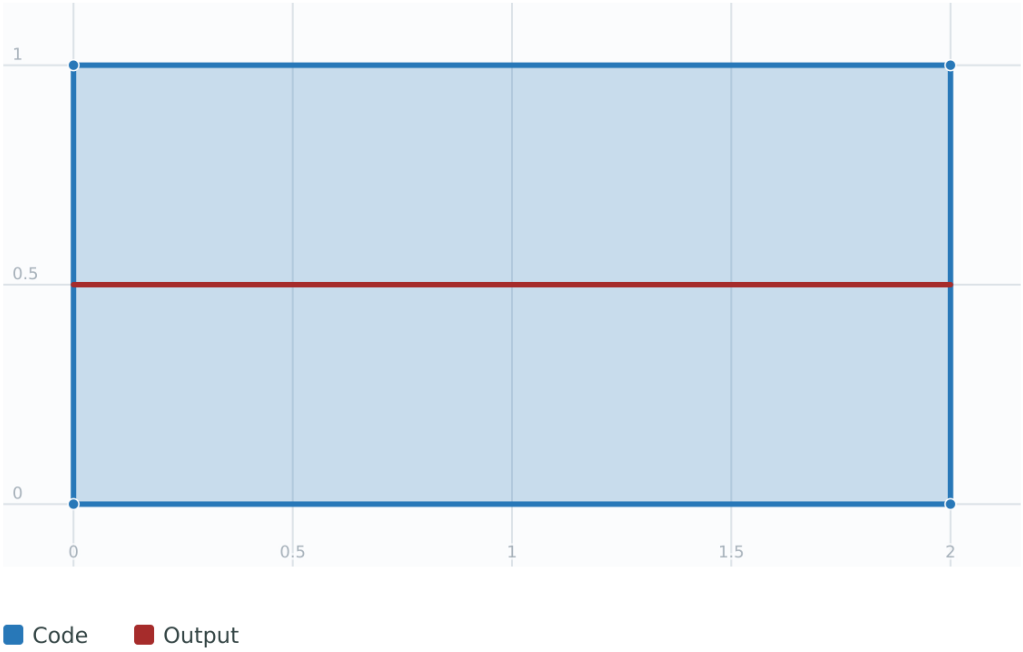
Code

```
SELECT CG_ApproximateMedialAxis('POLYGON((0 0,2 0,2 1,0 1,0 0))', true);
```

Output

```
MULTILINESTRING((0 0.5,0.5 0.5,1.5 0.5,2 0.5))
```

Figure



□□

CG_StraightSkeleton, CG_StraightSkeletonPartition

8.3.15 CG_GenerateFlatRoof

CG_GenerateFlatRoof — Generate a flat (box) roof from a footprint polygon.

□□

geometry **CG_GenerateFlatRoof**(geometry geom, float8 height);

□□

Generate a flat (box-shaped) roof as a 3D `PolyhedralSurface Z` from a 2D footprint polygon. The roof height above the ground plane is given by `height` (default 3.0).

Availability: 3.7.0 - requires SFCGAL >= 2.3.0.

This method needs SFCGAL backend.

This function supports Polyhedral surfaces.

□□

Generate a flat roof two units above a rectangular footprint.

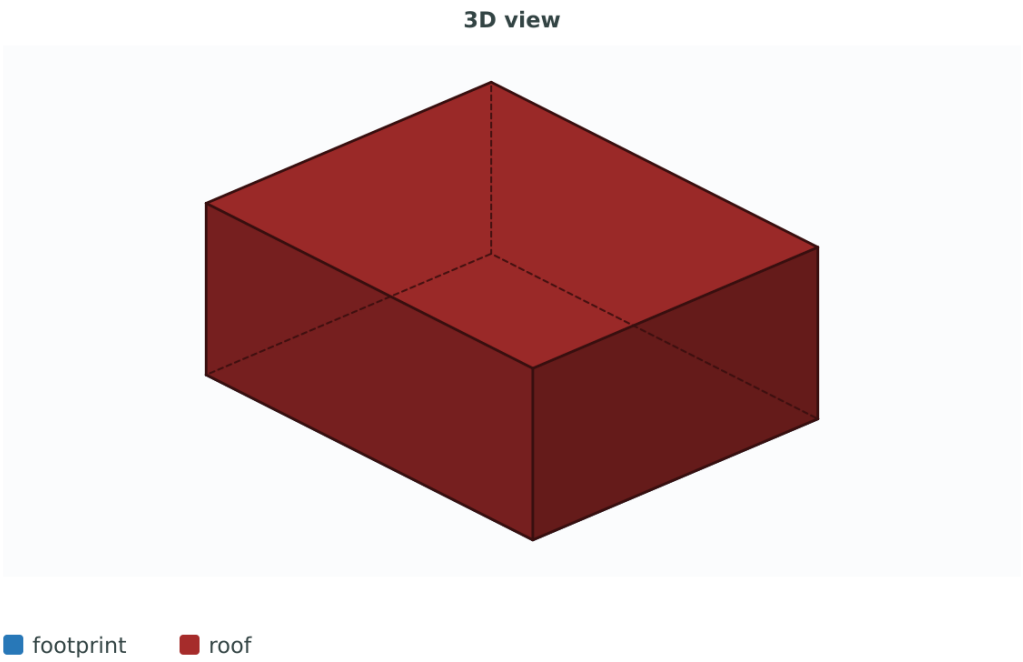
Code

```
WITH data AS (
  SELECT 'POLYGON((0 0,5 0,5 4,0 4,0 0))'::geometry AS footprint
)
SELECT CG_GenerateFlatRoof(footprint, 2.0) AS roof
FROM data;
```

Output

```
POLYHEDRALSURFACE Z (((0 0 0,0 4 0,5 4 0,5 0 0,0 0 0)),((0 0 2,5 0 2,5 4 2,0 4 2,0 0 2)),((0 0 0,0 0 2,0 4 2,0 4 0,0 0 ↩
     0)),((0 4 0,0 4 2,5 4 2,5 4 0,0 4 0)),((5 4 0,5 4 2,5 0 2,5 0 0,5 4 0)),((5 0 0,5 0 2,0 0 2,0 0 0,5 0 0)))
```

Figure



□□

CG_GenerateHippedRoof, CG_GenerateGableRoof, CG_GenerateSkillionRoof, CG_GenerateRoof

8.3.16 CG_GenerateHippedRoof

CG_GenerateHippedRoof — Generate a hipped roof from a footprint polygon.

□□

geometry **CG_GenerateHippedRoof**(geometry geom, float8 height);

□□

Generate a hipped roof as a 3D `PolyhedralSurface Z` from a 2D footprint polygon. All sides slope upward to meet at a ridge or apex. The roof height is given by `height` (default 3.0).

Availability: 3.7.0 - requires SFCGAL >= 2.3.0.

This method needs SFCGAL backend.

This function supports Polyhedral surfaces.

□□

The footprint and hipped roof are shown side-by-side so the generated 3D faces do not hide behind the input footprint.

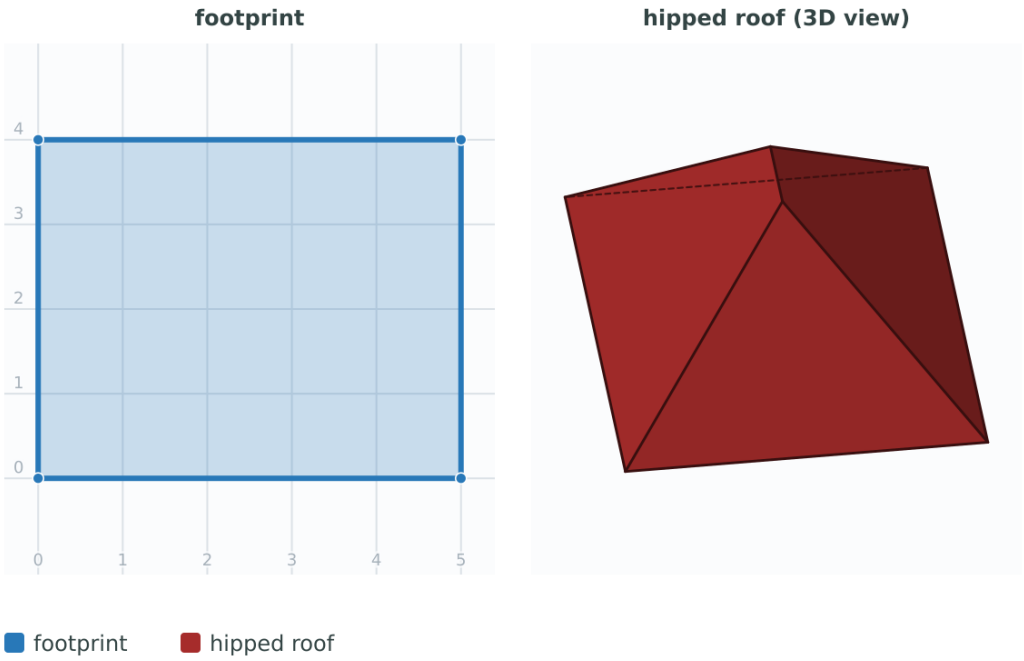
Code

```
WITH data AS (
  SELECT 'POLYGON((0 0,5 0,5 4,0 4,0 0))'::geometry AS input_footprint
)
SELECT input_footprint AS input_footprint,
       CG_GenerateHippedRoof(input_footprint, 2.0) AS hipped_roof
FROM data;
```

Output

```
POLYGON((0 0,5 0,5 4,0 4,0 0)) | POLYHEDRALSURFACE Z (((0 4 0,5 4 0,5 0 0,0 0 0,0 4 0)),((0 4 0,0 0 0,2 2 2,0 4 0)) ↩
    ,((0 0 0,5 0 0,3 2 2,2 2 2,0 0 0)),((5 0 0,5 4 0,3 2 2,5 0 0)),((5 4 0,0 4 0,2 2 2,3 2 2,5 4 0)))
```

Figure



☒☒

CG_GenerateFlatRoof, CG_GenerateGableRoof, CG_GenerateSkillionRoof, CG_GenerateRoof

8.3.17 CG_GenerateGableRoof

CG_GenerateGableRoof — Generate a gable roof from a footprint polygon.

☒☒

geometry **CG_GenerateGableRoof**(geometry geom, float8 height, float8 slope_angle);

☒☒

Generate a gable roof as a 3D `PolyhedralSurface Z` from a 2D footprint polygon. Two opposite sides are vertical gable ends; the other two sides slope to a horizontal ridge. `height` sets the ridge height

above the ground (default 3.0) and `slope_angle` controls the pitch in degrees (default 30.0). The SRID of the input geometry is preserved.

Availability: 3.7.0 - requires SFCGAL >= 2.3.0.

This method needs SFCGAL backend.

This function supports Polyhedral surfaces.

☒☒

Generate a gable roof using a height of 2 and a 30 degree slope.

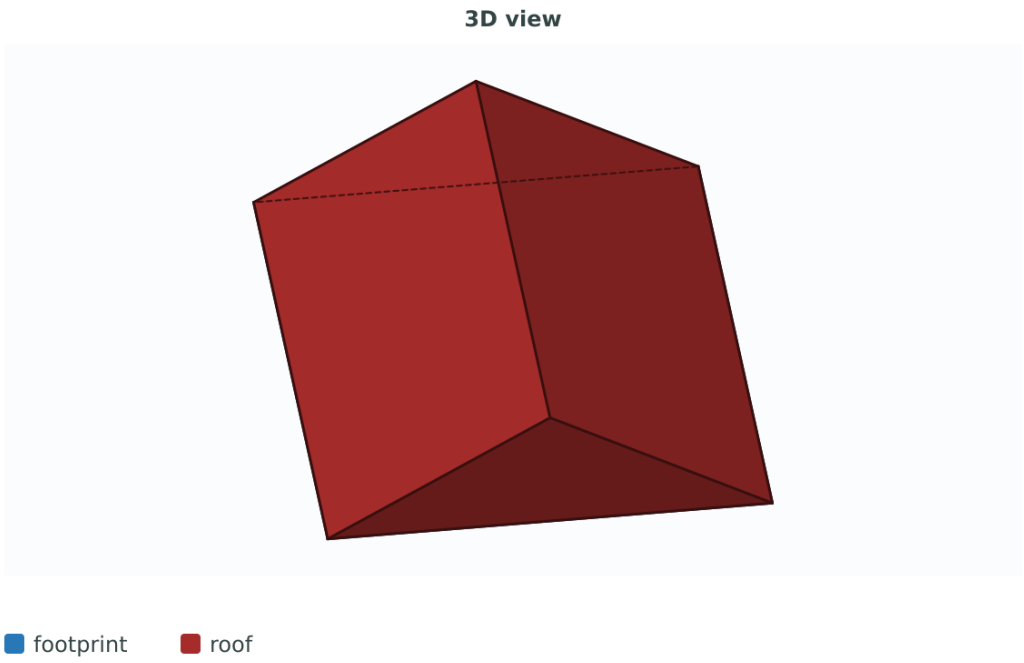
Code

```
WITH data AS (
  SELECT 'POLYGON((0 0,5 0,5 4,0 4,0 0))'::geometry AS footprint
)
SELECT CG_GenerateGableRoof(footprint, 2.0, 30.0) AS roof
FROM data;
```

Output

```
POLYHEDRALSURFACE Z (((5 4 0,5 0 0,0 0 0,0 4 0,5 4 0)),((0 4 0,0 0 0,0 2 1.155,0 4 0)),((0 0 0,5 0 0,5 2 1.155,0 2 ↩
    1.155,0 0 0)),((5 0 0,5 4 0,5 2 1.155,5 0 0)),((5 4 0,0 4 0,0 2 1.155,5 2 1.155,5 4 0)))
```

Figure



☒☒

CG_GenerateFlatRoof, CG_GenerateHippedRoof, CG_GenerateSkillionRoof, CG_GenerateRoof

8.3.18 CG_GenerateSkillionRoof

CG_GenerateSkillionRoof — Generate a skillion (single-slope) roof from a footprint polygon.

□□

geometry **CG_GenerateSkillionRoof**(geometry geom, float8 height, float8 slope_angle, integer primary_edge_index);

□□

Generate a skillion (single-slope, shed-style) roof as a 3D `PolyhedralSurface Z` from a 2D footprint polygon. The entire roof surface slopes in one direction. `height` sets the maximum ridge height (default 3.0), `slope_angle` sets the pitch in degrees (default 30.0), and `primary_edge_index` selects which polygon edge defines the high side (0-based, default 0).

When the combination of `height` and `slope_angle` would cause a side wall to vanish (the slope reaches the far edge before the full height is achieved), that face is omitted from the result. This means the number of output patches can vary depending on the parameters —a lower height relative to the footprint depth produces more faces than a taller, steeper configuration.

Availability: 3.7.0 - requires SFCGAL >= 2.3.0.

This method needs SFCGAL backend.

This function supports Polyhedral surfaces.

□□

Height 2.0 produces 6 faces (all walls present); height 4.0 produces 5 faces (one side wall is filtered out because the slope reaches the far edge before the full height).

Code

```
SELECT
  ST_NumPatches(CG_GenerateSkillionRoof(
    'POLYGON((0 0,5 0,5 4,0 4,0 0))', 2.0)) AS "height 2",
  ST_NumPatches(CG_GenerateSkillionRoof(
    'POLYGON((0 0,5 0,5 4,0 4,0 0))', 4.0)) AS "height 4";
```

Output

```
6 | 5
```

The height 2.0 result is shown with its footprint.

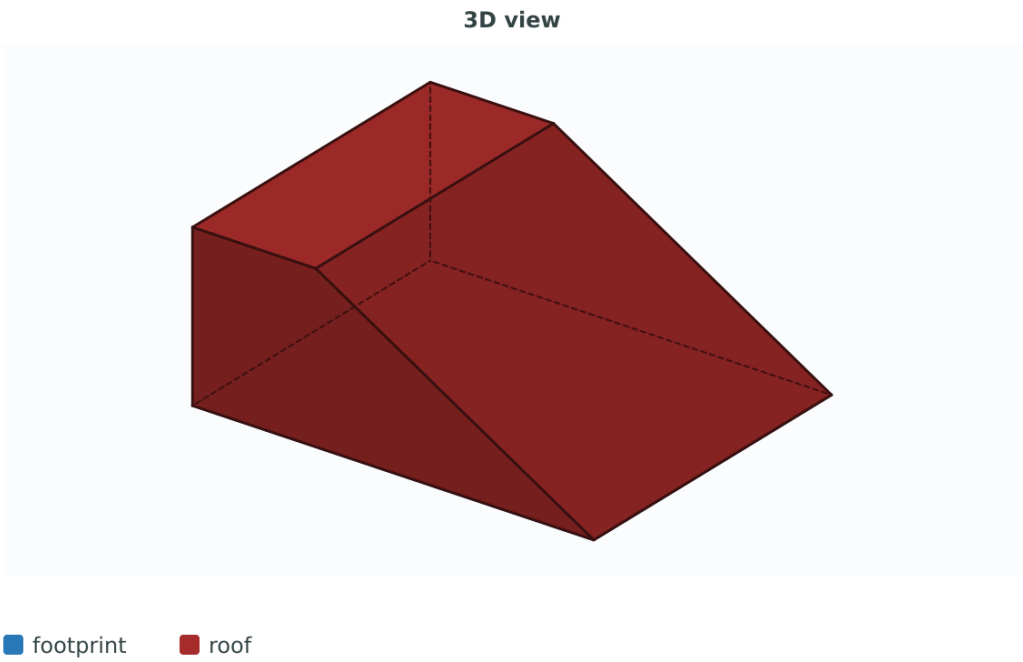
Code

```
WITH data AS (
  SELECT 'POLYGON((0 0,5 0,5 4,0 4,0 0))'::geometry AS footprint
)
SELECT CG_GenerateSkillionRoof(footprint, 2.0) AS roof
FROM data;
```

Output

```
POLYHEDRALSURFACE Z (((0 4 0,5 4 0,5 0 0,0 0 0,0 4 0)),((5 0 2,5 4 2,3.464 4 2,3.464 0 2,5 0 2)),((0 4 0,0 0 0,3.464 0 ↩
     2,3.464 4 2,0 4 0)),((0 0 0,5 0 0,5 0 2,3.464 0 2,0 0 0)),((5 0 0,5 4 0,5 4 2,5 0 2,5 0 0)),((5 4 0,0 4 0,3.464  ↩
    4 2,5 4 2,5 4 0)))
```

Figure



☒☒

CG_GenerateFlatRoof, CG_GenerateHippedRoof, CG_GenerateGableRoof, CG_GenerateRoof

8.3.19 CG_GenerateRoof

CG_GenerateRoof — Generate a roof of the requested type from a footprint polygon.

☒☒

geometry **CG_GenerateRoof**(geometry geom, text roof_type, float8 height, float8 slope_angle, integer primary_edge_index);

☒☒

Generate a roof as a 3D `PolyhedralSurface Z` from a 2D footprint polygon. The `roof_type` parameter selects the roof style (default `'HIPPED'`):

- `FLAT` —flat box roof.
- `HIPPED` —hipped roof (all sides slope to apex).
- `GABLE` —gable roof (two sloping sides, two vertical ends).
- `SKILLION` —single-slope (shed) roof.

`height` sets the roof height (default 3.0), `slope_angle` sets the pitch in degrees (default 30.0), and `primary_edge_index` selects the reference edge for skillion roofs (default 0).

Availability: 3.7.0 - requires SFCGAL >= 2.3.0.

This method needs SFCGAL backend.

This function supports Polyhedral surfaces.

□□

The generic dispatcher returns the same 3D roofs as the specialized helpers for the same footprint.

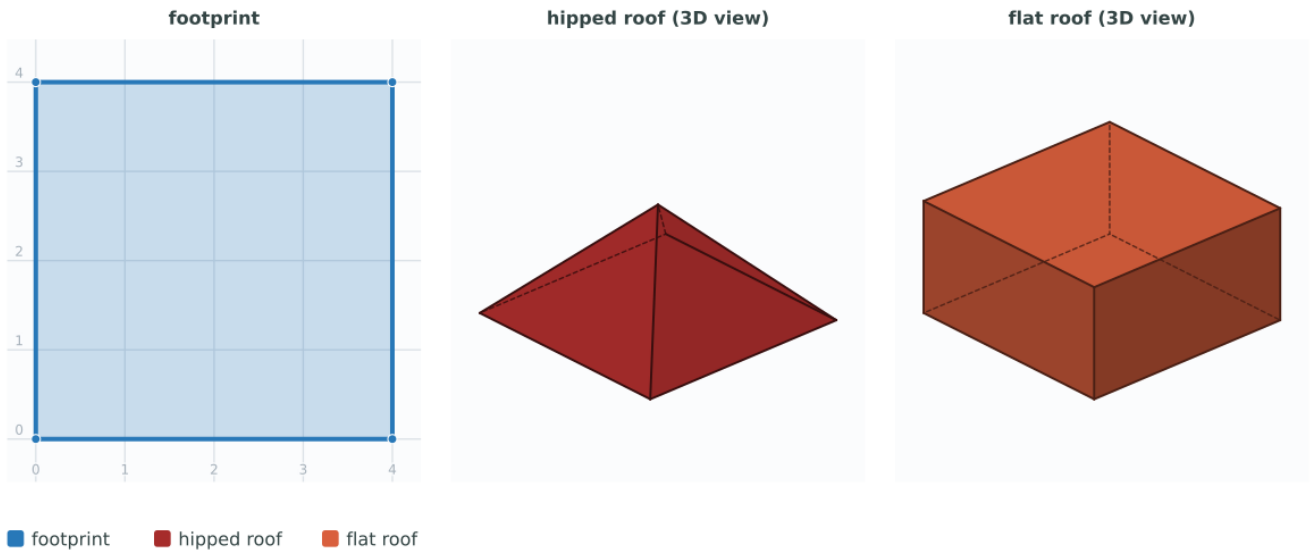
Code

```
WITH data AS (
  SELECT 'POLYGON((0 0,4 0,4 4,0 4,0 0))'::geometry AS input_footprint
)
SELECT input_footprint AS input_footprint,
       CG_GenerateRoof(input_footprint, 'HIPPED', 2.0, 30.0, 0) AS hipped_roof,
       CG_GenerateRoof(input_footprint, 'FLAT', 2.0, 30.0, 0) AS flat_roof
FROM data;
```

Output

```
POLYGON((0 0,4 0,4 4,0 4,0 0)) | POLYHEDRALSURFACE Z (((0 4 0,4 4 0,4 0 0,0 0 0,0 4 0)),((0 4 0,0 0 0,2 2 2,0 4 0)) ↩
    ,((0 0 0,4 0 0,2 2 2,0 0 0)),((4 0 0,4 4 0,2 2 2,4 0 0)),((4 4 0,0 4 0,2 2 2,4 4 0))) | POLYHEDRALSURFACE Z (((0  ↩
    0 0,0 4 0,4 4 0,4 0 0,0 0 0)),((0 0 2,4 0 2,4 4 2,0 4 2,0 0 2)),((0 0 0,0 0 2,0 4 2,0 4 0,0 0 0)),((0 4 0,0 4 2,4 ↩
     4 2,4 4 0,0 4 0)),((4 4 0,4 4 2,4 0 2,4 0 0,4 4 0)),((4 0 0,4 0 2,0 0 2,0 0 0,4 0 0)))
```

Figure



□□

CG_GenerateFlatRoof, CG_GenerateHippedRoof, CG_GenerateGableRoof, CG_GenerateSkillionRoof

8.3.20 CG_PolygonRepair

CG_PolygonRepair — Repair an invalid polygon or multipolygon.

□□

geometry **CG_PolygonRepair**(geometry geom, text rule);

□□

Repair an invalid polygon or multipolygon using CGAL's 2D Polygon Repair algorithm. Returns a `MultiPolygon`.

The `rule` parameter controls the repair strategy:

- `EVEN_ODD` (default) —areas covered an odd number of times are kept. Available with CGAL 6.0+.
- `NON_ZERO` —areas with non-zero winding number are kept. Requires CGAL 6.1+.
- `UNION` —union of all input polygons. Requires CGAL 6.1+.
- `INTERSECTION` —intersection of all input polygons. Requires CGAL 6.1+.

Availability: 3.7.0 - requires SFCGAL >= 2.3.0 and CGAL >= 6.0.

 This method needs SFCGAL backend.

 This function supports Polyhedral surfaces.

□□

Repair a self-intersecting bowtie polygon with the default `EVEN_ODD` rule.

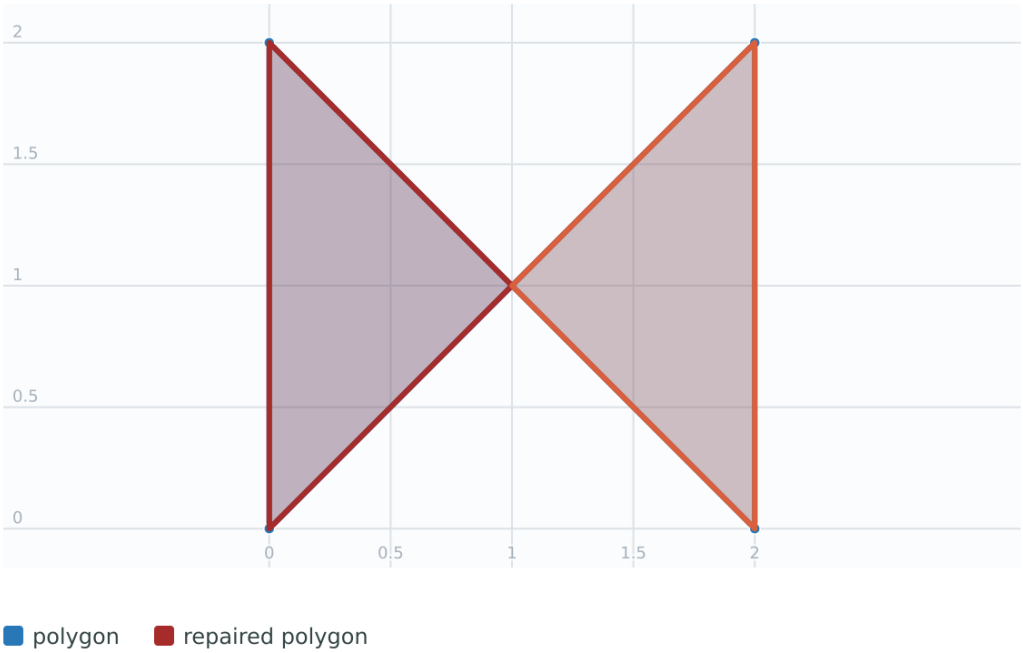
Code

```
WITH data AS (
  SELECT 'POLYGON((0 0,2 2,2 0,0 2,0 0))'::geometry AS input_polygon
)
SELECT input_polygon AS input_polygon,
       CG_PolygonRepair(input_polygon) AS repaired_polygon
FROM data;
```

Output

```
POLYGON((0 0,2 2,2 0,0 2,0 0)) | MULTIPOLYGON(((0 0,1 1,0 2,0 0)),((1 1,2 0,2 2,1 1)))
```

Figure



With the `UNION` rule, all winding area is kept. This rule requires CGAL 6.1 or later.

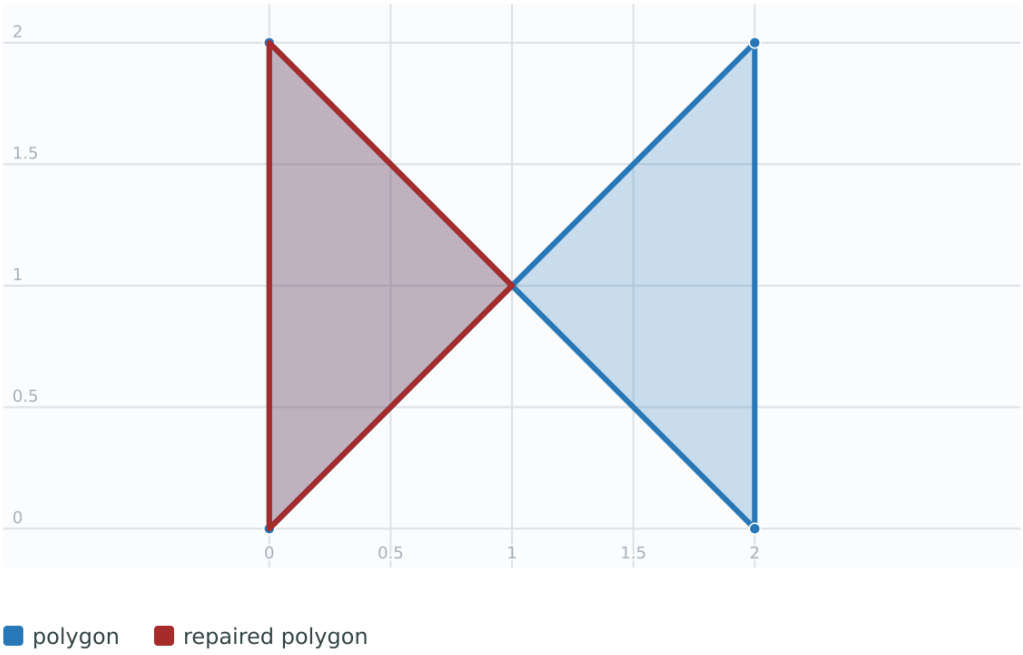
Code

```
WITH data AS (
  SELECT 'POLYGON((0 0,2 2,2 0,0 2,0 0))'::geometry AS input_polygon
)
SELECT input_polygon AS input_polygon,
       CG_PolygonRepair(input_polygon, 'UNION') AS repaired_polygon
FROM data;
```

Output

```
POLYGON((0 0,2 2,2 0,0 2,0 0)) | MULTIPOLYGON(((0 0,1 1,0 2,0 0)))
```

Figure



☒☒

ST_IsValid, ST_MakeValid

8.3.21 CG_ConstrainedDelaunayTriangles

CG_ConstrainedDelaunayTriangles — Return a constrained Delaunay triangulation around the given input geometry.

☒☒

geometry **CG_ConstrainedDelaunayTriangles**(geometry g1);

☒☒

Return a Constrained Delaunay triangulation around the vertices of the input geometry. Output is a TIN.

This method needs SFCGAL backend.

2.1.0 ☒☒☒☒☒☒☒☒☒☒☒☒.

This function supports 3d and will not drop the z-index.

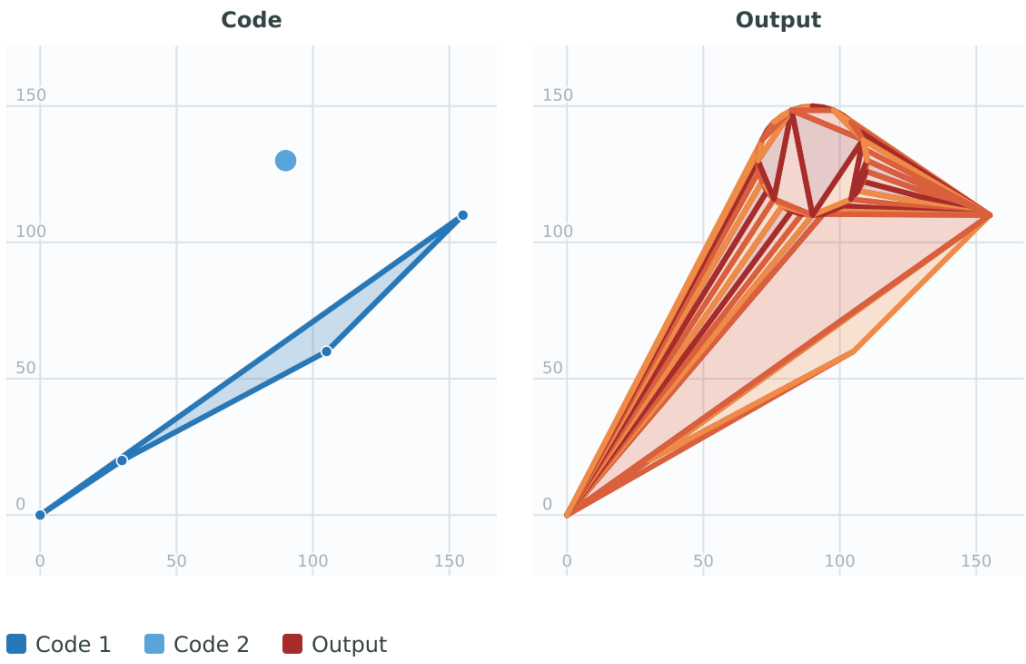
☒☒

Constrained Delaunay triangulation of the polygon and buffered point.

Code

```
SELECT
  CG_ConstrainedDelaunayTriangles(
    ST_Union(
      'POLYGON((175 150,20 40,50 60,125 100,175 150))'::geometry,
      ST_Buffer('POINT(110 170)'::geometry, 20)
    )
  );
```

Figure

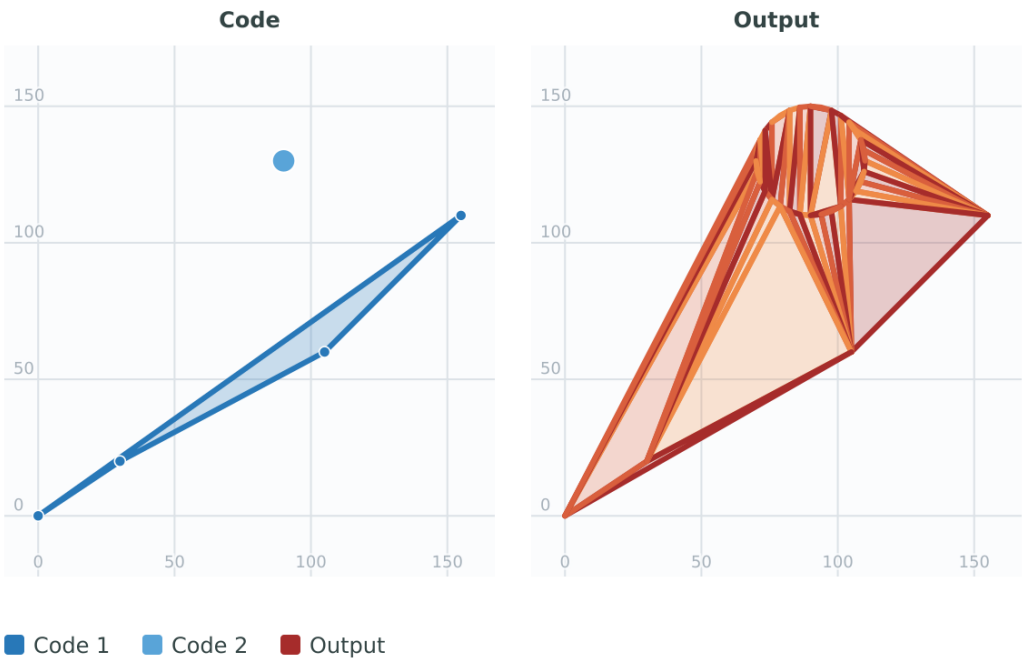


ST_DelaunayTriangles applied to the same input. Triangle edges cross polygon boundaries.

Code

```
SELECT
  ST_DelaunayTriangles(
    ST_Union(
      'POLYGON((175 150,20 40,50 60,125 100,175 150))'::geometry,
      ST_Buffer('POINT(110 170)'::geometry, 20)
    )
  );
```

Figure



☒☒

ST_DelaunayTriangles, ST_TriangulatePolygon, CG_Tesselate, ST_ConcaveHull, ST_Dump

8.3.22 CG_Extrude

CG_Extrude — ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.

☒☒

geometry **CG_Extrude**(geometry geom, float x, float y, float z);

☒☒

Extrudes the input along the vector (*x*, *y*, *z*) and connects it to its translated copy. Extrusion can raise the dimensionality of the result, for example from a surface to a volume.

Availability: 3.5.0

This method needs SFCGAL backend.

This function supports 3d and will not drop the z-index.

This function supports Polyhedral surfaces.

This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).

□□

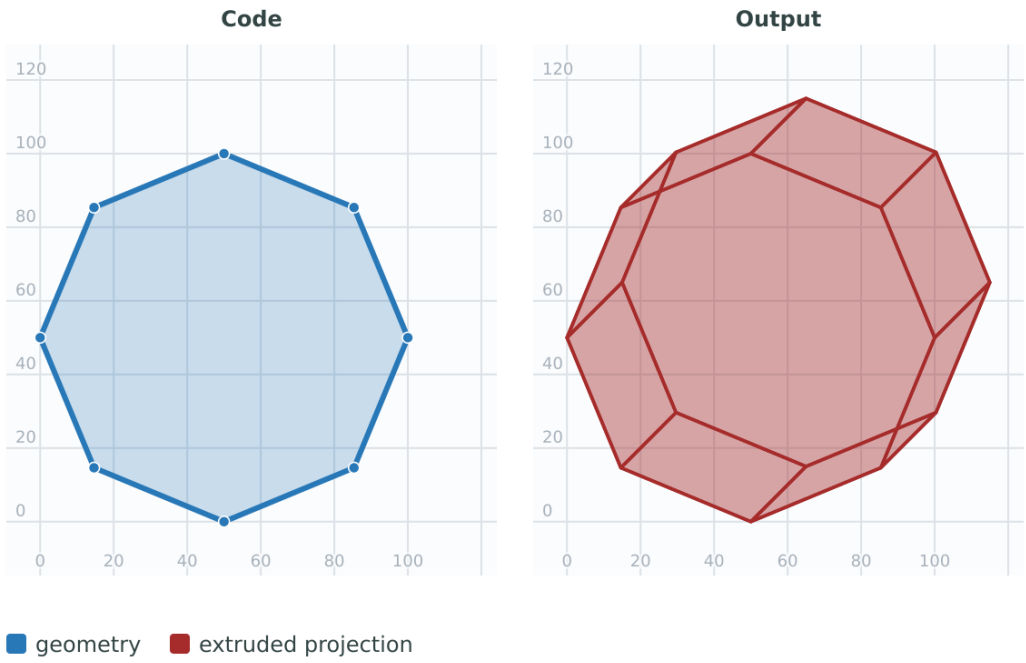
The figures use ST_Affine to project Z diagonally into XY after extrusion. The returned geometry from CG_Extrude remains a 3D PolyhedralSurface.

Extruding an octagonal Polygon 30 units along Z.

Code

```
WITH data AS (
  SELECT ST_Buffer('POINT(100 90)'::geometry, 50, 'quad_segs=2') AS geom
), projected AS (
  SELECT ST_Force2D(ST_Affine(geom,
           1, 0, 0.5, 0, 1, 0.5, 0, 0, 0, 0, 0, 0)) AS input_geom,
         ST_Force2D(ST_Affine(CG_Extrude(geom, 0, 0, 30),
           1, 0, 0.5, 0, 1, 0.5, 0, 0, 0, 0, 0, 0)) AS output_geom
  FROM data
)
SELECT input_geom AS input_geometry,
       output_geom AS extruded_projection
FROM projected;
```

Figure



Extruding a LineString 10 units along Z.

Code

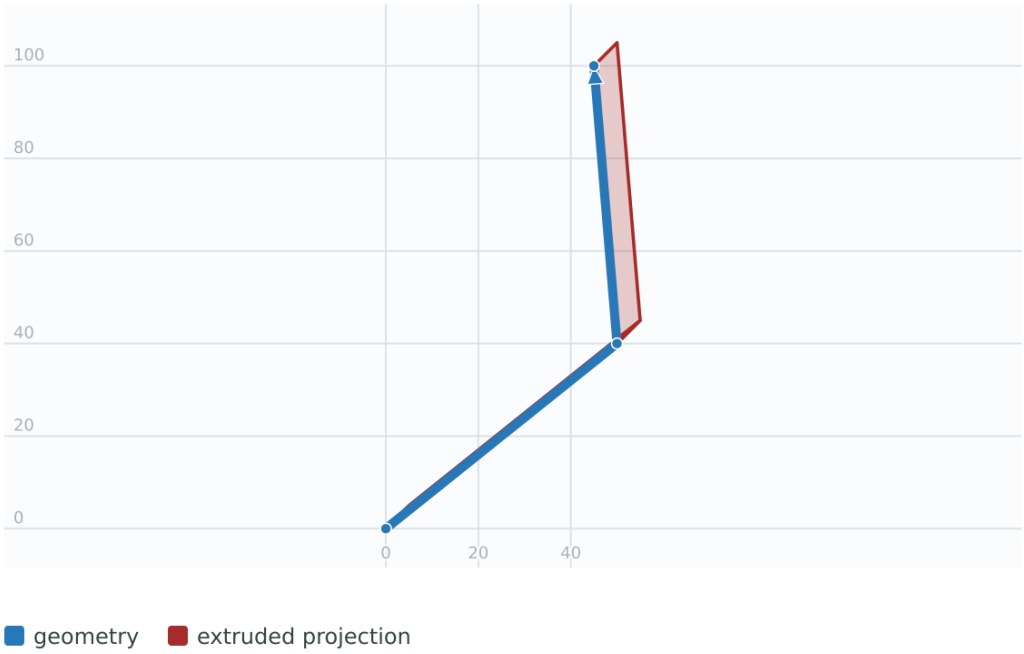
```
WITH data AS (
  SELECT 'LINESTRING(50 50,100 90,95 150)'::geometry AS geom
), projected AS (
  SELECT ST_Force2D(ST_Affine(geom,
           1, 0, 0.5, 0, 1, 0.5, 0, 0, 0, 0, 0, 0)) AS input_geom,
         ST_Force2D(ST_Affine(CG_Extrude(geom, 0, 0, 10),
           1, 0, 0.5, 0, 1, 0.5, 0, 0, 0, 0, 0, 0)) AS output_geom
  FROM data
)
SELECT input_geom AS input_geometry,
```

```
       output_geom AS extruded_projection
FROM projected;
```

Output

```
LINESTRING(50 50,100 90,95 150) | POLYHEDRALSURFACE(((50 50,100 90,105 95,55 55,50 50)),((100 90,95 150,100 155,105 ↩
    95,100 90)))
```

Figure



☒☒

ST_AsX3D, CG_ExtrudeStraightSkeleton

8.3.23 CG_ExtrudeStraightSkeleton

CG_ExtrudeStraightSkeleton — Straight Skeleton Extrusion

☒☒

geometry **CG_ExtrudeStraightSkeleton**(geometry geom, float roof_height, float body_height = 0);

☒☒

Computes an extrusion with a maximal height of the polygon geometry.



☒☒

Perhaps the first (historically) use-case of straight skeletons: given a polygonal roof, the straight skeleton directly gives the layout of each tent. If each skeleton edge is lifted from the plane a height equal to its offset distance, the resulting roof is ”correct” in that water will always fall down to the contour edges (the roof’s border), regardless of where it falls on the roof. The function computes this extrusion aka ”roof” on a polygon. If the argument body_height > 0, so the polygon is extruded like with CG_Extrude(polygon, 0, 0, body_height). The result is an union of these polyhedralsurfaces.

Availability: 3.5.0 - requires SFCGAL >= 1.5.0.

Requires SFCGAL >= 1.5.0

 This method needs SFCGAL backend.

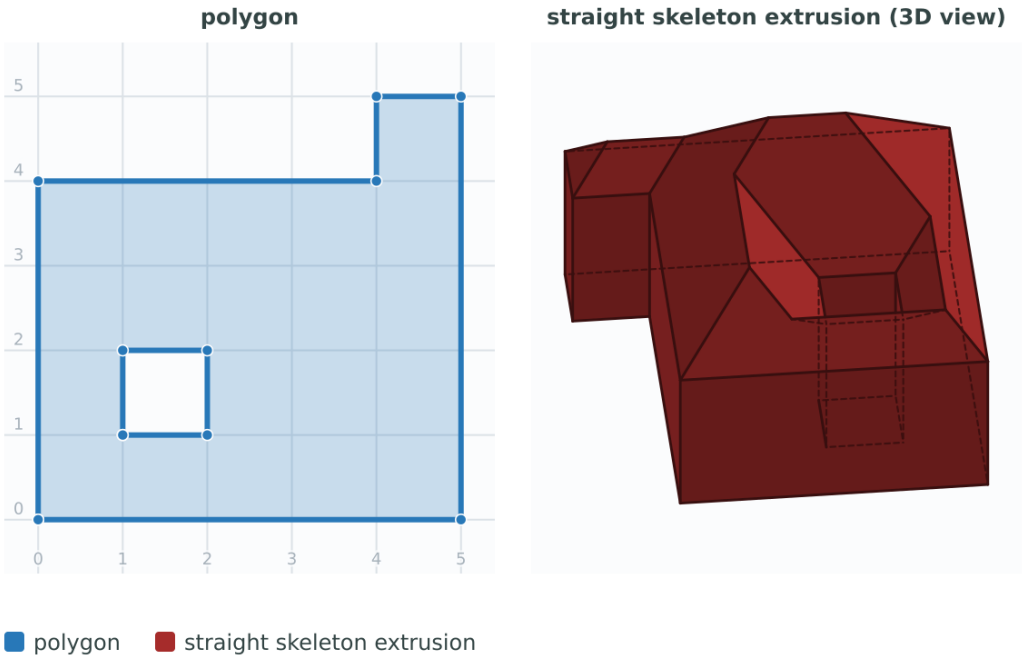
☒☒

Extrude a polygon with a hole into a body and straight-skeleton roof.

Code

```
WITH data AS (
  SELECT 'POLYGON ((0 0,5 0,5 5,4 5,4 4,0 4,0 0),(1 1,1 2,2 2,2 1,1 1))'::geometry AS ↩
    input_polygon
)
SELECT input_polygon AS input_polygon,
       CG_ExtrudeStraightSkeleton(input_polygon, 3.0, 2.0) AS straight_skeleton_extrusion
FROM data;
```

Figure



☒☒

CG_Extrude, CG_ExtrudeStraightSkeleton, CG_StraightSkeleton, CG_StraightSkeletonPartition

8.3.24 CG_GreeneApproxConvexPartition

CG_GreeneApproxConvexPartition — Computes approximal convex partition of the polygon geometry

□□

geometry **CG_GreeneApproxConvexPartition**(geometry geom);

□□

Computes approximal monotone convex partition of the polygon geometry.



□□
A partition of a polygon P is a set of polygons such that the interiors of the polygons do not intersect and the union of the polygons is equal to the interior of the original polygon P. CG_ApproxConvexPartition and CG_GreeneApproxConvexPartition functions produce approximately optimal convex partitions. Both these functions produce convex decompositions by first decomposing the polygon into simpler polygons; CG_ApproxConvexPartition uses a triangulation and CG_GreeneApproxConvexPartition a monotone partition. These two functions both guarantee that they will produce no more than four times the optimal number of convex pieces but they differ in their runtime complexities. Though the triangulation-based approximation algorithm often results in fewer convex pieces, this is not always the case.

Availability: 3.5.0 - requires SFCGAL >= 1.5.0.

Requires SFCGAL >= 1.5.0

 This method needs SFCGAL backend.

□□

Greene's monotone-based approximation uses the same polygon and stays directly comparable in one panel.

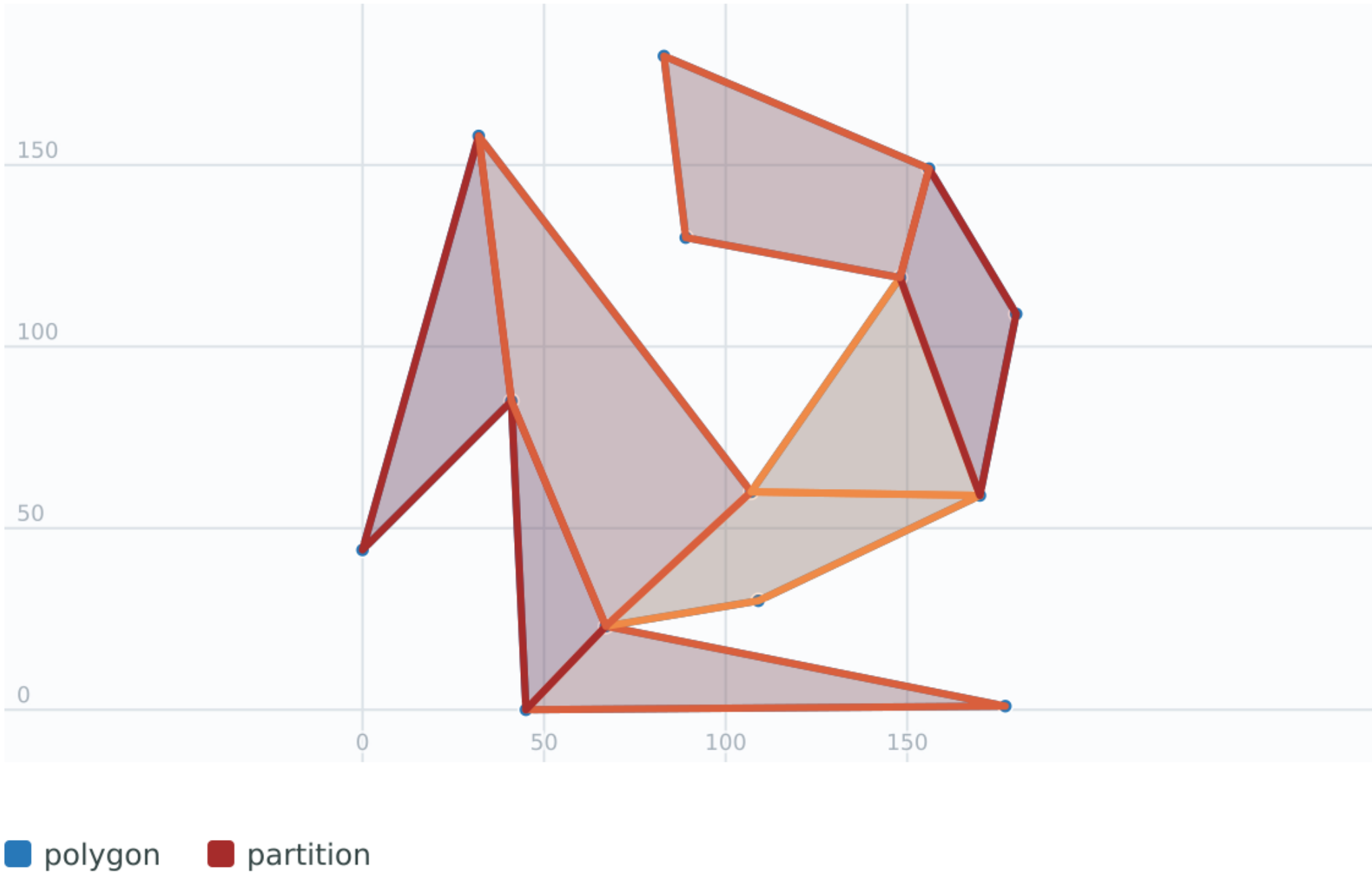
Code

```
WITH data AS (
  SELECT 'POLYGON((156 150,83 181,89 131,148 120,107 61,32 159,0 45,41 86,45 1,177 2,67 ↩
    24,109 31,170 60,180 110,156 150))'::geometry AS input_polygon
)
SELECT input_polygon AS input_polygon,
       CG_GreeneApproxConvexPartition(input_polygon) AS partition
FROM data;
```

Output

```
POLYGON((156 150,83 181,89 131,148 120,107 61,32 159,0 45,41 86,45 1,177 2,67 24,109 31,170 60,180 110,156 150)) | ↩
    GEOMETRYCOLLECTION(POLYGON((32 159,0 45,41 86,32 159)), POLYGON((45 1,177 2,67 24,45 1)), POLYGON((67 24,109 ↩
    31,170 60,107 61,67 24)), POLYGON((41 86,45 1,67 24,41 86)), POLYGON((107 61,32 159,41 86,67 24,107 61)), POLYGON ↩
    ((148 120,107 61,170 60,148 120)), POLYGON((148 120,170 60,180 110,156 150,148 120)), POLYGON((156 150,83 181,89 ↩
    131,148 120,156 150)))
```

Figure



⊠⊠

CG_YMonotonePartition, CG_ApproxConvexPartition, CG_OptimalConvexPartition

8.3.25 CG_MinkowskiSum

CG_MinkowskiSum — ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

geometry **CG_MinkowskiSum**(geometry geom1, geometry geom2);

⊠⊠

⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠ 2 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠⊠ A ⊠ B ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ A ⊠ B ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠ (motion planning) ⊠ CAD(computer-aided design) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠ Wikipedia Minkowski addition ⊠⊠⊠⊠⊠⊠⊠.

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ 2 ⊠⊠⊠⊠ (⊠⊠⊠, ⊠⊠⊠⊠⊠, ⊠⊠⊠) ⊠⊠⊠⊠⊠⊠⊠. ⊠⊠ 3 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, Z ⊠⊠⊠ 0 ⊠⊠⊠⊠⊠⊠ 2 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠ 2 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠⊠⊠⊠ CGAL 2D Minkowskisum ⊠⊠⊠⊠⊠⊠.

Availability: 3.5.0

This method needs SFCGAL backend.

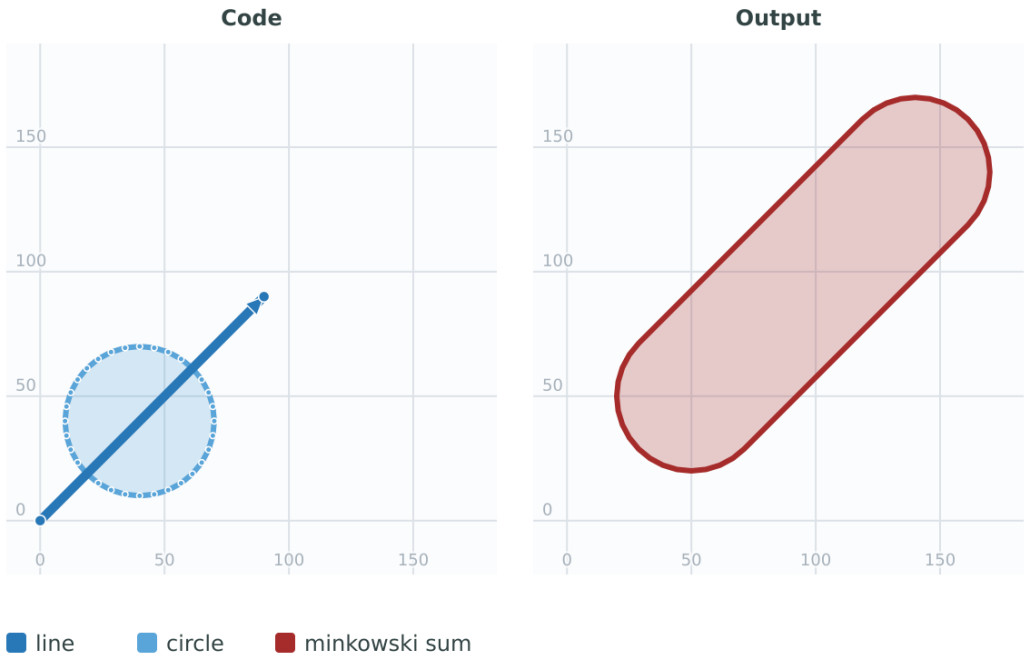
□□

Minkowski sum of a LineString and a circular polygon. The line crosses the polygon, and the result sweeps the polygon along the line.

Code

```
WITH data AS (
  SELECT ST_MakeLine(ST_Point(10, 10), ST_Point(100, 100)) AS line,
         ST_Buffer('POINT(50 50)'::geometry, 30) AS circle
)
SELECT line AS input_line,
       circle AS input_circle,
       CG_MinkowskiSum(line, circle) AS minkowski_sum
FROM data;
```

Figure



Minkowski sum of a MultiPoint and a polygon. The result contains one translated copy of the polygon for each point.

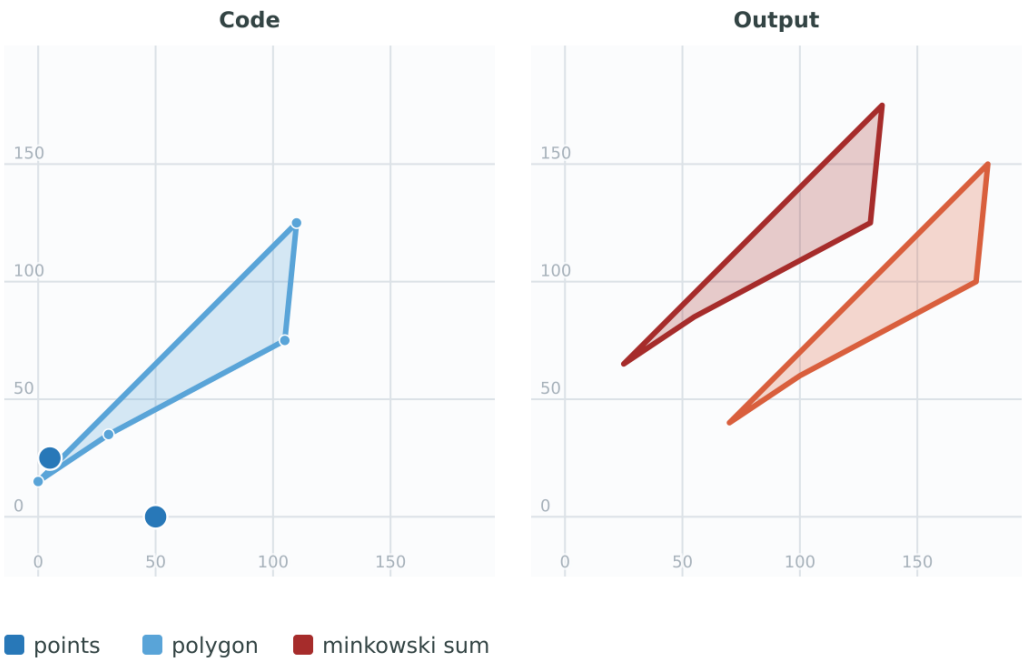
Code

```
WITH data AS (
  SELECT 'MULTIPOINT(25 50,70 25)'::geometry AS points,
         'POLYGON((130 150,20 40,50 60,125 100,130 150))'::geometry AS polygon
)
SELECT points AS input_points,
       polygon AS input_polygon,
       CG_MinkowskiSum(points, polygon) AS minkowski_sum
FROM data;
```

Output

```
MULTIPOINT((25 50),(70 25)) | POLYGON((130 150,20 40,50 60,125 100,130 150)) | MULTIPOLYGON(((45 90,75 110,150 150,155 ↩
     200,45 90)),((90 65,120 85,195 125,200 175,90 65)))
```

Figure



8.3.26 CG_OptimalAlphaShape

CG_OptimalAlphaShape — Computes an Alpha-shape enclosing a geometry using an ”optimal” alpha value.

☒☒

geometry **CG_OptimalAlphaShape**(geometry geom, boolean allow_holes = false, integer nb_components = 1);

☒☒

Computes the ”optimal” alpha-shape of the points in a geometry. The alpha-shape is computed using a value of α chosen so that:

1. the number of polygon elements is equal to or smaller than `nb_components` (which defaults to 1)
2. all input points are contained in the shape

The result will not contain holes unless the optional `allow_holes` argument is specified as true.

Availability: 3.5.0 - requires SFCGAL >= 1.4.1.

This method needs SFCGAL backend.

☒☒

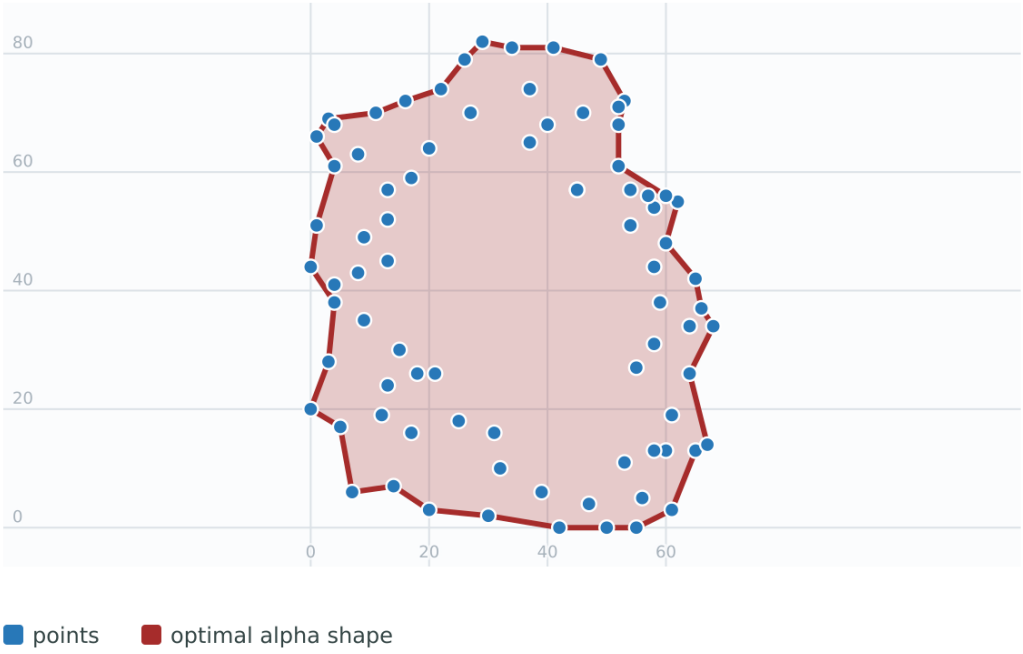
The original point cloud and the optimal alpha-shape share one comparison frame.

Code

```
WITH data AS (
  SELECT 'MULTIPOINT((63 84),(76 88),(68 73),(53 18),(91 50),(81 70),
(88 29),(24 82),(32 51),(37 23),(27 54),(84 19),(75 87),(44 42),(77 67),(90 30),(36 61),(32 ↩
    65),
(81 47),(88 58),(68 73),(49 95),(81 60),(87 50),
(78 16),(79 21),(30 22),(78 43),(26 85),(48 34),(35 35),(36 40),(31 79),(83 29),(27 84),(52 ↩
    98),(72 95),(85 71),
(75 84),(75 77),(81 29),(77 73),(41 42),(83 72),(23 36),(89 53),(27 57),(57 97),(27 77),(39 ↩
    88),(60 81),
(80 72),(54 32),(55 26),(62 22),(70 20),(76 27),(84 35),(87 42),(82 54),(83 64),(69 86),(60 ↩
    90),(50 86),(43 80),(36 73),
(36 68),(40 75),(24 67),(23 60),(26 44),(28 33),(40 32),(43 19),(65 16),(73 16),(38 46),(31 ↩
    59),(34 86),(45 90),(64 97))'::geometry AS input_points
)
SELECT input_points AS input_points,

        CG_OptimalAlphaShape(input_points) AS optimal_alpha_shape
FROM data;
```

Figure



Allowing holes keeps the same outer shell but carves a central hole.

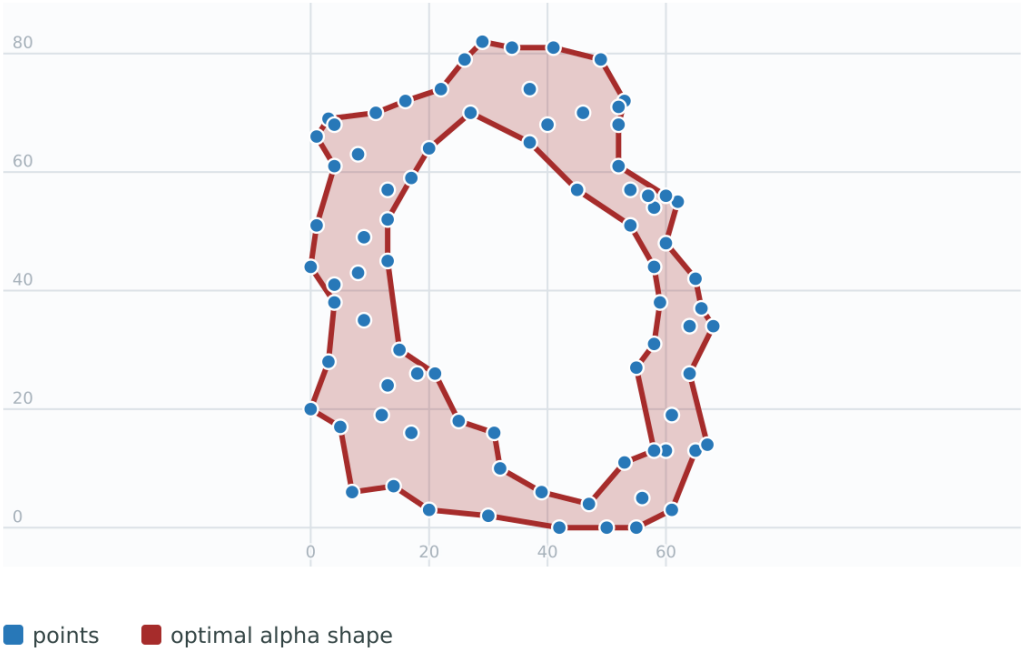
Code

```
WITH data AS (
  SELECT 'MULTIPOINT((63 84),(76 88),(68 73),(53 18),(91 50),(81 70),(88 29),(24 82),(32  ↩
    51),(37 23),(27 54),(84 19),(75 87),(44 42),(77 67),(90 30),(36 61),(32 65),(81 47),(88  ↩
    58),(68 73),(49 95),(81 60),(87 50),
(78 16),(79 21),(30 22),(78 43),(26 85),(48 34),(35 35),(36 40),(31 79),(83 29),(27 84),(52 ↩
    98),(72 95),(85 71),
(75 84),(75 77),(81 29),(77 73),(41 42),(83 72),(23 36),(89 53),(27 57),(57 97),(27 77),(39 ↩
    88),(60 81),
(80 72),(54 32),(55 26),(62 22),(70 20),(76 27),(84 35),(87 42),(82 54),(83 64),(69 86),(60 ↩
    90),(50 86),(43 80),(36 73),
(36 68),(40 75),(24 67),(23 60),(26 44),(28 33),(40 32),(43 19),(65 16),(73 16),(38 46),(31 ↩
    59),(34 86),(45 90),(64 97))'::geometry AS input_points
```

```
)
SELECT input_points AS input_points,

         CG_OptimalAlphaShape(input_points, allow_holes => true) AS optimal_alpha_shape
FROM data;
```

Figure



□□

ST_ConcaveHull, CG_AlphaShape

8.3.27 CG_OptimalConvexPartition

CG_OptimalConvexPartition — Computes an optimal convex partition of the polygon geometry

□□

geometry **CG_OptimalConvexPartition**(geometry geom);

□□

Computes an optimal convex partition of the polygon geometry.



□□
A partition of a polygon P is a set of polygons such that the interiors of the polygons do not intersect and the union of the polygons is equal to the interior of the original polygon P. CG_OptimalConvexPartition produces a partition that is optimal in the number of pieces.

Availability: 3.5.0 - requires SFCGAL >= 1.5.0.

Requires SFCGAL >= 1.5.0

This method needs SFCGAL backend.

□□

The optimal convex partition uses the same polygon and produces the smallest documented piece count.

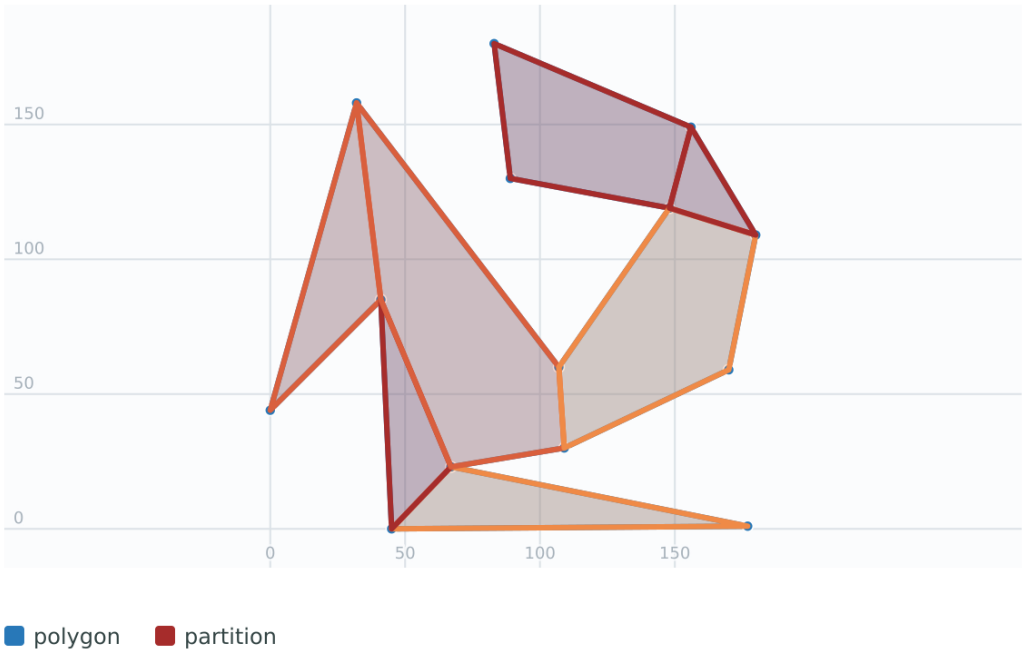
Code

```
WITH data AS (
  SELECT 'POLYGON((156 150,83 181,89 131,148 120,107 61,32 159,0 45,41 86,45 1,177 2,67 ↩
    24,109 31,170 60,180 110,156 150))'::geometry AS input_polygon
)
SELECT input_polygon AS input_polygon,
       CG_OptimalConvexPartition(input_polygon) AS partition
FROM data;
```

Output

```
POLYGON((156 150,83 181,89 131,148 120,107 61,32 159,0 45,41 86,45 1,177 2,67 24,109 31,170 60,180 110,156 150)) | ↩
    GEOMETRYCOLLECTION(POLYGON((156 150,83 181,89 131,148 120,156 150)), POLYGON((32 159,0 45,41 86,32 159)), POLYGON ↩
    ((45 1,177 2,67 24,45 1)), POLYGON((41 86,45 1,67 24,41 86)), POLYGON((107 61,32 159,41 86,67 24,109 31,107 61)), ↩
     POLYGON((148 120,107 61,109 31,170 60,180 110,148 120)), POLYGON((156 150,148 120,180 110,156 150)))
```

Figure



□□

CG_YMonotonePartition, CG_ApproxConvexPartition, CG_GreeneApproxConvexPartition

8.3.28 CG_StraightSkeleton

CG_StraightSkeleton — □□□□□□□□□□ (straight skeleton) □□□□□□.

□□

geometry **CG_StraightSkeleton**(geometry geom, boolean use_distance_as_m = false);

□□

Computes the straight skeleton of a polygon by tracing the inward propagation of its edges. When *use_distance_as_m* is `true`, the M coordinate of each output point stores its distance from the input boundary.

Availability: 3.5.0

Requires SFCGAL >= 1.3.8 for option use_distance_as_m



□□
This function ignores the Z dimension. It always gives a 2D result even when used on a 3D geometry.

 This method needs SFCGAL backend.

 This function supports Polyhedral surfaces.

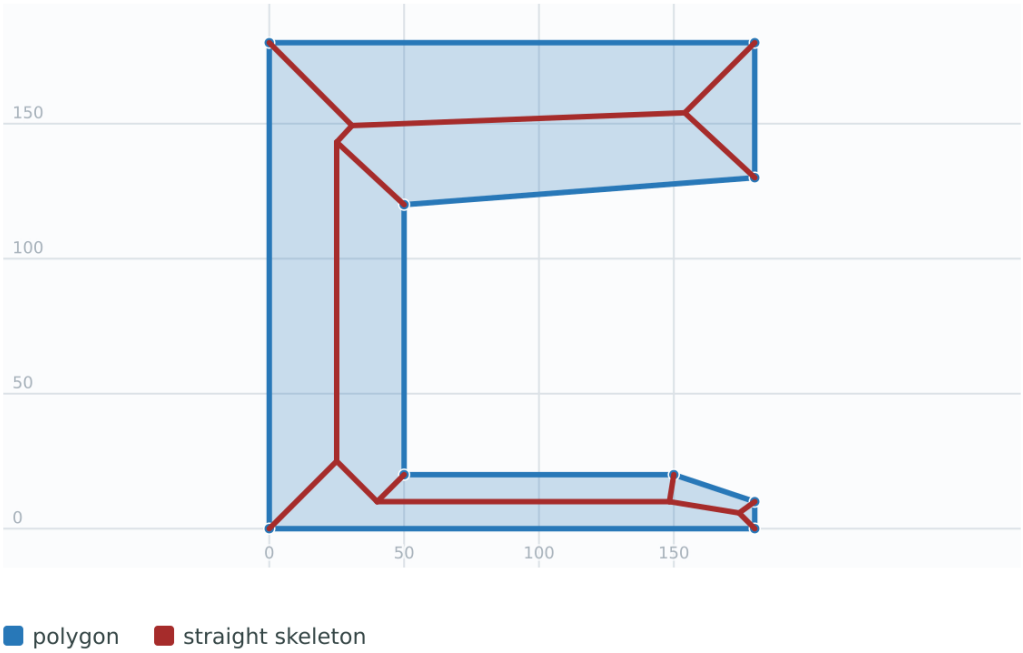
 This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).

□□

Code

```
WITH data AS (
  SELECT 'POLYGON((190 190,10 190,10 10,190 10,190 20,160 30,60 30,60 130,190 140,190 190)) ↩
    '::geometry AS polygon
)
SELECT CG_StraightSkeleton(polygon) AS straight_skeleton
FROM data;
```

Figure



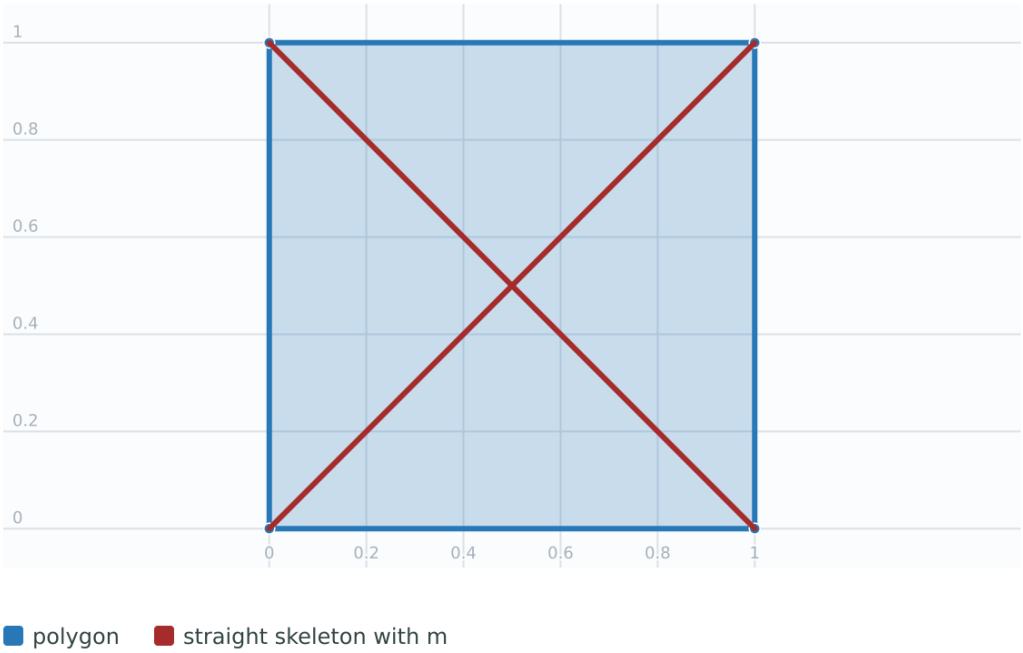
Code

```
WITH data AS (
  SELECT 'POLYGON((0 0,1 0,1 1,0 1,0 0))'::geometry AS polygon
)
SELECT CG_StraightSkeleton(polygon, true) AS straight_skeleton_with_m
FROM data;
```

Output

```
MULTILINESTRING M ((0 0 0,0.5 0.5 0.5),(1 0 0,0.5 0.5 0.5),(1 1 0,0.5 0.5 0.5),(0 1 0,0.5 0.5 0.5))
```

Figure



Note that valid inputs with rings that touch at a single point will raise an error.

Code

```
SELECT CG_StraightSkeleton('POLYGON((0 0,3 0,3 3,0 3,0 0),(0 0,1 2,2 1,0 0))'));
```

Output

```
NOTICE:  During straight_skeleton(A) :
NOTICE:    with A: POLYGON((0/1 0/1,3/1 0/1,3/1 3/1,0/1 3/1,0/1 0/1),(0/1 0/1,1/1 2/1,2/1 1/1,0/1 0/1))
ERROR:  straight skeleton of Polygon with point touching rings is not implemented.
```

□□

CG_StraightSkeletonPartition, CG_ExtrudeStraightSkeleton

8.3.29 CG_Tesselate

CG_Tesselate — □□□□□□□□□□□□□□□□□□ (tessellation) □□□□□□□□ TIN □□ TIN □□□□ □□□□□□.

□□

geometry **CG_Tesselate**(geometry geom);

□□

[□□] □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ (□□□□) □□□□□□ TIN □□□□ □□□□□□.



☒☒
ST_TriangulatePolygon does similar to this function except that it returns a geometry collection of polygons instead of a TIN and also only works with 2D geometries.

Availability: 3.5.0

 This method needs SFCGAL backend.

 This function supports 3d and will not drop the z-index.

 This function supports Polyhedral surfaces.

 This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).

☒☒

Tessellation divides each face of a cube into triangles.

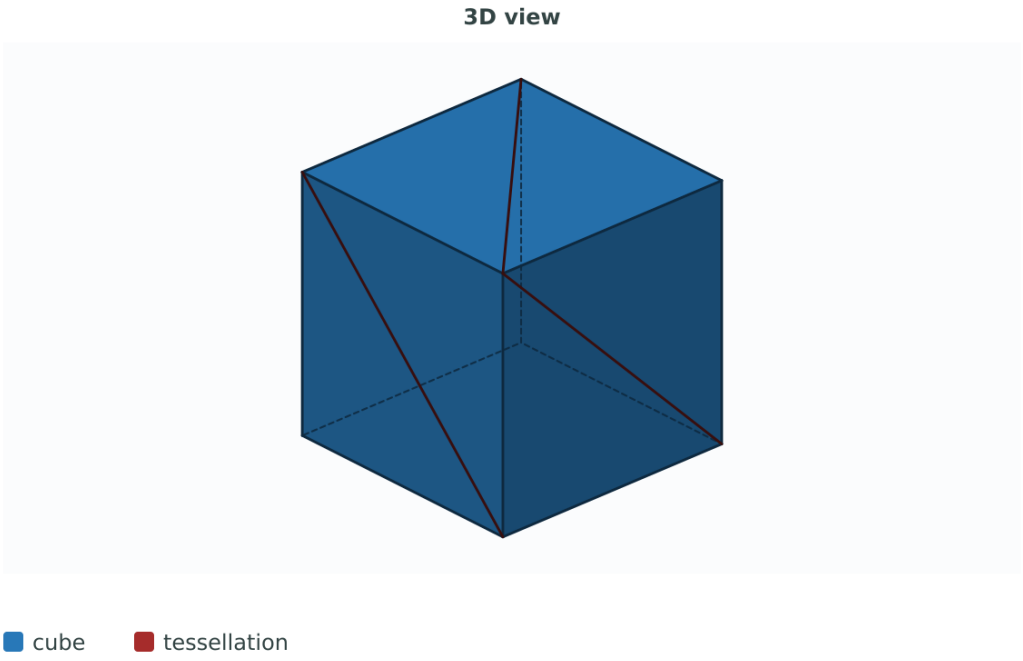
Code

```
WITH data AS (
  SELECT 'POLYHEDRALSURFACE Z( ((0 0 0,0 0 1,0 1 1,0 1 0,0 0 0)),
((0 0 0,0 1 0,1 1 0,1 0 0,0 0 0)),((0 0 0,1 0 0,1 0 1,0 0 1,0 0 0)),
((1 1 0,1 1 1,1 0 1,1 0 0,1 1 0)),
((0 1 0,0 1 1,1 1 1,1 1 0,0 1 0)),((0 0 1,1 0 1,1 1 1,0 1 1,0 0 1)) )'::geometry AS cube
)
SELECT CG_Tesselate(cube) AS tessellation
FROM data;
```

Output

```
TIN Z (((0 0 0,0 0 1,0 1 1,0 0 0)),((0 1 0,0 0 0,0 1 1,0 1 0)),((0 0 0,0 1 0,1 1 0,0 0 0)),((1 0 0,0 0 0,1 1 0,1 0 0)) ↩
    ,((0 0 1,1 0 0,1 0 1,0 0 1)),((0 0 1,0 0 0,1 0 0,0 0 1)),((1 1 0,1 1 1,1 0 1,1 1 0)),((1 0 0,1 1 0,1 0 1,1 0 0)) ↩
    ,((0 1 0,0 1 1,1 1 1,1 0 1)),((1 1 0,0 1 0,1 1 1,1 1 0)),((0 1 1,1 0 1,1 1 1,0 1 1)),((0 1 1,0 0 1,1 0 1,0 1 1)))
```

Figure



Tessellation of a Polygon.

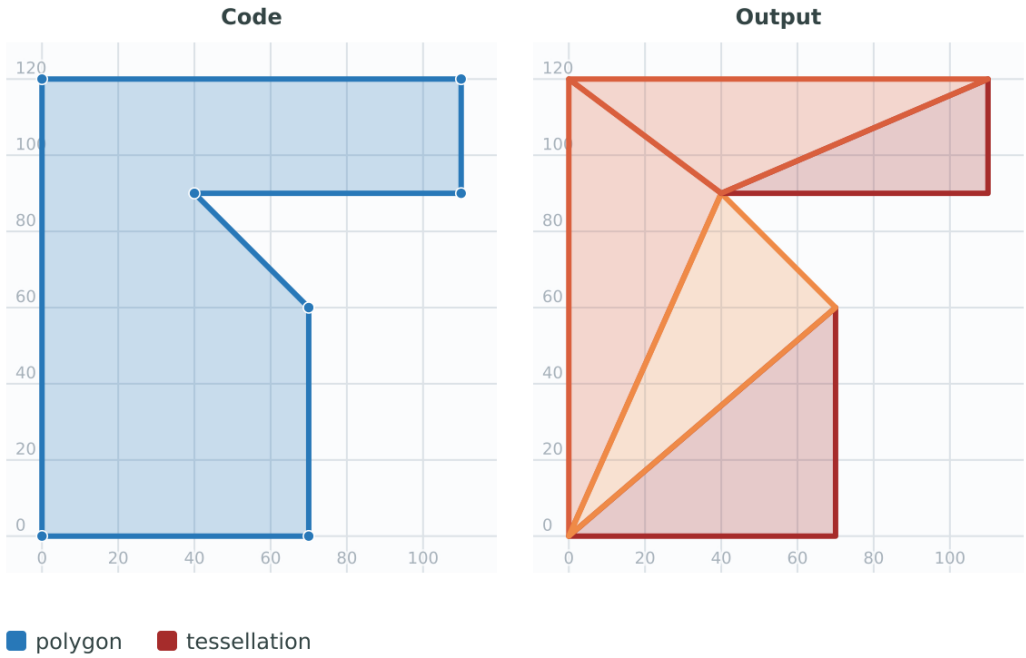
Code

```
WITH data AS (
  SELECT 'POLYGON((10 190,10 70,80 70,80 130,50 160,120 160,120 190,10 190))'::geometry AS
    polygon
)
SELECT polygon AS input_polygon,
       CG_Tesselate(polygon) AS tessellation
FROM data;
```

Output

```
POLYGON((10 190,10 70,80 70,80 130,50 160,120 160,120 190,10 190)) | TIN(((80 130,10 70,80 70,80 130)),((50 160,10
    190,10 70,50 160)),((80 130,50 160,10 70,80 130)),((120 160,120 190,50 160,120 160)),((120 190,10 190,50 160,120
    190)))
```

Figure



□□

CG_ConstrainedDelaunayTriangles, ST_DelaunayTriangles, ST_TriangulatePolygon

8.3.30 CG_Triangulate

CG_Triangulate — Triangulates a polygonal geometry

□□

geometry **CG_Triangulate**(geometry geom);

□□

Triangulates a polygonal geometry.
Performed by the SFCGAL module



□□
NOTE: this function returns a geometry representing the triangulated result.

Availability: 3.5.0



This method needs SFCGAL backend.

□□□□

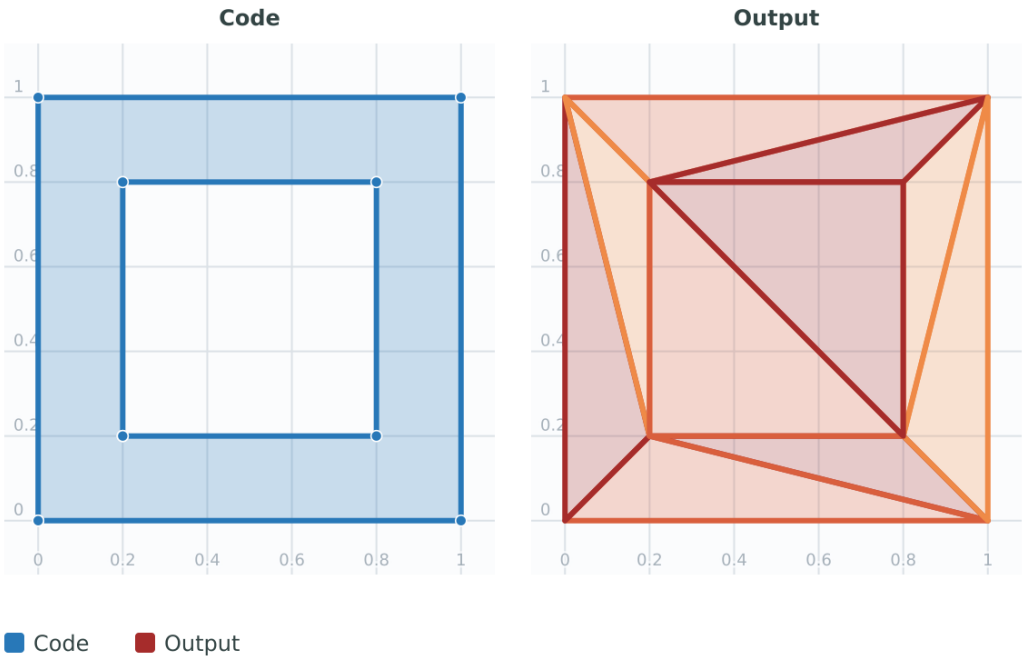
Code

```
SELECT CG_Triangulate(
  'POLYGON((0 0,1 0,1 1,0 1,0 0),
           (0.2 0.2,0.2 0.8,0.8 0.8,0.8 0.2,0.2 0.2))'
);
```

Output

```
TIN(((0.8 0.2,0.2 0.2,1 0,0.8 0.2)),((0.2 0.2,0 0,1 0,0.2 0.2)),((1 1,0.8 0.8,0.8 0.2,1 1)),((0 1,0 0,0.2 0.2,0 1)) ↩
    ,((0 1,0.2 0.8,1 1,0 1)),((0 1,0.2 0.2,0.2 0.8,0 1)),((0.2 0.8,0.8 0.8,1 1,0.2 0.8)),((0.2 0.8,0.2 0.2,0.8 ↩
    0.2,0.2 0.8)),((1 1,0.8 0.2,1 0,1 1)),((0.8 0.8,0.2 0.8,0.8 0.2,0.8 0.8)))
```

Figure



□□

CG_ConstrainedDelaunayTriangles, ST_DelaunayTriangles, ST_TriangulatePolygon

8.3.31 CG_Visibility

CG_Visibility — Compute a visibility polygon from a point or a segment in a polygon geometry

□□

geometry **CG_Visibility**(geometry polygon, geometry point);
geometry **CG_Visibility**(geometry polygon, geometry pointA, geometry pointB);

□□

Returns the portion of a polygon visible from an observer point, or from the observer segment defined by *pointA* and *pointB*.

Availability: 3.5.0 - requires SFCGAL >= 1.5.0.

Requires SFCGAL >= 1.5.0

This method needs SFCGAL backend.

This function supports 3d and will not drop the z-index.

This function supports Polyhedral surfaces.

This function supports Triangles and Triangulated Irregular Network Surfaces (TIN).

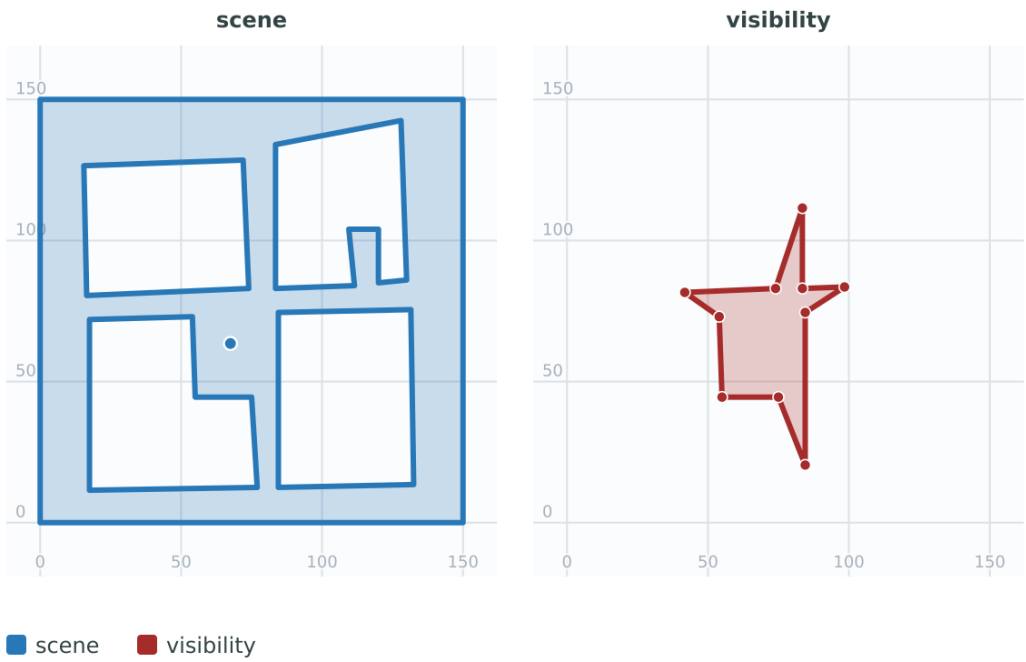
□□

Visibility from a point inside a polygon with holes.

Code

```
WITH data AS (
  SELECT 'POLYGON((23.5 23.5,23.5 173.5,173.5 173.5,173.5 23.5,23.5 23.5),(108 98,108 ↩
    36,156 37,155 99,108 98),(107 157.5,107 106.5,135 107.5,133 127.5,143.5 127.5,143.5 ↩
    108.5,153.5 109.5,151.5 166,107 157.5),(41 95.5,41 35,100.5 36,98.5 68,78.5 68,77.5 ↩
    96.5,41 95.5),(39 150,40 104,97.5 106.5,95.5 152,39 150))'::geometry AS polygon,
         'POINT(91 87)'::geometry AS observer
)
SELECT ST_Collect(polygon, observer) AS input_scene,
       CG_Visibility(polygon, observer) AS visibility
FROM data;
```

Figure

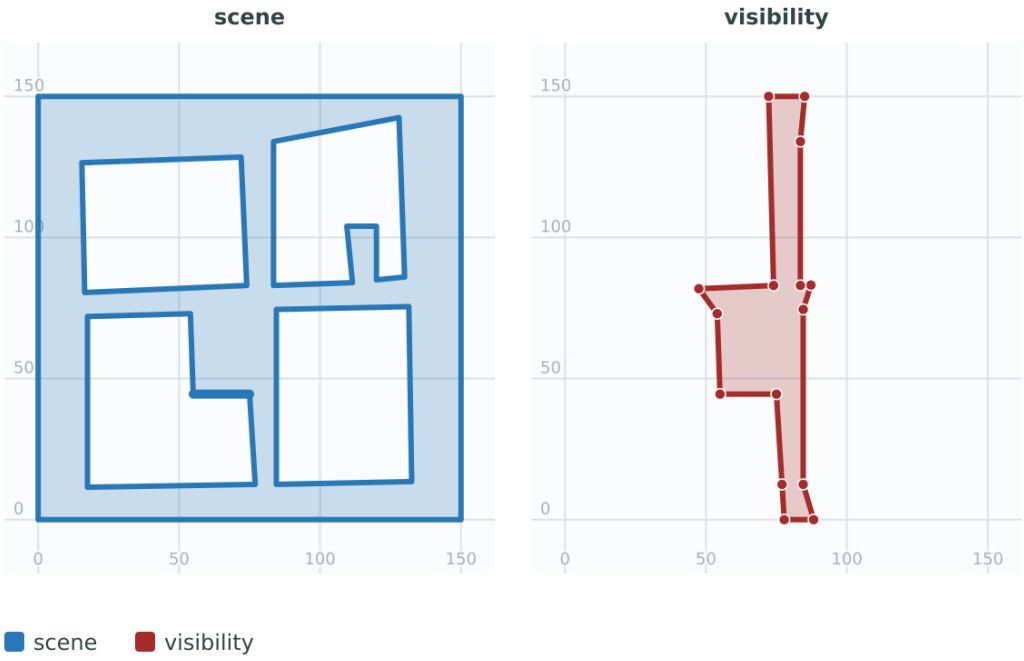


Visibility from a segment inside the same polygon.

Code

```
WITH data AS (
  SELECT 'POLYGON((23.5 23.5,23.5 173.5,173.5 173.5,173.5 23.5,23.5 23.5),(108 98,108 ↩
    36,156 37,155 99,108 98),(107 157.5,107 106.5,135 107.5,133 127.5,143.5 127.5,143.5 ↩
    108.5,153.5 109.5,151.5 166,107 157.5),(41 95.5,41 35,100.5 36,98.5 68,78.5 68,77.5 ↩
    96.5,41 95.5),(39 150,40 104,97.5 106.5,95.5 152,39 150))'::geometry AS polygon,
         'LINESTRING(78.5 68,98.5 68)'::geometry AS observer
)
SELECT ST_Collect(polygon, observer) AS input_scene,
       CG_Visibility(polygon,
                     ST_StartPoint(observer),
                     ST_EndPoint(observer)) AS visibility
FROM data;
```

Figure



8.3.32 CG_YMonotonePartition

CG_YMonotonePartition — Computes y-monotone partition of the polygon geometry

☒☒

geometry **CG_YMonotonePartition**(geometry geom);

☒☒

Computes y-monotone partition of the polygon geometry.



☒☒
A partition of a polygon P is a set of polygons such that the interiors of the polygons do not intersect and the union of the polygons is equal to the interior of the original polygon P. A y-monotone polygon is a polygon whose vertices v1,...,vn can be divided into two chains v1,...,vk and vk,...,vn,v1, such that any horizontal line intersects either chain at most once. This algorithm does not guarantee a bound on the number of polygons produced with respect to the optimal number.

Availability: 3.5.0 - requires SFCGAL >= 1.5.0.

Requires SFCGAL >= 1.5.0

 This method needs SFCGAL backend.

☒☒

The same polygon can also be partitioned into y-monotone pieces for downstream processing.

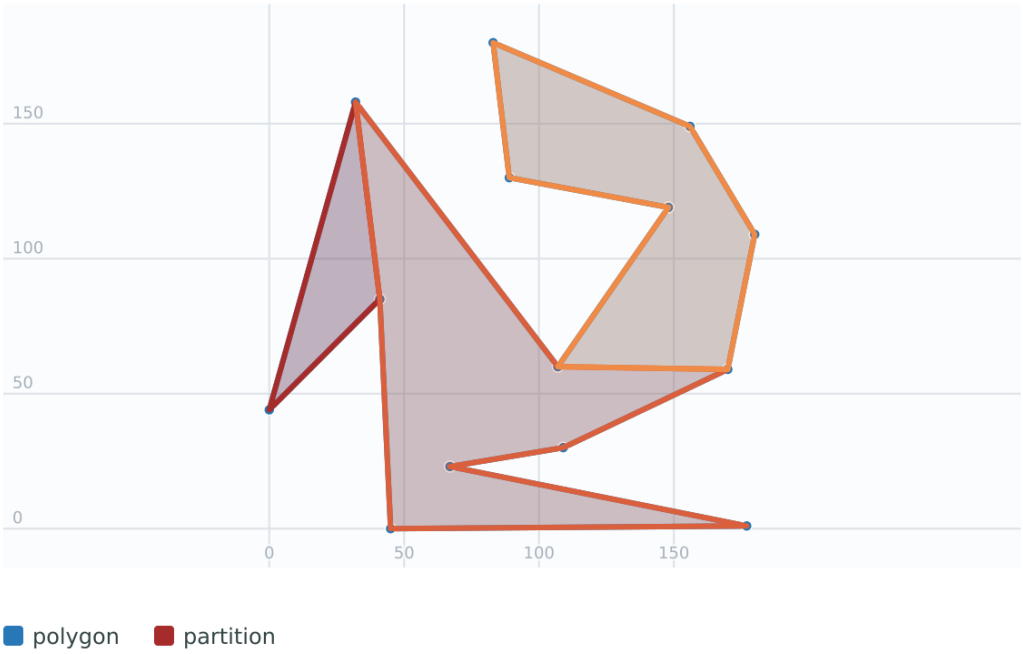
Code

```
WITH data AS (
  SELECT 'POLYGON((156 150,83 181,89 131,148 120,107 61,32 159,0 45,41 86,45 1,177 2,67 ↩
    24,109 31,170 60,180 110,156 150))'::geometry AS input_polygon
)
SELECT input_polygon AS input_polygon,
       CG_YMonotonePartition(input_polygon) AS partition
FROM data;
```

Output

```
POLYGON((156 150,83 181,89 131,148 120,107 61,32 159,0 45,41 86,45 1,177 2,67 24,109 31,170 60,180 110,156 150)) | ↩
    GEOMETRYCOLLECTION(POLYGON((32 159,0 45,41 86,32 159)), POLYGON((107 61,32 159,41 86,45 1,177 2,67 24,109 31,170 ↩
    60,107 61)), POLYGON((156 150,83 181,89 131,148 120,107 61,170 60,180 110,156 150)))
```

Figure



☒☒

CG_ApproxConvexPartition, CG_GreeneApproxConvexPartition, CG_OptimalConvexPartition

8.3.33 CG_StraightSkeletonPartition

CG_StraightSkeletonPartition — Computes the straight skeleton partition of a polygon.

☒☒

geometry **CG_StraightSkeletonPartition**(geometry geom, boolean auto_orientation);

□□

Computes the straight skeleton partition of the input polygon geometry *geom*. The straight skeleton is a partitioning of the polygon into faces formed by tracing the collapse of its edges. If *auto_orientation* is set to true, the function will automatically adjust the orientation of the input polygon to ensure correct results.

Availability: 3.6.0 - requires SFCGAL >= 2.0.0.

This method needs SFCGAL backend.

□□

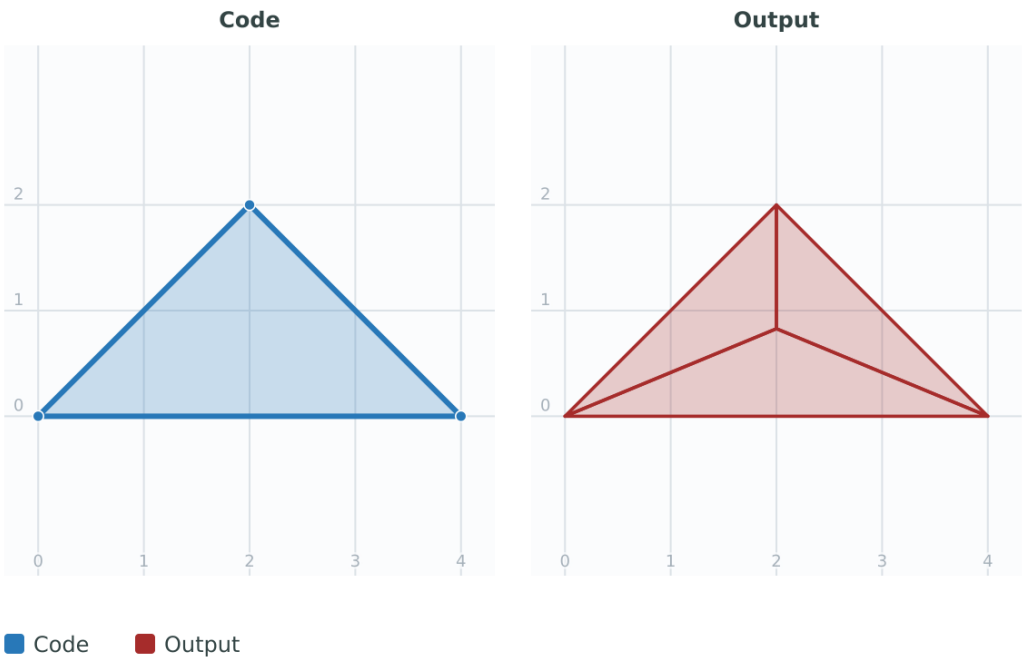
Code

```
SELECT CG_StraightSkeletonPartition('POLYGON((0 0,4 0,2 2,0 0))', true);
```

Output

```
POLYHEDRALSURFACE(((0 0,2 0.82842712474619,2 2,0 0)),((4 0,2 0.82842712474619,0 0,4 0)),((2 2,2 0.82842712474619,4 0,2 ↩
    2)))
```

Figure

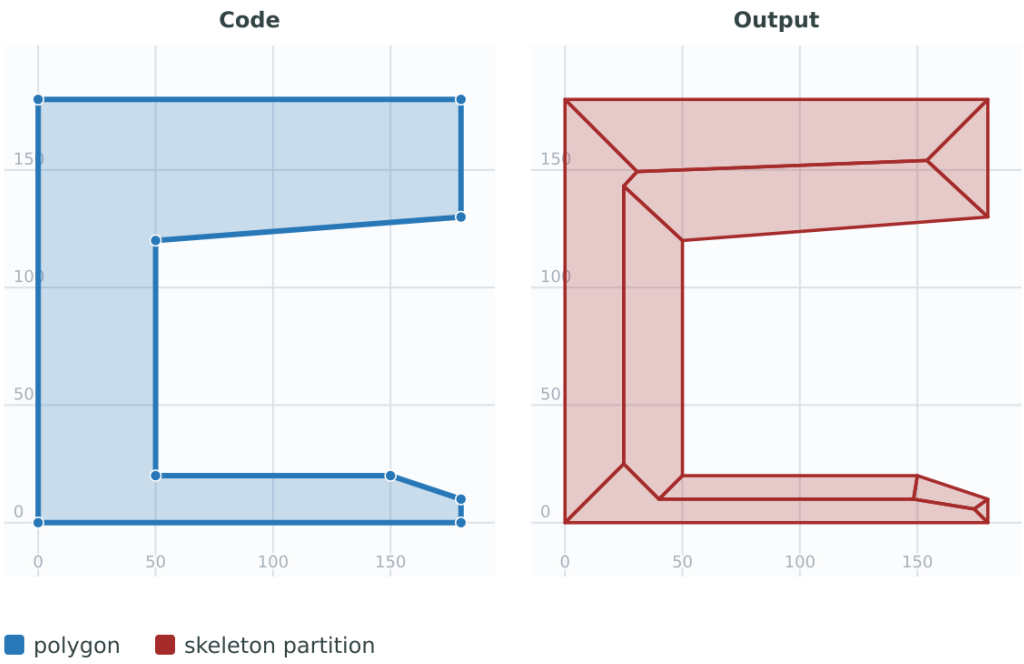


Straight skeleton partition of a polygon.

Code

```
WITH data AS (
  SELECT 'POLYGON((190 190,10 190,10 10,190 10,190 20,160 30,60 30,60 130,190 140,190 190)) ↩
    '::geometry AS polygon
)
SELECT polygon AS input_polygon,
       CG_StraightSkeletonPartition(polygon, true) AS skeleton_partition
FROM data;
```

Figure



☒☒

CG_StraightSkeleton, ST_Polygonize

8.3.34 CG_3DBuffer

CG_3DBuffer — Computes a 3D buffer around a geometry.

☒☒

geometry **CG_3DBuffer**(geometry geom, float8 radius, integer segments, integer buffer_type);

☒☒

Generates a 3D buffer around the input geometry *geom* with a specified *radius*. The buffer is constructed in 3D space, creating a volumetric representation of the geometry's surroundings. The *segments* parameter defines the number of segments used to approximate the curved sections of the buffer, with a minimum value of 4 segments required. The *buffer_type* specifies the type of buffer to create: 0: Rounded buffer (default) 1: Flat buffer 2: Square buffer

Input geometry must be a Point or LineString.

Availability: 3.6.0 - requires SFCGAL >= 2.0.0

This method needs SFCGAL backend.

☒☒

Code

```
SELECT CG_3DBuffer('POINT(0 0 0)', 1, 8, 0);
```

Output

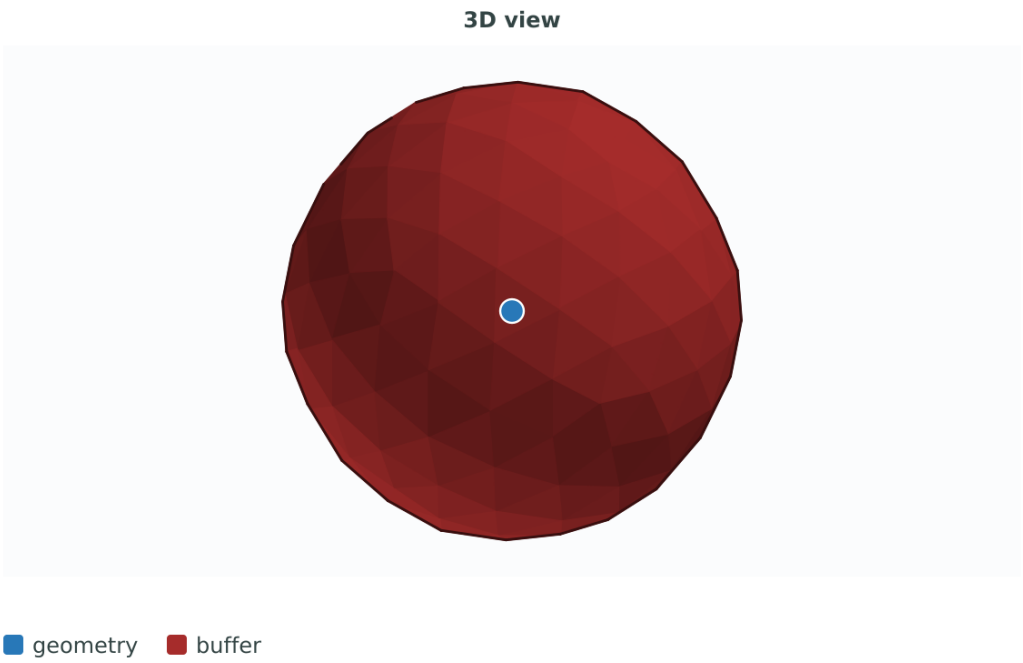
```
POLYHEDRALSURFACE Z (((0 0 1, 0.5 -0.5 0.71, 0 -0.71 0.71, 0 0 1)), ... )
```

A rounded point buffer using 32 segments.

Code

```
SELECT
  'POINT(100 90)'::geometry AS input_geometry,
  CG_3DBuffer('POINT(100 90)'::geometry, 50, 32, 0) AS buffer;
```

Figure

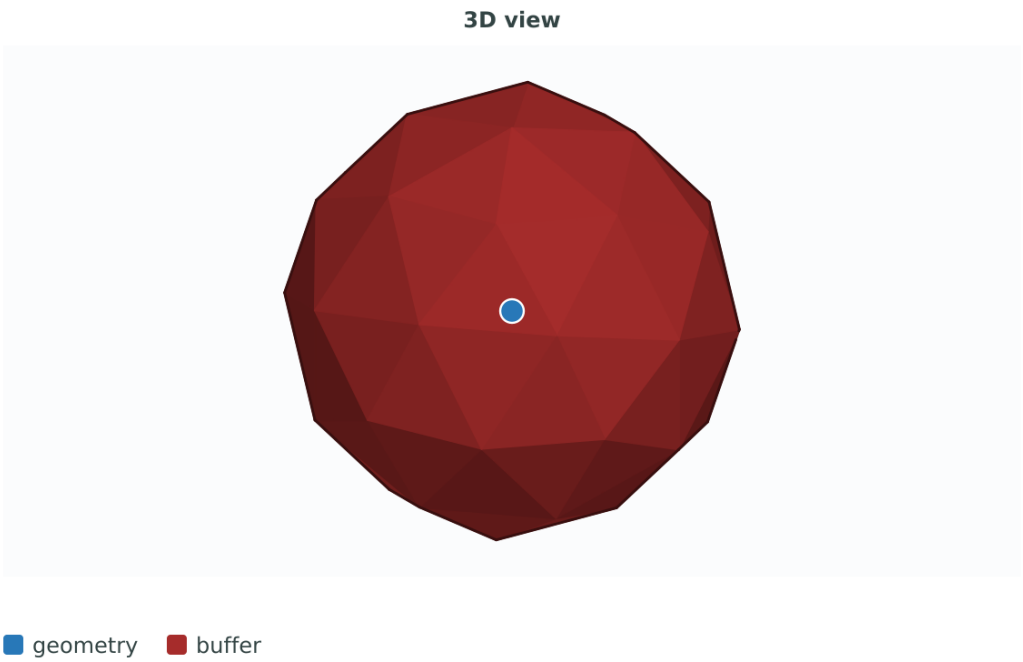


A coarser rounded point buffer using 5 segments.

Code

```
SELECT
  'POINT(100 90)'::geometry AS input_geometry,
  CG_3DBuffer('POINT(100 90)'::geometry, 50, 5, 0) AS buffer;
```

Figure

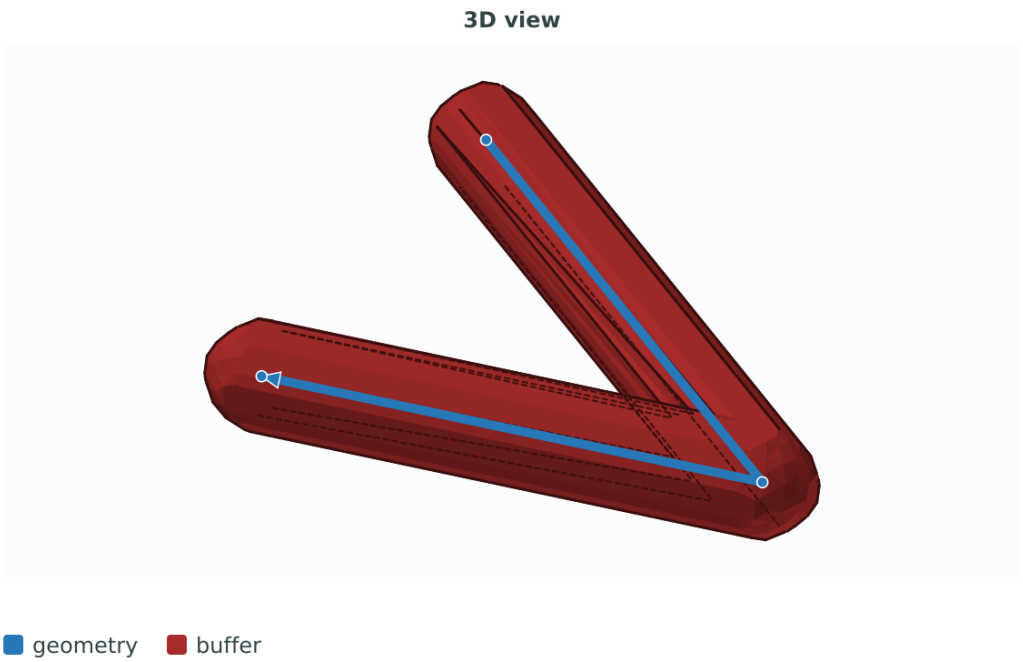


A rounded buffer around a line using 32 segments.

Code

```
SELECT
  'LINESTRING(50 50,150 150,150 50)'::geometry AS input_geometry,
  CG_3DBuffer('LINESTRING(50 50,150 150,150 50)'::geometry, 10, 32, 0) AS buffer;
```

Figure

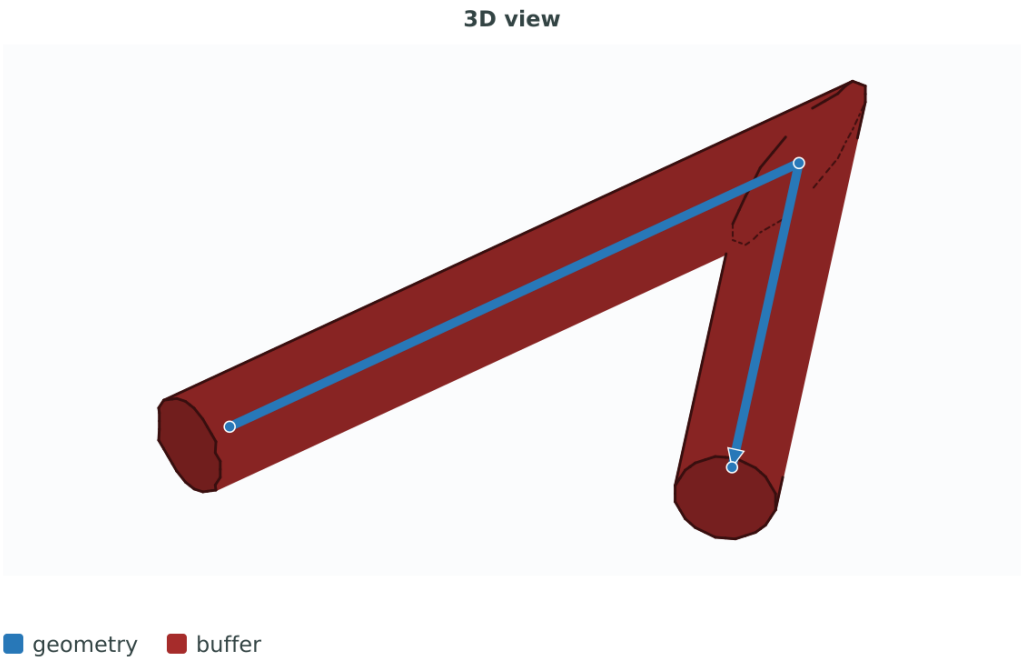


A square buffer around the same line using 32 segments.

Code

```
SELECT
  'LINESTRING(50 50,150 150,150 50)'::geometry AS input_geometry,
  CG_3DBuffer('LINESTRING(50 50,150 150,150 50)'::geometry, 10, 32, 2) AS buffer;
```

Figure



□□

ST_Buffer, CG_3DConvexHull, ST_AsX3D

8.3.35 CG_Rotate

CG_Rotate — Rotates a geometry by a given angle around the origin (0,0).

□□

geometry **CG_Rotate**(geometry geom, float8 angle);

□□

Rotates the input geometry *geom* by *angle* radians around the origin point (0,0). The rotation is performed in 2D space; Z coordinates are not modified. Positive angles rotate the geometry counterclockwise.

Availability: 3.6.0 - requires SFCGAL >= 2.0.0

This method needs SFCGAL backend.

□□

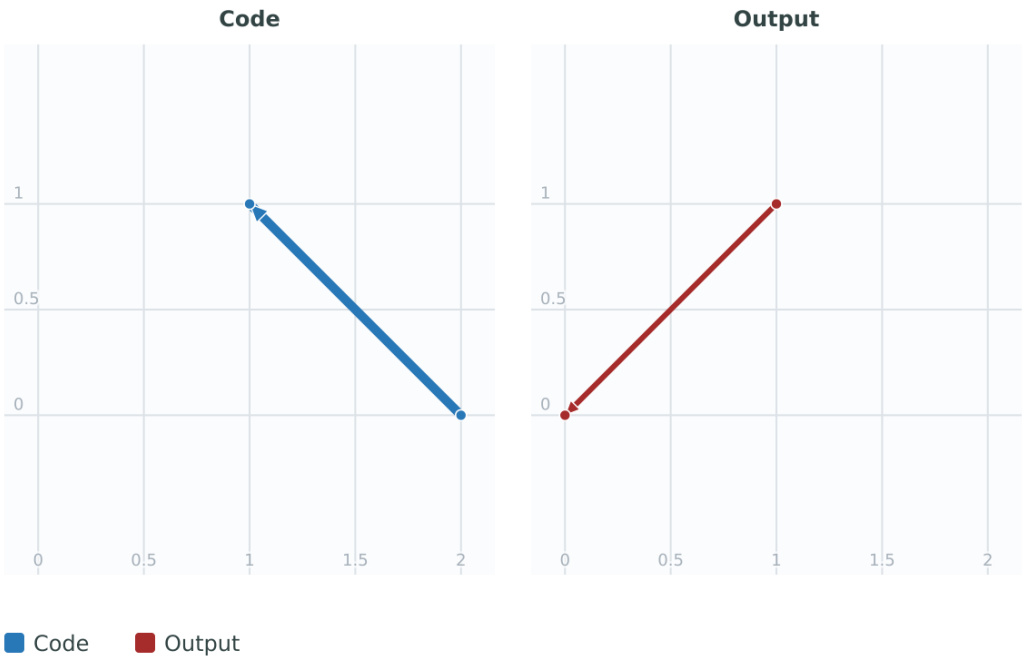
Code

```
SELECT CG_Rotate('LINESTRING(1 0,0 1)', pi()/2);
```

Output

```
LINESTRING(6.123233995736766e-17 1,-1 6.123233995736766e-17)
```

Figure



☒☒

CG_2DRotate, ST_Rotate

8.3.36 CG_2DRotate

CG_2DRotate — Rotates a geometry by a given angle around a specified point in 2D.

☒☒

geometry **CG_2DRotate**(geometry geom, float8 angle, float8 cx, float8 cy);

☒☒

Rotates the input geometry *geom* by *angle* radians around the point (*cx*, *cy*). The rotation is performed in 2D space; Z coordinates are dropped. Positive angles rotate the geometry counter-clockwise.

Availability: 3.6.0 - requires SFCGAL >= 2.0.0

This method needs SFCGAL backend.

☒☒

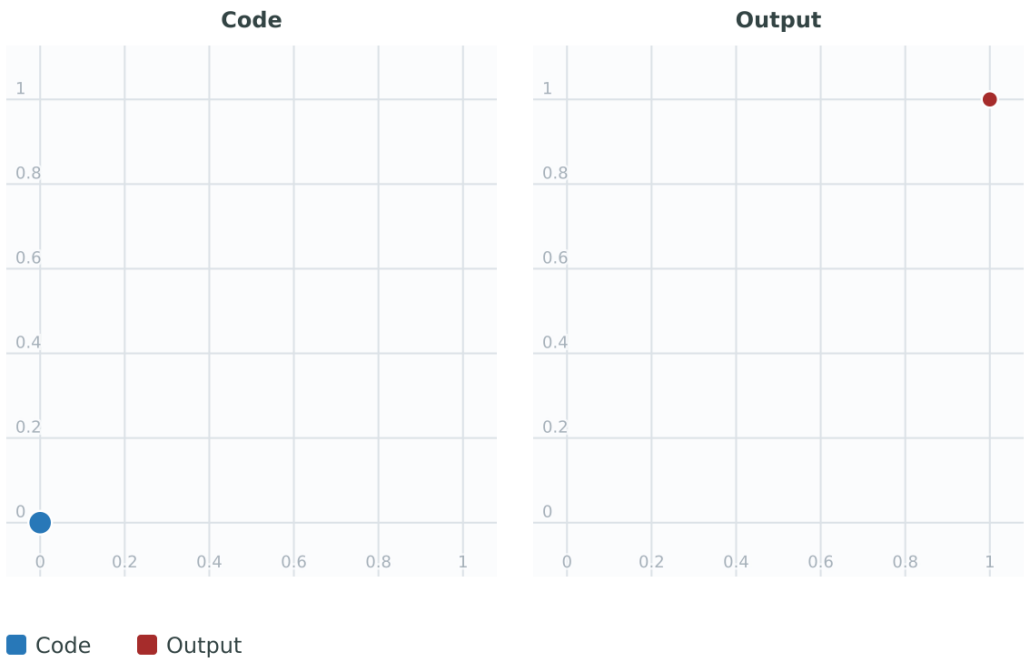
Code

```
SELECT CG_2DRotate('POINT(1 0)', pi()/2, 1, 1);
```

Output

```
POINT(2 1)
```

Figure



☒☒

CG_Rotate, CG_3DRotate

8.3.37 CG_3DRotate

CG_3DRotate — Rotates a geometry in 3D space around an axis vector.

☒☒

geometry **CG_3DRotate**(geometry geom, float8 angle, float8 ax, float8 ay, float8 az);

☒☒

Rotates the input geometry *geom* by *angle* radians around an axis defined by the vector (*ax*, *ay*, *az*) passing through the origin (0,0,0).

Availability: 3.6.0 - requires SFCGAL >= 2.0.0

This method needs SFCGAL backend.

This function supports 3d and will not drop the z-index.

□□

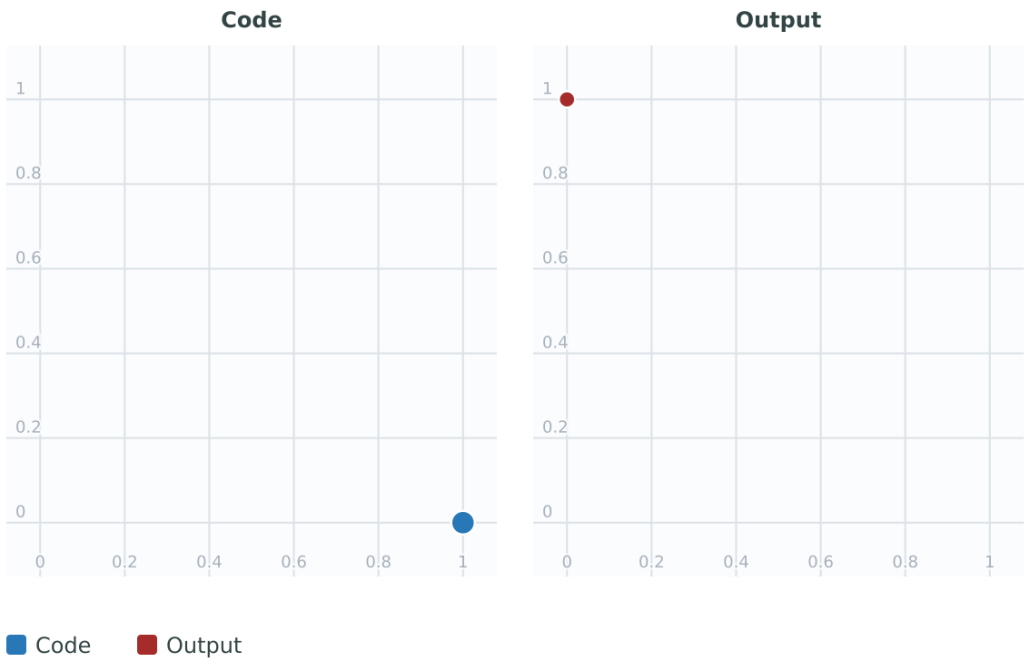
Code

```
SELECT CG_3DRotate('POINT(1 0 0)', pi()/2, 0, 0, 1);
```

Output

```
POINT Z (6.123233995736766e-17 1 0)
```

Figure



□□

CG_RotateX, CG_RotateY, CG_RotateZ

8.3.38 CG_RotateX

CG_RotateX — Rotates a geometry around the X-axis by a given angle.

□□

geometry **CG_RotateX**(geometry geom, float8 angle);

□□

Rotates the input geometry *geom* by *angle* radians around the X-axis.
Availability: 3.6.0 - requires SFCGAL >= 2.0.0

This method needs SFCGAL backend.

This function supports 3d and will not drop the z-index.

□□

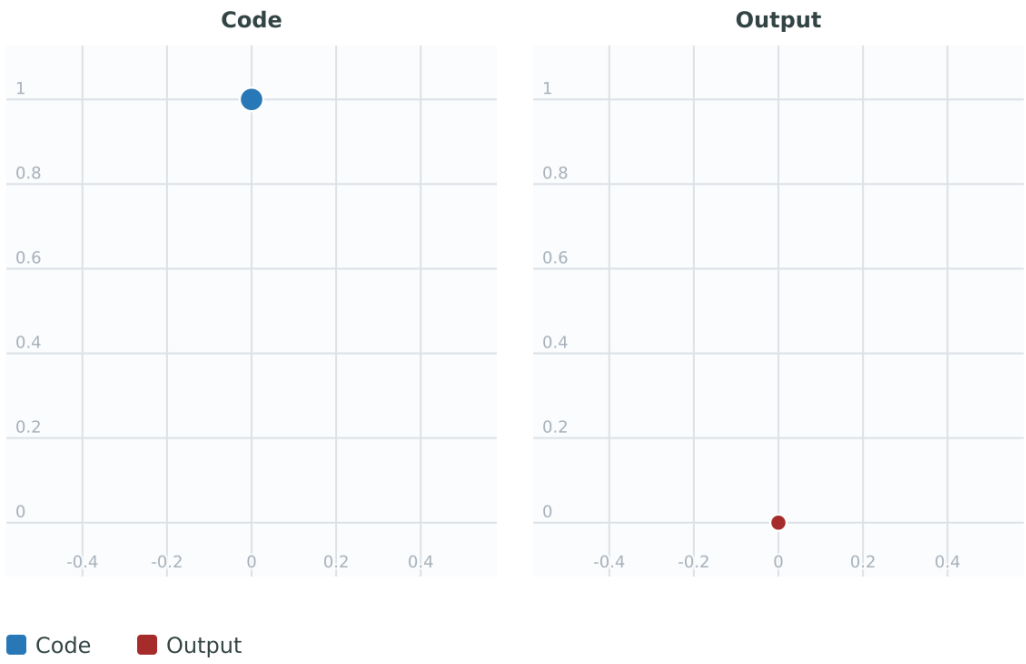
Code

```
SELECT CG_RotateX('POINT(0 1 0)', pi()/2);
```

Output

```
POINT Z (0 6.123233995736766e-17 1)
```

Figure



□□

CG_RotateY, CG_RotateZ, CG_3DRotate

8.3.39 CG_RotateY

CG_RotateY — Rotates a geometry around the Y-axis by a given angle.

□□

geometry **CG_RotateY**(geometry geom, float8 angle);

□□

Rotates the input geometry *geom* by *angle* radians around the Y-axis passing.
Availability: 3.6.0 - requires SFCGAL >= 2.0.0

This method needs SFCGAL backend.

This function supports 3d and will not drop the z-index.

□□

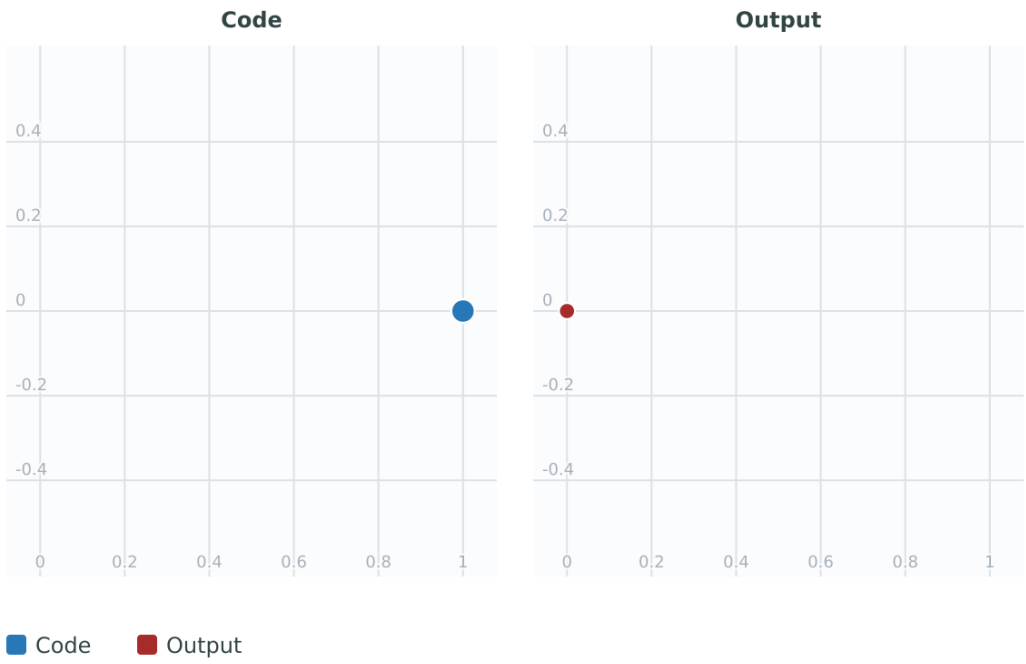
Code

```
SELECT CG_RotateY('POINT(1 0 0)', pi()/2);
```

Output

```
POINT Z (6.123233995736766e-17 0 -1)
```

Figure



□□

CG_RotateX, CG_RotateZ, CG_3DRotate

8.3.40 CG_RotateZ

CG_RotateZ — Rotates a geometry around the Z-axis by a given angle.

□□

geometry **CG_RotateZ**(geometry geom, float8 angle);

□□

Rotates the input geometry *geom* by *angle* radians around the Z-axis.
Availability: 3.6.0 - requires SFCGAL >= 2.0.0

This method needs SFCGAL backend.

This function supports 3d and will not drop the z-index.

□□

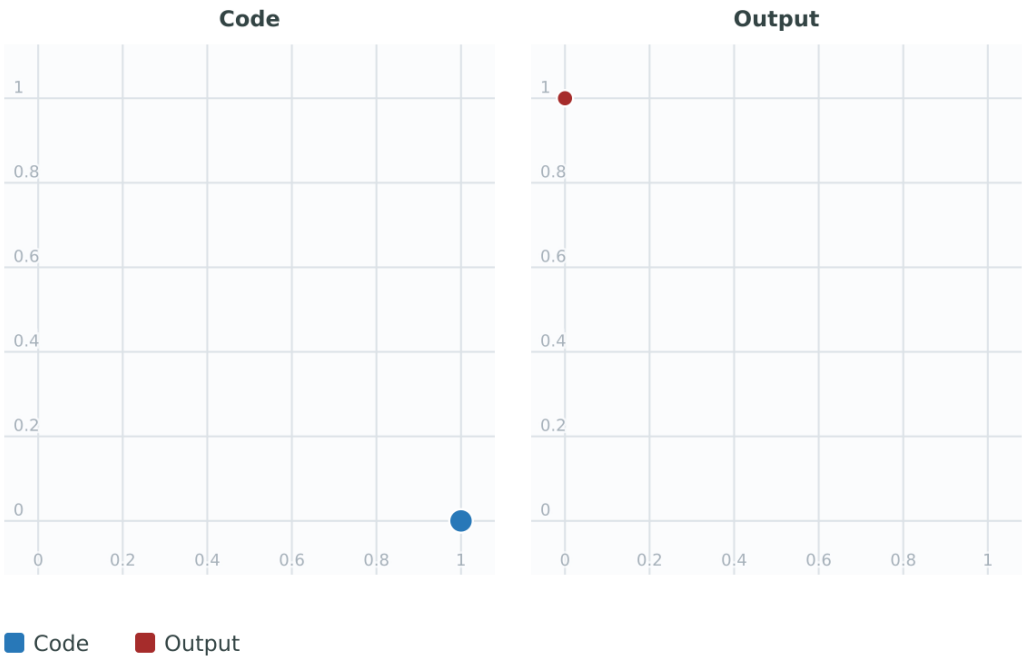
Code

```
SELECT CG_RotateZ('POINT(1 0 0)', pi()/2);
```

Output

```
POINT Z (6.123233995736766e-17 1 0)
```

Figure



□□

CG_RotateX, CG_RotateY, CG_3DRotate

8.3.41 CG_Scale

CG_Scale — Scales a geometry uniformly in all dimensions by a given factor.

□□

geometry **CG_Scale**(geometry geom, float8 factor);

□□

Scales the input geometry *geom* by a uniform scale *factor* in all dimensions (X, Y, and Z). The scaling is performed relative to the origin point (0,0,0).

Availability: 3.6.0 - requires SFCGAL >= 2.0.0

This method needs SFCGAL backend.

□□

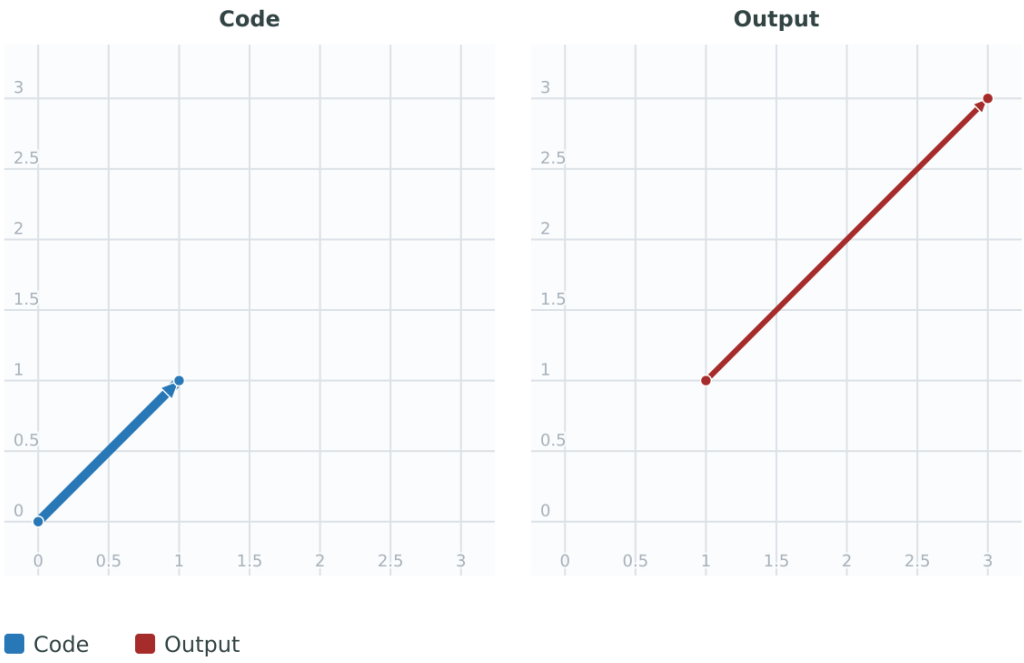
Code

```
SELECT CG_Scale('LINESTRING(1 1,2 2)', 2);
```

Output

```
LINESTRING(2 2,4 4)
```

Figure



□□

CG_3DScale, CG_3DScaleAroundCenter, ST_Scale

8.3.42 CG_3DScale

CG_3DScale — Scales a geometry by separate factors along X, Y, and Z axes.

□□

geometry **CG_3DScale**(geometry geom, float8 factorX, float8 factorY, float8 factorZ);

□□

Scales the input geometry *geom* by different factors along the X, Y, and Z axes. The scaling is performed relative to the origin point (0,0,0).

Availability: 3.6.0 - requires SFCGAL >= 2.0.0

This method needs SFCGAL backend.

This function supports 3d and will not drop the z-index.

□□

Code

```
SELECT CG_3DScale('POINT(1 1 1)', 2, 3, 4);
```

Output

```
POINT Z (2 3 4)
```

Figure



□□

CG_Scale, CG_3DScaleAroundCenter

8.3.43 CG_3DScaleAroundCenter

CG_3DScaleAroundCenter — Scales a geometry in 3D space around a specified center point.

□□

geometry **CG_3DScaleAroundCenter**(geometry geom, float8 factorX, float8 factorY, float8 factorZ, float8 centerX, float8 centerY, float8 centerZ);

□□

Scales the input geometry *geom* by different factors along the X, Y, and Z axes, relative to a specified center point (*centerX*, *centerY*, *centerZ*).

Availability: 3.6.0 - requires SFCGAL >= 2.0.0

This method needs SFCGAL backend.

This function supports 3d and will not drop the z-index.

□□

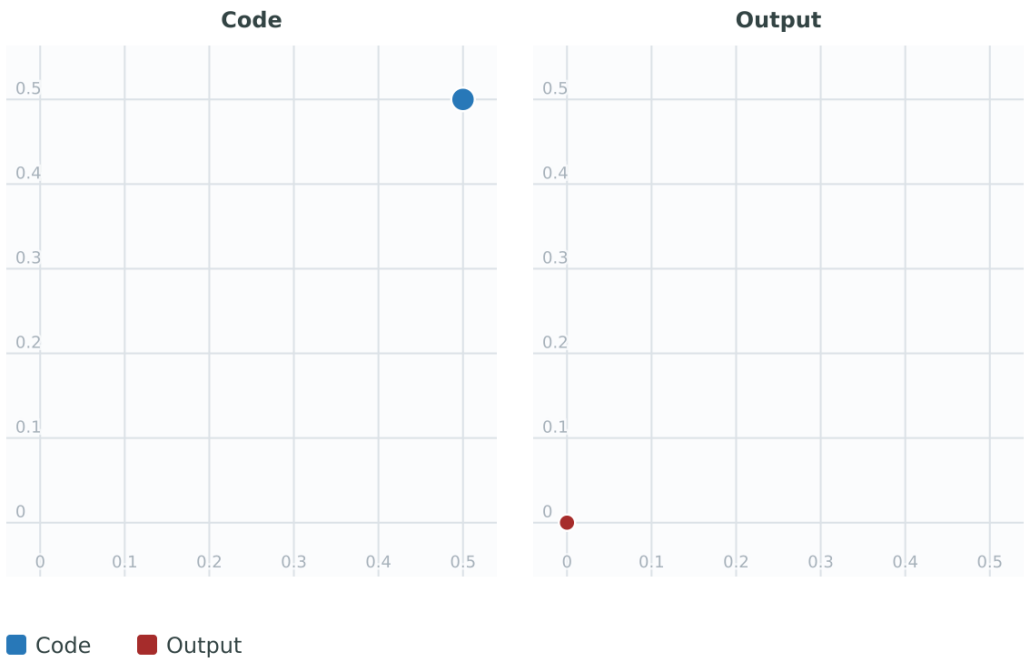
Code

```
SELECT CG_3DScaleAroundCenter('POINT(2 2 2)', 0.5, 0.5, 0.5, 1, 1, 1);
```

Output

```
POINT Z (1.5 1.5 1.5)
```

Figure



□□

CG_Scale, CG_3DScale

8.3.44 CG_Translate

CG_Translate — Translates (moves) a geometry by given offsets in 2D space.

□□

geometry **CG_Translate**(geometry geom, float8 deltaX, float8 deltaY);

□□

Translates the input geometry *geom* by adding *deltaX* to the X coordinates and *deltaY* to the Y coordinates. Z coordinates are dropped.

Availability: 3.6.0 - requires SFCGAL >= 2.0.0

This method needs SFCGAL backend.

□□

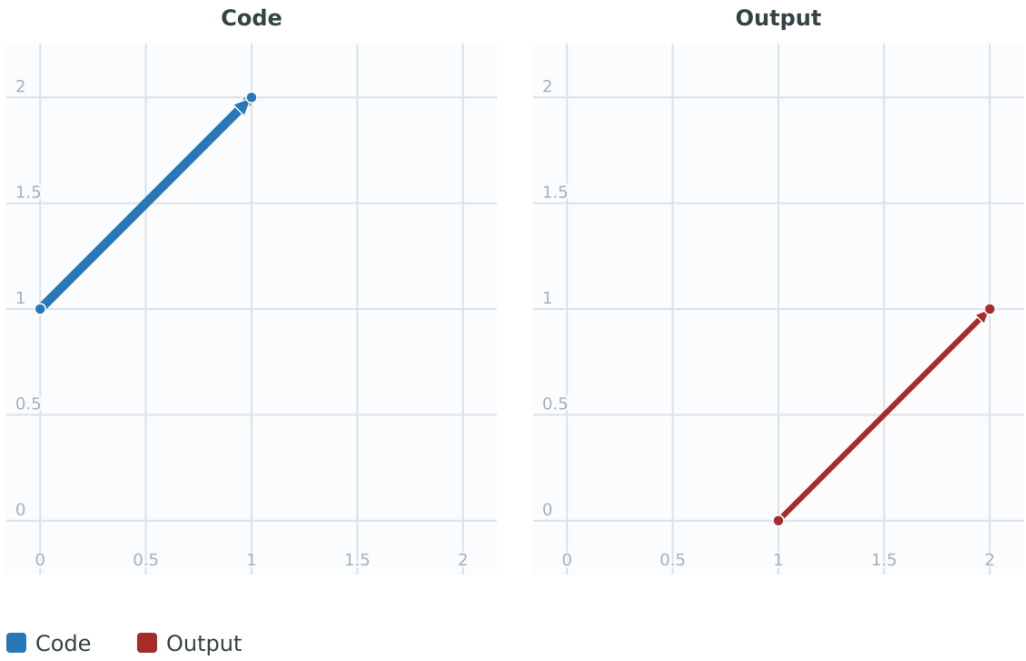
Code

```
SELECT CG_Translate('LINESTRING(1 1,2 2)', 1, -1);
```

Output

```
LINESTRING(2 0,3 1)
```

Figure



□□

CG_3DTranslate, ST_Translate

8.3.45 CG_3DTranslate

CG_3DTranslate — Translates (moves) a geometry by given offsets in 3D space.

□□

geometry **CG_3DTranslate**(geometry geom, float8 deltaX, float8 deltaY, float8 deltaZ);

□□

Translates the input geometry *geom* by adding *deltaX* to the X coordinates, *deltaY* to the Y coordinates, and *deltaZ* to the Z coordinates.

Availability: 3.6.0 - requires SFCGAL >= 2.0.0

This method needs SFCGAL backend.

This function supports 3d and will not drop the z-index.

□□

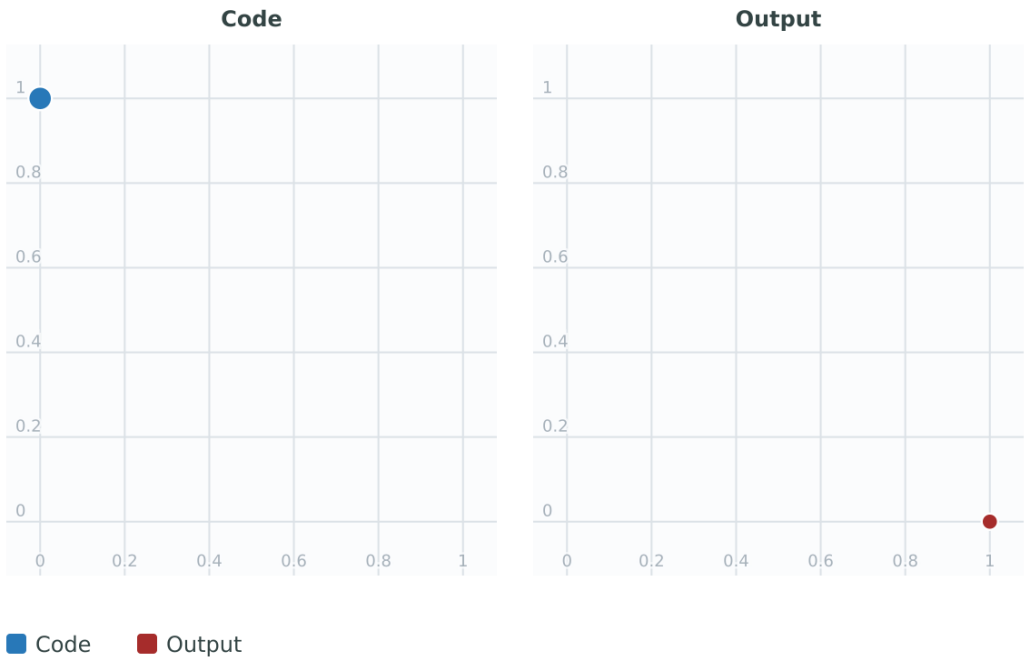
Code

```
SELECT CG_3DTranslate('POINT(1 1 1)', 1, -1, 2);
```

Output

```
POINT Z (2 0 3)
```

Figure



□□

CG_Translate, ST_Translate

8.3.46 CG_Simplify

CG_Simplify — Reduces the complexity of a geometry while preserving essential features and Z/M values.

□□

geometry **CG_Simplify**(geometry geom, double precision threshold, boolean preserveTopology = false);

□□

Simplifies a geometry using SFCGAL's simplification algorithm, which reduces the number of points or vertices while preserving the essential features of the geometry. This function preserves Z and M values during simplification.

The algorithm is based on constrained triangulation and uses the CGAL Polyline Simplification 2 library with additional handling to preserve Z and M coordinates. When topology is preserved and geometries intersect, Z and M values are interpolated at intersection points.

This function works well with 3D terrain-like geometries (2.5D) but is not designed for vertical surfaces like walls.

Availability: 3.6.0 - requires SFCGAL >= 2.1.0

This method needs SFCGAL backend.

This function supports 3d and will not drop the z-index.

This function supports M coordinates.

Parameters

geom Input geometry

threshold Maximum distance threshold (in geometry unit) for simplification. The higher this value, the more simplified the resulting geometry will be.

preserveTopology If set to true, the function ensures that the topology of the geometry is preserved. When geometries intersect in this mode, Z and M values at intersection points are interpolated. The default value is false.

Return Value

Returns a simplified geometry with preserved Z and M values.

□□

This example simplifies a polygon with a threshold of 0.5.

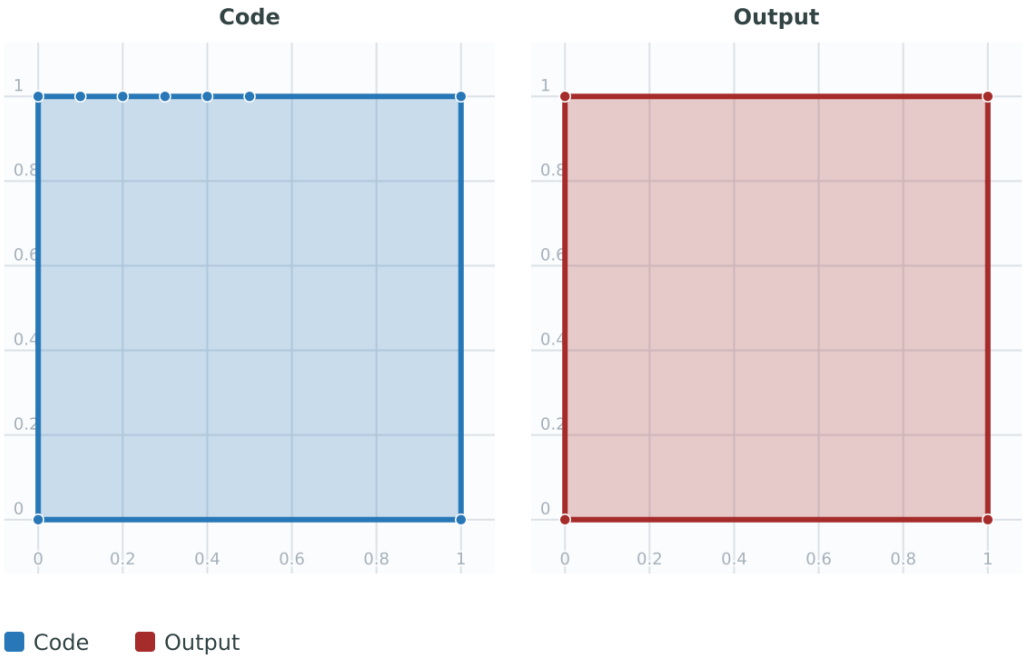
Code

```
SELECT CG_Simplify(
    'POLYGON((0 0,0 1,0.1 1,0.2 1,0.3 1,0.4 1,0.5 1,1 1,1 0,0 0))',
    0.5
  );
```

Output

```
POLYGON((0 0,0 1,1 1,1 0,0 0))
```

Figure



Preserving topology can keep additional support points in a mixed ZM collection.

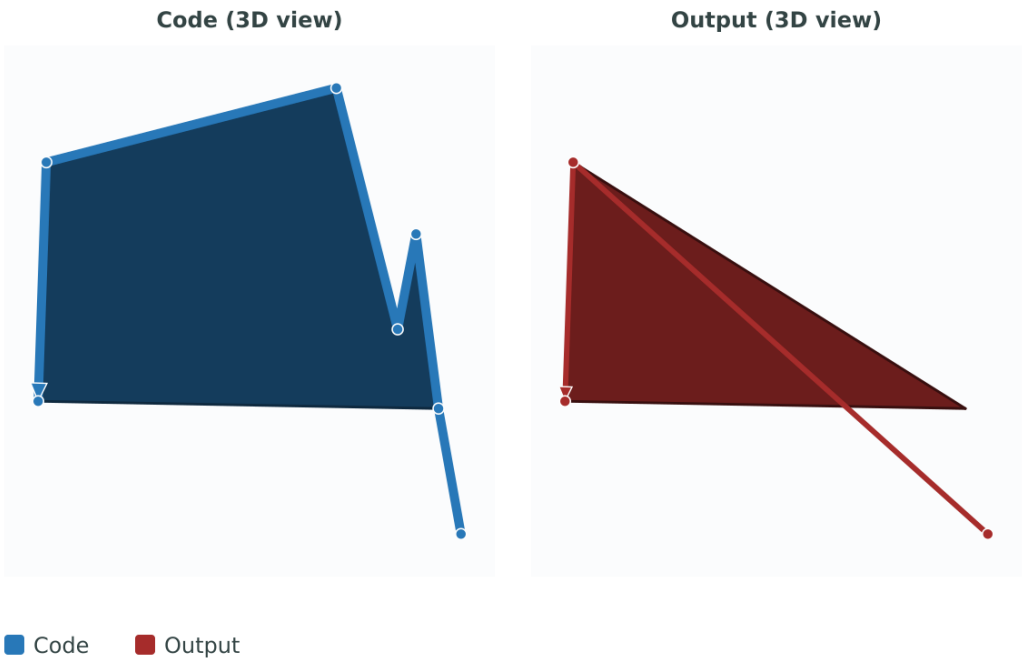
Code

```
SELECT CG_Simplify(
    'GEOMETRYCOLLECTION ZM(
      LINESTRING ZM(-1 -1 3 4,0 0 10 100,1 1 20 200,0 2 15 150,0 5 30 300,2 19 25 250,-4 20 ↩
     15 150),
      POLYGON ZM((0 0 10 100,1 1 20 200,0 2 15 150,0 5 30 300,2 19 25 250,-4 20 15 150,0 0  ↩
    10 100))
    )',
    2,
    false
  );
```

Output

```
GEOMETRYCOLLECTION ZM (LINESTRING ZM (-1 -1 3 4,2 19 25 250,-4 20 15 150),POLYGON ZM ((0 0 10 100,2 19 25 250,-4 20 15 ↩
     150,0 0 10 100)))
```

Figure



The topology-preserving variant keeps the shared anchor point at the collection intersection.

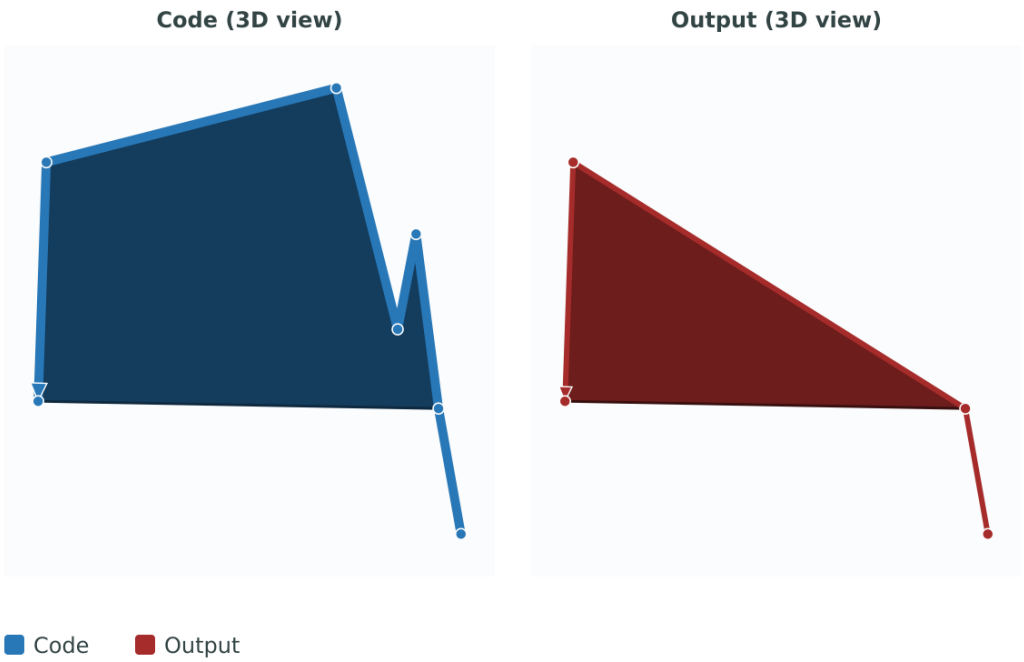
Code

```
SELECT CG_Simplify(
    'GEOMETRYCOLLECTION ZM(
      LINESTRING ZM(-1 -1 3 4,0 0 10 100,1 1 20 200,0 2 15 150,0 5 30 300,2 19 25 250,-4 20 ↩
     15 150),
      POLYGON ZM((0 0 10 100,1 1 20 200,0 2 15 150,0 5 30 300,2 19 25 250,-4 20 15 150,0 0  ↩
    10 100))
    )',
    2,
    true
  );
```

Output

```
GEOMETRYCOLLECTION ZM (LINESTRING ZM (-1 -1 3 4,0 0 10 100,2 19 25 250,-4 20 15 150),POLYGON ZM ((0 0 10 100,2 19 25  ↩
     250,-4 20 15 150,0 0 10 100)))
```

Figure

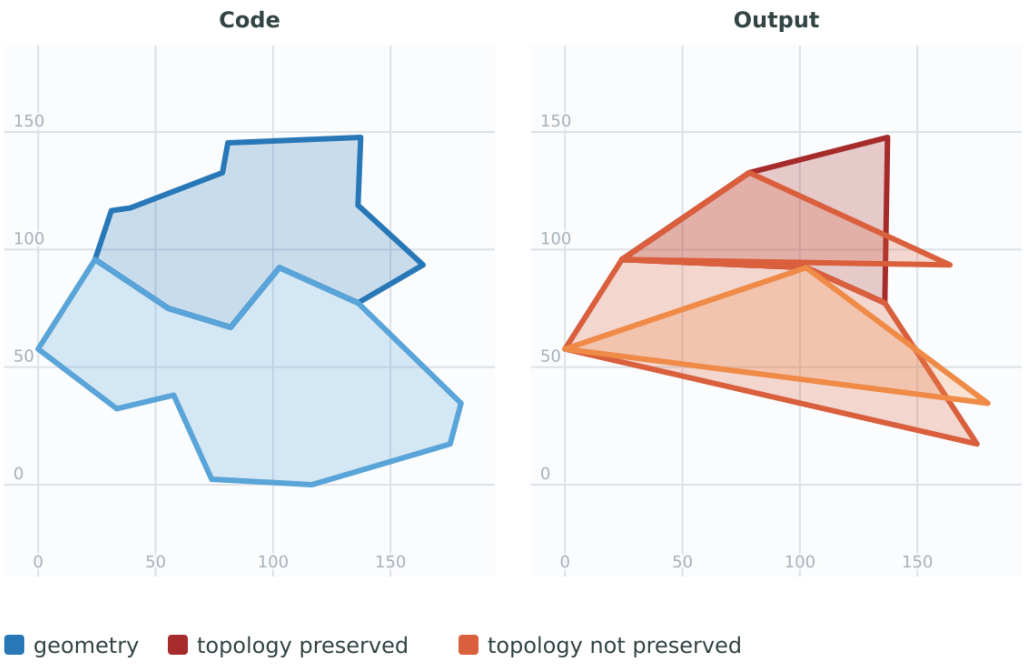


Compare simplification with and without topology preservation, using a threshold of 50.

Code

```
WITH data AS (
  SELECT ST_GeomFromText('GEOMETRYCOLLECTION(
    POLYGON((88.46 158.85,90.77 171.54,147.31 173.85,146.15 145,173.85 119.62,146.15 ↩
    103.46,112.69 118.46,91.92 93.08,65.38 101.15,34.23 121.92,41.15 142.69,49.23 ↩
    143.85,88.46 158.85)),
    POLYGON((112.69 118.46,146.15 103.46,190 60.77,185.38 43.46,126.54 26.15,83.85 ↩
    28.46,67.69 64.23,43.46 58.46,10 83.85,34.23 121.92,65.38 101.15,91.92 93.08,112.69 ↩
    118.46)))
  ') AS geom
)
SELECT geom AS input_geometry,
       CG_Simplify(geom, 50, true) AS topology_preserved,
       CG_Simplify(geom, 50, false) AS topology_not_preserved
FROM data;
```

Figure



☒☒

ST_Simplify, ST_SimplifyPreserveTopology

8.3.47 CG_3DAlphaWrapping

CG_3DAlphaWrapping — Computes a 3D Alpha-wrapping strictly enclosing a geometry.

☒☒

geometry **CG_3DAlphaWrapping**(geometry geom, integer relative_alpha, integer relative_offset);

☒☒

Computes the 3D Alpha Wrapping of the points in a geometry. An alpha wrapping is a watertight and orientable surface mesh that strictly encloses the input. It can be seen as an extension or refinement of an alpha-shape.

The `relative_alpha` parameter controls which features will appear in the output. It can have values from 0 to infinity. Higher relative_alpha values produce coarser wraps; lower values retain more detail.

The `relative_offset` parameter controls the tightness of the result. It can have values from 0 to infinity. If this parameter is set to 0, its value is automatically determined based on the relative_alpha parameter.

Availability: 3.6.0 - requires SFCGAL >= 2.1.0

This method needs SFCGAL backend.

This function supports 3d and will not drop the z-index.

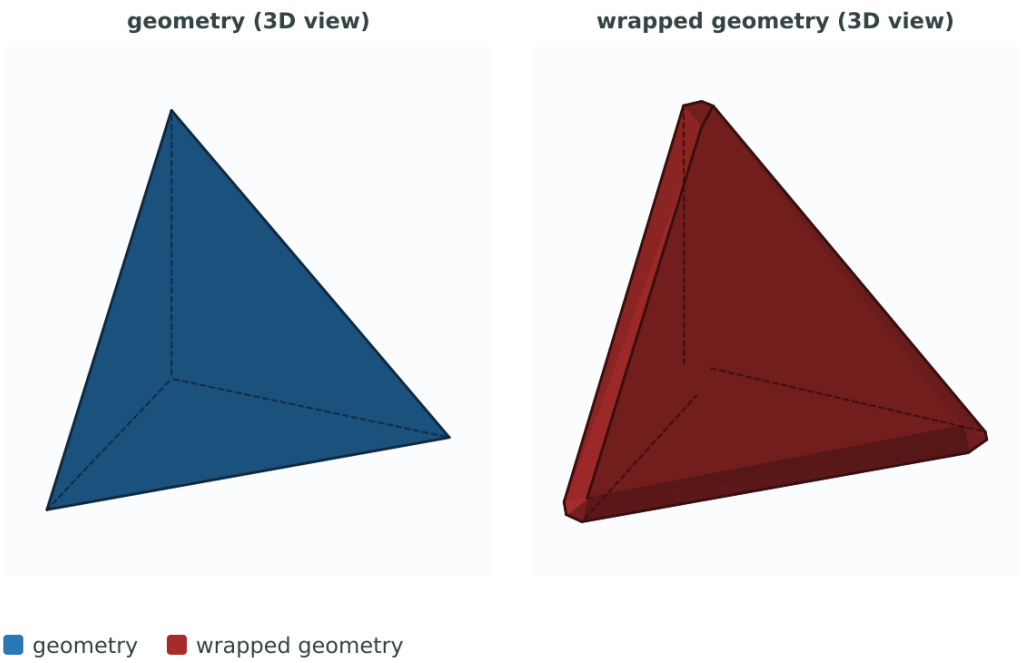
□□

Wrap a tetrahedral surface and show the native 3D input beside the actual enclosing mesh.

Code

```
WITH data AS (
  SELECT 'POLYHEDRALSURFACE Z (
    ((0 0 0,4 0 0,0 4 0,0 0 0)),
    ((0 0 0,0 0 4,4 0 0,0 0 0)),
    ((0 0 0,0 4 0,0 0 4,0 0 0)),
    ((4 0 0,0 0 4,0 4 0,4 0 0))
  )'::geometry AS input_geometry
)
SELECT input_geometry AS input_geometry,
       CG_3DAlphaWrapping(input_geometry, 1)
         AS wrapped_geometry
FROM data;
```

Figure



□□

CG_AlphaShape

8.3.48 CG_NurbsCurveInterpolate

CG_NurbsCurveInterpolate — Creates an interpolating NURBS curve passing through all given data points

□□

geometry **CG_NurbsCurveInterpolate**(geometry data_points, integer degree);

□□

Creates a NURBS curve that interpolates through all provided data points. The resulting curve passes exactly through each point in the input geometry.

Parameters:

- `data_points` - A LINESTRING geometry containing the points to interpolate through.
- `degree` - Polynomial degree of the NURBS curve (typically 2 or 3).

Availability: 3.7.0 - requires SFCGAL >= 2.3.0.

This method needs SFCGAL backend.

This function supports 3d and will not drop the z-index.

□□

Interpolate a degree-3 NURBS curve through four 2D points.

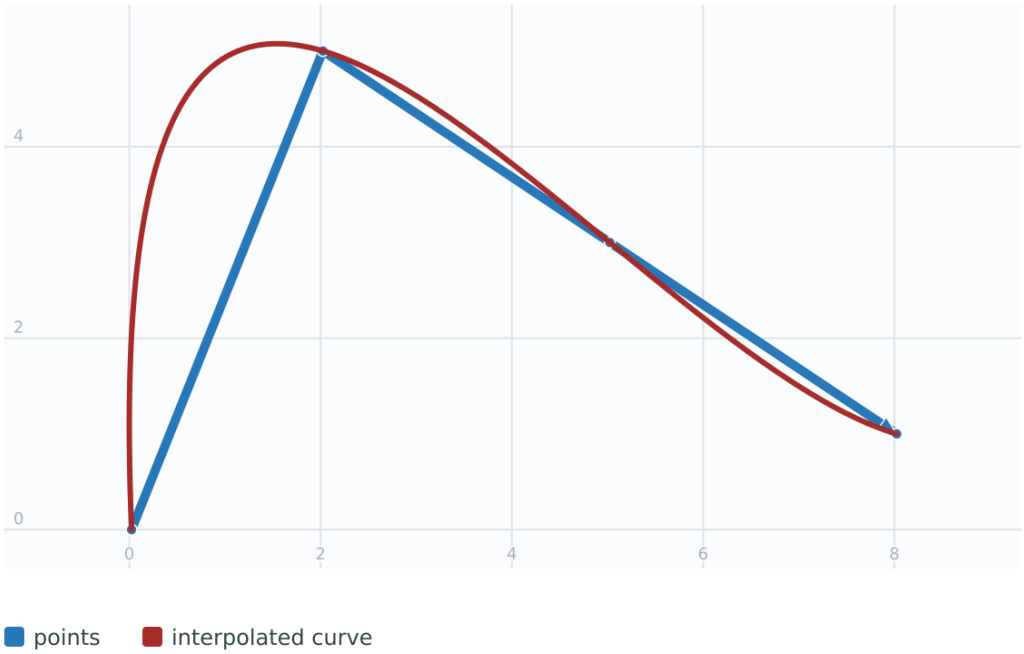
Code

```
WITH data AS (
  SELECT 'LINESTRING(0 0,2 5,5 3,8 1)'::geometry AS input_points
)
SELECT input_points AS input_points,
       CG_NurbsCurveInterpolate(input_points, 3) AS interpolated_curve
FROM data;
```

Output

```
LINESTRING(0 0,2 5,5 3,8 1) | NURBSCURVE(DEGREE 3, CONTROLPOINTS(NURBSPOINT(WEIGHTEDPOINT(0 0),WEIGHT 1), NURBSPOINT( ↩
    WEIGHTEDPOINT(-0.43 10.29),WEIGHT 1), NURBSPOINT(WEIGHTEDPOINT(4.95 1.9),WEIGHT 1), NURBSPOINT(WEIGHTEDPOINT(8 1) ↩
    ,WEIGHT 1)),KNOTS (KNOT(0,4), KNOT(1,4)))
```

Figure



Interpolate a 3D degree-3 NURBS curve through the documented control points.

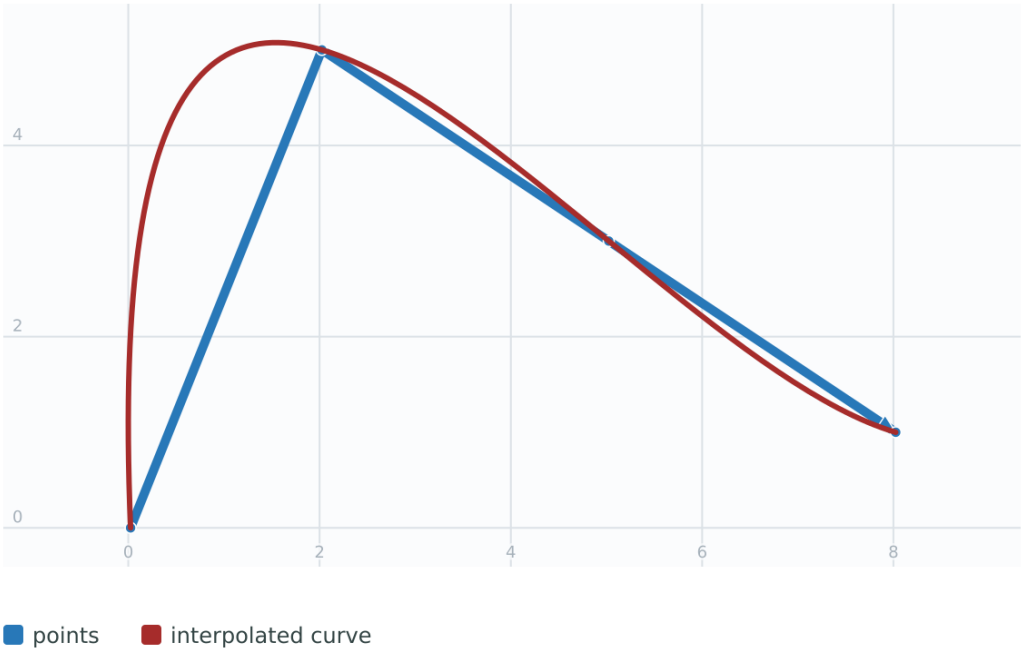
Code

```
WITH data AS (
  SELECT 'LINESTRING Z (0 0 0,2 5 2,5 3 1,8 1 0)'::geometry AS input_points
)
SELECT input_points AS input_points,
       CG_NurbsCurveInterpolate(input_points, 3) AS interpolated_curve
FROM data;
```

Output

```
LINESTRING Z (0 0 0,2 5 2,5 3 1,8 1 0) | NURBSCURVE Z (DEGREE 3,CONTROLPOINTS Z (NURBSPOINT(WEIGHTEDPOINT Z (0 0 0), ↩
    WEIGHT 1), NURBSPOINT(WEIGHTEDPOINT Z (-0.43 10.29 4.36),WEIGHT 1), NURBSPOINT(WEIGHTEDPOINT Z (4.95 1.9 0.54), ↩
    WEIGHT 1), NURBSPOINT(WEIGHTEDPOINT Z (8 1 0),WEIGHT 1)),KNOTS (KNOT(0,4), KNOT(1,4)))
```

Figure



☒☒

CG_NurbsCurveApproximate, ST_MakeNurbsCurve

8.3.49 CG_NurbsCurveApproximate

CG_NurbsCurveApproximate — Creates an approximating NURBS curve fitting data points within a tolerance

☒☒

geometry **CG_NurbsCurveApproximate**(geometry data_points, integer degree, float8 tolerance, integer max_control_points=100);

□□

Creates a NURBS curve that approximates the given data points within the specified tolerance. Unlike interpolation, the curve does not necessarily pass through all points but provides a smooth fit with fewer control points.

Parameters:

- `data_points` - A LINESTRING geometry containing the points to approximate.
- `degree` - Polynomial degree of the NURBS curve.
- `tolerance` - Maximum allowed distance between the curve and data points.
- `max_control_points` - Optional maximum number of control points (default: 100). Limits curve complexity.

Availability: 3.7.0 - requires SFCGAL >= 2.3.0.

 This method needs SFCGAL backend.

 This function supports 3d and will not drop the z-index.

□□

Approximate noisy 2D data with a degree-2 curve and a tolerance of 0.5.

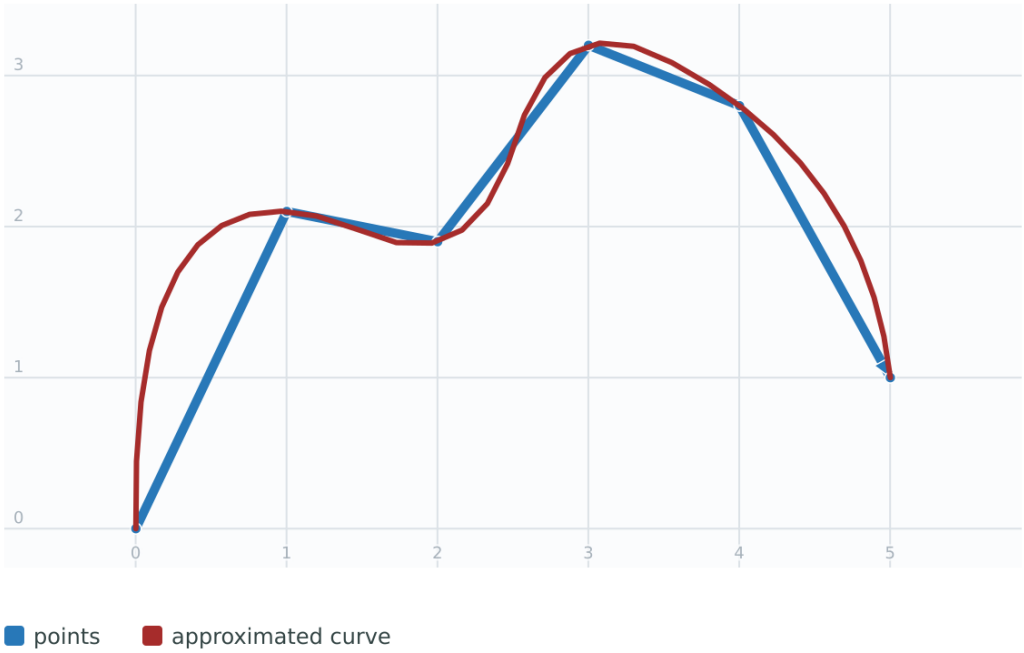
Code

```
WITH data AS (
  SELECT 'LINESTRING(0 0,1 2.1,2 1.9,3 3.2,4 2.8,5 1)'::geometry AS input_points
)
SELECT input_points AS input_points,
       CG_NurbsCurveApproximate(input_points, 2, 0.5) AS approximated_curve
FROM data;
```

Output

```
LINESTRING(0 0,1 2.1,2 1.9,3 3.2,4 2.8,5 1) | NURBSCURVE(DEGREE 2, CONTROLPOINTS(NURBSPOINT(WEIGHTEDPOINT(0 0),WEIGHT ↩
    1), NURBSPOINT(WEIGHTEDPOINT(-0.05 2.64),WEIGHT 1), NURBSPOINT(WEIGHTEDPOINT(2.24 1.65),WEIGHT 1), NURBSPOINT( ↩
    WEIGHTEDPOINT(2.79 3.48),WEIGHT 1), NURBSPOINT(WEIGHTEDPOINT(4.84 2.43),WEIGHT 1), NURBSPOINT(WEIGHTEDPOINT(5 1), ↩
    WEIGHT 1)),KNOTS (KNOT(0,3), KNOT(0.35,1), KNOT(0.51,1), KNOT(0.68,1), KNOT(1,3)))
```

Figure



Approximate a 3D point set while preserving the Z dimension in the fitted curve.

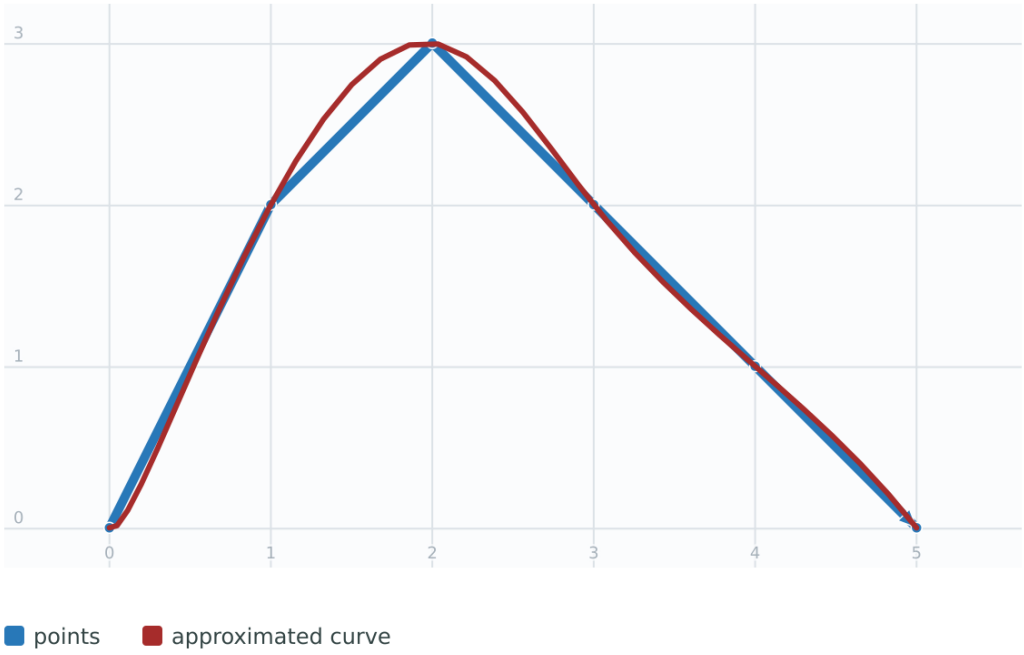
Code

```
WITH data AS (
  SELECT 'LINESTRING Z (0 0 0,1 2 1,2 3 2,3 2 1,4 1 0,5 0 0)'::geometry AS input_points
)
SELECT input_points AS input_points,
       CG_NurbsCurveApproximate(input_points, 3, 0.1) AS approximated_curve
FROM data;
```

Output

```
LINESTRING Z (0 0 0,1 2 1,2 3 2,3 2 1,4 1 0,5 0 0) | NURBSCURVE Z (DEGREE 3,CONTROLPOINTS Z (NURBSPOINT(WEIGHTEDPOINT ↩
    Z (0 0 0),WEIGHT 1), NURBSPOINT(WEIGHTEDPOINT Z (0.18 -0.15 -1.29),WEIGHT 1), NURBSPOINT(WEIGHTEDPOINT Z (1.49 ↩
    4.36 3.35),WEIGHT 1), NURBSPOINT(WEIGHTEDPOINT Z (3.33 1.33 0.38),WEIGHT 1), NURBSPOINT(WEIGHTEDPOINT Z (4.34 ↩
    0.83 -0.44),WEIGHT 1), NURBSPOINT(WEIGHTEDPOINT Z (5 0 0),WEIGHT 1)),KNOTS (KNOT(0,4), KNOT(0.46,1), KNOT(0.64,1) ↩
    , KNOT(1,4)))
```

Figure



☒☒

CG_NurbsCurveInterpolate, ST_MakeNurbsCurve

8.3.50 CG_NurbsCurveDerivative

CG_NurbsCurveDerivative — Computes the derivative (tangent, curvature) of a NURBS curve at a given parameter

☒☒

geometry **CG_NurbsCurveDerivative**(geometry nurbscurve, float8 parameter, integer derivative_order);

☒☒

Computes the derivative of a NURBS curve at a specific parameter value. The derivative represents the rate of change at that point:

- Order 1: Tangent vector (direction and speed)
- Order 2: Curvature vector (how the tangent changes)
- Order 3: Third-order derivative (supported maximum)

Parameters:

- `nurbscurve` - A NURBS curve geometry.
- `parameter` - Parameter value for evaluation (typically 0.0 to 1.0).

• `derivative_order` - Order of derivative to compute (1, 2, or 3).

Returns a POINT geometry representing the derivative vector at the specified parameter.

Availability: 3.7.0 - requires SFCGAL >= 2.3.0.

This method needs SFCGAL backend.

This function supports 3d and will not drop the z-index.

□□

The first derivative gives the tangent vector at the evaluation parameter.

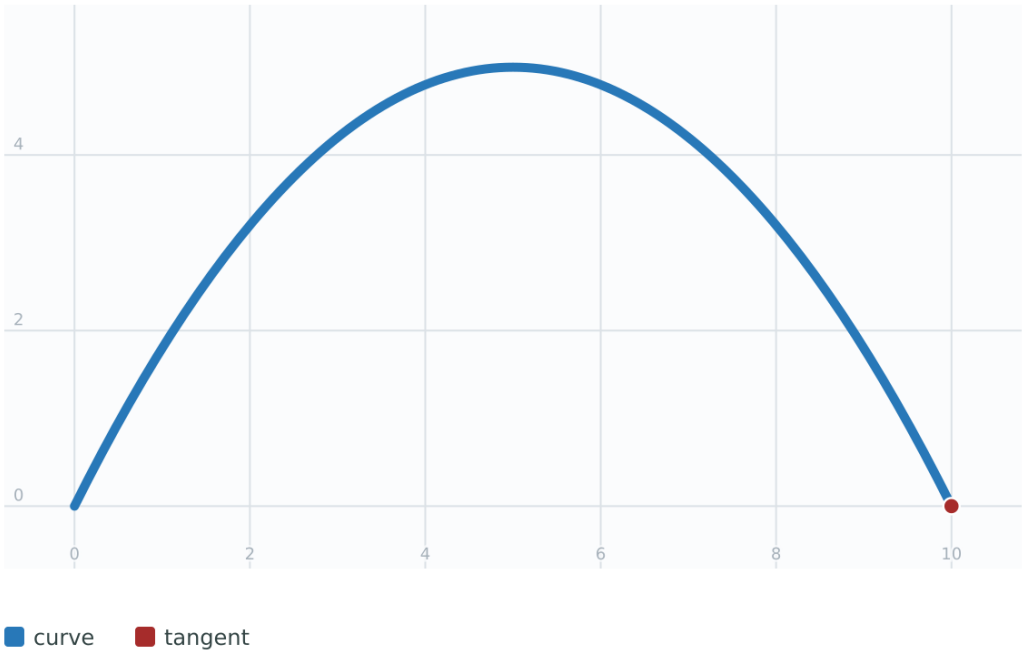
Code

```
WITH data AS (
  SELECT 'NURBSCURVE (2, (0.00 0.00, 5.00 10.00, 10.00 0.00))'::geometry AS input_curve
)
SELECT input_curve AS input_curve,
       CG_NurbsCurveDerivative(input_curve, 0.5, 1) AS tangent
FROM data;
```

Output

```
NURBSCURVE(DEGREE 2, CONTROLPOINTS(NURBSPOINT(WEIGHTEDPOINT(0 0),WEIGHT 1), NURBSPOINT(WEIGHTEDPOINT(5 10),WEIGHT 1),  ↩
    NURBSPOINT(WEIGHTEDPOINT(10 0),WEIGHT 1)),KNOTS (KNOT(0,3), KNOT(1,3))) | POINT Z (10 0 0)
```

Figure



The second derivative exposes curvature change for a higher-degree curve.

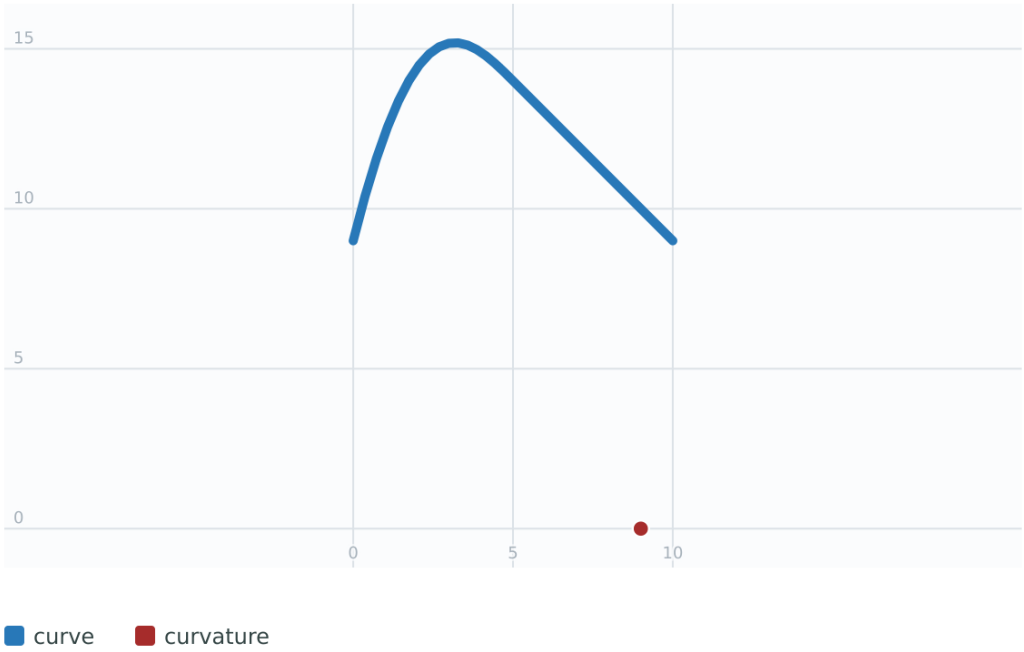
Code

```
WITH data AS (
  SELECT 'NURBSCURVE (3, (0.00 0.00, 2.00 8.00, 5.00 5.00, 8.00 2.00, 10.00 0.00))':: ↩
    geometry AS input_curve
)
SELECT input_curve AS input_curve,
       CG_NurbsCurveDerivative(input_curve, 0.5, 2) AS curvature
FROM data;
```

Output

```
NURBSCURVE(DEGREE 3, CONTROLPOINTS(NURBSPOINT(WEIGHTEDPOINT(0 0),WEIGHT 1), NURBSPOINT(WEIGHTEDPOINT(2 8),WEIGHT 1),  ↩
    NURBSPOINT(WEIGHTEDPOINT(5 5),WEIGHT 1), NURBSPOINT(WEIGHTEDPOINT(8 2),WEIGHT 1), NURBSPOINT(WEIGHTEDPOINT(10 0), ↩
    WEIGHT 1)),KNOTS (KNOT(0,4), KNOT(0.5,1), KNOT(1,4))) | POINT Z (9 -9 0)
```

Figure



The same derivative call also works for 3D NURBS curves.

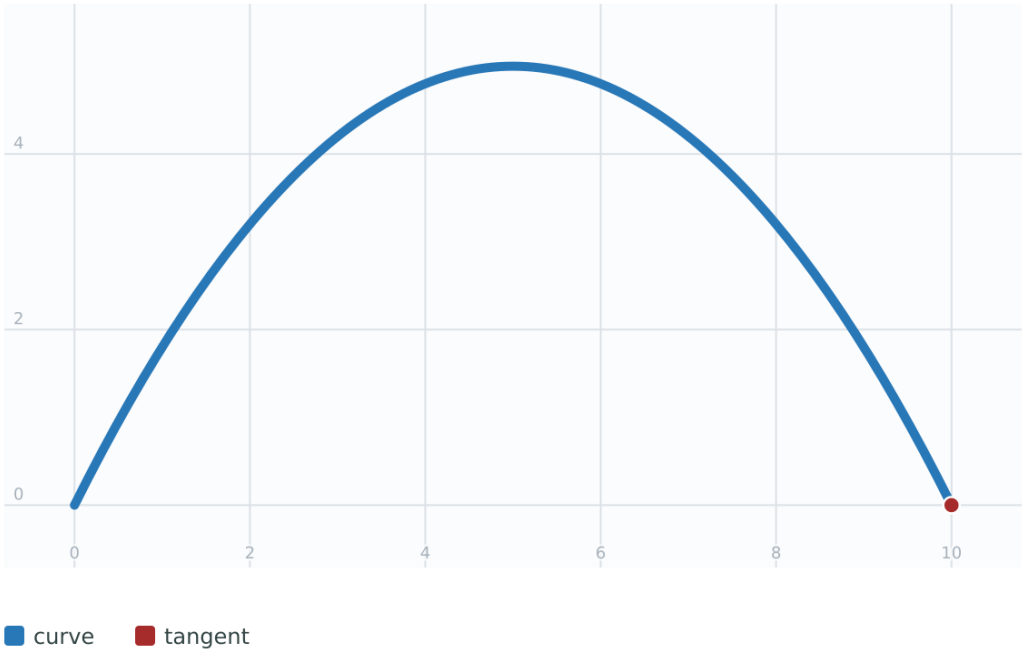
Code

```
WITH data AS (
  SELECT 'NURBSCURVE Z (2, (0.00 0.00 0.00, 5.00 10.00 5.00, 10.00 0.00 0.00))'::geometry  ↩
    AS input_curve
)
SELECT input_curve AS input_curve,
       CG_NurbsCurveDerivative(input_curve, 0.5, 1) AS tangent
FROM data;
```

Output

```
NURBSCURVE Z (DEGREE 2,CONTROLPOINTS Z (NURBSPOINT(WEIGHTEDPOINT Z (0 0 0),WEIGHT 1), NURBSPOINT(WEIGHTEDPOINT Z (5 10 ↩
     5),WEIGHT 1), NURBSPOINT(WEIGHTEDPOINT Z (10 0 0),WEIGHT 1)),KNOTS (KNOT(0,3), KNOT(1,3))) | POINT Z (10 0 0)
```

Figure



□□

ST_Evaluate, ST_MakeNurbsCurve

8.4 Deprecated SFCGAL Function Aliases

These ST_ names remain available for compatibility and emit deprecation notices. Their entries contain only the legacy signature and migration link. Use the corresponding CG_ pages for behavior, examples, and geometry support details.

8.4.1 ST_3DIntersection

ST_3DIntersection — Deprecated alias for CG_3DIntersection.

□□

geometry **ST_3DIntersection**(geometry geom1, geometry geom2);

Description



□□
ST_3DIntersection is deprecated as of 3.5.0. Use CG_3DIntersection instead.

Availability: 2.1.0

 This method needs SFCGAL backend.

8.4.2 ST_3DDifference

ST_3DDifference — Deprecated alias for `CG_3DDifference`.

□□

geometry **ST_3DDifference**(geometry geom1, geometry geom2);

Description



□□
`ST_3DDifference` is deprecated as of 3.5.0. Use CG_3DDifference instead.

Availability: 2.2.0

 This method needs SFCGAL backend.

8.4.3 ST_3DUnion

ST_3DUnion — Deprecated alias for `CG_3DUnion`.

□□

geometry **ST_3DUnion**(geometry geom1, geometry geom2);
geometry **ST_3DUnion**(geometry set g1field);

Description



□□
`ST_3DUnion` is deprecated as of 3.5.0. Use CG_3DUnion instead.

Availability: 2.2.0

 This method needs SFCGAL backend.

8.4.4 ST_Tesselate

ST_Tesselate — Deprecated alias for `CG_Tesselate`.

□□

geometry **ST_Tesselate**(geometry geom);

Description



□□
ST_Tesselate is deprecated as of 3.5.0. Use CG_Tesselate instead.

Availability: 2.1.0

 This method needs SFCGAL backend.

8.4.5 ST_3DArea

ST_3DArea — Deprecated alias for CG_3DArea.

□□

float**ST_3DArea**(geometry geom1);

Description



□□
ST_3DArea is deprecated as of 3.5.0. Use CG_3DArea instead.

Availability: 2.1.0

 This method needs SFCGAL backend.

8.4.6 ST_Extrude

ST_Extrude — Deprecated alias for CG_Extrude.

□□

geometry **ST_Extrude**(geometry geom, float x, float y, float z);

Description



☒☒
ST_Extrude is deprecated as of 3.5.0. Use CG_Extrude instead.

Availability: 2.1.0



This method needs SFCGAL backend.

8.4.7 ST_ForceLHR

ST_ForceLHR — Deprecated alias for CG_ForceLHR.

☒☒

geometry **ST_ForceLHR**(geometry geom);

Description



☒☒
ST_ForceLHR is deprecated as of 3.5.0. Use CG_ForceLHR instead.

Availability: 2.1.0



This method needs SFCGAL backend.

8.4.8 ST_Orientation

ST_Orientation — Deprecated alias for CG_Orientation.

☒☒

integer **ST_Orientation**(geometry geom);

Description



☒☒
ST_Orientation is deprecated as of 3.5.0. Use CG_Orientation instead.

Availability: 2.1.0



This method needs SFCGAL backend.

8.4.9 ST_MinkowskiSum

ST_MinkowskiSum — Deprecated alias for `CG_MinkowskiSum`.

□□

geometry **ST_MinkowskiSum**(geometry geom1, geometry geom2);

Description



□□
`ST_MinkowskiSum` is deprecated as of 3.5.0. Use CG_MinkowskiSum instead.

Availability: 2.1.0



This method needs SFCGAL backend.

8.4.10 ST_StraightSkeleton

ST_StraightSkeleton — Deprecated alias for `CG_StraightSkeleton`.

□□

geometry **ST_StraightSkeleton**(geometry geom);

Description



□□
`ST_StraightSkeleton` is deprecated as of 3.1.0. Use CG_StraightSkeleton instead.



□□
Like `CG_StraightSkeleton`, this function ignores the Z dimension and always gives a 2D result, even when used on a 3D geometry.

Availability: 2.1.0



This method needs SFCGAL backend.

8.4.11 ST_ApproximateMedialAxis

ST_ApproximateMedialAxis — Deprecated alias for `CG_ApproximateMedialAxis`.

□□

geometry **ST_ApproximateMedialAxis**(geometry geom);

Description



□□
ST_ApproximateMedialAxis is deprecated as of 3.5.0. Use CG_ApproximateMedialAxis instead.



□□
Like CG_ApproximateMedialAxis, this function ignores the Z dimension and always gives a 2D result, even when used on a 3D geometry.

Availability: 2.2.0


This method needs SFCGAL backend.

8.4.12 ST_IsPlanar

ST_IsPlanar — Deprecated alias for CG_IsPlanar.

□□

boolean **ST_IsPlanar**(geometry geom);

Description



□□
ST_IsPlanar is deprecated as of 3.5.0. Use CG_IsPlanar instead.

Availability: 2.2.0: This was documented in 2.1.0 but got accidentally left out in 2.1 release.


This method needs SFCGAL backend.

8.4.13 ST_Volume

ST_Volume — Deprecated alias for CG_Volume.

□□

float **ST_Volume**(geometry geom1);

Description



☒☒
ST_Volume is deprecated as of 3.5.0. Use CG_Volume instead.

Availability: 2.2.0



This method needs SFCGAL backend.

8.4.14 ST_MakeSolid

ST_MakeSolid — Deprecated alias for CG_MakeSolid.

☒☒

geometry **ST_MakeSolid**(geometry geom1);

Description



☒☒
ST_MakeSolid is deprecated as of 3.5.0. Use CG_MakeSolid instead.

Availability: 2.2.0



This method needs SFCGAL backend.

8.4.15 ST_IsSolid

ST_IsSolid — Deprecated alias for CG_IsSolid.

☒☒

boolean **ST_IsSolid**(geometry geom1);

Description



☒☒
ST_IsSolid is deprecated as of 3.5.0. Use CG_IsSolid instead.

Availability: 2.2.0



This method needs SFCGAL backend.

8.4.16 ST_ConstrainedDelaunayTriangles

ST_ConstrainedDelaunayTriangles — Deprecated alias for `CG_ConstrainedDelaunayTriangles`.

□□

geometry **ST_ConstrainedDelaunayTriangles**(geometry g1);

Description

□□
`ST_ConstrainedDelaunayTriangles` is deprecated as of 3.5.0. Use CG_ConstrainedDelaunayTriangles instead.

Availability: 3.0.0

This method needs SFCGAL backend.

8.4.17 ST_3DConvexHull

ST_3DConvexHull — Deprecated alias for `CG_3DConvexHull`.

□□

geometry **ST_3DConvexHull**(geometry geom1);

Description

□□
`ST_3DConvexHull` is deprecated as of 3.5.0. Use CG_3DConvexHull instead.

Availability: 3.3.0

This method needs SFCGAL backend.

8.4.18 ST_AlphaShape

ST_AlphaShape — Deprecated alias for `CG_AlphaShape`.

□□

geometry **ST_AlphaShape**(geometry geom, float alpha , boolean allow_holes = false);

Description



☒☒
ST_AlphaShape is deprecated as of 3.5.0. Use CG_AlphaShape instead.



This method needs SFCGAL backend.

8.4.19 ST_OptimalAlphaShape

ST_OptimalAlphaShape — Deprecated alias for `CG_OptimalAlphaShape`.

☒☒

geometry **ST_OptimalAlphaShape**(geometry geom, boolean allow_holes = false , integer nb_components = 1);

Description



☒☒
ST_OptimalAlphaShape is deprecated as of 3.5.0. Use CG_OptimalAlphaShape instead.

Availability: 3.3.0 - requires SFCGAL >= 1.4.1.



This method needs SFCGAL backend.

Chapter 9

□□ (topology)

PostGIS □□□□□□□□□□ (face), □□□□ (edge), □□ (node) □□□□□□□□□□□□□□□□□□□□.

Sandro Santilli's presentation at PostGIS Day Paris 2011 conference gives a good synopsis of PostGIS Topology and where it is headed Topology with PostGIS 2.0 slide deck.

Vincent Picavet provides a good synopsis and overview of what is Topology, how is it used, and various FOSS4G tools that support it in PostGIS Topology PGConf EU 2012.

An example of a topologically based GIS database is the US Census Topologically Integrated Geographic Encoding and Referencing System (TIGER) database. If you want to experiment with PostGIS topology and need some data.

The PostGIS topology module has existed for a long time but was not always part of the official documentation. Extensive cleanup removed deprecated functions, fixed known usability issues, documented the features and functions, added new functionality, and improved SQL-MM compliance.

This chapter is the maintained reference for the topology module, including its primitive tables, TopoGeometry type, topology management functions, editing functions, validation functions, and conversion helpers.

□□□□□□□□□□□□□□□□□ `topology` □□□□□□□□□□□□.

SQL/MM □□□□□□□□□□□□□□ ST_ □□□□□□□□□□□□, PostGIS □□□□□□□□□□□□□□ □□□□□□□□□□.

Topology support is built by default and can be disabled by specifying the --without-topology configure option at build time as described in 2□

9.1 Building Faces from Linework

This example builds topology faces from noded linework and associates point attributes with the resulting faces. It uses ST_CreateTopoGeo to populate an empty topology, ValidateTopology to check the result, GetFaceByPoint to locate each attribute point, and ST_GetFaceGeometry to materialize ordinary polygon geometries.

Create an empty topology and populate it with linework that encloses two faces:

Code

```
CREATE EXTENSION IF NOT EXISTS postgis_topology;

SELECT topology.CreateTopology('topo_example', 3857);

WITH lines(geom) AS (VALUES
```

```
    ('SRID=3857;LINESTRING(0 0, 2 0)'::geometry),
    ('SRID=3857;LINESTRING(0 1, 2 1)'::geometry),
    ('SRID=3857;LINESTRING(0 0, 0 1)'::geometry),
    ('SRID=3857;LINESTRING(1 0, 1 1)'::geometry),
    ('SRID=3857;LINESTRING(2 0, 2 1)'::geometry)
)
SELECT topology.ST_CreateTopoGeo(
    'topo_example',
    ST_Collect(geom)
)
FROM lines;

SELECT * FROM topology.ValidateTopology('topo_example');
```

An empty result from `ValidateTopology` means that no topology errors were found. Attribute points can now be matched to faces. A face identifier of `0` denotes the universal face, so the point is outside every bounded face.

Code

```
WITH labels(name, geom) AS (VALUES
    ('west', 'SRID=3857;POINT(0.5 0.5)'::geometry),
    ('east', 'SRID=3857;POINT(1.5 0.5)'::geometry),
    ('outside', 'SRID=3857;POINT(3 0.5)'::geometry)
),
located AS (
    SELECT name,
           topology.GetFaceByPoint('topo_example', geom, 0) AS face_id
    FROM labels
)
SELECT name,
       face_id,
       CASE WHEN face_id = 0 THEN NULL
            ELSE topology.ST_GetFaceGeometry('topo_example', face_id)
       END AS geom
FROM located
ORDER BY name;
```

Before copying attributes into a face table, check both for points outside bounded faces and for multiple source points assigned to the same face:

Code

```
WITH labels(name, geom) AS (VALUES
    ('west', 'SRID=3857;POINT(0.5 0.5)'::geometry),
    ('east', 'SRID=3857;POINT(1.5 0.5)'::geometry),
    ('outside', 'SRID=3857;POINT(3 0.5)'::geometry)
),
located AS (
    SELECT name,
           topology.GetFaceByPoint('topo_example', geom, 0) AS face_id
    FROM labels
)
SELECT 'outside face' AS problem,
       array_agg(name ORDER BY name) AS labels
FROM located
WHERE face_id = 0
HAVING count(*) > 0
UNION ALL
SELECT 'duplicate face ' || face_id,
       array_agg(name ORDER BY name)
FROM located
```

```
WHERE face_id != 0
GROUP BY face_id
HAVING count(*) > 1;
```

Use DropTopology to remove the example topology when it is no longer needed:

Code

```
SELECT topology.DropTopology('topo_example');
```

9.2 Topology Primitive Tables

The core primitives of any topology are stored in the `edge_data`, `node`, and `face` tables that live in the schema created by CreateTopology. Each row of `edge_data` represents an oriented edge: it records a directed curve from `start_node` to `end_node` together with the identifier of the face encountered on the left of that direction (`left_face`) and the face encountered on the right (`right_face`). The same geometric segment may therefore appear twice—once for each orientation—when it belongs to two faces.

The `next_left_edge` and `next_right_edge` columns complete this orientation information by encoding how to keep walking around a face. They store signed integers whose absolute value is the identifier of the next oriented edge and whose sign determines whether the stored orientation has to be followed as-is or reversed when traversing. Formally, the following rules hold for every edge `e`:

- `abs(next_left_edge)` is the identifier of the edge reached by continuing around the face that lies to the left of `e`. If the value is positive the walk continues from the end node of `e` along the stored orientation of the referenced edge; if it is negative the referenced edge must be followed backwards so that the shared face remains on the walker’ s left.
- `abs(next_right_edge)` analogously follows the boundary of the face located on the right of `e`. A positive value means that the next edge is taken with its recorded orientation starting from the end node of `e`, whereas a negative value instructs to traverse the referenced edge in reverse, starting from its end node, so that the right-hand face is preserved.
- A zero value indicates that the edge is dangling on the corresponding side (for example an isolated edge whose incident face is the universal face `0`). The `abs_next_left_edge` and `abs_next_right_edge` columns exposed by the `edge` view are convenience projections of these absolute values.

This representation is a variant of a doubly connected edge list and is exploited by many topology routines. Functions such as GetRingEdges and ValidateTopology rely on it to reconstruct face boundaries and to diagnose inconsistencies—hence the “invalid next_left_edge” and “invalid next_right_edge” diagnostics reported during validation. Constructors like AddEdge initialise the `next_*` attributes with trivial self references, while editing routines including ST_AddEdgeModFace and ST_RemEdgeModFace update the links as edges are inserted or removed. Other bulk operations (for example Polygonize) may intentionally leave the fields unset, which is why the documentation flags their behaviour explicitly.

9.3 ☒☒☒☒

9.3.1 getfaceedges_returntype

getfaceedges_returntype — A composite type that consists of a sequence number and an edge number.

Description

A composite type that consists of a sequence number and an edge number. This is the return type for `ST_GetFaceEdges` and `GetNodeEdges` functions.

1. `sequence` ⊠⊠⊠⊠⊠⊠⊠⊠: ⊠⊠⊠⊠⊠⊠ SRID ⊠⊠⊠⊠⊠ topology.topology ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
2. `edge` ⊠⊠⊠⊠⊠⊠⊠: ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

9.3.2 TopoGeometry

TopoGeometry — A composite type representing a topologically defined geometry.

Description

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠ ID ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. TopoGeometry ⊠⊠⊠⊠⊠⊠⊠⊠ topology_id, layer_id, id, type ⊠⊠⊠⊠⊠⊠⊠⊠⊠.

1. `topology_id` ⊠⊠⊠⊠⊠⊠⊠⊠: ⊠⊠⊠⊠⊠⊠ SRID ⊠⊠⊠⊠⊠ topology.topology ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
2. `layer_id` ⊠⊠⊠⊠⊠⊠⊠⊠: TopoGeometry ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ layer_id ⊠⊠⊠. topology_id ⊠ layer_id ⊠⊠⊠⊠⊠ topology.layers ⊠⊠⊠⊠⊠⊠⊠⊠⊠ (unique reference) ⊠⊠⊠⊠⊠⊠.
3. `id` ⊠⊠⊠⊠⊠⊠⊠⊠: ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
4. 1 ⊠⊠ 4 ⊠⊠⊠ `type` ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. 1: [⊠⊠] ⊠⊠⊠, 2: [⊠⊠] ⊠⊠, 3: [⊠⊠] ⊠⊠⊠, 4: ⊠⊠⊠⊠.

⊠⊠⊠⊠⊠

⊠⊠.

⊠⊠⊠⊠⊠	⊠⊠
⊠⊠	⊠⊠⊠

⊠⊠

CreateTopoGeom

9.3.3 validatetopology_returntype

validatetopology_returntype — A composite type that consists of an error message and id1 and id2 to denote location of error. This is the return type for `ValidateTopology`.

Description

□□□□□□□□□ 2 □□□□□□□□□□□□□. ValidateTopology □□□□□□□□□□□□□□□□□□ □□□□ ID □□□□□ id1 □ id2 □□□□□□□□□□□□□□.

Field	Type	Description
error	varchar	Denotes the type of error. Current error descriptors are: coincident nodes, edge crosses node, edge not simple, edge end node geometry mismatch, edge start node geometry mismatch, face overlaps face, face within face.
id1	integer	Denotes the identifier of the edge, face, or node in error.
id2	integer	For errors that involve two objects, denotes the secondary edge or node.

□□

ValidateTopology

9.4 □□□□□

9.4.1 TopoElement

TopoElement — □□□□□ TopoGeometry □□□□□□□□□□□□□□□□ 2 □□□□□□□.

Description

□□□□□□ TopoGeometry □□□□□ 1 □□□□□□□□□□□□□ 2 □□□□□□□.

□□ TopoGeometry □□□, □□□□□□□□□□□□□□□□□ (topological primitive) □□□□□□□□□□□ □□□□□□□□□ (1: node, 2: edge, 3: face) □□□□□□□. □□ TopoGeometry □□□□□□□□□□□□□ □□ TopoGeometry □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□.

> **□□**
> □□□□ TopoGeometry □□□□□□□□, □□□□ TopoGeometry □□□□, □□ TopoGeometry □ □□□□□□□ topology.layer □□□□□□□□□□□□□□□□□□□□□□□□□.

□□

Code

```
SELECT te[1] AS id, te[2] AS type FROM
( SELECT ARRAY[1, 2]::topology.topoelement AS te ) f;
```

Output

```
id | type
----+------
  1 |    2
```

Code

```
SELECT ARRAY[1, 2]::topology.topoelement;
```

Output

```
te
-------
 {1,2}
```

This example shows what happens when casting a three-element array to topology.topoelement. The domain requires a two-element array, so the dimension check fails.

Code

```
SELECT ARRAY[1, 2, 3]::topology.topoelement;
```

Output

```
ERROR:  value for domain topology.topoelement violates check constraint "dimensions"
```

□□

GetTopoGeomElements, TopoElementArray, TopoGeometry, TopoGeom_addElement, TopoGeom_remEleme

9.4.2 TopoElementArray

TopoElementArray — An array of TopoElement objects.

Description

1 □□□□ TopoElement □□□□□□□□, □□□□□ TopoElement □□□□□□□□□□□□□□□□□□□□.

□□

Code

```
SELECT '{{1,2},{4,3}}'::topology.topoelementarray AS tea;
```

Output

```
{{1,2},{4,3}}
```

This is the more verbose equivalent.

Code

```
SELECT ARRAY[ARRAY[1, 2], ARRAY[4, 3]]::topology.topoelementarray AS tea;
```

Output

```
{{1,2},{4,3}}
```

This example uses the array aggregate function packaged with topology.

Code

```
SELECT topology.TopoElementArray_Agg(ARRAY[e, t] ORDER BY e, t) As tea
FROM generate_series(1, 4) As e CROSS JOIN generate_series(1, 3) As t;
```

Output

```
{{1,1},{1,2},{1,3},{2,1},{2,2},{2,3},{3,1},{3,2},{3,3},{4,1},{4,2},{4,3}}
```

Code

```
SELECT '{{1,2,4},{3,4,5}}'::topology.topoelementarray AS tea;
```

Output

```
ERROR:  value for domain topology.topoelementarray violates check constraint "dimensions"
```

□□

TopoElement, GetTopoGeomElementArray, TopoElementArray_Agg

9.5 □□□ TopoGeometry □□

9.5.1 AddTopoGeometryColumn

AddTopoGeometryColumn — □□□□□□ TopoGeometry □□□□□□, topology.layer □□□□□□□□ □□□□□, □ layer_id □□□□□□.

□□

integer **AddTopoGeometryColumn**(name topology_name, name schema_name, name table_name, name column_name, varchar feature_type, integer child_layer);
integer **AddTopoGeometryColumn**(name topology_name, regclass tab, name column_name, integer layer_id, varchar feature_type, integer child_layer);

Description

□ TopoGeometry □□□□□□□□□□□□□□□□□□. TopoGeometry □□□□□□□□□□□□□□□□□□□□□□□□. □□□□□□□□□□□□□□□□□□□□□. □□□□□□□□□□□□□□. AddTopoGeometryColumn() □□□□□□□□□□□□□□□□□□□□:

□□□□□□□□□□□□□□□□□□ topology.layer □□□□□□□□□□□□□□□□□□□□□□□□□□□□.

[child_layer] □□□□□□□□□ (□□ NULL □□□□) □□, □□□□□□□ (□□□□□□□□□□□□□□) □□ TopoGeometry □□□□□□□. □□□□□□□, □□□□□ (child_layer □□□□ TopoGeometry □□□□□□□) □□ TopoGeometry □□□□□□□.

□□□□□□□□□ (AddTopoGeometryColumn □□□□□□□ ID □□□□□□) □□□□□□ TopoGeometry □□□□□□□□□□□□□□□.

Valid `feature_types` are: POINT, MULTIPOINT, LINE, MULTILINE, POLYGON, MULTIPOLYGON, COLLECTION

Availability: 1.1

□□

This example creates the new table in the `ma_topo` schema. The table could be created in a different schema, in which case `topology_name` and `schema_name` would be different.

Code

```
CREATE SCHEMA ma;
CREATE TABLE ma.parcels(gid serial, parcel_id varchar(20) PRIMARY KEY, address text);
SELECT topology.AddTopoGeometryColumn('ma_topo', 'ma', 'parcels', 'topo', 'POLYGON');
```

Code

```
CREATE SCHEMA ri;
CREATE TABLE ri.roads(gid serial PRIMARY KEY, road_name text);
SELECT topology.AddTopoGeometryColumn('ri_topo', 'ri', 'roads', 'topo', 'LINE');
```

□□

DropTopoGeometryColumn, toTopoGeom, CreateTopology, CreateTopoGeom

9.5.2 RenameTopoGeometryColumn

RenameTopoGeometryColumn — Renames a topogeometry column

□□

topology.layer **RenameTopoGeometryColumn**(regclass layer_table, name feature_column, name new_nam

Description

This function changes the name of an existing TopoGeometry column ensuring metadata information about it is updated accordingly.

Availability: 3.4.0

□□

Code

```
SELECT topology.RenameTopoGeometryColumn('public.parcels', 'topogeom', 'tgeom');
```

☒☒

AddTopoGeometryColumn, RenameTopology

9.5.3 DropTopology

DropTopology — ☒☒☒☒☒☒☒☒☒☒: ☒☒☒☒☒☒☒☒☒☒ topology.topology ☒☒☒☒☒☒☒☒☒☒, ☒☒ ☒ geometry_columns ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.

☒☒

integer **DropTopology**(varchar topology_schema_name);

Description

☒☒☒☒☒☒☒☒☒☒ topology.topology ☒☒☒☒☒☒☒☒☒☒, ☒☒☒ geometry_columns ☒☒☒☒☒☒☒ ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒ ☒. ☒☒☒☒☒☒☒☒☒☒☒☒, ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.

Availability: 1.1

☒☒

ma_topo ☒☒☒☒☒☒☒☒ topology.topology ☒ geometry_columns ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒ ☒☒☒☒.

Code

```
SELECT topology.DropTopology('ma_topo');
```

☒☒

DropTopoGeometryColumn

9.5.4 RenameTopology

RenameTopology — Renames a topology

☒☒

varchar **RenameTopology**(varchar old_name, varchar new_name);

Description

Renames a topology schema, updating its metadata record in the `topology.topology` table.

Availability: 3.4.0

□□

Rename a topology from `topo_stage` to `topo_prod`.

Code

```
SELECT topology.RenameTopology('topo_stage', 'topo_prod');
```

□□

CopyTopology, RenameTopoGeometryColumn

9.5.5 DropTopoGeometryColumn

DropTopoGeometryColumn — `schema_name` □□□□□ `table_name` □□□□□□□□ Topogeometry □ □□□□□□ topology.layer □□□□□□□□□□□□□□□□□□.

□□

text **DropTopoGeometryColumn**(varchar schema_name, varchar table_name, varchar column_name);

Description

`schema_name` □□□□□ `table_name` □□□□□□□□ Topogeometry □□□□□□ topology.layer □□□ □□□□□□□□□□□□□□□□. □□□□□□□□□□□□□□□□. □□: □□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□ NULL □□□□□□.

Availability: 1.1

□□

Code

```
SELECT topology.DropTopoGeometryColumn('ma_topo', 'parcel_topo', 'topo');
```

□□

AddTopoGeometryColumn

9.5.6 FixCorruptTopoGeometryColumn

FixCorruptTopoGeometryColumn — Fixes topogeometry corruption caused by upgrade to postgis_topology 3.6.0 and higher

□□

text **FixCorruptTopoGeometryColumn**(name layerSchema, name layerTable, name layerColumn);

Description

When upgrading from PostGIS topology <3.6.0 to version >3.6.0+, the topogeometry column definition was changed. This caused corruption in topogeometries created before the upgrade. This function fixes this corruption in affected tables.

The function refuses to process tables with expression or partial indexes; drop or rebuild those indexes manually around the repair.

Availability: 3.6.1

□□

Fix all topology columns

Code

```
SELECT topology.FixCorruptTopoGeometryColumn(schema_name, table_name, feature_column)
    FROM topology.layer;
```

□□

UpgradeTopology

9.5.7 Populate_Topology_Layer

Populate_Topology_Layer — Adds missing entries to topology.layer table by reading metadata from topo tables.

□□

table(schema_name text, table_name text, feature_column text) **Populate_Topology_Layer**();

Description

Adds missing entries to the `topology.layer` table by inspecting topology constraints on tables. This function is useful for fixing up entries in topology catalog after restores of schemas with topo data.

It returns the list of entries created. Returned columns are `schema_name`, `table_name`, `feature_column`.

2.3.0 □□□□□□□□□□□□.

□□

Code

```
SELECT CreateTopology('strk_topo');
CREATE SCHEMA strk;
CREATE TABLE strk.parcels(gid serial, parcel_id varchar(20) PRIMARY KEY, address text);
SELECT topology.AddTopoGeometryColumn('strk_topo', 'strk', 'parcels', 'topo', 'POLYGON');
```

Output

```
createtopology
----------------
              1
(1 row)

CREATE SCHEMA
CREATE TABLE
 addtopogeometrycolumn
-----------------------
                     1
(1 row)
```

This query returns no records because the feature is already registered.

Code

```
SELECT *
FROM topology.Populate_Topology_Layer();
```

Output

```
(0 rows)
```

Rebuild the topology layer catalog.

Code

```
TRUNCATE TABLE topology.layer;
```

Output

```
TRUNCATE TABLE
```

Code

```
SELECT *
FROM topology.Populate_Topology_Layer();
```

Output

```
schema_name | table_name | feature_column
-------------+------------+----------------
 strk        | parcels    | topo
(1 row)
```

Code

```
SELECT topology_id, layer_id,
       schema_name AS sn, table_name AS tn, feature_column AS fc
FROM topology.layer;
```

Output

```
topology_id | layer_id |  sn  |   tn    |  fc
------------+----------+------+---------+------
          1 |        1 | strk | parcels | topo
(1 row)
```

□□

AddTopoGeometryColumn

9.5.8 TopologySummary

TopologySummary — Takes a topology name and provides summary totals of types of objects in topology.

□□

text **TopologySummary**(varchar topology_schema_name);

Description

Takes a topology name and provides summary totals of types of objects in topology.
2.0.0 □□□□□□□□□□□□.

□□

Code

```
SELECT topology.TopologySummary('city_data');
```

Output

```
Topology city_data (329), SRID 4326, precision: 0
22 nodes, 24 edges, 10 faces, 29 topogeoms in 5 layers
Layer 1, type Polygonal (3), 9 topogeoms
 Deploy: features.land_parcels.feature
Layer 2, type Puntal (1), 8 topogeoms
 Deploy: features.traffic_signs.feature
Layer 3, type Lineal (2), 8 topogeoms
 Deploy: features.city_streets.feature
Layer 4, type Polygonal (3), 3 topogeoms
 Hierarchy level 1, child layer 1
 Deploy: features.big_parcels.feature
Layer 5, type Puntal (1), 1 topogeoms
 Hierarchy level 1, child layer 2
 Deploy: features.big_signs.feature
```

9.5.9 ValidateTopology

ValidateTopology — Returns a set of validatetopology_returntype objects detailing issues with topology.

□□

setof validatetopology_returntype **ValidateTopology**(varchar toponame, geometry bbox);

Description

Returns a set of validatetopology_returntype objects detailing issues with topology, optionally limiting the check to the area specified by the `bbox` parameter.

List of possible errors, what they mean and what the returned ids represent are displayed below:

□□	**id1**	**id2**	**Meaning**
coincident nodes	Identifier of first node.	Identifier of second node.	Two nodes have the same geometry.
edge crosses node(□□□□□□□□□□□)	Identifier of the edge.	Identifier of the node.	An edge has a node in its interior. See ST_Relate.
invalid edge(□□□□□□□□□)	Identifier of the edge.		An edge geometry is invalid. See ST_IsValid.
edge not simple(□□□□□□□□)	Identifier of the edge.		An edge geometry has self-intersections. See ST_IsSimple.
edge crosses edge(□□□□□□□□□□□□)	Identifier of first edge.	Identifier of second edge.	Two edges have an interior intersection. See ST_Relate.
edge start node geometry mismatch(□□□□□□□□□)	Identifier of the edge.	Identifier of the indicated start node.	The geometry of the node indicated as the starting node for an edge does not match the first point of the edge geometry. See ST_StartPoint.
edge end node geometry mismatch(□□□□□□□□□)	Identifier of the edge.	Identifier of the indicated end node.	The geometry of the node indicated as the ending node for an edge does not match the last point of the edge geometry. See ST_EndPoint.
face without edges(□□□□□□□)	Identifier of the orphaned face.		No edge reports an existing face on either of its sides (left_face, right_face).
face has no rings(□□□□□□)	Identifier of the partially-defined face.		Edges reporting a face on their sides do not form a ring.
face has wrong mbr	Identifier of the face with wrong mbr cache.		Minimum bounding rectangle of a face does not match minimum bounding box of the collection of edges reporting the face on their sides.

□□	**id1**	**id2**	**Meaning**
hole not in advertised face	Signed identifier of an edge, identifying the ring. See GetRingEdges.		A ring of edges reporting a face on its exterior is contained in different face.
not-isolated node has not- containing_face	Identifier of the ill-defined node.		A node which is reported as being on the boundary of one or more edges is indicating a containing face.
isolated node has containing_face	Identifier of the ill-defined node.		A node which is not reported as being on the boundary of any edges is lacking the indication of a containing face.
isolated node has wrong containing_face	Identifier of the misrepresented node.		A node which is not reported as being on the boundary of any edges indicates a containing face which is not the actual face containing it. See GetFaceContaining-Point.
invalid next_right_edge	Identifier of the misrepresented edge.	Signed id of the edge which should be indicated as the next right edge.	The edge indicated as the next edge encountered walking on the right side of an edge is wrong.
invalid next_left_edge	Identifier of the misrepresented edge.	Signed id of the edge which should be indicated as the next left edge.	The edge indicated as the next edge encountered walking on the left side of an edge is wrong.
mixed face labeling in ring	Signed identifier of an edge, identifying the ring. See GetRingEdges.		Edges in a ring indicate conflicting faces on the walking side. This is also known as a ”Side Location Conflict”.
non-closed ring	Signed identifier of an edge, identifying the ring. See GetRingEdges.		A ring of edges formed by following next_left_edge/next_right_edg attributes starts and ends on different nodes.
face has multiple shells	Identifier of the contended face.	Signed identifier of an edge, identifying the ring. See GetRingEdges.	More than a one ring of edges indicate the same face on its interior.

1.0.0 □□□□□□□□□□□□.

□□□□: 2.0.0 □□□□□□□□□□□□□□□□□□□□□□□□□□□□, □□□□□□□□□□□□ (false positive) □□□□□□□.

□□□□: 2.2.0 □□□□'edge crosses node' □□□□□□□□□□□□□□□ id1 □ id2 □□□□□□□□□□□□.

Changed: 3.2.0 added optional bbox parameter, perform face labeling and edge linking checks.

□□

Code

```
SELECT * FROM topology.ValidateTopology('ma_topo');
```

Output

```
error        | id1 | id2
-------------------+-----+-----
face without edges |   1 |
```

□□

validatetopology_returntype

9.5.10 ValidateTopologyRelation

ValidateTopologyRelation — Returns info about invalid topology relation records

□□

table(error text, layer_id int, topogeo_id bigint, element_id bigint) **ValidateTopologyRelation**(varchar toponame);

Description

Returns a set records giving information about invalidities in the relation table of the topology.

Availability: 3.2.0

□□

ValidateTopology

9.5.11 ValidateTopologyPrecision

ValidateTopologyPrecision — Returns non-precise vertices in the topology.

□□

geometry **ValidateTopologyPrecision**(name toponame, geometry bbox, float8 gridSize);

Description

Returns all vertices that are not rounded to the topology or given `gridSize` as a puntal geometry, optionally limiting the check to the area specified by the `bbox` parameter.

Availability: 3.6.0

□□

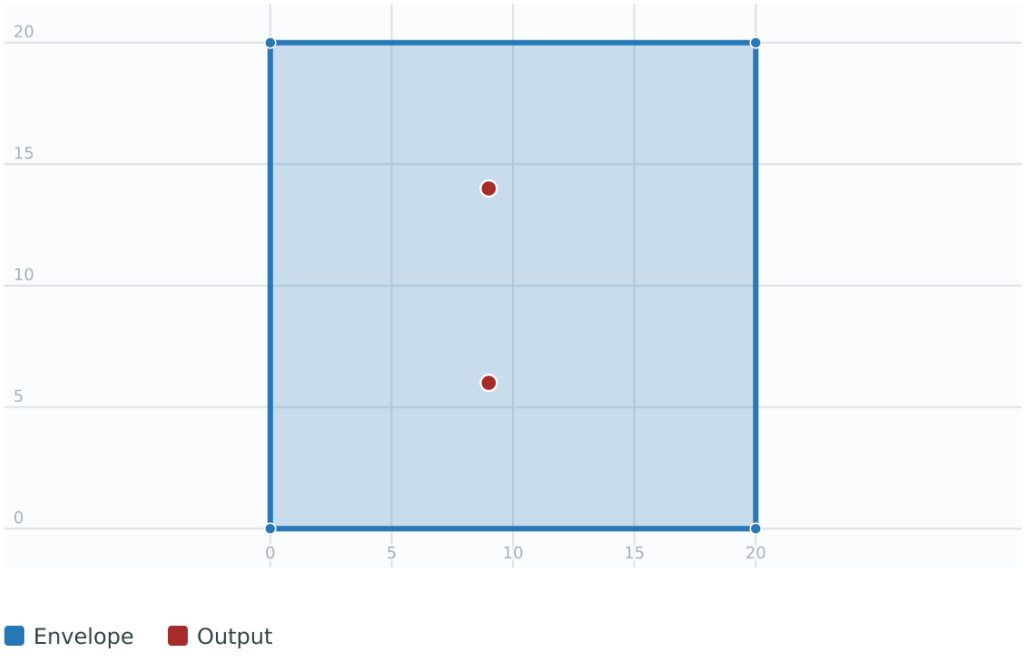
Code

```
SELECT g
FROM topology.ValidateTopologyPrecision(
    'city_data',
    gridSize =
> 2,
    bbox =
> ST_MakeEnvelope(0, 0, 20, 20)
) g;
```

Output

```
MULTIPOINT(9 6,9 14)
```

Figure



□□

MakeTopologyPrecise

9.5.12 MakeTopologyPrecise

MakeTopologyPrecise — Snap topology vertices to precision grid.

□□

void **MakeTopologyPrecise**(name toponame, geometry bbox, float8 gridSize);

Description

Snaps all vertices of a topology to the topology precision grid or to the grid whose size is specified with the `gridSize` parameter, optionally limiting the operation to the objects intersecting the area specified by the `bbox` parameter.

Using a `gridSize` larger than the smallest edge extent raises an exception rather than collapsing the edge into an empty geometry. This safeguards adjacent topological relationships by preventing precision snapping from removing primitives altogether.



□□
Snapping could make the topology invalid, so it is recommended to check the outcome of operation with ValidateTopology.

Availability: 3.6.0

□□

Code

```
SELECT topology.MakeTopologyPrecise(
    'city_data',
    gridSize =
> 2
);
```

□□

ValidateTopologyPrecision, ValidateTopology

9.5.13 FindTopology

FindTopology — Returns a topology record by different means.

□□

topology **FindTopology**(topogeometry topogeom);
topology **FindTopology**(regclass layerTable, name layerColumn);
topology **FindTopology**(name layerSchema, name layerTable, name layerColumn);
topology **FindTopology**(text topoName);
topology **FindTopology**(int id);

Description

Takes a topology identifier or the identifier of a topology-related object and returns a topology.topology record.

Availability: 3.2.0

□□

Code

```
SELECT name(FindTopology('features.land_parcels', 'feature'));
```

Output

```
city_data
```

□□

FindLayer

9.5.14 FindLayer

FindLayer — Returns a topology.layer record by different means.

□□

topology.layer **FindLayer**(topogeometry tg);
topology.layer **FindLayer**(regclass layer_table, name feature_column);
topology.layer **FindLayer**(name schema_name, name table_name, name feature_column);
topology.layer **FindLayer**(integer topology_id, integer layer_id);

Description

Takes a layer identifier or the identifier of a topology-related object and returns a topology.layer record.

Availability: 3.2.0

□□

Code

```
SELECT layer_id(FindLayer('features.land_parcels', 'feature'));
```

Output

```
1
```

□□

FindTopology

9.5.15 TotalTopologySize

TotalTopologySize — Total disk space used by the specified topology, including all indexes and TOAST data.

□□

int8 **TotalTopologySize**(name toponame);

Description

Takes a topology name and provides the total disk space used by all its tables, including indexes and TOAST data.

Availability: 3.6.0

□□

Code

```
SELECT topology.TopologySummary('city_data');
```

Output

```
Topology city_data (329), SRID 4326, precision: 0
22 nodes, 24 edges, 10 faces, 29 topogeoms in 5 layers
Layer 1, type Polygonal (3), 9 topogeoms
 Deploy: features.land_parcels.feature
Layer 2, type Puntal (1), 8 topogeoms
 Deploy: features.traffic_signs.feature
Layer 3, type Lineal (2), 8 topogeoms
 Deploy: features.city_streets.feature
Layer 4, type Polygonal (3), 3 topogeoms
 Hierarchy level 1, child layer 1
 Deploy: features.big_parcels.feature
Layer 5, type Puntal (1), 1 topogeoms
 Hierarchy level 1, child layer 2
 Deploy: features.big_signs.feature
```

9.5.16 UpgradeTopology

UpgradeTopology — Upgrades the specified topology to support large ids (int8) for topology and primitive ids.

□□

void **UpgradeTopology**(name toponame);

Description

Takes a topology name and upgrades it to support large ids (int8) for topology and primitive ids. The function upgrades the following: - face (face_id column from int4 to int8, face_id_seq from int4 to int8) - node (node_id column from int4 to int8, containing_face column from int4 to int8, node_id_seq from int4 to int8) - edge_data (edge_id column from int4 to int8, edge_data_edge_id_seq from int4 to int8, left_face and right_face columns from int4 to int8, start_node and end_node columns from int4 to int8, next_left_edge and next_right_edge columns from int4 to int8) - relation (topogeo_id column from int4 to int8, element_id from int4 to int8) - topology (useslargeids column set to true)

Availability: 3.6.0

□□

Code

```
SELECT topology.UpgradeTopology('city_data');
```

9.6 Topology Statistics Management

Adding elements to a topology triggers many database queries for finding existing edges that will be split, adding nodes and updating edges that will node with the new linework. For this reason it is useful that statistics about the data in the topology tables are up-to-date.

PostGIS Topology population and editing functions do not automatically update the statistics because a updating stats after each and every change in a topology would be overkill, so it is the caller's duty to take care of that.



□□
That the statistics updated by autovacuum will NOT be visible to transactions which started before autovacuum process completed, so long-running transactions will need to run ANALYZE themselves, to use updated statistics.

9.7 □□□□□

9.7.1 CreateTopology

CreateTopology — Creates a new topology schema and registers it in the topology.topology table.

□□

integer **CreateTopology**(name topology_schema_name, integer srid, double precision prec=0, boolean hasz=false, integer topoid=0, boolean useslargeids=false);

Description

Creates a new topology schema with name `topology_name` and registers it in the `topology.topology` table. Topologies must be uniquely named. The topology tables (`edge_data`, `face`, `node`,and `relation` are created in the schema. It returns the id of the topology.

The `srid` is the spatial reference system SRID for the topology. The SRID defaults to -1 (unknown) if not specified.

The tolerance `prec` is measured in the units of the spatial reference system. The tolerance defaults to 0.

`hasz` defaults to false if not specified.

`topoid` optional explicit identifier (allows deterministic topology id assignment, needs to be unique)

`useslargeids` optional, defaults to false. If true, the topology will be created to support large ids (int8) for topology and primitive ids.

This is similar to the SQL/MM ST_InitTopoGeo but has more functionality.

Availability: 1.1

Enhanced: 2.0 added support for `hasZ`.

Enhanced: 3.6.0 added `topoid` and `usesLargeIDs` parameters.

□□

Create a topology schema called `ma_topo` that stores edges and nodes in Massachusetts State Plane-meters (SRID = 26986). The tolerance represents 0.5 meters since the spatial reference system is meter-based.

Code

```
SELECT topology.CreateTopology('ma_topo', 26986, 0.5);
```

Create a topology for Rhode Island called `ri_topo` in spatial reference system State Plane-feet (SRID = 3438)

Code

```
SELECT topology.CreateTopology('ri_topo', 3438) AS topoid;
```

Output

```
2
```

□□

4.5□, ST_InitTopoGeo

9.7.2 CopyTopology

CopyTopology — Makes a copy of a topology (nodes, edges, faces, layers and TopoGeometries) into a new schema

□□

integer **CopyTopology**(varchar existing_topology_name, varchar new_name);

Description

Creates a new topology with name `new_name`, with SRID and precision copied from `existing_topology_name`. The nodes, edges and faces in `existing_topology_name` are copied into the new topology, as well as Layers and their associated TopoGeometries.



□□
The new rows in the `topology.layer` table contain synthetic values for schema_name, table_name and feature_column. This is because the TopoGeometry objects exist only as a definition and are not yet available in a user-defined table.

2.0.0 □□□□□□□□□□□□.

⊠⊠

Make a backup of a topology called ma_topo.

Code

```
SELECT topology.CopyTopology('ma_topo', 'ma_topo_backup');
```

⊠⊠

4.5⊠, CreateTopology, RenameTopology

9.7.3 ST_InitTopoGeo

ST_InitTopoGeo — Creates a new topology schema and registers it in the topology.topology table.

⊠⊠

text **ST_InitTopoGeo**(varchar topology_schema_name);

Description

This is the SQL-MM equivalent of CreateTopology. It lacks options for spatial reference system and tolerance. it returns a text description of the topology creation, instead of the topology id.

Availability: 1.1

This method implements the SQL/MM specification. SQL-MM 3 Topo-Geo and Topo-Net 3: Routine Details: X.3.17

⊠⊠

Code

```
SELECT topology.ST_InitTopoGeo('topo_schema_to_create') AS topocreation;
```

Output

```
Topology-Geometry 'topo_schema_to_create' (id:7) created.
```

⊠⊠

CreateTopology

9.7.4 ST_CreateTopoGeo

ST_CreateTopoGeo — ⊠⊠ ⊠⊠⊠⊠.

⊠⊠

text **ST_CreateTopoGeo**(varchar atopology, geometry acollection);

Description

⊠⊠⊠.
⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
2.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

This method implements the SQL/MM specification. SQL-MM: Topo-Geo and Topo-Net 3: Routine Details -- X.3.18

⊠⊠

Populate the topology.

Code

```
SELECT topology.ST_CreateTopoGeo(
'ri_topo',
ST_GeomFromText(
 'MULTILINESTRING((384744 236928,384750 236923,384769 236911,384799 236895,384811 ↩
    236890,384833 236884,
 384844 236882,384866 236881,384879 236883,384954 236898,385087 236932,385117 236938,
 385167 236938,385203 236941,385224 236946,385233 236950,385241 236956,385254 236971,
 385260 236979,385268 236999,385273 237018,385273 237037,385271 237047,385267 237057,
 385225 237125,385210 237144,385192 237161,385167 237192,385162 237202,385159 237214,
 385159 237227,385162 237241,385166 237256,385196 237324,385209 237345,385234 237375,
 385237 237383,385238 237399,385236 237407,385227 237419,385213 237430,385193 237439,
 385174 237451,385170 237455,385169 237460,385171 237475,385181 237503,385190 237521,
 385200 237533,385206 237538,385213 237541,385221 237542,385235 237540,385242 237541,
 385249 237544,385260 237555,385270 237570,385289 237584,385292 237589,385291 237596,385284 ↩
     237630))',
 3438)
 );
```

Output

```
Topology ri_topo populated
```

Create the tables and topogeometries.

Code

```
CREATE TABLE ri.roads(gid serial PRIMARY KEY, road_name text);

SELECT topology.AddTopoGeometryColumn('ri_topo', 'ri', 'roads', 'topo', 'LINE');
```

⊠⊠

TopoGeo_LoadGeometry, AddTopoGeometryColumn, CreateTopology, DropTopology

9.7.5 TopoGeo_AddPoint

TopoGeo_AddPoint — ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (split) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠.

⊠⊠

bigint **TopoGeo_AddPoint**(varchar atopology, geometry apoint, float8 tolerance);

Description

Adds a point to an existing topology and returns its identifier. The given point will snap to existing nodes or edges within given tolerance. An existing edge may be split by the snapped point.

2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

TopoGeo_AddLineString, TopoGeo_AddPolygon, TopoGeo_LoadGeometry, AddNode, CreateTopology

9.7.6 TopoGeo_AddLineString

TopoGeo_AddLineString — Adds a linestring to an existing topology using a tolerance and possibly splitting existing edges/faces.

⊠⊠

SETOF bigint **TopoGeo_AddLineString**(varchar atopology, geometry aline, float8 tolerance, int max_edges

Description

Adds a linestring to an existing topology and returns a set of signed edge identifiers forming it up (negative identifies mean the edge goes in the opposite direction of the input linestring).

The given line will snap to existing nodes or edges within given tolerance. Existing edges and faces may be split by the line. New nodes and faces may be added, in addition to new edges.

The returned edge identifiers may be either existing edges or newly created edges as needed to fully represent the input line as closely as possible.

The number of newly created edges (either covering space previously uncovered or resulting from split of existing edges) may be limited by the `max_edges` parameter.

> ⊠⊠
> When adding many lines, the load order can have a large impact on performance because each call checks the line against topology elements already loaded. In bulk workflows, adding large closed or enclosing lines before smaller contained lines can avoid expensive later splits. For loading a collection in one call, see TopoGeo_LoadGeometry.



□□
Updating statistics about topologies being loaded via this function is up to caller, see maintaining statistics during topology editing and population.

2.0.0 □□□□□□□□□□□□.

Enhanced: 3.2.0 added support for returning signed identifier.

Enhanced: 3.7.0 added support for limiting the number of new edges created in the topology.

□□

TopoGeo_AddPoint, TopoGeo_AddPolygon, TopoGeo_LoadGeometry, AddEdge, CreateTopology

9.7.7 TopoGeo_AddPolygon

TopoGeo_AddPolygon — Adds a polygon to an existing topology using a tolerance and possibly splitting existing edges/faces. Returns face identifiers.

□□

SETOF bigint **TopoGeo_AddPolygon**(varchar atopology, geometry apoly, float8 tolerance);

Description

Adds a polygon to an existing topology and returns a set of face identifiers forming it up. The boundary of the given polygon will snap to existing nodes or edges within given tolerance. Existing edges and faces may be split by the boundary of the new polygon.



□□
Updating statistics about topologies being loaded via this function is up to caller, see maintaining statistics during topology editing and population.

2.0.0 □□□□□□□□□□□□.

□□

TopoGeo_AddPoint, TopoGeo_AddLineString, TopoGeo_LoadGeometry, AddFace, CreateTopology

9.7.8 TopoGeo_LoadGeometry

TopoGeo_LoadGeometry — Load a geometry into an existing topology, snapping and splitting as needed.

□□

void **TopoGeo_LoadGeometry**(varchar atopology, geometry ageom, float8 tolerance);

Description

Loads a geometry into an existing topology. The given geometry will snap to existing nodes or edges within given tolerance. Existing edges and faces may be split as a consequence of the load.

> ⊠⊠
> Updating statistics about topologies being loaded via this function is up to caller, see maintaining statistics during topology editing and population.

Availability: 3.5.0

⊠⊠

TopoGeo_AddPoint, TopoGeo_AddLineString, TopoGeo_AddPolygon, CreateTopology

9.8 ⊠⊠⊠⊠⊠

9.8.1 ST_AddIsoNode

ST_AddIsoNode — ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (isolated) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ID ⊠⊠⊠⊠⊠⊠. ⊠⊠⊠ NULL ⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

bigint **ST_AddIsoNode**(varchar atopology, bigint aface, geometry apoint);

Description

`atopology` ⊠⊠⊠ `aface` ID(faceid) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ `apoint` ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ID(nodeid) ⊠⊠⊠⊠⊠⊠.

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (SRID) ⊠⊠⊠⊠⊠, `apoint` ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠ NULL ⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (⊠⊠⊠⊠⊠⊠⊠⊠⊠) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

`aface` ⊠ NULL ⊠⊠⊠⊠ `apoint` ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠.

Availability: 1.1

This method implements the SQL/MM specification. SQL-MM: Topo-Net Routines: X+1.3.1

⊠⊠

⊠⊠

AddNode, CreateTopology, DropTopology, ST_Intersects

9.8.2 ST_AddIsoEdge

ST_AddIsoEdge — ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ anode ⊠ anothernode ⊠⊠⊠⊠⊠ alinestring ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ID ⊠⊠⊠⊠⊠⊠.

⊠⊠

bigint **ST_AddIsoEdge**(varchar atopology, bigint anode, bigint anothernode, geometry alinestring);

Description

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ anode ⊠ anothernode ⊠⊠⊠⊠⊠ alinestring ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ID(edgeid) ⊠⊠⊠⊠⊠⊠.

alinestring ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (SRID) ⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ NULL ⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠.

alinestring ⊠ anode ⊠ anothernode ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠.

anode ⊠ anothernode ⊠ alinestring ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

Availability: 1.1

This method implements the SQL/MM specification. SQL-MM: Topo-Geo and Topo-Net 3: Routine Details: X.3.4

⊠⊠

⊠⊠

ST_AddIsoNode, ST_IsSimple, ST_Within

9.8.3 ST_AddEdgeNewFaces

ST_AddEdgeNewFaces — ⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ 2 ⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

bigint **ST_AddEdgeNewFaces**(varchar atopology, bigint anode, bigint anothernode, geometry acurve);

Description

⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ 2 ⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠ ID ⊠⊠⊠⊠⊠⊠.

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠⊠⊠⊠ NULL ⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ node ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠), acurve ⊠ LINESTRING ⊠⊠⊠⊠⊠, anode ⊠ anothernode ⊠ acurve ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

acurve ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (SRID) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

2.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

This method implements the SQL/MM specification. SQL-MM: Topo-Geo and Topo-Net 3: Routine Details: X.3.12

⊠⊠

⊠⊠

ST_RemEdgeNewFace

ST_AddEdgeModFace

9.8.4 ST_AddEdgeModFace

ST_AddEdgeModFace — ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

bigint **ST_AddEdgeModFace**(varchar atopology, bigint anode, bigint anothernode, geometry acurve);

Description

⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

> **Note!** ⊠⊠
> ⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠ (⊠⊠⊠) ⊠⊠⊠⊠ (universe face) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠⊠⊠⊠⊠⊠⊠⊠ ID ⊠⊠⊠⊠⊠⊠.

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠⊠⊠⊠ NULL ⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ node ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠), `acurve` ⊠ `LINESTRING` ⊠⊠⊠⊠⊠, `anode` ⊠ `anothernode` ⊠ `acurve` ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

`acurve` ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (SRID) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

2.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

This method implements the SQL/MM specification. SQL-MM: Topo-Geo and Topo-Net 3: Routine Details: X.3.13

⊠⊠

⊠⊠

ST_RemEdgeModFace

ST_AddEdgeNewFaces

9.8.5 ST_RemEdgeNewFace

ST_RemEdgeNewFace — ⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

bigint **ST_RemEdgeNewFace**(varchar atopology, bigint anedge);

Description

⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠⊠⊠⊠⊠⊠⊠⊠ ID ⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ NULL ⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠, ⊠⊠ (⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

Refuses to remove an edge participating in the definition of an existing TopoGeometry. Refuses to heal two faces if any TopoGeometry is defined by only one of them (and not the other).

⊠⊠⊠⊠⊠⊠ NULL ⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ edge ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠), ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

2.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

This method implements the SQL/MM specification. SQL-MM: Topo-Geo and Topo-Net 3: Routine Details: X.3.14

⊠⊠

⊠⊠

ST_RemEdgeModFace

ST_AddEdgeNewFaces

9.8.6 ST_RemEdgeModFace

ST_RemEdgeModFace — Removes an edge, and if the edge separates two faces deletes one face and modifies the other face to cover the space of both.

⊠⊠

bigint **ST_RemEdgeModFace**(varchar atopology, bigint anedge);

Description

Removes an edge, and if the removed edge separates two faces deletes one face and modifies the other face to cover the space of both. Preferentially keeps the face on the right, to be consistent with ST_AddEdgeModFace. Returns the id of the face which is preserved.

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

Refuses to remove an edge participating in the definition of an existing TopoGeometry. Refuses to heal two faces if any TopoGeometry is defined by only one of them (and not the other).

⊠⊠⊠⊠⊠⊠ NULL ⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ edge ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠), ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

2.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

This method implements the SQL/MM specification. SQL-MM: Topo-Geo and Topo-Net 3: Routine Details: X.3.15

☒☒

☒☒

ST_AddEdgeModFace

ST_RemEdgeNewFace

9.8.7 ST_ChangeEdgeGeom

ST_ChangeEdgeGeom — ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.

☒☒

text **ST_ChangeEdgeGeom**(varchar atopology, bigint anedge, geometry acurve);

Description

☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.

If any arguments are null, the given edge does not exist in the edge table of the topology schema, the acurve is not a LINESTRING, or the modification would change the underlying topology then an error is thrown.

acurve ☒☒☒☒☒☒☒☒☒☒☒☒☒ (SRID) ☒☒☒☒☒☒☒☒☒☒☒☒☒.

☒ acurve ☒☒☒☒☒☒☒☒☒☒, ☒☒☒☒☒☒☒☒.

☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.

1.1.0 ☒☒☒☒☒☒☒☒☒☒☒☒.

☒☒☒☒: 2.0.0 ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.

This method implements the SQL/MM specification. SQL-MM: Topo-Geo and Topo-Net 3: Routine Details X.3.6

☒☒

Code

```
SELECT topology.ST_ChangeEdgeGeom(
    'ma_topo',
    1,
        ST_GeomFromText(
            'LINESTRING(227591.9 893900.4,227622.6 893844.3,227641.6 893816.6,227704.5 ↩
    893778.5)',
            26986)
);
```

Output

```
Edge 1 changed
```

☒☒

ST_AddEdgeModFace

ST_RemEdgeModFace

ST_ModEdgeSplit

9.8.8 ST_ModEdgeSplit

ST_ModEdgeSplit — ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒, ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒ ☒☒☒☒☒☒.

☒☒

bigint **ST_ModEdgeSplit**(varchar atopology, bigint anedge, geometry apoint);

Description

☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒, ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒ ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.

Availability: 1.1

☒☒☒☒: 2.0 ☒☒☒☒☒☒, ☒☒☒☒ ST_ModEdgesSplit ☒☒☒☒☒☒☒☒☒☒☒☒☒.

This method implements the SQL/MM specification. SQL-MM: Topo-Geo and Topo-Net 3: Routine Details: X.3.9

☒☒

Add an edge to split.

Code

```
SELECT topology.AddEdge('ma_topo', ST_GeomFromText('LINESTRING(227592 893910,227600 893910) ↩
    ', 26986)) AS edgeid;
```

Output

```
3
```

Split the edge with a new node.

Code

```
SELECT topology.ST_ModEdgeSplit('ma_topo', 3, ST_SetSRID(ST_Point(227594, 893910), 26986)) ↩
    AS node_id;
```

Output

```
7
```

□□

ST_NewEdgesSplit, ST_ModEdgeHeal, ST_NewEdgeHeal, AddEdge

9.8.9 ST_ModEdgeHeal

ST_ModEdgeHeal — Heals two edges by deleting the node connecting them, modifying the first edge and deleting the second edge. Returns the id of the deleted node.

□□

bigint **ST_ModEdgeHeal**(varchar atopology, bigint anedge, bigint anotheredge);

Description

Heals two edges by deleting the node connecting them, modifying the first edge and deleting the second edge. Returns the id of the deleted node. Updates all existing joined edges and relationships accordingly.

2.0 □□□□□□□□□□□□□.

 This method implements the SQL/MM specification. SQL-MM: Topo-Geo and Topo-Net 3: Routine Details: X.3.9

□□

ST_ModEdgeSplit ST_NewEdgesSplit

9.8.10 ST_NewEdgeHeal

ST_NewEdgeHeal — Heals two edges by deleting the node connecting them, deleting both edges, and replacing them with an edge whose direction is the same as the first edge provided.

□□

bigint **ST_NewEdgeHeal**(varchar atopology, bigint anedge, bigint anotheredge);

Description

Heals two edges by deleting the node connecting them, deleting both edges, and replacing them with an edge whose direction is the same as the first edge provided. Returns the id of the new edge replacing the healed ones. Updates all existing joined edges and relationships accordingly.

2.0 □□□□□□□□□□□□□.

 This method implements the SQL/MM specification. SQL-MM: Topo-Geo and Topo-Net 3: Routine Details: X.3.9

□□

ST_ModEdgeHeal ST_ModEdgeSplit ST_NewEdgesSplit

9.8.11 ST_MoveIsoNode

ST_MoveIsoNode — Moves an isolated node in a topology from one point to another. If new `apoint` geometry exists as a node an error is thrown. Returns description of move.

□□

text **ST_MoveIsoNode**(varchar atopology, bigint anode, geometry apoint);

Description

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□. □ `apoint` □□□□□□□□□□□□□□□□□□□□□□□□□□□□□.

If any arguments are null, the `apoint` is not a point, the existing node is not isolated (is a start or end point of an existing edge), new node location intersects an existing edge (even at the end points) or the new location is in a different face (since 3.2.0) then an exception is thrown.

□□□□□□□□□□□□□□□□□□ (SRID) □□□□□□□□□□□□□□□□.

2.0.0 □□□□□□□□□□□□□□.

Enhanced: 3.2.0 ensures the nod cannot be moved in a different face

This method implements the SQL/MM specification. SQL-MM: Topo-Net Routines: X.3.2

□□

Add an isolated node with no face.

Code

```
SELECT topology.ST_AddIsoNode('ma_topo', NULL, ST_GeomFromText('POINT(227579 893916)', ↩
    26986)) AS nodeid;
```

Output

```
7
```

Move the new node.

Code

```
SELECT topology.ST_MoveIsoNode('ma_topo', 7, ST_GeomFromText('POINT(227579.5 893916.5)', ↩
    26986)) AS descrip;
```

Output

```
Isolated Node 7 moved to location 227579.5,893916.5
```

□□

ST_AddIsoNode

9.8.12 ST_NewEdgesSplit

ST_NewEdgesSplit — ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ 2 ⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ID ⊠⊠⊠⊠⊠⊠.

⊠⊠

bigint **ST_NewEdgesSplit**(varchar atopology, bigint anedge, geometry apoint);

Description

⊠⊠⊠⊠⊠⊠⊠⊠ `apoint` ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ 2 ⊠⊠⊠⊠⊠ ⊠⊠⊠⊠ ID `anedge` ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ID ⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (SRID) ⊠⊠⊠⊠⊠, `apoint` ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠ NULL ⊠⊠ ⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

Availability: 1.1

This method implements the SQL/MM specification. SQL-MM: Topo-Net Routines: X.3.8

⊠⊠

Add an edge to split.

Code

```
SELECT topology.AddEdge('ma_topo', ST_GeomFromText('LINESTRING(227575 893917,227592 893900) ↩
    ', 26986)) AS edgeid;
```

Output

```
2
```

Split the new edge.

Code

```
SELECT topology.ST_NewEdgesSplit('ma_topo', 2, ST_GeomFromText('POINT(227578.5 893913.5)',  ↩
    26986)) AS newnodeid;
```

Output

```
6
```

⊠⊠

ST_ModEdgeSplit ST_ModEdgeHeal ST_NewEdgeHeal AddEdge

9.8.13 ST_RemoveIsoNode

ST_RemoveIsoNode — ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠ (⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠) ⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠.

□□

text **ST_RemoveIsoNode**(varchar atopology, bigint anode);

Description

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□. □□□□□□□□□□ (□□□□□□□□□□□□□□□□) □□, □□□□□□□□□□.

Availability: 1.1

This method implements the SQL/MM specification. SQL-MM: Topo-Geo and Topo-Net 3: Routine Details: X+1.3.3

□□

Remove an isolated node with no face.

Code

```
SELECT topology.ST_RemoveIsoNode('ma_topo', 7) AS result;
```

Output

```
Isolated node 7 removed
```

□□

ST_AddIsoNode

9.8.14 ST_RemoveIsoEdge

ST_RemoveIsoEdge — Removes an isolated edge and returns description of action. If the edge is not isolated, then an exception is thrown.

□□

text **ST_RemoveIsoEdge**(varchar atopology, bigint anedge);

Description

Removes an isolated edge and returns description of action. If the edge is not isolated, then an exception is thrown.

Availability: 1.1

This method implements the SQL/MM specification. SQL-MM: Topo-Geo and Topo-Net 3: Routine Details: X+1.3.3

□□

Remove an isolated edge with no face.

Code

```
SELECT topology.ST_RemoveIsoEdge('ma_topo', 7) AS result;
```

Output

```
Isolated edge 7 removed
```

□□

ST_AddIsoNode

9.9 □□□□□

9.9.1 GetEdgeByPoint

GetEdgeByPoint — Finds the edge-id of an edge that intersects a given point.

□□

bigint **GetEdgeByPoint**(varchar atopology, geometry apoint, float8 tol1);

Description

Retrieves the id of an edge that intersects a Point.

□□□□□□□, □□□□, □□□□□□□□□□□□□□□□ (edgeid) □□□□□□□. tolerance = 0 □□□□□□□□□ □□□□□□□□□□□□□.

If `apoint` doesn't intersect an edge, returns 0 (zero).

If use tolerance > 0 and there is more than one edge near the point then an exception is thrown.



□□
□□□□□ tolerance = 0 □□□□ ST_Intersects □, □□□□□□□□ ST_DWithin □□□□□□□.

GEOS □□□□□□

2.0.0 □□□□□□□□□□□□□□.

□□

□□□□□□□ AddEdge □□□□□□□□□□□□□□□□□□□.

Code

```
SELECT topology.GetEdgeByPoint('ma_topo', geom, 1) As with1mtol, topology.GetEdgeByPoint(' ↩
    ma_topo', geom, 0) As withnotol
FROM ST_GeomFromEWKT('SRID=26986;POINT(227622.6 893843)') As geom;
```

Output

```
with1mtol | withnotol
-----------+-----------
         2 |         0
```

This query errors because two or more edges are found.

Code

```
SELECT topology.GetEdgeByPoint('ma_topo', geom, 1) As nearnode
FROM ST_GeomFromEWKT('SRID=26986;POINT(227591.9 893900.4)') As geom;
```

Output

```
ERROR:  Two or more edges found
```

□□

AddEdge, GetNodeByPoint, GetFaceByPoint

9.9.2 GetFaceByPoint

GetFaceByPoint — Finds face intersecting a given point.

□□

bigint **GetFaceByPoint**(varchar atopology, geometry apoint, float8 tol1);

Description

Finds a face referenced by a Point, with given tolerance.

The function will effectively look for a face intersecting a circle having the point as center and the tolerance as radius.

If no face intersects the given query location, 0 is returned (universal face).

If more than one face intersect the query location an exception is thrown.

2.0.0 □□□□□□□□□□□□.

Enhanced: 3.2.0 more efficient implementation and clearer contract, stops working with invalid topologies.

□□

Code

```
SELECT topology.GetFaceByPoint('ma_topo', geom, 10) As with1mtol, topology.GetFaceByPoint(' ↩
    ma_topo', geom, 0) As withnotol
FROM ST_GeomFromEWKT('POINT(234604.6 899382.0)') As geom;
```

Output

```
with1mtol | withnotol
-----------+-----------
         1 |         0
```

This query errors because two or more faces are found.

Code

```
SELECT topology.GetFaceByPoint('ma_topo', geom, 1) As nearnode
FROM ST_GeomFromEWKT('POINT(227591.9 893900.4)') As geom;
```

Output

```
ERROR:  Two or more faces found
```

□□

GetFaceContainingPoint, AddFace, GetNodeByPoint, GetEdgeByPoint

9.9.3 GetFaceContainingPoint

GetFaceContainingPoint — Finds the face containing a point.

□□

bigint **GetFaceContainingPoint**(text atopology, geometry apoint);

Description

Returns the id of the face containing a point.

An exception is thrown if the point falls on a face boundary.



□□

The function relies on a valid topology, using edge linking and face labeling.

Availability: 3.2.0

□□

ST_GetFaceGeometry

9.9.4 GetNodeByPoint

GetNodeByPoint — Finds the node-id of a node at a point location.

□□

bigint **GetNodeByPoint**(varchar atopology, geometry apoint, float8 tol1);

Description

Retrieves the id of a node at a point location.

The function returns an integer (id-node) given a topology, a POINT and a tolerance. If tolerance = 0 means exact intersection, otherwise retrieves the node from an interval.

If `apoint` doesn't intersect a node, returns 0 (zero).

If use tolerance > 0 and there is more than one node near the point then an exception is thrown.



□□
□□□□ tolerance = 0 □□□ ST_Intersects □, □□□□□□□ ST_DWithin □□□□□□.

GEOS □□□□□
2.0.0 □□□□□□□□□□□□.

□□

□□□□□□ AddEdge □□□□□□□□□□□□□□□□.

Code

```
SELECT topology.GetNodeByPoint('ma_topo', geom, 1) As nearnode
FROM ST_GeomFromEWKT('SRID=26986;POINT(227591.9 893900.4)') As geom;
```

Output

```
2
```

This query errors because the tolerance matches two or more nodes.

Code

```
SELECT topology.GetNodeByPoint('ma_topo', geom, 1000) As too_much_tolerance
FROM ST_GeomFromEWKT('SRID=26986;POINT(227591.9 893900.4)') As geom;
```

Output

```
ERROR:  Two or more nodes found
```

□□

AddEdge, GetEdgeByPoint, GetFaceByPoint

9.9.5 GetTopologyID

GetTopologyID — □□□□□□□□□□ topology.topology □□□□□□□□□ ID □□□□□□.

□□

integer **GetTopologyID**(varchar toponame);

Description

□□□□□□□□□□ topology.topology □□□□□□□□□ ID □□□□□□.
Availability: 1.1

□□

Code

```
SELECT topology.GetTopologyID('ma_topo') AS topo_id;
```

Output

```
1
```

□□

CreateTopology, DropTopology, GetTopologyName, GetTopologySRID

9.9.6 GetTopologySRID

GetTopologySRID — □□□□□□□□□□ topology.topology □□□□□□□□□ SRID □□□□□□.

□□

integer **GetTopologyID**(varchar toponame);

Description

□□□□□□□□□□ topology.topology □□□□□□□□□□□□□□□□□□□□□□□□.
2.0.0 □□□□□□□□□□□□.

□□

Code

```
SELECT topology.GetTopologySRID('ma_topo') AS srid;
```

Output

```
4326
```

☒☒

CreateTopology, DropTopology, GetTopologyName, GetTopologyID

9.9.7 GetTopologyName

GetTopologyName — ☒☒☒ ID ☒☒☒☒☒☒☒☒ (☒☒☒) ☒☒☒☒☒☒☒☒☒.

☒☒

varchar **GetTopologyName**(integer topology_id);

Description

☒☒☒☒☒ ID ☒☒☒☒☒ topology.topology ☒☒☒☒☒☒☒☒☒☒☒ (☒☒☒) ☒☒☒☒☒☒.

Availability: 1.1

☒☒

Code

```
SELECT topology.GetTopologyName(1) AS topo_name;
```

Output

```
ma_topo
```

☒☒

CreateTopology, DropTopology, GetTopologyID, GetTopologySRID

9.9.8 ST_GetFaceEdges

ST_GetFaceEdges — aface ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.

☒☒

getfaceedges_returntype **ST_GetFaceEdges**(varchar atopology, bigint aface);

Description

aface ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒ (sequence) ☒☒☒☒ ID(edgeid) ☒☒☒☒☒☒☒. ☒☒☒☒☒ 1 ☒☒☒☒☒☒☒.

☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒ (☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒).

2.0 ☒☒☒☒☒☒☒☒☒☒☒☒.

This method implements the SQL/MM specification. SQL-MM 3 Topo-Geo and Topo-Net 3: Routine Details: X.3.5

□□

This query returns the edges that bound face 1.

Code

```
SELECT (topology.ST_GetFaceEdges('tt', 1)).*;
```

Output

```
sequence | edge
----------+------
        1 |   -4
        2 |    5
        3 |    7
        4 |   -6
        5 |    1
        6 |    2
        7 |    3
```

This query returns the sequence, edge identifier, and geometry of the edges that bound face 1. Use ST_GetFaceGeometry if only the geometry and sequence are needed.

Code

```
SELECT t.seq, t.edge, geom
FROM topology.ST_GetFaceEdges('tt', 1) As t(seq, edge)
    INNER JOIN tt.edge AS e ON abs(t.edge) = e.edge_id;
```

□□

GetRingEdges, AddFace, ST_GetFaceGeometry

9.9.9 ST_GetFaceGeometry

ST_GetFaceGeometry — □□□□□□□□□□□□□□ ID □□□□□□□□□□□□□.

□□

geometry **ST_GetFaceGeometry**(varchar atopology, bigint aface);

Description

□□□□□□□□□□□□□□ ID □□□□□□□□□□□□□. □□□□□□□□□□□□□□□□□□□□□□□□□□□□□.

Availability: 1.1

This method implements the SQL/MM specification. SQL-MM 3 Topo-Geo and Topo-Net 3: Routine Details: X.3.16

□□

This query returns the WKT of the polygon added with AddFace.

Code

```
SELECT topology.ST_GetFaceGeometry('ma_topo', 1) AS facegeomwkt;
```

Output

```
POLYGON((234776.9 899563.7,234896.5 899456.7,234914 899436.4,234946.6 899356.9,
234872.5 899328.7,234891 899285.4,234992.5 899145,234890.6 899069,
234755.2 899255.4,234612.7 899379.4,234776.9 899563.7))
```

Figure



□□

AddFace

9.9.10 GetRingEdges

GetRingEdges — □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□.

□□

getfaceedges_returntype **GetRingEdges**(varchar atopology, bigint aring, integer max_edges=null);

Description

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□. □□□□□□□□ (sequence) □ □□□□□□□□ ID(edgeid) □□□□□□□□. □□□□□□ 1 □□□□□□□□.

□□□□□□□□□□□ ID □□□□□□□, □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□. □□ □□□□□□□□□ ID □□□□□□□, □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□.

`max_edges` □ NULL □□□□□□□□□□□□□□□□□□□□□□□□□□□. □□□□□□□□□□□□□ □□□□□□□□□□□□□□□□□□□□□□□□□□□.

⊠⊠
⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

ST_GetFaceEdges, GetNodeEdges getfaceedges_returntype

9.9.11 GetNodeEdges

GetNodeEdges — ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

getfaceedges_returntype **GetNodeEdges**(varchar atopology, bigint anode);

Description

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ID ⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠ 1 ⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (⊠⊠⊠⊠⊠⊠⊠⊠) ⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠
⊠⊠⊠.

2.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

getfaceedges_returntype, GetRingEdges, ST_Azimuth

9.9.12 FindVertexSegmentPairsBelowDistance

FindVertexSegmentPairsBelowDistance — Find pairs of topology vertex/segment that are closer than tolerated distance

⊠⊠

table(seg_edge bigint, segno int, vert_edge bigint, vertno int, vert_geom geometry) **FindVertexSegmentPairsBelowDistance**(varchar atopology, float8 tolerance);

Description

Returns a relation in which each tuple is a pair of vertex/segment where the vertex is NOT an endpoint of the segment but the distance between them is below the given tolerance.

Availability: 3.7

☒☒

ValidateTopologyPrecision

9.10 ☒☒☒☒☒☒

9.10.1 Polygonize

Polygonize — Finds and registers all faces defined by topology edges.

☒☒

text **Polygonize**(varchar toponame);

Description

Registers all faces that can be built out a topology edge primitives.

☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.

> **Note!**
> ☒☒
> ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒, ☒☒☒☒☒☒☒☒ Polygonize ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.

> **Note!**
> ☒☒
> ☒☒☒☒ edge ☒☒☒☒ next_left_edge ☒ next_right_edge ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.

2.0.0 ☒☒☒☒☒☒☒☒☒☒☒☒.

☒☒

AddFace, ST_Polygonize

9.10.2 AddNode

AddNode — ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒ ID(nodeid) ☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒, ☒☒☒☒ ID ☒☒☒☒☒☒.

⊠⊠

bigint **AddNode**(varchar toponame, geometry apoint, boolean allowEdgeSplitting=false, boolean computeContainingFace=false);

Description

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. AddEdge ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, `allowEdgeSplitting` ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

`computeContainingFace` ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

> **⊠⊠**
> `apoint` ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ID(nodeid) ⊠⊠⊠⊠⊠⊠.

2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

Code

```
SELECT topology.AddNode('ma_topo', ST_GeomFromText('POINT(227641.6 893816.5)', 26986)) AS ↩
    nodeid;
```

Output

```
4
```

⊠⊠

AddEdge, CreateTopology

9.10.3 AddEdge

AddEdge — ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠ (⊠⊠⊠⊠) ⊠⊠⊠⊠⊠⊠⊠ ID(edgeid) ⊠⊠⊠⊠⊠⊠.

⊠⊠

bigint **AddEdge**(varchar toponame, geometry aline);

Description

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ toponame ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠ (⊠⊠⊠⊠) ⊠⊠⊠⊠⊠⊠⊠ ID(edgeid) ⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠” ⊠⊠ (universe)” ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

> **⊠⊠**
> ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ aline ⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

> **⊠⊠**
> aline ⊠⊠⊠ srid ⊠⊠⊠⊠⊠⊠⊠⊠⊠ srid ⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

GEOS ⊠⊠⊠⊠⊠

> **⊠⊠**
> AddEdge is deprecated as of 3.5.0. Use TopoGeo_AddLineString instead.

2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

Code

```
SELECT topology.AddEdge('ma_topo', ST_GeomFromText('LINESTRING(227575.8 893917.2,227591.9 ↩
    893900.4)', 26986)) AS edgeid;
```

Output

```
1
```

Code

```
SELECT topology.AddEdge('ma_topo', ST_GeomFromText('LINESTRING(227591.9 893900.4,227622.6 ↩
    893844.2,227641.6 893816.5,227704.5 893778.5)', 26986)) AS edgeid;
```

Output

```
2
```

This query errors because the edge intersects an existing edge away from the endpoints.

Code

```
SELECT topology.AddEdge('ma_topo', ST_GeomFromText('LINESTRING(227591.2 893900,227591.9 ↩
    893900.4,227704.5 893778.5)', 26986)) AS edgeid;
```

Output

```
ERROR:  Edge intersects (not on endpoints) with existing edge 1
```

⊠⊠

TopoGeo_AddLineString, CreateTopology, 4.5⊠

9.10.4 AddFace

AddFace — ⊠⊠⊠⊠⊠⊠⊠⊠⊠ (face primitive) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

bigint **AddFace**(varchar toponame, geometry apolygon, boolean force_new=false);

Description

⊠⊠⊠⊠⊠⊠⊠⊠⊠ (face primitive) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠⊠⊠⊠⊠⊠⊠⊠ left_face ⊠ right_face ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ containing_face ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠.

> ⊠⊠
> ⊠⊠⊠⊠⊠ edge ⊠⊠⊠⊠⊠ next_left_edge ⊠ next_right_edge ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠) ⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

`apolygon` ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, `force_new` ⊠ (⊠⊠⊠⊠⊠) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ID ⊠⊠⊠⊠⊠ ⊠, `force_new` ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ID ⊠⊠⊠⊠⊠⊠⊠⊠⊠.

> ⊠⊠
> ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (force_new = true) ⊠⊠⊠, ⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ MBR ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

> ⊠⊠
> `apolygon` ⊠⊠⊠ `srid` ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ `srid` ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

First add the edges using `generate_series` as an iterator. This query only works for polygons with fewer than 10000 points because of the maximum value in `generate_series`.

Code

```
SELECT topology.AddEdge('ma_topo', ST_MakeLine(ST_PointN(geom,i), ST_PointN(geom, i + 1) )) ↩
     As edgeid
FROM (
  SELECT ST_NPoints(geom) AS npt, geom
  FROM (
    SELECT ST_Boundary(ST_GeomFromText('POLYGON((234896.5 899456.7,234914 899436.4,234946.6 ↩
     899356.9,234872.5 899328.7,
                234891 899285.4,234992.5 899145, 234890.6 899069,234755.2 899255.4,
                234612.7 899379.4,234776.9 899563.7,234896.5 899456.7))', 26986) )  As geom
  ) AS geoms
) AS facen
CROSS JOIN generate_series(1,10000) AS i
WHERE i < npt;
```

Output

```
3
4
5
6
7
8
9
10
11
12
```

Then add the face.

Code

```
SELECT topology.AddFace(
    'ma_topo',
    ST_GeomFromText(
    'POLYGON((234896.5 899456.7,234914 899436.4,234946.6 899356.9,234872.5 899328.7,
    234891 899285.4,234992.5 899145, 234890.6 899069,234755.2 899255.4,
    234612.7 899379.4,234776.9 899563.7,234896.5 899456.7))',
    26986) ) As faceid;
```

Output

```
1
```

☒☒

AddEdge, CreateTopology, 4.5☒

9.10.5 ST_Simplify

ST_Simplify — ☒☒☒☒-☒☒ (Douglas-Peucker) ☒☒☒☒☒☒☒☒☒ TopoGeometry ☒” ☒☒☒☒” ☒☒☒ ☒☒☒☒☒☒☒☒.

☒☒

geometry **ST_Simplify**(topogeometry tg, float8 tolerance);

Description

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠-⊠⊠ (Douglas-Peucker) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ TopoGeometry ⊠” ⊠⊠⊠ ⊠” ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

> **Note**
> ⊠⊠
> ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
> ⊠⊠⊠/⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

GEOS ⊠⊠⊠⊠⊠

2.1.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

⊠⊠ ST_Simplify, ST_IsSimple, ST_IsValid, ST_ModEdgeSplit

9.10.6 RemoveUnusedPrimitives

RemoveUnusedPrimitives — Removes topology primitives which not needed to define existing TopoGeometry objects.

⊠⊠

bigint **RemoveUnusedPrimitives**(text topology_name, geometry bbox);

Description

Finds all primitives (nodes, edges, faces) that are not strictly needed to represent existing TopoGeometry objects and removes them, maintaining topology validity (edge linking, face labeling) and TopoGeometry space occupation.

No new primitive identifiers are created, but rather existing primitives are expanded to include merged faces (upon removing edges) or healed edges (upon removing nodes).

Availability: 3.3.0

⊠⊠

ST_ModEdgeHeal, ST_RemEdgeModFace

9.11 TopoGeometry ⊠⊠⊠

9.11.1 CreateTopoGeom

CreateTopoGeom — ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. tg_type ⊠ 1: [⊠⊠] ⊠⊠⊠, 2: [⊠⊠] ⊠⊠, 3: [⊠⊠] ⊠⊠⊠, 4: ⊠⊠⊠⊠⊠⊠⊠.

□□

topogeometry **CreateTopoGeom**(varchar toponame, integer tg_type, integer layer_id, topoelementarray tg_objs, bigint tg_id);
topogeometry **CreateTopoGeom**(varchar toponame, integer tg_type, integer layer_id);

Description

Creates a topogeometry object for layer denoted by `layer_id` and registers it in the relations table in the `toponame` schema.

`tg_type` is an integer: 1:[multi]point (punctal), 2:[multi]line (lineal), 3:[multi]poly (areal), 4:collection. `layer_id` is the layer id in the topology.layer table.

□□□□□□□□□□□□□□□□□□, □□□□□□□□□□□□□□□□□□, □□□□□□□□□□□□□□□□□□□□, □□, □□□□, □□□□□□□□□□□□□□□□□□□□□□□□.

□□□□□□□□□□□□□□ TopoGeometry □□□□□□□□□.

Availability: 1.1

□□

Form from existing edges.

Create a topogeom in ri_topo schema for layer 2 (our ri_roads), of type (2) LINE, for the first edge (we loaded in `ST_CreateTopoGeo`).

Code

```
INSERT INTO ri.ri_roads(road_name, topo)
VALUES (
    'Unknown',
    topology.CreateTopoGeom(
        'ri_topo',
        2,
        2,
        '{{1,2}}'::topology.topoelementarray
    )
);
```

Convert an areal geometry to best guess topogeometry.

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□. □□□□ blockgroups □□□□□□□□□□□□□ TopoGeometry □□□□□□□□. □□□□□□□□□□□□□□□□□, □□□□□□□□□□□□:

Create the topogeometry column.

Code

```
SELECT topology.AddTopoGeometryColumn(
'topo_boston',
'boston', 'blockgroups', 'topo', 'POLYGON');
```

Output

```
1
```

Update the column assuming everything is perfectly aligned with the edges.

Code

```
UPDATE boston.blockgroups AS bg
SET topo = topology.CreateTopoGeom(
    'topo_boston',
    3,
    1,
    foo.bfaces
)
FROM (
    SELECT
        b.gid,
        topology.TopoElementArray_Agg(ARRAY[f.face_id, 3]) AS bfaces
    FROM boston.blockgroups AS b
    INNER JOIN topo_boston.face AS f
        ON b.geom && f.mbr
    WHERE ST_Covers(
        b.geom,
        topology.ST_GetFaceGeometry('topo_boston', f.face_id)
    )
    GROUP BY b.gid
) AS foo
WHERE foo.gid = bg.gid;
```

The world is rarely perfect, so this version allows some error. It counts a face if 50 percent of it falls within the expected block group boundary.

Code

```
WITH candidate_faces AS (
    SELECT
        b.gid,
        b.geom AS block_geom,
        f.face_id,
        topology.ST_GetFaceGeometry('topo_boston', f.face_id) AS face_geom
    FROM boston.blockgroups AS b
    INNER JOIN topo_boston.face AS f
        ON b.geom && f.mbr
),
block_faces AS (
    SELECT
        gid,
        topology.TopoElementArray_Agg(ARRAY[face_id, 3]) AS bfaces
    FROM candidate_faces
    WHERE ST_Covers(block_geom, face_geom)
        OR (
            ST_Intersects(block_geom, face_geom)
            AND ST_Area(ST_Intersection(block_geom, face_geom))

> ST_Area(face_geom) * 0.5
        )
    GROUP BY gid
)
UPDATE boston.blockgroups AS bg
SET topo = topology.CreateTopoGeom(
    'topo_boston',
    3,
    1,
    block_faces.bfaces
)
FROM block_faces
WHERE block_faces.gid = bg.gid;
```

To convert the topogeometry back to a denormalized geometry aligned with faces and edges, cast the topogeometry to a geometry. The resulting geometries are aligned with the TIGER street centerlines.

Code

```
UPDATE boston.blockgroups SET new_geom = topo::geometry;
```

⊠⊠

AddTopoGeometryColumn, toTopoGeom ST_CreateTopoGeo, ST_GetFaceGeometry, TopoElementArray, TopoElementArray_Agg

9.11.2 toTopoGeom

toTopoGeom — Converts a simple Geometry into a topo geometry.

⊠⊠

topogeometry **toTopoGeom**(geometry geom, varchar toponame, integer layer_id, float8 tolerance=-1);
topogeometry **toTopoGeom**(geometry geom, topogeometry topogeom, float8 tolerance=-1);

Description

⊠⊠⊠⊠⊠ TopoGeometry ⊠⊠⊠⊠⊠⊠.

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠ `relation` ⊠⊠⊠⊠⊠⊠⊠ TopoGeometry ⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠ TopoGeometry ⊠⊠⊠ (`topogeom` ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

When `tolerance` is given it will be used to snap the input geometry to existing primitives. The special value -1 (default) means use the topology precision as tolerance.

⊠⊠ 1 ⊠⊠⊠⊠⊠⊠⊠ (`toponame`) ⊠⊠⊠⊠⊠⊠ (`layer_id`) ⊠⊠⊠⊠ TopoGeometry ⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠ 2 ⊠⊠⊠, ⊠⊠ TopoGeometry(`toponame`) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠ TopoGeometry ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ clearTopoGeom ⊠⊠⊠⊠⊠⊠⊠.

2.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠⊠⊠: 2.1.0 ⊠⊠⊠⊠⊠⊠ TopoGeometry ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

Changed: 3.7.0 changes meaning of tolerance 0 to really mean 0 and -1 (new default) to request using the topology precision.

⊠⊠

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (workflow) ⊠⊠⊠.

Create the topology without allowing any tolerance.

Code

```
SELECT topology.CreateTopology('topo_boston_test', 2249);
```

Create a new table.

Code

```
CREATE TABLE nei_topo(gid serial primary key, nei varchar(30));
```

Add a topogeometry column to the new table.

Code

```
SELECT topology.AddTopoGeometryColumn('topo_boston_test', 'public', 'nei_topo', 'topo', ' ↩
    MULTIPOLYGON') AS new_layer_id;
```

Output

```
1
```

Use the new layer identifier to populate the topogeometry column.

Code

```
INSERT INTO nei_topo(nei, topo)
SELECT nei, topology.toTopoGeom(geom, 'topo_boston_test', 1)
FROM neighborhoods
WHERE gid BETWEEN 1 and 15;
```

Verify what happened.

Code

```
SELECT * FROM
    topology.TopologySummary('topo_boston_test');
```

Output

```
Topology topo_boston_test (5), SRID 2249, precision 0
61 nodes, 87 edges, 35 faces, 15 topogeoms in 1 layers
Layer 1, type Polygonal (3), 15 topogeoms
 Deploy: public.nei_topo.topo
```

Shrink all TopoGeometry polygons by 10 meters.

Code

```
UPDATE nei_topo SET topo = toTopoGeom(ST_Buffer(topo, -10), clearTopoGeom(topo), 0);
```

Find faces that are left without any TopoGeometry reference after the shrink operation.

Code

```
SELECT ST_GetFaceGeometry('topo_boston_test', f.face_id)
  FROM topo_boston_test.face f
  WHERE f.face_id
> 0
  AND NOT EXISTS (
    SELECT * FROM topo_boston_test.relation
    WHERE layer_id = 1 AND element_id = f.face_id
  );
```

□□

CreateTopology, AddTopoGeometryColumn, CreateTopoGeom, TopologySummary, clearTopoGeom

9.11.3 TopoElementArray_Agg

TopoElementArray_Agg — Returns a `topoelementarray` for a set of element_id, type arrays (topoelements).

□□

topoelementarray **TopoElementArray_Agg**(topoelement set tefield);

Description

TopoElement □□□□□□ TopoElementArray □□□□□□□□□□.
2.0.0 □□□□□□□□□□□□.

□□

Code

```
SELECT topology.TopoElementArray_Agg(ARRAY[e, t] ORDER BY e, t) As tea
FROM generate_series(1, 3) As e CROSS JOIN generate_series(1, 4) As t;
```

Output

```
{{1,1},{1,2},{1,3},{1,4},{2,1},{2,2},{2,3},{2,4},{3,1},{3,2},{3,3},{3,4}}
```

□□

TopoElement, TopoElementArray

9.11.4 TopoElement

TopoElement — Converts a topogeometry to a topoelement.

□□

topoelement **TopoElement**(topogeometry topo);

Description

Converts a TopoGeometry to a TopoElement.
Availability: 3.4.0

□□

□□□□□□□□□□□□□□□□□□□□ (workflow) □□□.
Return the TopoElement representation of each topogeometry.

Code

```
SELECT TopoElement(topo)
FROM neighborhoods;
```

Use a cast to aggregate topogeometries into a TopoElementArray.

Code

```
SELECT topology.TopoElementArray_Agg(topo::topoelement)
FROM neighborhoods
GROUP BY city;
```

⊠⊠

TopoElementArray_Agg, TopoGeometry, TopoElement

9.12 TopoGeometry ⊠⊠⊠

9.12.1 clearTopoGeom

clearTopoGeom — Clears the content of a topo geometry.

⊠⊠

topogeometry **clearTopoGeom**(topogeometry topogeom);

Description

TopoGeometry ⊠⊠⊠⊠⊠⊠⊠⊠⊠ TopoGeometry ⊠⊠⊠⊠⊠. toTopoGeom ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

2.1 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

Shrink all TopoGeometry polygons by 10 meters.

Code

```
UPDATE nei_topo SET topo = toTopoGeom(ST_Buffer(topo, -10), clearTopoGeom(topo), 0);
```

⊠⊠

toTopoGeom

9.12.2 TopoGeom_addElement

TopoGeom_addElement — Adds an element to the definition of a TopoGeometry.

⊠⊠

topogeometry **TopoGeom_addElement**(topogeometry tg, topoelement el);

Description

TopoGeometry ⊠⊠⊠⊠⊠⊠ TopoElement ⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠.

2.3 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

Add edge 5 to TopoGeometry `tg`.

Code

```
UPDATE mylayer SET tg = TopoGeom_addElement(tg, '{5,2}');
```

⊠⊠

TopoGeom_remElement, CreateTopoGeom

9.12.3 TopoGeom_remElement

TopoGeom_remElement — Removes an element from the definition of a TopoGeometry.

⊠⊠

topogeometry **TopoGeom_remElement**(topogeometry tg, topoelement el);

Description

TopoGeometry ⊠⊠⊠⊠⊠⊠⊠ TopoElement ⊠⊠⊠⊠⊠⊠.

2.3 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

Remove face 43 from TopoGeometry `tg`.

Code

```
UPDATE mylayer SET tg = TopoGeom_remElement(tg, '{43,3}');
```

⊠⊠

TopoGeom_addElement, CreateTopoGeom

9.12.4 TopoGeom_addTopoGeom

TopoGeom_addTopoGeom — Adds element of a TopoGeometry to the definition of another TopoGeometry.

☒☒

topogeometry **TopoGeom_addTopoGeom**(topogeometry tgt, topogeometry src);

Description

Adds the elements of a TopoGeometry to the definition of another TopoGeometry, possibly changing its cached type (type attribute) to a collection, if needed to hold all elements in the source object.

The two TopoGeometry objects need be defined against the *same* topology and, if hierarchically defined, need be composed by elements of the same child layer.

Availability: 3.2

☒☒

Set an overall TopoGeometry value to be composed of all elements from specific TopoGeometry values.

Code

```
UPDATE mylayer SET tg_overall = TopoGeom_addTopoGeom(
    TopoGeom_addTopoGeom(
        clearTopoGeom(tg_overall),
        tg_specific1
    ),
    tg_specific2
);
```

☒☒

TopoGeom_addElement, clearTopoGeom, CreateTopoGeom

9.12.5 toTopoGeom

toTopoGeom — Adds a geometry shape to an existing topo geometry.

Description

Refer to toTopoGeom.

9.13 TopoGeometry ☒☒☒

9.13.1 GetTopoGeomElementArray

GetTopoGeomElementArray — Returns a `topoelementarray` (an array of topoelements) containing the topological elements and type of the given TopoGeometry (primitive elements).

□□

topoelementarray **GetTopoGeomElementArray**(varchar toponame, integer layer_id, bigint tg_id);

topoelementarray **GetTopoGeomElementArray**(topogeometry tg);

Description

□□□□□□□□ TopoGeometry □□□ (□□□□□) □□□□□ TopoElementArray □□□□□□. □□□□ □□□□□□□□□□□□□□□□□□□□□□□□□□□□ GetTopoGeomElements □□□□□□□□.

tg_id □ topology.layer □□□□□□ `layer_id` □□□□□□□□□□□□□□□□□□ TopoGeometry □□ □ TopoGeometry ID □□□.

Availability: 1.1

□□

□□

GetTopoGeomElements, TopoElementArray

9.13.2 GetTopoGeomElements

GetTopoGeomElements — Returns a set of `topoelement` objects containing the topological element_id,eleme of the given TopoGeometry (primitive elements).

□□

setof topoelement **GetTopoGeomElements**(varchar toponame, integer layer_id, bigint tg_id);

setof topoelement **GetTopoGeomElements**(topogeometry tg);

Description

Returns a set of element_id,element_type (topoelements) corresponding to primitive topology elements TopoElement (1: nodes, 2: edges, 3: faces) that a given topogeometry object in `toponame` schema is composed of.

tg_id □ topology.layer □□□□□□ `layer_id` □□□□□□□□□□□□□□□□□□ TopoGeometry □□ □ TopoGeometry ID □□□.

2.0.0 □□□□□□□□□□□□.

□□

□□

GetTopoGeomElementArray, TopoElement, TopoGeom_addElement, TopoGeom_remElement

9.13.3 ST_SRID

ST_SRID — Returns the spatial reference identifier for a topogeometry.

□□

integer **ST_SRID**(topogeometry tg);

Description

Returns the spatial reference identifier of the topology containing the TopoGeometry.
The identifier is recorded for the topology when it is created and refers to an entry in 4.5□.
Availability: 3.2.0

□□

Return the SRID of a TopoGeometry stored in the `ri_topo` topology.

Code

```
SELECT topology.ST_SRID(topo) AS srid
FROM ri.roads
WHERE road_name = 'Unknown';
```

Output

```
srid
------
 3438
```

□□

4.5□, ST_SetSRID, ST_Transform, ST_SRID

9.14 TopoGeometry □□□

9.14.1 AsGML

AsGML — TopoGeometry □ GML □□□□□□□□□.

□□

text **AsGML**(topogeometry tg);
text **AsGML**(topogeometry tg, text nsprefix_in);
text **AsGML**(topogeometry tg, regclass visitedTable);
text **AsGML**(topogeometry tg, regclass visitedTable, text nsprefix);
text **AsGML**(topogeometry tg, text nsprefix_in, integer precision, integer options);
text **AsGML**(topogeometry tg, text nsprefix_in, integer precision, integer options, regclass visitedTable);
text **AsGML**(topogeometry tg, text nsprefix_in, integer precision, integer options, regclass visitedTable, text idprefix);
text **AsGML**(topogeometry tg, text nsprefix_in, integer precision, integer options, regclass visitedTable, text idprefix, int gmlversion);

Description

TopoGeometry ⊠ GML ⊠⊠⊠⊠ GML3 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. nsprefix_in ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ gml ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. nsprefix ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (non-qualified) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠. ⊠⊠⊠ (⊠⊠⊠ 15) ⊠⊠⊠ (⊠⊠⊠ 1) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠ ST_AsGML ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

visitedTable ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠ (xlink:xref) ⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠'element_type' ⊠'element_id' (⊠⊠⊠) 2 ⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, element_type ⊠ element_id ⊠, ⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠:

Code

```
CREATE TABLE visited (
  element_type integer, element_id integer,
  unique(element_type, element_id)
);
```

idprefix ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

gmlver ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠ ST_AsGML ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠ 3 ⊠⊠⊠.

2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

⊠⊠⊠⊠⊠ CreateTopoGeom ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

Code

```
SELECT topology.AsGML(topo) As rdgml
  FROM ri.roads
  WHERE road_name = 'Unknown';
```

Output

```
<gml:TopoCurve>
    <gml:directedEdge>
        <gml:Edge gml:id="E1">
            <gml:directedNode orientation="-">
                <gml:Node gml:id="N1"/>
            </gml:directedNode>
            <gml:directedNode
></gml:directedNode>
            <gml:curveProperty>
                <gml:Curve srsName="urn:ogc:def:crs:EPSG::3438">
                    <gml:segments>
                        <gml:LineStringSegment>
                            <gml:posList srsDimension="2"
>384744 236928 384750 236923 384769 236911 384799 236895 384811 236890
                            384833 236884 384844 236882 384866 236881 384879 236883 384954 236898 385087 236932 385117 ←
      236938
                            385167 236938 385203 236941 385224 236946 385233 236950 385241 236956 385254 236971
                            385260 236979 385268 236999 385273 237018 385273 237037 385271 237047 385267 237057 385225 ←
      237125
                            385210 237144 385192 237161 385167 237192 385162 237202 385159 237214 385159 237227 385162 ←
      237241
                            385166 237256 385196 237324 385209 237345 385234 237375 385237 237383 385238 237399 385236 ←
      237407
                            385227 237419 385213 237430 385193 237439 385174 237451 385170 237455 385169 237460 385171 ←
      237475
                            385181 237503 385190 237521 385200 237533 385206 237538 385213 237541 385221 237542 385235 ←
      237540 385242 237541
                            385249 237544 385260 237555 385270 237570 385289 237584 385292 237589 385291 237596 385284 ←
      237630</gml:posList>
                        </gml:LineStringSegment>
                    </gml:segments>
```

```
                </gml:Curve>
            </gml:curveProperty>
        </gml:Edge>
    </gml:directedEdge>
</gml:TopoCurve>
```

□□□□□□□□□□□□□□□□□□□□□□□□□□□□.

Code

```
SELECT topology.AsGML(topo,'') As rdgml
  FROM ri.roads
  WHERE road_name = 'Unknown';
```

Output

```
<TopoCurve>
    <directedEdge>
        <Edge id="E1">
            <directedNode orientation="-">
                <Node id="N1"/>
            </directedNode>
            <directedNode
></directedNode>
            <curveProperty>
                <Curve srsName="urn:ogc:def:crs:EPSG::3438">
                    <segments>
                        <LineStringSegment>
                            <posList srsDimension="2"
>384744 236928 384750 236923 384769 236911 384799 236895 384811 236890
                            384833 236884 384844 236882 384866 236881 384879 236883 384954 236898 385087 236932 385117 ↩
      236938
                            385167 236938 385203 236941 385224 236946 385233 236950 385241 236956 385254 236971
                            385260 236979 385268 236999 385273 237018 385273 237037 385271 237047 385267 237057 385225 ↩
      237125
                            385210 237144 385192 237161 385167 237192 385162 237202 385159 237214 385159 237227 385162 ↩
      237241
                            385166 237256 385196 237324 385209 237345 385234 237375 385237 237383 385238 237399 385236 ↩
      237407
                            385227 237419 385213 237430 385193 237439 385174 237451 385170 237455 385169 237460 385171 ↩
      237475
                            385181 237503 385190 237521 385200 237533 385206 237538 385213 237541 385221 237542 385235 ↩
      237540 385242 237541
                            385249 237544 385260 237555 385270 237570 385289 237584 385292 237589 385291 237596 385284 ↩
      237630</posList>
                        </LineStringSegment>
                    </segments>
                </Curve>
            </curveProperty>
        </Edge>
    </directedEdge>
</TopoCurve>
```

□□

CreateTopoGeom, ST_CreateTopoGeo

9.14.2 AsTopoJSON

AsTopoJSON — TopoGeometry □ TopoJSON □□□□□□□□□.

□□

text **AsTopoJSON**(topogeometry tg, regclass edgeMapTable);

Description

TopoGeometry □ TopoJSON □□□□□□□□□. `edgeMapTable` □ NULL □□□□□, □□□□□□□□□ (arc) □□□□□□□□□□□□/□□ (lookup/storage) □□□□□□□□□□. □□□□□□□□□ (compact) ” □□” □□□□□□□□□□□□□□□□□□.

□□□□□□□□□□, □□□□□” □□ (serial)” □□”arc_id” □□□□□□□”edge_id” □□□□□□□□□□□□□. □□□”edge_id” □□.

> **□□**
>
> TopoJSON □□□□□□□□□□□□□□ 0-□□□□□□”edgeMapTable” □□□□□□□ 1-□□□□□□.

□□□ TopoJSON □□□, □□□□□□□□□□□□□□ (snippet) □□□, □□□□□□□□□□□□□□□□□□□□□□□□□□□. TopoJSON □□□ □□□□□□□□.

2.1.0 □□□□□□□□□□□□□□.

□□□□: 2.2.1 □□□□□□□ (puntal) □□□□□□□□□□□□□□□.

□□

ST_AsGeoJSON

□□

Create a temporary edge map for the TopoJSON output.

Code

```
CREATE TEMP TABLE edgemap(arc_id serial, edge_id int unique);
```

Build the TopoJSON header, objects, arcs, and footer.

Code

```
SELECT '{ "type": "Topology", "transform": { "scale": [1,1], "translate": [0,0] }, "objects ↩
    ": {'

UNION ALL SELECT '"' || feature_name || '": ' || AsTopoJSON(feature, 'edgemap')
FROM features.big_parcels WHERE feature_name = 'P3P4';

WITH edges AS (
  SELECT m.arc_id, e.geom FROM edgemap m, city_data.edge e
  WHERE e.edge_id = m.edge_id
), points AS (
  SELECT arc_id, (ST_DumpPoints(geom)).* FROM edges
), compare AS (
  SELECT p2.arc_id,
         CASE WHEN p1.path IS NULL THEN p2.geom
              ELSE ST_Translate(p2.geom, -ST_X(p1.geom), -ST_Y(p1.geom))
         END AS geom
  FROM points p2 LEFT OUTER JOIN points p1
  ON ( p1.arc_id = p2.arc_id AND p2.path[1] = p1.path[1]+1 )
  ORDER BY arc_id, p2.path
), arcsdump AS (
  SELECT arc_id, (regexp_matches(ST_AsGeoJSON(geom), '\[.*\]'))[1] as t
```

```
  FROM compare
), arcs AS (
  SELECT arc_id, '[' || array_to_string(array_agg(t), ',') || ']' as a FROM arcsdump
  GROUP BY arc_id
  ORDER BY arc_id
)
SELECT '}, "arcs": [' UNION ALL
SELECT array_to_string(array_agg(a), E',\n') from arcs

UNION ALL SELECT ']}'::text as t;
```

Output

```
{ "type": "Topology", "transform": { "scale": [1,1], "translate": [0,0] }, "objects": {
"P3P4": { "type": "MultiPolygon", "arcs": [[[-1]],[[6,5,-5,-4,-3,1]]]}
}, "arcs": [
 [[25,30],[6,0],[0,10],[-14,0],[0,-10],[8,0]],
 [[35,6],[0,8]],
 [[35,6],[12,0]],
 [[47,6],[0,8]],
 [[47,14],[0,8]],
 [[35,22],[12,0]],
 [[35,14],[0,8]]
 ]}
```

9.15 □□□□□□

9.15.1 Equals

Equals — □ TopoGeometry □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□.

□□

boolean **Equals**(topogeometry tg1, topogeometry tg2);

Description

□ TopoGeometry □□□□□□□□□□ (□□, □□□□, □□) □□□□□□□□□□□□□□□□□□□□.



□□
□□□□□□□□□□ TopoGeometry □□□□□□□□□□. □□□□□□□□□□□□□□□□ TopoGeometry □□□□□□□□□□.

1.1.0 □□□□□□□□□□□□□□.



This function supports 3d and will not drop the z-index.

□□

The following example uses the `nei_topo` table created in the toTopoGeom examples.

Code

```
SELECT topology.Equals(a.topo, b.topo)
FROM nei_topo AS a
JOIN nei_topo AS b ON b.gid = a.gid
WHERE a.gid = 1;
```

Output

```
t
```

⊠⊠

GetTopoGeomElements, ST_Equals

9.15.2 Intersects

Intersects — ⊠ TopoGeometry ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

boolean **Intersects**(topogeometry tg1, topogeometry tg2);

Description

⊠ TopoGeometry ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

> **⊠⊠**
> This function not supported for topogeometries that are geometry collections. It also can not compare topogeometries from different topologies. Also not currently supported for hierarchical topogeometries (topogeometries composed of other topogeometries).

1.1.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

This function supports 3d and will not drop the z-index.

⊠⊠

The following example uses the `nei_topo` table created in the toTopoGeom examples.

Code

```
SELECT topology.Intersects(a.topo, b.topo)
FROM nei_topo AS a
JOIN nei_topo AS b ON b.gid = a.gid
WHERE a.gid = 1;
```

Output

```
t
```

□□

ST_Intersects

9.16 Importing and exporting Topologies

Once you have created topologies, and maybe associated topological layers, you might want to export them into a file-based format for backup or transfer into another database.

Using the standard dump/restore tools of PostgreSQL is problematic because topologies are composed by a set of tables (4 for primitives, an arbitrary number for layers) and records in metadata tables (topology.topology and topology.layer). Additionally, topology identifiers are not univoque across databases so that parameter of your topology will need to be changes upon restoring it.

In order to simplify export/restore of topologies a pair of executables are provided: `pgtopo_export` and `pgtopo_import`. Example usage:

Code

```
pgtopo_export dev_db topo1 | pgtopo_import topo1 | psql staging_db
```

9.16.1 Using the Topology exporter

The `pgtopo_export` script takes the name of a database and a topology and outputs a dump file which can be used to import the topology (and associated layers) into a new database.

By default `pgtopo_export` writes the dump file to the standard output so that it can be piped to `pgtopo_import` or redirected to a file (refusing to write to terminal). You can optionally specify an output filename with the `-f` commandline switch.

By default `pgtopo_export` includes a dump of all layers defined against the given topology. This may be more data than you need, or may be non-working (in case your layer tables have complex dependencies) in which case you can request skipping the layers with the `--skip-layers` switch and deal with those separately.

Invoking `pgtopo_export` with the `--help` (or `-h` for short) switch will always print short usage string.

The dump file format is a compressed tar archive of a `pgtopo_export` directory containing at least a `pgtopo_dump_version` file with format version info. As of version 1 the directory contains tab-delimited CSV files with data of the topology primitive tables (node, edge_data, face, relation), the topology and layer records associated with it and (unless `--skip-layers` is given) a custom-format PostgreSQL dump of tables reported as being layers of the given topology.

9.16.2 Using the Topology importer

The `pgtopo_import` script takes a `pgtopo_export` format topology dump and a name to give to the topology to be created and outputs an SQL script reconstructing the topology and associated layers.

The generated SQL file will contain statements that create a topology with the given name, load primitive data in it, restores and registers all topology layers by properly linking all TopoGeometry values to their correct topology.

By default `pgtopo_import` reads the dump from the standard input so that it can be used in conjunction with `pgtopo_export` in a pipeline. You can optionally specify an input filename with the `-f` commandline switch.

By default `pgtopo_import` includes in the output SQL file the code to restore all layers found in the dump.

This may be unwanted or non-working in case your target database already have tables with the same name as the ones in the dump. In that case you can request skipping the layers with the `--skip-layers` switch and deal with those separately (or later).

SQL to only load and link layers to a named topology can be generated using the `--only-layers` switch. This can be useful to load layers AFTER resolving the naming conflicts or to link layers to a different topology (say a spatially-simplified version of the starting topology).

If the target topology already exists and you want it dropped upfront you can pass the `--drop-topology` switch (since PostGIS-3.6.0).

Chapter 10

□□□□□□□□□, □□□□□

10.1 □□□□□□□□

□□□□□□, □□□□□□□□□□□□ raster2pgsql □□□□□□□□□□□□□□□□□□□□□□□□ PostGIS □ □□□□□□□□□□□□.

10.1.1 raster2pgsql □□□□□□□□□□□□□

The raster2pgsql is a raster loader executable that loads GDAL supported raster formats into SQL suitable for loading into a PostGIS raster table. It is capable of loading folders of raster files as well as creating overviews of rasters.

Since the raster2pgsql is compiled as part of PostGIS most often (unless you compile your own GDAL library), the raster types supported by the executable will be the same as those compiled in the GDAL dependency library. To get a list of raster types your particular raster2pgsql supports use the -G switch.



□□
□□□□□□□□□□□□□□□ (factor) □□□□□□□□□□□□, □□□□□□□□□□□□□□□□□□ □. http://trac.osgeo.org/postgis/ticket/1764 □□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□.

10.1.1.1 Example Usage

□□□□□□□□□□□□□□□ 100x100 □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□:

The loader assigns SRID 4326, creates an index and standard constraints, analyzes the loaded table, records source filenames, and stores the input rasters in 100 by 100 pixel tiles.

Code

```
raster2pgsql -s 4326 -I -C -M -F -t 100x100 *.tif public.demelevation
> elev.sql
psql -d gisdb -f elev.sql
```



□□
If you do not specify the schema as part of the target table name, the table will be created in the default schema of the database or user you are connecting with.

UNIX ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠:

Code

```
raster2pgsql -s 4326 -I -C -M -F -t 100x100 \
  *.tif public.demelevation |
  psql -d gisdb
```

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ aerial ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠ 2 ⊠⊠⊠ 4 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠ (⊠⊠⊠⊠⊠⊠⊠⊠ DB ⊠⊠⊠) ⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ -e ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (⊠⊠). ⊠⊠⊠⊠ 128x128 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ -F ⊠⊠⊠⊠”filename” ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

Code

```
raster2pgsql -I -C -e -Y -F -s 26986 -t 128x128 -l 2,4 \
  bostonaerials2008/*.jpg aerials.boston |
  psql -U postgres -d gisdb -h localhost -p 5432
```

List the raster formats supported by this build:

Code

```
raster2pgsql -G
```

The `-G` command outputs a list something like:

Output

```
Available GDAL raster formats:
  Virtual Raster
  GeoTIFF
  National Imagery Transmission Format
  Raster Product Format TOC format
  ECRG TOC format
  Erdas Imagine Images (.img)
  CEOS SAR Image
  CEOS Image
  ...
  Arc/Info Export E00 GRID
  ZMap Plus Grid
  NOAA NGS Geoid Height Grids
```

10.1.1.2 raster2pgsql options

Long options that take values also accept the `--option=value` spelling.

-?, --help ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

-G, --gdal-formats ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

c|a|d|p -- ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠:

- **-c** ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (⊠) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- **-a** ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (⊠) ⊠⊠⊠⊠⊠⊠.
- **-d** ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (⊠) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- **-p** ⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

--if-not-exists Use `IF NOT EXISTS` for table creation in `-c` and `-p` modes. When `-I` or `--create-index` is also specified, use `IF NOT EXISTS` for index creation too. This option cannot be used with `-d`. Append mode requires an explicit creation action.

Atomic loader actions The short modes are presets over explicit actions. The equivalent long options are `--drop-table`, `--create-table`, `--load-data`, `--create-index`, `--add-constraints`, `--vacuum`, `--analyze`, and `--no-transaction`. Add `--if-not-exists` to make active creation actions use `IF NOT EXISTS`. `--drop-table` emits `DROP TABLE IF EXISTS` before the selected actions. With no mode specified, the default create/load actions still apply.

□□□□□□□: □□□□□□□□□□□□□□□□□□□□□□□□□

-C, --add-constraints `raster_columns` □□□□□□□□□□□□□□□ SRID, □□□□□□□□ □□□□□□□□□□□□□.

-x, --no-extent Disable setting the max extent constraint. Only applied if -C or --add-constraints is also used.

-r, --regular-blocking Set the constraints (spatially unique and coverage tile) for regular blocking. Only applied if -C or --add-constraints is also used.

□□□□□□□: □□□□□□□□□□□□□□□□□□□□□□□□□

-s, --srid [<FROM_SRID>:]<SRID> Assign the output raster to the specified SRID. When FROM_SRID is also provided, raster2pgsql marks the input raster with FROM_SRID and emits SQL that reprojects the raster to the target SRID. If the source SRID is omitted or is zero, the raster metadata will be checked to determine an appropriate source SRID. Reprojection cannot be used with copy mode.

-b, --band BAND □□□□□□□□□ (1-□□) □□□□□□□□□□. □□□□□□□□□□□□□□□□, □□ (,) □□□□□□□□. □□□□□□□□□□□□□□□□□□□□□□□□□.

-t, --tile-size TILE_SIZE □□□□□□□□□□□□□□□□□□□□□□□□□□□□. `TILE_SIZE` □□□□ x □□□□□□□□□, □□"auto" □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□□.

-P, --pad □□□□□□□□□□□□□□□□□□□□□ (padding) □□□□□□□□□□□□□□□□□□□□ □□□□□.

-R, --register □□□□□□□□□□ (DB □□) □□□□□□□□□□. □□□□□□□□□□□□ (□□□□□□□) □□□□□□□□□□□□□□□□□□□□□.

-l, --overview-factor OVERVIEW_FACTOR Create overview of the raster. For more than one factor, separate with comma(,). Overview table name follows the pattern o_`overview_factor`_`tabl` where `overview_factor` is a placeholder for numerical overview factor and `table` is replaced with the base table name. If a source raster already has an overview with matching dimensions, that overview is reused. Created overview is stored in the database and is not affected by -R. Note that your generated sql file will contain both the main table and overview tables.

-N, --nodata NODATA "NODATA" □□□□□□□□□□□ NODATA □□□□□.

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

-f, --raster-column COLUMN □□□□□□□□□□□□□□□□□. □□□□'rast' □□□.

-F, --filename □□□□□□□□□□□□□□□.

-n, --filename-column COLUMN □□□□□□□□□□□□□□□. -F □□□□□□□□□□□□□.

-q, --quote PostgreSQL □□□□□□□□□□□□□.

-I Alias for `--create-index`.

--create-index Create a GiST index on the raster column at the end of this raster2pgsql run. With repeated `-a` append runs, create the index on the final run or after loading; add `--if-not-exis` to make reruns tolerate an existing index.

-M □□□□□□□□□□□□□ (vacuum analyze) □□□□.

-k, --skip-nodata-check Keeps empty tiles and skips NODATA value checks for each raster band. Note you save time in checking, but could end up with far more junk rows in your database and those junk rows are not marked as empty tiles.

-T, --tablespace tablespace ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. -X ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠ (⊠⊠⊠⊠⊠⊠⊠) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

-X, --index-tablespace tablespace ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. -I ⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

-Y, --copy [MAX_ROWS_PER_COPY] Use copy statements instead of insert statements. Optionally specify `max_rows_per_copy`; default 50 when not specified.

-e, --no-transaction ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠ (transaction) ⊠⊠⊠⊠⊠⊠⊠⊠⊠.

-E, --endian ENDIAN ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (endianness) ⊠⊠⊠⊠⊠⊠⊠. XDR ⊠ 0, ⊠⊠⊠⊠ NDR ⊠ 1 ⊠⊠⊠⊠⊠⊠⊠. ⊠⊠, NDR ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

-V, --wkb-version version ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠ 0 ⊠⊠⊠. ⊠⊠, 0 ⊠⊠⊠⊠⊠⊠⊠.

10.1.2 PostGIS ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠

⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

1. ⊠⊠:

   Code

   ```
   CREATE TABLE myrasters(rid serial primary key, rast raster);
   ```

2. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ST_MakeEmptyRast ⊠⊠⊠ ST_AddBand ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

   ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠ ST_AsRaster ⊠⊠⊠, ⊠⊠⊠ ST_Union , ST_MapAlgebraF ⊠⊠⊠⊠⊠⊠ (algebra) ⊠⊠.

   ⊠⊠. ⊠⊠⊠⊠⊠ ST_Transform ⊠ ⊠⊠⊠.

3. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠:

   Code

   ```
   CREATE INDEX myrasters_rast_st_convexhull_idx ON myrasters USING gist(ST_ConvexHull( ↩
      rast));
   ```

   ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (convex hull) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ST_ConvexHull ⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

   > ⊠⊠
   > PostGIS 2.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (envelop) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠. ⊠⊠.

4. AddRasterConstraints ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠

10.1.3 Using ”out db” cloud rasters

The `raster2pgsql` tool uses GDAL to access raster data, and can take advantage of a key GDAL feature: the ability to read from rasters that are stored remotely in cloud ”object stores” (e.g. AWS S3, Google Cloud Storage).

Efficient use of cloud stored rasters requires the use of a ”cloud optimized” format. The most well-known and widely used is the ”cloud optimized GeoTIFF” format. Using a non-cloud format, like a JPEG, or an un-tiled TIFF will result in very poor performance, as the system will have to download the entire raster each time it needs to access a subset.

First, load your raster into the cloud storage of your choice. Once it is loaded, you will have a URI to access it with, either an ”http” URI, or sometimes a URI specific to the service. (e.g., ”s3://bucket/object”). To access non-public buckets, you will need to supply GDAL config options to authenticate your connection. Note that this command is *reading* from the cloud raster and *writing* to the database.

Code

```
AWS_ACCESS_KEY_ID=xxxxxxxxxxxxxxxxxxx \
AWS_SECRET_ACCESS_KEY=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx \
raster2pgsql \
  -s 990000 \
  -t 256x256 \
  -I \
  -R \
  /vsis3/your.bucket.com/your_file.tif \
  your_table \
  | psql your_db
```

Once the table is loaded, you need to give the database permission to read from remote rasters, by setting two permissions, postgis.enable_outdb_rasters and postgis.gdal_enabled_drivers.

Code

```
SET postgis.enable_outdb_rasters = true;
SET postgis.gdal_enabled_drivers TO 'ENABLE_ALL';
```

To make the changes sticky, set them directly on your database. You will need to re-connect to experience the new settings.

Code

```
ALTER DATABASE your_db SET postgis.enable_outdb_rasters = true;
ALTER DATABASE your_db SET postgis.gdal_enabled_drivers TO 'ENABLE_ALL';
```

For non-public rasters, you may have to provide access keys to read from the cloud rasters. The same keys you used to write the `raster2pgsql` call can be set for use inside the database, with the postgis.gdal_vsi_options configuration. Note that multiple options can be set by space-separating the `key=value` pairs.

Code

```
SET postgis.gdal_vsi_options = 'AWS_ACCESS_KEY_ID=xxxxxxxxxxxxxxxxxxx
AWS_SECRET_ACCESS_KEY=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx';
```

Once you have the data loaded and permissions set you can interact with the raster table like any other raster table, using the same functions. The database will handle all the mechanics of connecting to the cloud data when it needs to read pixel data.

10.2 Raster Coverages and Raster/Vector Analysis

A coverage is a way to interpret spatial data as a mapping from a *domain* of locations to a *range* of values. Evaluating a coverage means asking for the value at a location; an inverse query asks for the

locations whose values match a condition. PostGIS does not expose a generic ISO 19123 coverage SQL type, but geometry tables and raster values provide the building blocks for both directions.

A coverage can be continuous, with values calculated or interpolated for arbitrary locations, or discrete, with values supplied by stored spatial objects or grid cells. The coverage model describes how locations map to values, independently of whether those values come from an equation, a table, or one or more raster values. A time column or function argument can add a temporal dimension without changing that distinction.

In a vector coverage, a geometry column supplies the domain and other columns supply attributes in the range. A vector range can contain text, numbers, or another spatial value, provided its meaning and type are consistent wherever the coverage is defined. In a raster, each cell location is part of the domain and the band values at that cell form the range. Other columns in the same table row describe the raster value or tile as a whole; they do not attach separate values to individual cells.

A logical raster coverage may be stored as one raster value or as many tiles in a table. Tiles that belong to one coverage should have compatible SRIDs, scales, alignment, band counts, pixel types, and NODATA conventions. The constraints exposed by 10.3.1☒ record and validate those properties.

Raster table arrangements. A raster column can represent an image warehouse, a tiled coverage, or a collection of rasterized objects. These arrangements have different assumptions about tile size, alignment, gaps, and overlaps. PostGIS does not infer the intended arrangement from the column type alone.

Arrangement	Row relationship	Typical use
Image warehouse	Independent rasters; sizes, alignment, and extents may differ or overlap	Cataloging unrelated source images
Irregular tiled coverage	One logical coverage with gaps allowed, variable tile sizes, and normally no overlap	Incremental or source-driven mosaics
Regular tiled coverage	One alignment and tile size with no overlap; gaps may remain at the boundary or inside the coverage	Indexed analysis over a large coverage
Rectangular regular coverage	Equal aligned tiles fill a rectangular extent without gaps or overlaps	Predictable block processing and overview generation
Separate tiled images	Each image is internally regular but stored in its own table or partition	Keeping source-image identity and lifecycle separate
Raster-object coverage	Each row is a rasterized feature; extents and sizes may differ and overlap	Object models where cells replace or supplement vector geometry

Use AddRasterConstraints to record common SRID, scale, alignment, band, extent, and regular-blocking properties, and use spatial indexes and application constraints for relationships that the catalog cannot express.

Queries against a tiled coverage normally have two stages:

1. Use the raster spatial index and ST_Intersects to select candidate tiles.
2. Read, clip, aggregate, or vectorize values inside those tiles with functions such as ST_Value, ST_Clip, ST_SummaryStatsAgg, and ST_DumpAsPolygons.

ST_DumpAsPolygons returns geomval rows that pair geometries with cell values. This allows ordinary geometry overlay and filtering to answer inverse coverage queries, but polygonizing many cells is usually more expensive than keeping the operation in raster space.

Choose boundary semantics deliberately. Clipping and aggregating raster pixels is usually the shorter and faster path. If a partial boundary cell must contribute in proportion to its covered area, intersect the geometry with the geomval rows and calculate an area-weighted result in an appropriate projected coordinate system. That exact partial-cell method costs more because it vectorizes the selected cells.

Tile size affects both sides of the indexed query: smaller tiles reduce the amount of raster data inspected for each candidate row, while increasing row and index overhead. Benchmark representative queries instead of treating one tile size as a universal default.

For a current end-to-end raster/vector pattern, load and tile data with 10.1.1☒, then see the building example in ST_SummaryStats. It selects intersecting tiles, clips them to each polygon, and combines per-tile statistics into one result per feature. For point observations such as animal locations, first buffer the points in an appropriate projected coordinate system, then apply the same tile selection, clipping, and aggregation pattern to an elevation or other continuous raster coverage.

10.3 ☒☒☒☒☒☒☒

PostGIS ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒☒☒, ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.

1. raster_columns ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.
2. raster_overviews ☒☒. ☒☒☒☒☒☒ -l ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.

10.3.1 ☒☒☒☒☒☒☒☒

raster_columns ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒, ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒, ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒ raster_columns ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.

☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒ -C ☒☒☒☒☒☒☒☒☒☒☒, ☒☒☒☒☒☒☒ raster_columns ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒ AddRasterConstraints ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.

- r_table_catalog ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.
- r_table_schema ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.
- r_table_name ☒☒☒☒☒☒☒☒☒☒.
- r_raster_column ☒☒☒☒☒☒ r_table_name ☒☒☒☒☒☒☒☒☒☒☒. PostGIS ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒, ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.
- srid ☒☒☒☒☒☒☒☒☒☒☒☒☒☒. 4.5☒☒☒☒☒☒☒☒☒☒☒☒☒☒.
- scale_x ☒☒☒☒☒☒☒☒☒☒☒☒☒☒ (☒☒) ☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒ scale_x ☒☒☒☒, scale_x ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒ ST_ScaleX ☒☒☒☒☒☒☒.
- scale_y ☒☒☒☒☒☒☒☒☒☒☒☒☒☒ (☒☒) ☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒ scale_y ☒☒☒☒, scale_y ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒ ST_ScaleY ☒☒☒☒☒☒☒.
- blocksize_x ☒☒☒☒☒☒☒☒☒ (☒☒☒☒☒☒☒☒☒) ☒☒☒. ☒☒☒☒☒☒ ST_Width ☒☒☒☒☒☒☒.

- `blocksize_y` ⊠⊠⊠⊠⊠⊠⊠⊠⊠ (⊠⊠⊠⊠⊠⊠⊠⊠⊠) ⊠⊠⊠. ⊠⊠⊠⊠⊠⊠ ST_Height ⊠⊠⊠⊠⊠⊠⊠.
- `same_alignment` ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠ ST_SameAlignment ⊠⊠⊠⊠⊠⊠⊠.
- `regular_blocking` ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- `num_bands` ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ST_NumBands
- `pixel_types` ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. pixel_types ⊠⊠⊠ ST_BandPixelType ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- `nodata_values` ⊠⊠⊠⊠⊠⊠ `nodata_value` ⊠⊠⊠⊠⊠⊠⊠⊠⊠ (double precision) ⊠ (數) ⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠ ST_BandNoDataValue ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- `out_db` ⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- `extent` ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (extent) ⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠ DropRasterConstraints ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ AddRasterConstraints ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- `spatial_index` ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

10.3.2 ⊠⊠⊠⊠⊠⊠

`raster_overviews` ⊠⊠. ⊠⊠⊠⊠⊠⊠⊠ `raster_columns` ⊠ `raster_overviews` ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠ `-l` ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠ AddOverviewConstraints ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠.

> **Note!** ⊠⊠
> `raster_overviews` ⊠⊠⊠⊠⊠⊠ `raster_columns` ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. `raster_columns` ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ `raster_overviews` ⊠ `raster_columns` ⊠⊠⊠⊠⊠ (join) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠:

1. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
2. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠ (rule-of-thumb) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

`raster_overviews` ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

- `o_table_catalog` ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- `o_table_schema` ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

- `o_table_name` ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- `o_raster_column` ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- `r_table_catalog` ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- `r_table_schema` ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- `r_table_name` ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- `r_raster_column` ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- `overview_factor` ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. raster2pgsql ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠ 1 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠ 2 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ 4 ⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ 125x125 ⊠⊠⊠⊠⊠ 5000x5000 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (5000*5000)/(125*125) = 1600 ⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠ o_2 (l=2) ⊠⊠⊠⊠⊠⊠ (1600/2^2) = 400 ⊠, o_3 (l=3) ⊠⊠⊠⊠⊠⊠ (1600/2^3) = 200 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠ (scrap) ⊠⊠ (⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠) ⊠⊠⊠⊠⊠⊠⊠⊠. raster2pgsql ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (⊠⊠⊠⊠⊠ 2^overview_factor ⊠⊠⊠⊠⊠⊠⊠) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

10.4 PostGIS ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠

The fact that PostGIS raster provides you with SQL functions to render rasters in known image formats gives you a lot of options for rendering them. For example you can use OpenOffice / LibreOffice for rendering as demonstrated in Rendering PostGIS Raster graphics with LibreOffice Base Reports. In addition you can use a wide variety of languages as demonstrated in this section.

10.4.1 Rendering Map Algebra and Reclass Results

ST_MapAlgebra (callback function version) and ST_Reclass can be used together to turn analysis rasters into viewable images. A common pattern is to build a derived raster with map algebra, convert numeric ranges into display classes with reclassification, and render the result with ST_AsPNG or another ST_AsGDALRaster output format.

The raster reference includes executable visual examples for these workflows:

- ST_MapAlgebraExpr compares two-raster intersection and union results.
- ST_MapAlgebra (callback function version) demonstrates neighborhood processing and renders the source and result rasters.

For a classification-style output, reclassify a continuous band into a small display palette and render it directly.

Code

```
WITH classified AS (
    SELECT ST_Reclass(
        rast,
        1,
        '[0-50]:1, (50-100]:80, (100-150]:160, (150-255]:255',
        '8BUI',
        0
    ) AS rast
    FROM my_analysis_rasters
```

```
    WHERE rid = 1
)
SELECT ST_AsPNG(rast) AS png
FROM classified;
```

10.4.2 PL/pgSQL Map Algebra Callback Example

Map algebra callbacks are ordinary PostgreSQL functions. The following example uses PL/pgSQL to scale one raster band while keeping the callback argument types compatible with the double precision[][][] value matrix passed by ST_MapAlgebra (callback function version).

Code

```
CREATE OR REPLACE FUNCTION plpgsql_scale_pixel(
    value double precision[][][],
    positions integer[][],
    VARIADIC userargs text[]
)
RETURNS double precision
AS $$
DECLARE
    pixel double precision := value[1][1][1];
    factor double precision := COALESCE(userargs[1]::double precision, 1);
BEGIN
    IF pixel IS NULL THEN
        RETURN NULL;
    END IF;

    RETURN greatest(0, least(255, pixel * factor));
END;
$$ LANGUAGE plpgsql IMMUTABLE;

SELECT ST_MapAlgebra(
    rast,
    1,
    'plpgsql_scale_pixel(double precision[], integer[], text[])'::regprocedure,
    '8BUI',
    'FIRST',
    NULL,
    0,
    0,
    '1.5'
) AS rast
FROM my_analysis_rasters
WHERE rid = 1;
```

The first callback argument is a PostgreSQL double precision[][][] value matrix. With no neighborhood distance, the current pixel value for the first raster is available as `value[1][1][1]`.

10.4.3 □□□□□□□□□□ ST_AsPNG □□□□□ PHP □□□□□□□

□□□□□, PHP PostgreSQL □□□□□ ST_AsGDALRaster □□□□□□□□□□□□□ 1, 2, 3 □□□ PHP □□□□□ (request stream) □□□□□□□□□□□□□□□□□. □□ PHP □□□□□□”img src” HTML □□□□□□□□□□□□.

□□□□□□□□□□□□□□□□□□ (combine) □□□□ WGS84 □□□□□□□□□□□□□□□□□□, ST_Union □□□□□□□□□□□□□□□□□□□□□□□□□ (union) □□, ST_Transform □□□□□□□ □□□□□□□□□□□□□□□, ST_AsPNG □□□□□□□ PNG □□□□□□□□□□□□□□□□.

You would call the script with `http://mywebserver/test_raster.php?srid=2249` to get the raster image in Massachusetts state plane feet.

Code

```
<?php
$conn_str = 'dbname=mydb host=localhost port=5432 user=myuser password=mypwd';
$dbconn = pg_connect($conn_str);
header('Content-Type: image/png');

if (!empty($_REQUEST['srid']) &amp;&amp; is_numeric($_REQUEST['srid'])) {
    $input_srid = intval($_REQUEST['srid']);
} else {
    $input_srid = 26986;
}

$sql = "SET bytea_output = 'escape';
SELECT ST_AsPNG(
    ST_Transform(
        ST_AddBand(
            ST_Union(rast, 1),
            ARRAY[ST_Union(rast, 2), ST_Union(rast, 3)]
        ),
        $input_srid
    )
) AS new_rast
FROM aerials.boston
WHERE ST_Intersects(
    rast,
    ST_Transform(
        ST_MakeEnvelope(-71.1217, 42.227, -71.1210, 42.218, 4326),
        26986
    )
)";

$result = pg_query($sql);
$row = pg_fetch_row($result);
pg_free_result($result);
if ($row === false) return;
echo pg_unescape_bytea($row[0]);
?>
```

10.4.4 □□□□□□□□□□ ST_AsPNG □□□□□ ASP.NET C# □□□□□□□

In this section, we'll demonstrate how to use Npgsql PostgreSQL .NET driver and the ST_AsGDALRaster family of functions to output band 1,2,3 of a raster to the HTTP response stream so it can be embedded in an `img` tag.

□□□□□□□□□ npgsql PostgreSQL .NET □□□□□□□□□□. http://npgsql.projects.postgresql.org/- □□□□□□□□□□□□□□□□□□. □□□□□□□□□□□□□□□□ ASP.NET bin □□□□□□□□□□ □□□□.

□□□□□□□□□□□□□□□□□□ (combine) □□□□ WGS84 □□□□□□□□□□□□□□□□□□, ST_Union □□□□□□□□□□□□□□□□□□□□□□□□□□ (union) □□, ST_Transform □□□□□□□ □□□□□□□□□□□□□□□, ST_AsPNG □□□□□□□ PNG □□□□□□□□□□□□□□□□.

□□□□ C# □□□□□□□□□□□□□□ 10.4.3□□□□□□□□□.

You would call the handler with `http://mywebserver/TestRaster.ashx?srid=2249` to get the raster image in Massachusetts state plane feet.

web.config connection string section:

Code

```
<connectionStrings>
    <add name="DSN"
        connectionString="server=localhost;database=mydb;Port=5432;User Id=myuser;password= ↩
    mypwd"/>
</connectionStrings>
```

Code for `TestRaster.ashx`:

Code

```
<%@ WebHandler Language="C#" Class="TestRaster" %>
using System;
using System.Data;
using System.Web;
using Npgsql;

public class TestRaster : IHttpHandler
{
    public void ProcessRequest(HttpContext context)
    {
        context.Response.ContentType = "image/png";
        context.Response.BinaryWrite(GetResults(context));
    }

    public bool IsReusable {
        get { return false; }
    }

    public byte[] GetResults(HttpContext context)
    {
        byte[] result = null;
        NpgsqlCommand command;
        string sql = null;
        int input_srid = 26986;
        try {
            string connectionString = System.Configuration.ConfigurationManager
                .ConnectionStrings["DSN"].ConnectionString;
            using (NpgsqlConnection conn = new NpgsqlConnection(connectionString)) {
                conn.Open();

                if (context.Request["srid"] != null)
                {
                    input_srid = Convert.ToInt32(context.Request["srid"]);
                }
                sql = @"SELECT ST_AsPNG(
                            ST_Transform(
                                ST_AddBand(
                                    ST_Union(rast, 1),
                                    ARRAY[ST_Union(rast, 2), ST_Union(rast, 3)]
                                ),
                                :input_srid
                            )
                        ) As new_rast
                        FROM aerials.boston
                        WHERE ST_Intersects(
                            rast,
                            ST_Transform(
                                ST_MakeEnvelope(
                                    -71.1217, 42.227,
                                    -71.1210, 42.218,
                                    4326
```

```
                                ),
                                26986
                            )
                        )";
                command = new NpgsqlCommand(sql, conn);
                command.Parameters.Add(new NpgsqlParameter("input_srid", input_srid));


                result = (byte[]) command.ExecuteScalar();
                conn.Close();
            }

        }
        catch (Exception ex)
        {
            result = null;
            context.Response.Write(ex.Message.Trim());
        }
        return result;
    }
}
```

10.4.5 ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒ Java ☒☒☒☒☒☒☒☒

☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒

```
public class SaveQueryImage {
  public static void main(String[] argv) {
      System.out.println("Checking if Driver is registered with DriverManager.");

      try {
        //java.sql.DriverManager.registerDriver (new org.postgresql.Driver());
        Class.forName("org.postgresql.Driver");
      }
      catch (ClassNotFoundException cnfe) {
        System.out.println("Couldn't find the driver!");
        cnfe.printStackTrace();
        System.exit(1);
      }

      Connection conn = null;

      try {
        conn = DriverManager.getConnection("jdbc:postgresql://localhost:5432/mydb", "myuser ↩
    ", "mypwd");
        conn.setAutoCommit(false);

        PreparedStatement sGetImg = conn.prepareStatement(argv[0]);

        ResultSet rs = sGetImg.executeQuery();

        FileOutputStream fout;
        try
        {
            rs.next();
            /** Output to file name requested by user **/
            fout = new FileOutputStream(new File(argv[1]) );
            fout.write(rs.getBytes(1));
            fout.close();
        }
        catch(Exception e)
        {
            System.out.println("Can't create file");
            e.printStackTrace();
        }

        rs.close();
        sGetImg.close();
        conn.close();
      }
      catch (SQLException se) {
        System.out.println("Couldn't connect: print out a stack trace and exit.");
        se.printStackTrace();
        System.exit(1);
      }
  }
}
```

10.4.6 PLPython □□□□□ SQL □□□□□□□□□□□

□□□□□□□□□□□□□□□□□□□□ PLPython □□□□□□□. PLPython □□□□□□□. PLPythonu □ PLPythonu3u □□□□□□□□□□.

Code

```
CREATE OR REPLACE FUNCTION write_file (param_bytes bytea, param_filepath text)
RETURNS text
AS $$
f = open(param_filepath, 'wb+')
f.write(param_bytes)
return param_filepath
$$ LANGUAGE plpythonu;
```

Write out 5 images to the PostgreSQL server in varying sizes. The PostgreSQL daemon account needs write access to the folder; this echoes back the file names created.

Code

```
SELECT write_file(
    ST_AsPNG(
        ST_AsRaster(
            ST_Buffer(ST_Point(1, 5), j * 5, 'quad_segs=2'),
            150 * j,
            150 * j,
            '8BUI',
            100
        )
    ),
    'C:/temp/slices' || j || '.png'
)
FROM generate_series(1, 5) AS j;
```

Output

```
write_file
--------------------
 C:/temp/slices1.png
 C:/temp/slices2.png
 C:/temp/slices3.png
 C:/temp/slices4.png
 C:/temp/slices5.png
```

10.4.7 PSQL □□□□□□□□□□□

Sadly PSQL doesn't have easy to use built-in functionality for outputting binaries. This is a bit of a hack that piggy backs on PostgreSQL somewhat legacy large object support. To use it, first launch your **psql** command line connected to your database.

□□□□□□□□□□, □□□□□□□□□□□□□□□□□□□□□□□□□□□□.

Code

```
SELECT
    oid,
    lowrite(lo_open(oid, 131072), png) AS num_bytes
FROM (VALUES (
    lo_create(0),
    ST_AsPNG((SELECT rast FROM aerials.boston WHERE rid = 1))
)) AS v(oid, png);
```

Output

```
oid   | num_bytes
---------+-----------
 2630819 |     74860
```

Note the large object OID and export it to a file on your local computer, replacing the example path with the desired file path.

Code

```
\lo_export 2630819 'C:/temp/aerial_samp.png'
```

This deletes the file from large object storage in the database.

Code

```
SELECT lo_unlink(2630819);
```

Chapter 11

□□□□□□□

raster is a PostGIS type for storing and analyzing raster data.
□□□□□□□□□□□□□□□□□□ 10.1□□□□□□□□.
Some examples in this reference use a raster table of dummy data, created with the following code:

Code

```
CREATE TABLE dummy_rast(rid integer, rast raster);
INSERT INTO dummy_rast(rid, rast)
VALUES (1,
('01' -- little endian (uint8 ndr)
||
'0000' -- version (uint16 0)
||
'0000' -- nBands (uint16 0)
||
'0000000000000040' -- scaleX (float64 2)
||
'0000000000000840' -- scaleY (float64 3)
||
'000000000000E03F' -- ipX (float64 0.5)
||
'000000000000E03F' -- ipY (float64 0.5)
||
'0000000000000000' -- skewX (float64 0)
||
'0000000000000000' -- skewY (float64 0)
||
'00000000' -- SRID (int32 0)
||
'0A00' -- width (uint16 10)
||
'1400' -- height (uint16 20)
)::raster
),
-- Raster: 5 x 5 pixels, 3 bands, PT_8BUI pixel type, NODATA = 0
(2, ('01000003009A9999999999A93F9A9999999999A9BF000000E02B274A' ||
'410000000077195641000000000000000000000000000000000 ↩
    FFFFFFFF050005000400FDFEFDFEFEFDFEFEFDF9FAFEF' ||
' ↩
    EFCF9FBFDFEFEFDFCFAFEFEFE04004E627AADD16076B4F9FE6370A9F5FE59637AB0E54F5861708704004656648 7A1506
    ')::raster);
```

The functions below are ones which a user of PostGIS Raster is likely to need. There are other raster support functions which are not of interest to a general user.

11.1 ⊠⊠⊠⊠⊠⊠⊠⊠⊠

11.1.1 geomval

geomval — (⊠⊠⊠⊠⊠⊠⊠⊠⊠) geom ⊠ (⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠) val, ⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠.

Description

geomval ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, .geom ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ val ⊠⊠⊠⊠⊠⊠. ST_DumpAsPolygon ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

13.6⊠

11.1.2 addbandarg

addbandarg — ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ST_AddBand ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠.

Description

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ST_AddBand ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

Field	**Type**	**Description**
`index`	integer	⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ 1-⊠⊠⊠ ⊠⊠⊠. NULL ⊠⊠⊠, ⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠.
`pixeltype`	text	Pixel type of the new band. One of the pixel types described in ST_BandPixelType.
`initialvalue`	double precision	Initial value that all pixels of the new band will be set to.
`nodataval`	double precision	⊠⊠⊠⊠ NODATA ⊠⊠⊠⊠. NULL ⊠⊠⊠, ⊠⊠⊠⊠ NODATA ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠.

⊠⊠

ST_AddBand

11.1.3 rastbandarg

rastbandarg — ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

Description

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

Field	Type	Description
rast	raster	The raster in question.
nband	integer	1-based value indicating the raster band.

⊠⊠

ST_MapAlgebra (callback function version)

11.1.4 raster

raster — ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

Description

raster is a spatial data type used to represent raster data such as those imported from JPEGs, TIFFs, PNGs, digital elevation models. Each raster has 1 or more bands each having a set of pixel values. Rasters can be georeferenced.

> ⊠⊠
> GDAL ⊠⊠⊠⊠⊠⊠⊠⊠⊠ PostGIS ⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ST_ConvexHull ⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠⊠⊠⊠

⊠⊠.

⊠⊠⊠⊠⊠	⊠⊠
⊠⊠	⊠⊠⊠

⊠⊠

11⊠

11.1.5 reclassarg

reclassarg — A composite type used as input into the ST_Reclass function defining the behavior of reclassification.

Description

A composite type used as input into the ST_Reclass function defining the behavior of reclassification.

Field	Type	Description
nband	integer	□□□□□□□□□□□□□□□.
reclassexpr	text	Comma-delimited range:map_range mappings that define how old band values map to new band values.
pixeltype	text	One of defined pixel types as described in ST_BandPixelType.
nodataval	double precision	NODATA □□□□□□□□□. □□□□□□□□□□□□□□□□ □□, □□□□□□□□□□□.

The reclassexpr range delimiters mean: (greater than,) less than,] less than or equal, and [greater than or equal.

- [a-b] means a <= x <= b.
- (a-b] means a < x <= b.
- [a-b) means a <= x < b.
- (a-b) means a < x < b.

Opening-parenthesis notation is optional, so a-b means the same as (a-b).

□□

Reclassify band 2 as an 8BUI where 255 is nodata value.

Code

```
SELECT ROW(2, '0-100:1-10, 101-500:11-150,501 - 10000: 151-254', '8BUI', 255)::reclassarg;
```

Reclassify band 1 as an 1BB and no nodata value defined.

Code

```
SELECT ROW(1, '0-100]:0, (100-255:1', '1BB', NULL)::reclassarg;
```

□□

ST_Reclass

11.1.6 summarystats

summarystats — ST_SummaryStats □ ST_SummaryStatsAgg □□□□□□□□□□□□□□□□.

Description

ST_SummaryStats ⊠ ST_SummaryStatsAgg ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

count integer ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

sum double precision ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

mean double precision ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

stddev double precision ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

min double precision ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

max double precision ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

ST_SummaryStats, ST_SummaryStatsAgg

11.1.7 unionarg

unionarg — ⊠⊠⊠⊠⊠⊠⊠⊠⊠ UNION ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ST_Union ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

Description

⊠⊠⊠⊠⊠⊠⊠⊠⊠ UNION ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ST_Union ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

Field	Type	Description
nband	integer	⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠ 1-⊠⊠⊠⊠⊠⊠.
uniontype	text	Type of union operation. One of the types described in ST_Union.

⊠⊠

ST_Union

11.2 ⊠⊠⊠⊠⊠

11.2.1 AddRasterConstraints

AddRasterConstraints — Adds raster constraints to a loaded raster table for a specific column that constrains spatial ref, scaling, blocksize, alignment, bands, band type and a flag to denote if raster column is regularly blocked. The table must be loaded with data for the constraints to be inferred. Returns true if the constraint setting was accomplished and issues a notice otherwise.

☒☒

boolean **AddRasterConstraints**(name rasttable, name rastcolumn, boolean srid=true, boolean scale_x=true, boolean scale_y=true, boolean blocksize_x=true, boolean blocksize_y=true, boolean same_alignment=true, boolean regular_blocking=false, boolean num_bands=true , boolean pixel_types=true , boolean nodata_values=true , boolean out_db=true , boolean extent=true);
boolean **AddRasterConstraints**(name rasttable, name rastcolumn, text[] VARIADIC constraints);
boolean **AddRasterConstraints**(name rastschema, name rasttable, name rastcolumn, text[] VARIADIC constraints);
boolean **AddRasterConstraints**(name rastschema, name rasttable, name rastcolumn, boolean srid=true, boolean scale_x=true, boolean scale_y=true, boolean blocksize_x=true, boolean blocksize_y=true, boolean same_alignment=true, boolean regular_blocking=false, boolean num_bands=true, boolean pixel_types=true, boolean nodata_values=true , boolean out_db=true , boolean extent=true);

Description

`raster_columns` ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒, ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒. `rastschema` ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒. `srid` ☒ SPATIAL_REF_SYS ☒☒☒☒☒ ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.

`raster2pgsql` ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.

☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒ 10.3.1☒☒☒☒☒☒☒☒.

- `blocksize` ☒ X ☒ Y ☒☒☒☒☒☒☒☒☒☒☒☒.
- `blocksize_x` ☒ X ☒☒ (☒☒☒☒☒☒☒☒☒☒) ☒☒☒☒☒☒.
- `blocksize_y` ☒ Y ☒☒ (☒☒☒☒☒☒☒☒☒☒) ☒☒☒☒☒☒.
- `extent` ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.
- `num_bands` ☒☒☒☒☒☒☒☒☒.
- `pixel_types` ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒ N ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.
- `regular_blocking` ☒☒☒☒☒☒☒☒☒☒☒ (☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒) ☒☒☒☒☒☒☒ ☒☒☒☒ (☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒) ☒☒☒☒☒☒.
- `same_alignment` ensures they all have same alignment meaning any two tiles you compare will return true for. Refer to ST_SameAlignment.
- `srid` ☒☒☒☒☒☒☒☒☒ SRID ☒☒☒☒☒☒☒☒☒☒.
- ☒☒ -- ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.

> ☒☒
> ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒☒, ☒☒☒☒☒ ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.

> ☒☒
> ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒, ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒ DropRasterConstraints ☒☒☒☒☒☒☒☒☒☒☒☒☒.

2.0.0 ☒☒☒☒☒☒☒☒☒☒☒☒.

⊠⊠

Apply all possible constraints on column based on data.

Verify the raster table is registered correctly in the raster_columns view.

Code

```
CREATE TABLE myrasters(rid SERIAL primary key, rast raster);
INSERT INTO myrasters(rast)
SELECT ST_AddBand(ST_MakeEmptyRaster(1000, 1000, 0.3, -0.3, 2, 2, 0, 0, 4326), 1, '8BSI':: ↩
    text, -129, NULL);

SELECT AddRasterConstraints('myrasters'::name, 'rast'::name);
SELECT srid, scale_x, scale_y, blocksize_x, blocksize_y, num_bands, pixel_types,  ↩
    nodata_values
     FROM raster_columns
     WHERE r_table_name = 'myrasters';
```

Output

```
srid | scale_x | scale_y | blocksize_x | blocksize_y | num_bands | pixel_types| nodata_values
------+---------+---------+-------------+-------------+-----------+-------------+--------------
 4326 |       2 |       2 |        1000 |        1000 |         1 | {8BSI}      | {0}
```

Apply single constraint.

Code

```
CREATE TABLE public.myrasters2(rid SERIAL primary key, rast raster);
INSERT INTO myrasters2(rast)
SELECT ST_AddBand(ST_MakeEmptyRaster(1000, 1000, 0.3, -0.3, 2, 2, 0, 0, 4326), 1, '8BSI':: ↩
    text, -129, NULL);

SELECT AddRasterConstraints('public'::name, 'myrasters2'::name, 'rast'::name, ' ↩
    regular_blocking', 'blocksize');
```

Output

```
NOTICE:  Adding regular blocking constraint
NOTICE:  Adding blocksize-X constraint
NOTICE:  Adding blocksize-Y constraint
```

⊠⊠

10.3.1⊠, ST_AddBand, ST_MakeEmptyRaster, DropRasterConstraints, ST_BandPixelType, ST_SRID

11.2.2 DropRasterConstraints

DropRasterConstraints — ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ PostGIS ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠.

⊠⊠

boolean **DropRasterConstraints**(name rasttable, name rastcolumn, boolean srid, boolean scale_x, boolean scale_y, boolean blocksize_x, boolean blocksize_y, boolean same_alignment, boolean regular_blocking, boolean num_bands=true, boolean pixel_types=true, boolean nodata_values=true, boolean out_db=true , boolean extent=true);

boolean **DropRasterConstraints**(name rastschema, name rasttable, name rastcolumn, boolean srid=true, boolean scale_x=true, boolean scale_y=true, boolean blocksize_x=true, boolean blocksize_y=true, boolean same_alignment=true, boolean regular_blocking=false, boolean num_bands=true, boolean pixel_types=true, boolean nodata_values=true, boolean out_db=true , boolean extent=true);
boolean **DropRasterConstraints**(name rastschema, name rasttable, name rastcolumn, text[] constraints);

Description

AddRasterConstraints ⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ PostGIS ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

To drop a raster table use standard SQL: `DROP TABLE mytable`.

To drop just a raster column and leave the rest of the table, use standard SQL: `ALTER TABLE mytable DROP COLUMN rast`.

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ raster_columns ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠ raster_columns ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

Code

```
SELECT DropRasterConstraints ('myrasters', 'rast');
```

Output

```
droprasterconstraints
----------------------
 t
(1 row)
```

Verify change in raster_columns.

Code

```
SELECT srid, scale_x, scale_y, blocksize_x, blocksize_y, num_bands, pixel_types, ↩
    nodata_values
    FROM raster_columns
    WHERE r_table_name = 'myrasters';
```

Output

```
srid | scale_x | scale_y | blocksize_x | blocksize_y | num_bands | pixel_types| nodata_values
------+---------+---------+-------------+-------------+-----------+-------------+--------------
    0 |         |         |             |             |           |             |
```

⊠⊠

AddRasterConstraints

11.2.3 AddOverviewConstraints

AddOverviewConstraints — ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (overview) ⊠⊠⊠⊠⊠⊠.

⊠⊠

boolean **AddOverviewConstraints**(name ovschema, name ovtable, name ovcolumn, name refschema, name reftable, name refcolumn, int ovfactor);
boolean **AddOverviewConstraints**(name ovtable, name ovcolumn, name reftable, name refcolumn, int ovfactor);

Description

`raster_overviews` ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

`ovfactor` ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (乘數) ⊠⊠⊠⊠⊠⊠. `ovfactor` ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

`ovschema` ⊠ `refschema` ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, `search_path` ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

Verify the raster table is registered correctly in the raster_overviews view.

Code

```
CREATE TABLE res1 AS SELECT
ST_AddBand(ST_MakeEmptyRaster(1000, 1000, 0, 0, 2),
  1, '8BSI'::text, -129, NULL
) r1;

CREATE TABLE res2 AS SELECT
ST_AddBand(ST_MakeEmptyRaster(500, 500, 0, 0, 4),
  1, '8BSI'::text, -129, NULL
) r2;

SELECT AddOverviewConstraints('res2', 'r2', 'res1', 'r1', 2);
SELECT o_table_name ot, o_raster_column oc,
       r_table_name rt, r_raster_column rc,
       overview_factor f
FROM raster_overviews WHERE o_table_name = 'res2';
```

Output

```
ot   | oc |  rt   | rc | f
------+----+------+----+---
 res2 | r2 | res1 | r1 | 2
(1 row)
```

⊠⊠

10.3.2⊠, DropOverviewConstraints, ST_CreateOverview, AddRasterConstraints

11.2.4 DropOverviewConstraints

DropOverviewConstraints — ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (overview) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

boolean **DropOverviewConstraints**(name ovschema, name ovtable, name ovcolumn);
boolean **DropOverviewConstraints**(name ovtable, name ovcolumn);

Description

`raster_overviews` ⊠⊠.

`ovschema` ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, `search_path` ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

10.3.2⊠, AddOverviewConstraints, DropRasterConstraints

11.2.5 PostGIS_GDAL_Version

PostGIS_GDAL_Version — PostGIS ⊠⊠⊠⊠⊠⊠⊠ GDAL ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

text **PostGIS_GDAL_Version**();

Description

PostGIS ⊠⊠⊠⊠⊠⊠⊠ GDAL ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠ GDAL ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

Code

```
SELECT PostGIS_GDAL_Version();
```

Output

```
GDAL 1.11dev, released 2013/04/13
```

⊠⊠

postgis.gdal_datapath

11.2.6 PostGIS_Raster_Lib_Build_Date

PostGIS_Raster_Lib_Build_Date — ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

text **PostGIS_Raster_Lib_Build_Date**();

Description

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

Code

```
SELECT PostGIS_Raster_Lib_Build_Date();
```

Output

```
2010-04-28 21:15:10
```

⊠⊠

PostGIS_Raster_Lib_Version

11.2.7 PostGIS_Raster_Lib_Version

PostGIS_Raster_Lib_Version — ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

text **PostGIS_Raster_Lib_Version**();

Description

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

Code

```
SELECT PostGIS_Raster_Lib_Version();
```

Output

```
2.0.0
```

⊠⊠

PostGIS_Lib_Version

11.2.8 ST_GDALDrivers

ST_GDALDrivers — Returns a list of raster formats supported by PostGIS through GDAL. Only those formats with can_write=True can be used by ST_AsGDALRaster

□□

setof record **ST_GDALDrivers**(integer OUT idx, text OUT short_name, text OUT long_name, text OUT can_read, text OUT can_write, text OUT create_options);

Description

Returns a list of raster formats short_name,long_name and creator options of each format supported by GDAL. Use the short_name as input in the `format` parameter of ST_AsGDALRaster. Options vary depending on what drivers your libgdal was compiled with. `create_options` returns an xml formatted set of CreationOptionList/Option consisting of name and optional `type`, `description` and set of `VALUE` for each creator option for the specific driver.

Changed: 2.5.0 - add can_read and can_write columns.

□□□□: 2.0.6, 2.1.3 □□ - GUC □□□□□□□ gdal_enabled_drivers □□□□□□□□□□, □□□□□□□□ □□□□□□□□□□□□□□□□□□□□.

2.0.0 □□□□□□□□□□□□□. GDAL 1.6.0 □□□□□□□□□□□.

□□

List of Drivers.

Code

```
SET postgis.gdal_enabled_drivers = 'ENABLE_ALL';
SELECT short_name, long_name, can_write
FROM ST_GDALDrivers()
ORDER BY short_name;
```

Output

```
short_name    |                          long_name                          | can_write
----------------+-------------------------------------------------------------+-----------
 AAIGrid        | Arc/Info ASCII Grid                                         | t
 ACE2           | ACE2                                                        | f
 ADRG           | ARC Digitized Raster Graphics                               | f
 AIG            | Arc/Info Binary Grid                                        | f
 AirSAR         | AirSAR Polarimetric Image                                   | f
 ARG            | Azavea Raster Grid format                                   | t
 BAG            | Bathymetry Attributed Grid                                  | f
 BIGGIF         | Graphics Interchange Format (.gif)                          | f
 BLX            | Magellan topo (.blx)                                        | t
 BMP            | MS Windows Device Independent Bitmap                        | f
 BSB            | Maptech BSB Nautical Charts                                 | f
 PAux           | PCI .aux Labelled                                           | f
 PCIDSK         | PCIDSK Database File                                        | f
 PCRaster       | PCRaster Raster File                                        | f
 PDF            | Geospatial PDF                                              | f
 PDS            | NASA Planetary Data System                                  | f
 PDS4           | NASA Planetary Data System 4                                | t
 PLMOSAIC       | Planet Labs Mosaics API                                     | f
 PLSCENES       | Planet Labs Scenes API                                      | f
 PNG            | Portable Network Graphics                                   | t
 PNM            | Portable Pixmap Format (netpbm)                             | f
 PRF            | Racurs PHOTOMOD PRF                                         | f
 R              | R Object Data Store                                         | t
 Rasterlite     | Rasterlite                                                  | t
 RDA            | DigitalGlobe Raster Data Access driver                      | f
```

```
 RIK             | Swedish Grid RIK (.rik)                                | f
 RMF             | Raster Matrix Format                                   | f
 ROI_PAC         | ROI_PAC raster                                         | f
 RPFTOC          | Raster Product Format TOC format                       | f
 RRASTER         | R Raster                                               | f
 RS2             | RadarSat 2 XML Product                                 | f
 RST             | Idrisi Raster A.1                                      | t
 SAFE            | Sentinel-1 SAR SAFE Product                            | f
 SAGA            | SAGA GIS Binary Grid (.sdat, .sg-grd-z)                | t
 SAR_CEOS        | CEOS SAR Image                                         | f
 SDTS            | SDTS Raster                                            | f
 SENTINEL2       | Sentinel 2                                             | f
 SGI             | SGI Image File Format 1.0                              | f
 SNODAS          | Snow Data Assimilation System                          | f
 SRP             | Standard Raster Product (ASRP/USRP)                    | f
 SRTMHGT         | SRTMHGT File Format                                    | t
 Terragen        | Terragen heightfield                                   | f
 TIL             | EarthWatch .TIL                                        | f
 TSX             | TerraSAR-X Product                                     | f
 USGSDEM         | USGS Optional ASCII DEM (and CDED)                     | t
 VICAR           | MIPL VICAR file                                        | f
 VRT             | Virtual Raster                                         | t
 WCS             | OGC Web Coverage Service                               | f
 WMS             | OGC Web Map Service                                    | t
 WMTS            | OGC Web Map Tile Service                               | t
 XPM             | X11 PixMap Format                                      | t
 XYZ             | ASCII Gridded XYZ                                      | t
 ZMap            | ZMap Plus Grid                                         | t
```

List of options for each driver.

Output the create options XML column of JPEG as a table. These creator options can be used in the ST_AsGDALRaster options argument.

Code

```
SELECT (xpath('@name', g.opt))[1]::text As oname,
       (xpath('@type', g.opt))[1]::text As otype,
       (xpath('@description', g.opt))[1]::text As descrip
FROM (SELECT unnest(xpath('/CreationOptionList/Option', create_options::xml)) As opt
FROM  ST_GDALDrivers()
WHERE short_name = 'JPEG') As g;
```

Output

```
oname        |  otype  |      descrip
-------------------+---------+-------------------------------------------------
 PROGRESSIVE       | boolean | whether to generate a progressive JPEG
 QUALITY           | int     | good=100, bad=0, default=75
 WORLDFILE         | boolean | whether to generate a worldfile
 INTERNAL_MASK     | boolean | whether to generate a validity mask
 COMMENT           | string  | Comment
 SOURCE_ICC_PROFILE | string  | ICC profile encoded in Base64
 EXIF_THUMBNAIL    | boolean | whether to generate an EXIF thumbnail(overview).
                                By default its max dimension will be 128
 THUMBNAIL_WIDTH   | int     | Forced thumbnail width
 THUMBNAIL_HEIGHT  | int     | Forced thumbnail height
(9 rows)
```

This example turns a few common GeoTIFF creator options into a compact table.

Code

```
WITH gtiff_options AS (
    SELECT
        option_name,
        option_type,
        default_value
    FROM ST_GDALDrivers() AS driver
    CROSS JOIN LATERAL XMLTABLE(
        '/CreationOptionList/Option'
        PASSING (driver.create_options::xml)
        COLUMNS
```

```
            option_name text PATH '@name',
            option_type text PATH '@type',
            default_value text PATH '@default'
    ) AS options
    WHERE driver.short_name = 'GTiff'
)
SELECT option_name AS name, option_type AS type, coalesce(default_value, '') AS ↩
    default_value
FROM gtiff_options
WHERE option_name = ANY (ARRAY[
    'COMPRESS',
    'INTERLEAVE',
    'TILED',
    'BLOCKXSIZE',
    'BLOCKYSIZE',
    'JPEG_QUALITY'
])
ORDER BY 1;
```

Output

```
name      |     type      | default_value
--------------+---------------+--------------
 BLOCKXSIZE    | int           |
 BLOCKYSIZE    | int           |
 COMPRESS      | string-select |
 INTERLEAVE    | string-select | PIXEL
 JPEG_QUALITY  | int           | 75
 TILED         | boolean       |
(6 rows)
```

Use XMLTABLE to expose selected GTiff creation options as rows. The same pattern can be used without the name filter to inspect the full XML document.

Code

```
SET postgis.gdal_enabled_drivers = 'ENABLE_ALL';

SELECT option_name, option_type
FROM ST_GDALDrivers() AS driver
CROSS JOIN LATERAL XMLTABLE(
    '/CreationOptionList/Option'
    PASSING (driver.create_options::xml)
    COLUMNS
        option_name text PATH '@name',
        option_type text PATH '@type'
) AS options
WHERE driver.short_name = 'GTiff'
  AND option_name IN ('COMPRESS', 'TILED', 'BIGTIFF')
ORDER BY option_name;
```

Output

```
option_name |  option_type
------------+--------------
 BIGTIFF     | string-select
 COMPRESS    | string-select
 TILED       | boolean
(3 rows)
```

☒☒

ST_AsGDALRaster, ST_SRID, postgis.gdal_enabled_drivers

11.2.9 UpdateRasterSRID

UpdateRasterSRID — SRID .

raster **UpdateRasterSRID**(name schema_name, name table_name, name column_name, integer new_srid);
raster **UpdateRasterSRID**(name table_name, name column_name, integer new_srid);

Description

 SRID . SRID
 (, SRID) .

> **Note**
>
> () . .

2.1.0 .

UpdateGeometrySRID

11.2.10 ST_CreateOverview

ST_CreateOverview — .

regclass **ST_CreateOverview**(regclass tab, name col, int factor, text algo='NearestNeighbor');

Description

. ,
 (`1/factor`).

`raster_overviews` , .

'NearestNeighbor', 'Bilinear', 'Cubic', 'CubicSpline', 'Lanczos' .
 GDAL Warp resampling methods .

2.2.0 .

Output to generally better quality but slower to product format

Code

```
SELECT ST_CreateOverview('mydata.mytable'::regclass, 'rast', 2, 'Lanczos');
```

Output to faster to process default nearest neighbor

Code

```
SELECT ST_CreateOverview('mydata.mytable'::regclass, 'rast', 2);
```

☒☒

ST_Retile, AddOverviewConstraints, AddRasterConstraints, 10.3.2☒

11.3 ☒☒☒☒☒☒ (constructor)

11.3.1 ST_AddBand

ST_AddBand — ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒ (☒) ☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒ ☒☒☒☒☒☒☒☒☒☒, ☒☒☒☒☒☒☒☒☒☒☒☒☒☒.

☒☒

(1) raster **ST_AddBand**(raster rast, addbandarg[] addbandargset);
(2) raster **ST_AddBand**(raster rast, integer index, text pixeltype, double precision initialvalue=0, double precision nodataval=NULL);
(3) raster **ST_AddBand**(raster rast, text pixeltype, double precision initialvalue=0, double precision nodataval=NULL);
(4) raster **ST_AddBand**(raster torast, raster fromrast, integer fromband=1, integer torastindex=at_end);
(5) raster **ST_AddBand**(raster torast, raster[] fromrasts, integer fromband=1, integer torastindex=at_end);
(6) raster **ST_AddBand**(raster rast, integer index, text outdbfile, integer[] outdbindex, double precision nodataval=NULL);
(7) raster **ST_AddBand**(raster rast, text outdbfile, integer[] outdbindex, integer index=at_end, double precision nodataval=NULL);

Description

Returns a raster with a new band added in given position (index), of given type, of given initial value, and of given nodata value. If no index is specified, the band is added to the end. If no `fromband` is specified, band 1 is assumed. Pixel type is a string representation of one of the pixel types specified in ST_BandPixelType. If an existing index is specified all subsequent bands >= that index are incremented by 1. If an initial value greater than the max of the pixel type is specified, then the initial value is set to the highest value allowed by the pixel type.

addbandarg ☒☒☒☒☒☒☒☒☒☒ 1 ☒☒☒, ☒☒ addbandarg ☒☒☒☒☒☒ addbandarg ☒☒☒☒☒☒☒☒ ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.

☒☒☒☒☒☒☒☒☒☒☒☒ 5 ☒☒☒, `torast` ☒ NULL ☒☒☒☒☒☒☒☒☒☒☒☒☒☒ `fromband` ☒☒☒☒☒☒ ☒☒☒☒ (累計) ☒☒☒.

`outdbfile` ☒☒☒☒☒☒☒ 6 ☒ 7 ☒☒☒, `outdbfile` ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒. PostgreSQL ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.

☒☒☒☒: 2.1.0 ☒☒☒☒ addbandarg ☒☒☒☒☒☒.

☒☒☒☒: 2.1.0 ☒☒☒☒☒☒☒ DB ☒☒☒☒☒☒☒☒☒☒.

□□

Single New Band.

This example adds another 8-bit unsigned integer band with pixels initialized to 200.

Code

```
UPDATE dummy_rast
    SET rast = ST_AddBand(rast, '8BUI'::text, 200)
WHERE rid = 1;
```

Create an empty raster 100x100 units, with upper left right at 0, add 2 bands (band 1 is 0/1 boolean bit switch, band2 allows values 0-15) Uses addbandargs.

Output metadata for the raster bands to verify they are correct.

Code

```
INSERT INTO dummy_rast(rid, rast)
    VALUES(10, ST_AddBand(ST_MakeEmptyRaster(100, 100, 0, 0, 1, -1, 0, 0, 0),
    ARRAY[
        ROW(1, '1BB'::text, 0, NULL),
        ROW(2, '4BUI'::text, 0, NULL)
            ]::addbandarg[]
     )
    );

SELECT  (bmd).*
FROM (SELECT ST_BandMetaData(rast, generate_series(1, 2)) As bmd
    FROM dummy_rast WHERE rid = 10) AS foo;
```

Output

```
pixeltype | nodatavalue | isoutdb | path
-----------+-------------+---------+------
 1BB       |             | f       |
 4BUI      |             | f       |
```

This example outputs metadata for the raster.

Code

```
SELECT  (rmd).width, (rmd).height, (rmd).numbands
FROM (SELECT ST_MetaData(rast) As rmd
    FROM dummy_rast WHERE rid = 10) AS foo;
```

Output

```
width | height | numbands
-------+--------+----------
   100 |    100 |        2
```

Multiple New Bands.

Code

```
SELECT
    *
FROM ST_BandMetaData(ST_AddBand(ST_MakeEmptyRaster(10, 10, 0, 0, 1, -1, 0, 0, 0),
        ARRAY[
            ROW(NULL, '8BUI', 255, 0),
            ROW(NULL, '16BUI', 1, 2),
```

```
            ROW(2, '32BUI', 100, 12),
            ROW(2, '32BF', 3.14, -1)
        ]::addbandarg[]
    ),
    ARRAY[]::integer[]
);
```

Output

```
bandnum | pixeltype | nodatavalue | isoutdb | path | outdbbandnum | filesize | filetimestamp
--------+-----------+-------------+---------+------+--------------+----------+--------------
      1 | 8BUI      |           0 | f       | null |         null |     null |          null
      2 | 32BF      |          -1 | f       | null |         null |     null |          null
      3 | 32BUI     |          12 | f       | null |         null |     null |          null
      4 | 16BUI     |           2 | f       | null |         null |     null |          null
```

Aggregate the 1st band of a table of like rasters into a single raster. With as many bands as there are test_types and as many rows (new rasters) as there are mice. Note that The ORDER BY test_type clause relies on aggregate ORDER BY support. The resulting raster will have a band for each test_type alphabetical by test_type. For mouse lovers: No mice were harmed in this exercise.

Code

```
SELECT
    mouse,
    ST_AddBand(NULL, array_agg(rast ORDER BY test_type), 1) As rast
FROM mice_studies
GROUP BY mouse;
```

New Out-db band.

Code

```
SELECT
    *
FROM ST_BandMetaData(ST_AddBand(ST_MakeEmptyRaster(10, 10, 0, 0, 1, -1, 0, 0, 0),
        '/home/raster/mytestraster.tif'::text, NULL::int[]
    ),
    ARRAY[]::integer[]
);
```

Output

```
bandnum | pixeltype | nodatavalue | isoutdb | path
--------+-----------+-------------+---------+------
      1 | 8BUI      |             | t       | /home/raster/mytestraster.tif
      2 | 8BUI      |             | t       | /home/raster/mytestraster.tif
      3 | 8BUI      |             | t       | /home/raster/mytestraster.tif
```

☒☒

ST_BandMetaData, ST_BandPixelType, ST_MakeEmptyRaster, ST_MetaData, ST_NumBands, ST_Reclass

11.3.2 ST_AsRaster

ST_AsRaster — PostGIS ☒☒☒ PostGIS ☒☒☒☒☒☒☒☒☒.

⊠⊠

raster **ST_AsRaster**(geometry geom, raster ref, text pixeltype, double precision value=1, double precision nodataval=0, boolean touched=false);
raster **ST_AsRaster**(geometry geom, raster ref, text[] pixeltype=ARRAY['8BUI'], double precision[] value=ARRAY[1], double precision[] nodataval=ARRAY[0], boolean touched=false);
raster **ST_AsRaster**(geometry geom, double precision scalex, double precision scaley, double precision gridx, double precision gridy, text pixeltype, double precision value=1, double precision nodataval=0, double precision skewx=0, double precision skewy=0, boolean touched=false);
raster **ST_AsRaster**(geometry geom, double precision scalex, double precision scaley, double precision gridx=NULL, double precision gridy=NULL, text[] pixeltype=ARRAY['8BUI'], double precision[] value=ARRAY[1], double precision[] nodataval=ARRAY[0], double precision skewx=0, double precision skewy=0, boolean touched=false);
raster **ST_AsRaster**(geometry geom, double precision scalex, double precision scaley, text pixeltype, double precision value=1, double precision nodataval=0, double precision upperleftx=NULL, double precision upperlefty=NULL, double precision skewx=0, double precision skewy=0, boolean touched=false);
raster **ST_AsRaster**(geometry geom, double precision scalex, double precision scaley, text[] pixeltype, double precision[] value=ARRAY[1], double precision[] nodataval=ARRAY[0], double precision upperleftx=NULL, double precision upperlefty=NULL, double precision skewx=0, double precision skewy=0, boolean touched=false);
raster **ST_AsRaster**(geometry geom, integer width, integer height, double precision gridx, double precision gridy, text pixeltype, double precision value=1, double precision nodataval=0, double precision skewx=0, double precision skewy=0, boolean touched=false);
raster **ST_AsRaster**(geometry geom, integer width, integer height, double precision gridx=NULL, double precision gridy=NULL, text[] pixeltype=ARRAY['8BUI'], double precision[] value=ARRAY[1], double precision[] nodataval=ARRAY[0], double precision skewx=0, double precision skewy=0, boolean touched=false);
raster **ST_AsRaster**(geometry geom, integer width, integer height, text pixeltype, double precision value=1, double precision nodataval=0, double precision upperleftx=NULL, double precision upperlefty=NULL, double precision skewx=0, double precision skewy=0, boolean touched=false);
raster **ST_AsRaster**(geometry geom, integer width, integer height, text[] pixeltype, double precision[] value=ARRAY[1], double precision[] nodataval=ARRAY[0], double precision upperleftx=NULL, double precision upperlefty=NULL, double precision skewx=0, double precision skewy=0, boolean touched=false);

Description

PostGIS ⊠⊠⊠ PostGIS ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠.

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (scalex, scaley ⊠ gridx, gridy), ⊠⊠⊠⊠, NODATA ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠, ⊠⊠.

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (scalex & scaley ⊠ skewx & skewy) ⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠ width & height ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠, ⊠⊠⊠ scalex & scaley ⊠⊠⊠⊠ PostgreSQL ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (width & height) ⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (scalex & scaley ⊠ skewx & skewy) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (gridx & gridy) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (upperleftx & upperlefty)) ⊠⊠⊠⊠⊠⊠⊠⊠.

☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒☒☒☒, ☒☒ ☒☒☒☒☒☒☒☒☒ (pixeltype[]), ☒☒☒☒☒☒☒ (value) ☒☒☒ NODTATA ☒☒☒☒ (nodataval) ☒☒☒☒ ☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒, ☒☒☒☒☒☒☒☒☒☒☒ 8BUI, ☒☒☒☒ 1, NODATA ☒☒ 0 ☒☒☒☒.

☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒ SRID ☒☒☒☒☒☒☒.

☒☒☒☒ touched ☒☒☒☒☒☒☒☒☒☒☒☒☒☒ GDAL ALL_TOUCHED ☒☒☒☒☒☒☒☒☒☒☒☒☒☒, ☒☒☒☒ ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒, ☒☒☒ ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.

☒☒☒☒ ST_AsPNG ☒☒☒ ST_AsGDALRaster ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒ ☒☒ JPEG ☒☒ PNG ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.

2.0.0 ☒☒☒☒☒☒☒☒☒☒☒☒☒. GDAL 1.6.0 ☒☒☒☒☒☒☒☒☒☒☒.

Curved geometry inputs are linearized before rasterization, including NURBSCurve inputs handled by PostGIS. TIN and PolyhedralSurface inputs are not supported.

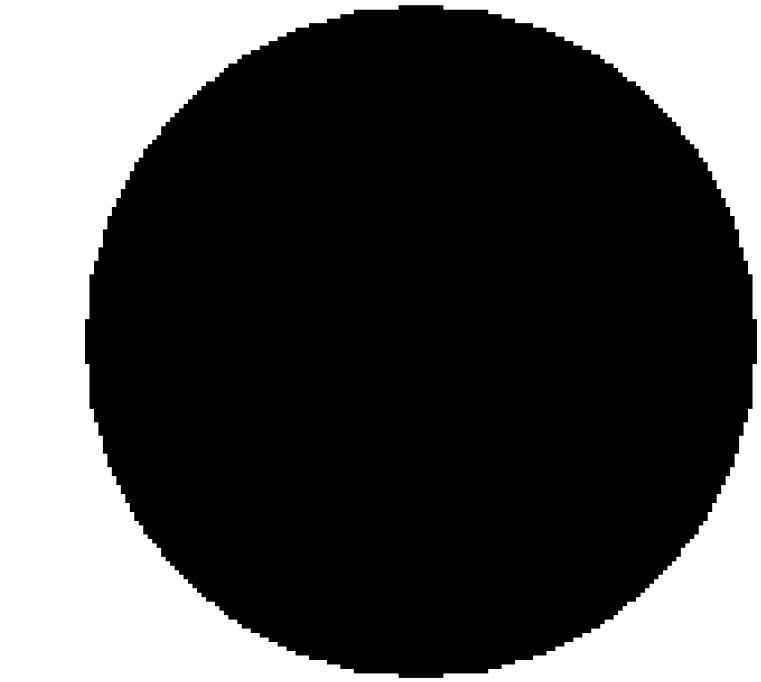
☒☒: PNG ☒☒☒☒☒☒☒☒

This example outputs a black circle on a white background, occupying 150 by 150 pixels.

Code

```
SELECT ST_AsPNG(
    ST_AsRaster(
        ST_Buffer(ST_Point(1, 5), 10),
        150, 150,
        ARRAY['8BUI', '8BUI', '8BUI'],
        ARRAY[0, 0, 0],
        ARRAY[255, 255, 255]
    )
) AS image;
```

Figure



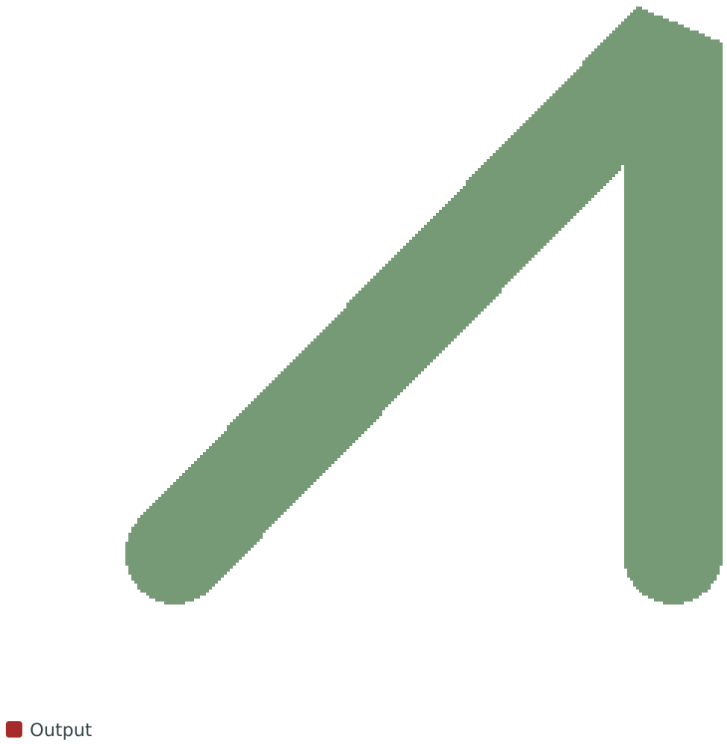
Output

This example maps bands to RGB channels, producing the teal value (118,154,118).

Code

```
SELECT ST_AsPNG(
    ST_AsRaster(
        ST_Buffer('LINESTRING(50 50,150 150,150 50)'::geometry, 10, 'join=bevel'),
        200, 200,
        ARRAY['8BUI', '8BUI', '8BUI'],
        ARRAY[118, 154, 118],
        ARRAY[255, 255, 255]
    )
) AS image;
```

Figure



☒☒

ST_BandPixelType, ST_Buffer, ST_GDALDrivers, ST_AsGDALRaster, ST_AsPNG, ST_AsJPEG, ST_SRID

11.3.3 ST_AsRasterAgg

ST_AsRasterAgg — Aggregate. Renders PostGIS geometries into a new raster.

☒☒

raster **ST_AsRasterAgg**(geometry geom, double precision val, raster ref, text pixeltype, double precision nodataval, text uniontype, boolean touched);

Description

Returns a single-band raster containing the rendered version of all incoming geometries, each with its associated value.

Availability: 3.6.0

☒☒

Code

```
WITH inp(g, v) AS (
    VALUES
        ( ST_Buffer(ST_MakePoint(10, 0), 10), 1 ),
        ( ST_Buffer(ST_MakePoint(20, 0), 10), 2 )
),
agg AS (
    SELECT ST_AsRasterAgg(
        g,
        v,
        ST_MakeEmptyRaster(0, 0, 0, 0, 1.0),
        '8BUI',
        99,
        'SUM',
        true
    ) r
    FROM inp
)
SELECT
    ST_Width(r) w,
    ST_Height(r) h,
    ST_Value(r, 'POINT(5 0)') v5_0,
    ST_Value(r, 'POINT(15 0)') v15_0,
    ST_Value(r, 'POINT(25 0)') v25_0
FROM agg;
```

Output

```
w  | h  | v5_0 | v15_0 | v25_0
----+----+------+-------+-------
 30 | 20 |    1 |     3 |     2
(1 row)
```

☒☒

ST_AsRaster, ST_DumpAsPolygons, ST_Union

11.3.4 ST_Band

ST_Band — ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.

☒☒

raster **ST_Band**(raster rast, integer[] nbands = ARRAY[1]);
raster **ST_Band**(raster rast, integer nband);
raster **ST_Band**(raster rast, text nbands, character delimiter=,);

Description

Returns one or more bands of an existing raster as a new raster. Useful for building new rasters from existing rasters or export of only selected bands of a raster or rearranging the order of bands in a raster. If no band is specified or any of specified bands does not exist in the raster, then all bands are returned. Used as a helper function in various functions such as for deleting a band.



□□
□□□□□□□□□□□□ nbands □□□, □□□□□□□ , □ '1,2,3' □□□□□□□□□□□□□□□□□□□□□□. □□□□□□□□□□□□□□□ ST_Band(rast, '1@2@3', '@') □□□□□□□□□□□□□□□□. □□□□□□□□□□□□□□□, □□□□□□□□□ ST_Band(rast, '{1,2,3}'::int[]); □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□. PostGIS □□□□□□□□□□□ text □□□□□□□□□□□□□□□□□□□□.

2.0.0 □□□□□□□□□□□□□□.

□□

This example creates two rasters: one containing band 1 of dummy_rast, and one containing band 2 reclassified as 2BUI.

Code

```
SELECT ST_NumBands(rast1) As numb1, ST_BandPixelType(rast1) As pix1,
 ST_NumBands(rast2) As numb2, ST_BandPixelType(rast2) As pix2
FROM (
    SELECT ST_Band(rast) As rast1, ST_Reclass(ST_Band(rast, 3), '100-200):1, [200-254:2', ' ↩
    2BUI') As rast2
        FROM dummy_rast
        WHERE rid = 2) As foo;
```

Output

```
numb1 | pix1 | numb2 | pix2
------+------+-------+------
    1 | 8BUI |     1 | 2BUI
```

Return bands 2 and 3 using array cast syntax.

Code

```
SELECT ST_NumBands(ST_Band(rast, '{2,3}'::int[])) As num_bands
FROM dummy_rast WHERE rid=2;
```

Output

```
2
```

Return bands 2 and 3 using array syntax.

Code

```
SELECT ST_NumBands(ST_Band(rast, ARRAY[2, 3])) As num_bands
    FROM dummy_rast
WHERE rid=2;
```

Output

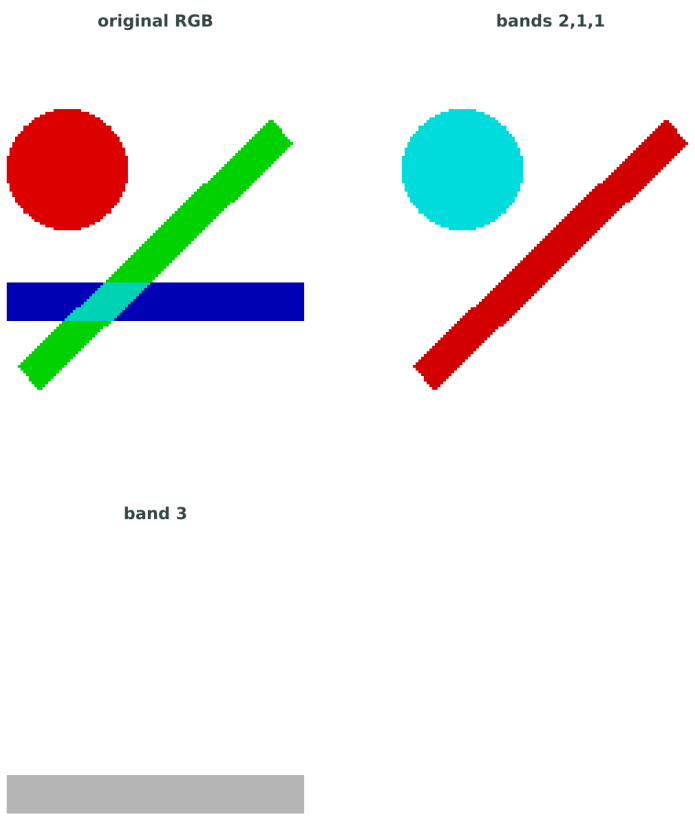
```
2
```

This example uses three distinct shapes as the red, green, and blue bands of a synthetic raster. It compares the original RGB raster with band 2 followed by band 1 twice, and with band 3 alone.

Code

```
WITH canvas AS (
    SELECT ST_AddBand(
        ST_AddBand(
            ST_AddBand(
                ST_MakeEmptyRaster(128, 128, 0, 128, 1, -1, 0, 0, 0),
                '8BUI'::text, 0::double precision, 0::double precision
            ),
            '8BUI'::text, 0::double precision, 0::double precision
        ),
        '8BUI'::text, 0::double precision, 0::double precision
    ) AS rast
), bands AS (
    SELECT ST_SetValue(
        ST_SetValue(
            ST_SetValue(rast, 1, ST_Buffer('POINT(32 96)'::geometry, 22), 220),
            2, ST_Buffer('LINESTRING(18 20,110 110)'::geometry, 6,
                         'endcap=flat join=bevel'), 210
        ),
        3, ST_Buffer('LINESTRING(10 48,118 48)'::geometry, 7,
                     'endcap=flat'), 180
    ) AS rast
    FROM canvas
)
SELECT ST_AsPNG(rast) AS "original RGB",
    ST_AsPNG(ST_Band(rast, ARRAY[2, 1, 1])) AS "bands 2,1,1",
    ST_AsPNG(ST_Band(rast, 3)) AS "band 3"
FROM bands;
```

Figure



Band selection from a multi-band raster.



□□ *(rast □)*



dupe_band



sing_band

□□

ST_AddBand, ST_NumBands, ST_Reclass, 11□

11.3.5 ST_MakeEmptyCoverage

ST_MakeEmptyCoverage — Cover georeferenced area with a grid of empty raster tiles.

□□

raster **ST_MakeEmptyCoverage**(integer tilewidth, integer tileheight, integer width, integer height, double precision upperleftx, double precision upperlefty, double precision scalex, double precision scaley, double precision skewx, double precision skewy, integer srid=unknown);

Description

Create a set of raster tiles with ST_MakeEmptyRaster. Grid dimension is `width` & `height`. Tile dimension is `tilewidth` & `tileheight`. The covered georeferenced area is from upper left corner (`upperleftx`, `upperlefty`) to lower right corner (`upperleftx` + `width` * `scalex`, `upperlefty` + `height` * `scaley`).



□□
Note that scaley is generally negative for rasters and scalex is generally positive. So lower right corner will have a lower y value and higher x value than the upper left corner.

2.2.0 □□□□□□□□□□□□.

□□□□

Create 16 tiles in a 4x4 grid to cover the WGS84 area from upper left corner (22, 77) to lower right corner (55, 33).

Code

```
SELECT (ST_MetaData(tile)).* FROM ST_MakeEmptyCoverage(1, 1, 4, 4, 22, 33, (55 - 22)/(4):: ↩
    float, (33 - 77)/(4)::float, 0., 0., 4326) tile;
```

Output

```
upperleftx | upperlefty | width | height | scalex | scaley | skewx | skewy | srid | numbands
-----------------------------------------------------------------------------------------
         22 |         33 |     1 |      1 |   8.25 |    -11 |     0 |     0 | 4326 |        0
      30.25 |         33 |     1 |      1 |   8.25 |    -11 |     0 |     0 | 4326 |        0
       38.5 |         33 |     1 |      1 |   8.25 |    -11 |     0 |     0 | 4326 |        0
      46.75 |         33 |     1 |      1 |   8.25 |    -11 |     0 |     0 | 4326 |        0
         22 |         22 |     1 |      1 |   8.25 |    -11 |     0 |     0 | 4326 |        0
      30.25 |         22 |     1 |      1 |   8.25 |    -11 |     0 |     0 | 4326 |        0
       38.5 |         22 |     1 |      1 |   8.25 |    -11 |     0 |     0 | 4326 |        0
      46.75 |         22 |     1 |      1 |   8.25 |    -11 |     0 |     0 | 4326 |        0
         22 |         11 |     1 |      1 |   8.25 |    -11 |     0 |     0 | 4326 |        0
      30.25 |         11 |     1 |      1 |   8.25 |    -11 |     0 |     0 | 4326 |        0
       38.5 |         11 |     1 |      1 |   8.25 |    -11 |     0 |     0 | 4326 |        0
      46.75 |         11 |     1 |      1 |   8.25 |    -11 |     0 |     0 | 4326 |        0
         22 |          0 |     1 |      1 |   8.25 |    -11 |     0 |     0 | 4326 |        0
      30.25 |          0 |     1 |      1 |   8.25 |    -11 |     0 |     0 | 4326 |        0
       38.5 |          0 |     1 |      1 |   8.25 |    -11 |     0 |     0 | 4326 |        0
      46.75 |          0 |     1 |      1 |   8.25 |    -11 |     0 |     0 | 4326 |        0
```

□□

ST_MakeEmptyRaster

11.3.6 ST_MakeEmptyRaster

ST_MakeEmptyRaster — ⊠⊠⊠⊠⊠ (⊠⊠ & ⊠⊠), ⊠⊠⊠ X ⊠ Y, ⊠⊠⊠⊠, ⊠⊠ (scalex, scaley, skewx & skewy) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (SRID) ⊠⊠⊠⊠⊠⊠ (⊠⊠⊠⊠⊠) ⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠ SRID ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. SRID ⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠ 0(unknown) ⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

raster **ST_MakeEmptyRaster**(raster rast);
raster **ST_MakeEmptyRaster**(integer width, integer height, float8 upperleftx, float8 upperlefty, float8 scalex, float8 scaley, float8 skewx, float8 skewy, integer srid=unknown);
raster **ST_MakeEmptyRaster**(integer width, integer height, float8 upperleftx, float8 upperlefty, float8 pixelsize);

Description

⊠⊠⊠⊠⊠ (⊠⊠ & ⊠⊠), ⊠⊠ (⊠⊠⊠⊠) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ X(upperleftx) ⊠⊠⊠⊠ Y(upperlefty), ⊠⊠⊠⊠, ⊠⊠ (scalex, scaley, skewx & skewy) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (SRID) ⊠⊠⊠⊠⊠ (⊠⊠⊠⊠⊠) ⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠⊠⊠⊠⊠⊠⊠⊠ (pixelsize) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. scalex ⊠⊠⊠⊠⊠⊠⊠⊠⊠, scaley ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. skewx ⊠ skewy ⊠ 0 ⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (⊠⊠⊠⊠⊠) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠ SRID ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ 0 ⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠ ST_AddBand ⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ST_SetValue ⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

Use an existing raster as a template and output metadata of the rasters we added.

Code

```
INSERT INTO dummy_rast(rid, rast)
VALUES(3, ST_MakeEmptyRaster(100, 100, 0.0005, 0.0005, 1, 1, 0, 0, 4326) );
INSERT INTO dummy_rast(rid, rast)
SELECT 4, ST_MakeEmptyRaster(rast)
FROM dummy_rast WHERE rid = 3;
SELECT
    rid,
    (md).width,
    (md).height,
    (md).scalex,
    (md).scaley,
    (md).srid
FROM (
    SELECT rid, ST_MetaData(rast) AS md
    FROM dummy_rast
    WHERE rid IN (3, 4)
) AS foo
ORDER BY rid;
```

Output

```
rid | width | height | scalex | scaley | srid
-----+-------+--------+--------+--------+------
   3 |   100 |    100 |      1 |      1 | 4326
   4 |   100 |    100 |      1 |      1 | 4326
```

⊠⊠

ST_AddBand, ST_MetaData, ST_ScaleX, ST_ScaleY, ST_SetValue, ST_SkewX, , ST_SkewY

11.3.7 ST_Tile

ST_Tile — ⊠⊠⊠.

⊠⊠

setof raster **ST_Tile**(raster rast, int[] nband, integer width, integer height, boolean padwithnodata=FALSE, double precision nodataval=NULL);
setof raster **ST_Tile**(raster rast, integer nband, integer width, integer height, boolean padwithnodata=FALSE, double precision nodataval=NULL);
setof raster **ST_Tile**(raster rast, integer width, integer height, boolean padwithnodata=FALSE, double precision nodataval=NULL);

Description

⊠⊠⊠.

`padwithnodata` = FALSE ⊠⊠⊠, ⊠⊠. `padwithnodata` = TRUE ⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ NODATA ⊠⊠⊠⊠⊠ (padding) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠ (⊠) ⊠⊠⊠⊠ NODATA ⊠ (⊠) ⊠⊠⊠⊠⊠⊠⊠⊠, `nodataval` ⊠⊠⊠⊠⊠ NODATA ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

> **⊠⊠**
> ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ DB ⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ DB ⊠⊠⊠⊠⊠⊠⊠⊠⊠.

2.1.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

Code

```
WITH foo AS (
    SELECT ST_AddBand(ST_AddBand(ST_MakeEmptyRaster(3, 3, 0, 0, 1, -1, 0, 0, 0), 1, '8BUI', ↩
     1, 0), 2, '8BUI', 10, 0) AS rast UNION ALL
    SELECT ST_AddBand(ST_AddBand(ST_MakeEmptyRaster(3, 3, 3, 0, 1, -1, 0, 0, 0), 1, '8BUI', ↩
     2, 0), 2, '8BUI', 20, 0) AS rast UNION ALL
    SELECT ST_AddBand(ST_AddBand(ST_MakeEmptyRaster(3, 3, 6, 0, 1, -1, 0, 0, 0), 1, '8BUI', ↩
     3, 0), 2, '8BUI', 30, 0) AS rast UNION ALL
```

```
    SELECT ST_AddBand(ST_AddBand(ST_MakeEmptyRaster(3, 3, 0, -3, 1, -1, 0, 0, 0), 1, '8BUI' ↩
    , 4, 0), 2, '8BUI', 40, 0) AS rast UNION ALL
    SELECT ST_AddBand(ST_AddBand(ST_MakeEmptyRaster(3, 3, 3, -3, 1, -1, 0, 0, 0), 1, '8BUI' ↩
    , 5, 0), 2, '8BUI', 50, 0) AS rast UNION ALL
    SELECT ST_AddBand(ST_AddBand(ST_MakeEmptyRaster(3, 3, 6, -3, 1, -1, 0, 0, 0), 1, '8BUI' ↩
    , 6, 0), 2, '8BUI', 60, 0) AS rast UNION ALL

    SELECT ST_AddBand(ST_AddBand(ST_MakeEmptyRaster(3, 3, 0, -6, 1, -1, 0, 0, 0), 1, '8BUI' ↩
    , 7, 0), 2, '8BUI', 70, 0) AS rast UNION ALL
    SELECT ST_AddBand(ST_AddBand(ST_MakeEmptyRaster(3, 3, 3, -6, 1, -1, 0, 0, 0), 1, '8BUI' ↩
    , 8, 0), 2, '8BUI', 80, 0) AS rast UNION ALL
    SELECT ST_AddBand(ST_AddBand(ST_MakeEmptyRaster(3, 3, 6, -6, 1, -1, 0, 0, 0), 1, '8BUI' ↩
    , 9, 0), 2, '8BUI', 90, 0) AS rast
), bar AS (
    SELECT ST_Union(rast) AS rast FROM foo
), baz AS (
    SELECT
        row_number() OVER (
            ORDER BY ST_UpperLeftY(rast) DESC, ST_UpperLeftX(rast)
        ) AS tile_id,
        rast
    FROM (
        SELECT ST_Tile(rast, 3, 3, TRUE) AS rast
        FROM bar
    ) AS tiled
)
SELECT
    tile_id,
    ST_UpperLeftX(rast) AS ulx,
    ST_UpperLeftY(rast) AS uly,
    ST_Value(rast, 1, 1, 1) AS band1,
    ST_Value(rast, 2, 1, 1) AS band2
FROM baz;
```

Output

```
tile_id | ulx | uly | band1 | band2
---------+-----+-----+-------+-------
       1 |   0 |   0 |     1 |    10
       2 |   3 |   0 |     2 |    20
       3 |   6 |   0 |     3 |    30
       4 |   0 |  -3 |     4 |    40
       5 |   3 |  -3 |     5 |    50
       6 |   6 |  -3 |     6 |    60
       7 |   0 |  -6 |     7 |    70
       8 |   3 |  -6 |     8 |    80
       9 |   6 |  -6 |     9 |    90
(9 rows)
```

Code

```
WITH foo AS (
    SELECT ST_AddBand(ST_AddBand(ST_MakeEmptyRaster(3, 3, 0, 0, 1, -1, 0, 0, 0), 1, '8BUI', ↩
     1, 0), 2, '8BUI', 10, 0) AS rast UNION ALL
    SELECT ST_AddBand(ST_AddBand(ST_MakeEmptyRaster(3, 3, 3, 0, 1, -1, 0, 0, 0), 1, '8BUI', ↩
     2, 0), 2, '8BUI', 20, 0) AS rast UNION ALL
    SELECT ST_AddBand(ST_AddBand(ST_MakeEmptyRaster(3, 3, 6, 0, 1, -1, 0, 0, 0), 1, '8BUI', ↩
     3, 0), 2, '8BUI', 30, 0) AS rast UNION ALL

    SELECT ST_AddBand(ST_AddBand(ST_MakeEmptyRaster(3, 3, 0, -3, 1, -1, 0, 0, 0), 1, '8BUI' ↩
    , 4, 0), 2, '8BUI', 40, 0) AS rast UNION ALL
    SELECT ST_AddBand(ST_AddBand(ST_MakeEmptyRaster(3, 3, 3, -3, 1, -1, 0, 0, 0), 1, '8BUI' ↩
    , 5, 0), 2, '8BUI', 50, 0) AS rast UNION ALL
    SELECT ST_AddBand(ST_AddBand(ST_MakeEmptyRaster(3, 3, 6, -3, 1, -1, 0, 0, 0), 1, '8BUI' ↩
    , 6, 0), 2, '8BUI', 60, 0) AS rast UNION ALL
```

```
    SELECT ST_AddBand(ST_AddBand(ST_MakeEmptyRaster(3, 3, 0, -6, 1, -1, 0, 0, 0), 1, '8BUI' ↩
    , 7, 0), 2, '8BUI', 70, 0) AS rast UNION ALL
    SELECT ST_AddBand(ST_AddBand(ST_MakeEmptyRaster(3, 3, 3, -6, 1, -1, 0, 0, 0), 1, '8BUI' ↩
    , 8, 0), 2, '8BUI', 80, 0) AS rast UNION ALL
    SELECT ST_AddBand(ST_AddBand(ST_MakeEmptyRaster(3, 3, 6, -6, 1, -1, 0, 0, 0), 1, '8BUI' ↩
    , 9, 0), 2, '8BUI', 90, 0) AS rast
), bar AS (
    SELECT ST_Union(rast) AS rast FROM foo
), baz AS (
    SELECT
        row_number() OVER (
            ORDER BY ST_UpperLeftY(rast) DESC, ST_UpperLeftX(rast)
        ) AS tile_id,
        rast
    FROM (
        SELECT ST_Tile(rast, ARRAY[2], 3, 3) AS rast
        FROM bar
    ) AS tiled
)
SELECT
    tile_id,
    ST_UpperLeftX(rast) AS ulx,
    ST_UpperLeftY(rast) AS uly,
    ST_Value(rast, 1, 1, 1) AS band_value
FROM baz;
```

Output

```
tile_id | ulx | uly | band_value
---------+-----+-----+------------
       1 |   0 |   0 |         10
       2 |   3 |   0 |         20
       3 |   6 |   0 |         30
       4 |   0 |  -3 |         40
       5 |   3 |  -3 |         50
       6 |   6 |  -3 |         60
       7 |   0 |  -6 |         70
       8 |   3 |  -6 |         80
       9 |   6 |  -6 |         90
(9 rows)
```

☒☒

ST_Union, ST_Retile

11.3.8 ST_Retile

ST_Retile — ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒, ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.

☒☒

SETOF raster **ST_Retile**(regclass tab, name col, geometry ext, float8 sfx, float8 sfy, int tw, int th, text algo='NearestNeighbor');

Description

⊠⊠⊠⊠⊠ (sfx, sfy) ⊠⊠⊠⊠⊠ (tw, th) ⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (tab, col) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (ext) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠⊠⊠⊠⊠⊠⊠'NearestNeighbor', 'Bilinear', 'Cubic', 'CubicSpline', ⊠⊠⊠'Lanczos' ⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠ GDAL Warp resampling methods ⊠⊠⊠⊠⊠⊠⊠.

2.2.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

ST_CreateOverview

11.3.9 ST_FromGDALRaster

ST_FromGDALRaster — ⊠⊠ GDAL ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

raster **ST_FromGDALRaster**(bytea gdaldata, integer srid=NULL);

Description

⊠⊠ GDAL ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. gdaldata ⊠ bytea ⊠⊠⊠⊠ GDAL ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

srid ⊠ NULL ⊠⊠⊠, ⊠⊠⊠⊠ GDAL ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ SRID ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. srid ⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ SRID ⊠⊠⊠⊠⊠⊠⊠⊠⊠.

2.1.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

Code

```
WITH foo AS (
    SELECT ST_AsPNG(ST_AddBand(ST_AddBand(ST_AddBand(ST_MakeEmptyRaster(2, 2, 0, 0, 0.1, ↩
    -0.1, 0, 0, 4326), 1, '8BUI', 1, 0), 2, '8BUI', 2, 0), 3, '8BUI', 3, 0)) AS png
),
bar AS (
    SELECT 1 AS rid, ST_FromGDALRaster(png) AS rast FROM foo
    UNION ALL
    SELECT 2 AS rid, ST_FromGDALRaster(png, 3310) AS rast FROM foo
)
SELECT
    rid,
    ST_MetaData(rast) AS metadata,
    ST_SummaryStats(rast, 1) AS stats1,
    ST_SummaryStats(rast, 2) AS stats2,
    ST_SummaryStats(rast, 3) AS stats3
FROM bar
ORDER BY rid;
```

Output

```
rid |         metadata          |    stats1     |    stats2     |     stats3
-----+---------------------------+---------------+---------------+----------------
   1 | (0,0,2,2,1,-1,0,0,0,3)    | (4,4,1,0,1,1) | (4,8,2,0,2,2) | (4,12,3,0,3,3)
   2 | (0,0,2,2,1,-1,0,0,3310,3) | (4,4,1,0,1,1) | (4,8,2,0,2,2) | (4,12,3,0,3,3)
(2 rows)
```

□□

ST_AsGDALRaster

11.4 □□□□□□ (accessor)

11.4.1 ST_GeoReference

ST_GeoReference — □□ (world) □□□□□□□□□□□□□□□□□□□□ GDAL □□ ESRI □□□□□□□□□□. □□□□ GDAL □□□.

□□

text **ST_GeoReference**(raster rast, text format=GDAL);

Description

□□□□ □□□□□□□□, □□□ (carriage) □□□□□□□□□□□□□□□□ GDAL □□ ESRI □□□□□□□□□□. □□□□□□□□□□□□□□□□□□ GDAL □□□. □□□□□□□'GDAL' □□'ESRI' □□□□□□□.

□□□□□□□□□□□□□□□□□□□□□□:

GDAL:

Output

```
scalex
skewy
skewx
scaley
upperleftx
upperlefty
```

ESRI:

Output

```
scalex
skewy
skewx
scaley
upperleftx + scalex*0.5
upperlefty + scaley*0.5
```

□□

Code

```
SELECT
    format,
    replace(ST_GeoReference(rast, format), E'\n', ' | ') AS georef
FROM dummy_rast
CROSS JOIN (VALUES ('ESRI'), ('GDAL')) AS formats(format)
WHERE rid = 1;
```

Output

```
format | georef
--------+-------------------------------------------------------------------------
 ESRI   | 2.0000000000 | 0.0000000000 | 0.0000000000 | 3.0000000000 | 1.5000000000 | 2.0000000000
 GDAL   | 2.0000000000 | 0.0000000000 | 0.0000000000 | 3.0000000000 | 0.5000000000 | 0.5000000000
(2 rows)
```

□□

ST_SetGeoReference, ST_ScaleX, ST_ScaleY

11.4.2 ST_Height

ST_Height — □□□□□□□□□□□□□□□□□□□□.

□□

integer **ST_Height**(raster rast);

Description

□□□□□□□□□□□□□□.

□□

Code

```
SELECT rid, ST_Height(rast) As rastheight
FROM dummy_rast;
```

Output

```
rid | rastheight
-----+------------
   1 |         20
   2 |          5
```

□□

ST_Width

11.4.3 ST_IsEmpty

ST_IsEmpty — ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (width = 0, height = 0) ⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

boolean **ST_IsEmpty**(raster rast);

Description

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (width = 0, height = 0) ⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

Code

```
SELECT ST_IsEmpty(ST_MakeEmptyRaster(100, 100, 0, 0, 0, 0, 0, 0))
```

Output

```
f
```

Code

```
SELECT ST_IsEmpty(ST_MakeEmptyRaster(0, 0, 0, 0, 0, 0, 0, 0))
```

Output

```
t
```

⊠⊠

ST_HasNoBand

11.4.4 ST_MemSize

ST_MemSize — ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (⊠⊠⊠⊠⊠⊠) ⊠⊠⊠⊠⊠.

⊠⊠

integer **ST_MemSize**(raster rast);

Description

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (⊠⊠⊠⊠⊠⊠) ⊠⊠⊠⊠⊠.

⊠⊠⊠⊠ PostgreSQL ⊠⊠⊠⊠⊠⊠ pg_column_size, pg_size_pretty, pg_relation_size, pg_total_relation_size ⊠⊠⊠⊠⊠⊠⊠⊠⊠.

> **⊠⊠**
>
> ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ pg_relation_size ⊠ ST_MemSize ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠ pg_relation_size ⊠⊠⊠ TOAST ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ TOAST ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. pg_column_size ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
> pg_total_relation_size ⊠⊠⊠⊠⊠⊠, TOAST ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

2.2.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

Code

```
SELECT ST_MemSize(ST_AsRaster(ST_Buffer(ST_Point(1, 5), 10, 1000), 150, 150, '8BUI')) AS ↩
    rast_mem;
```

Output

```
22568
```

⊠⊠

11.4.5 ST_MetaData

ST_MetaData — ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠ (skew), ⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

record **ST_MetaData**(raster rast);

Description

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠ (skew), ⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠: upperleftx | upperlefty | width | height | scalex | scaley | skewx | skewy | srid | numbands

⊠⊠

Code

```
SELECT
    rid,
    format('(%s, %s)', (md).upperleftx, (md).upperlefty) AS upper_left,
    format('%sx%s', (md).width, (md).height) AS size,
    format('(%s, %s)', (md).scalex, (md).scaley) AS scale,
    format('(%s, %s)', (md).skewx, (md).skewy) AS skew,
    (md).srid AS srid,
    (md).numbands AS numbands
FROM (
    SELECT rid, ST_MetaData(rast) AS md
    FROM dummy_rast
) AS foo
ORDER BY rid;
```

Output

```
rid |        upper_left        | size  |      scale      |  skew  | srid | numbands
-----+--------------------------+-------+-----------------+--------+------+----------
   1 | (0.5, 0.5)               | 10x20 | (2, 3)          | (0, 0) |    0 |        0
   2 | (3427927.75, 5793244)    | 5x5   | (0.05, -0.05)   | (0, 0) |    0 |        3
(2 rows)
```

□□

ST_BandMetaData, ST_NumBands

11.4.6 ST_NumBands

ST_NumBands — □□□□□□□□□□□□□□□□□□□□□□□□□□□□□.

□□

integer **ST_NumBands**(raster rast);

Description

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□.

□□

Code

```
SELECT rid, ST_NumBands(rast) As numbands
FROM dummy_rast;
```

Output

```
rid | numbands
----+----------
  1 |        0
  2 |        3
```

□□

ST_Value

11.4.7 ST_PixelHeight

ST_PixelHeight — □□□□□□□□□□□□□□□□□□□□□□□□□□□□.

□□

double precision **ST_PixelHeight**(raster rast);

Description

□□□□□□□□□□□□□□□□□□□□□□□□□□□□. □□□□□□□□□□□□□, □□□□□□□□□□□□□□
□□□□□□□□□□□□□□□.

□□□□□□□□□□□□□□□□□□□□ ST_PixelWidth □□□□□□□.

□□

Rasters with no skew.

Code

```
SELECT ST_Height(rast) As rastheight, ST_PixelHeight(rast) As pixheight,
 ST_ScaleX(rast) As scalex, ST_ScaleY(rast) As scaley, ST_SkewX(rast) As skewx,
        ST_SkewY(rast) As skewy
FROM dummy_rast;
```

Output

```
rastheight | pixheight | scalex | scaley | skewx | skewy
------------+-----------+--------+--------+-------+----------
         20 |         3 |      2 |      3 |     0 |        0
          5 |      0.05 |   0.05 |  -0.05 |     0 |        0
```

Rasters with skew different than 0.

Code

```
SELECT ST_Height(rast) As rastheight, ST_PixelHeight(rast) As pixheight,
 ST_ScaleX(rast) As scalex, ST_ScaleY(rast) As scaley, ST_SkewX(rast) As skewx,
        ST_SkewY(rast) As skewy
FROM (SELECT ST_SetSkew(rast, 0.5, 0.5) As rast
        FROM dummy_rast) As skewed;
```

Output

```
rastheight |     pixheight     | scalex | scaley | skewx | skewy
-----------+-------------------+--------+--------+-------+----------
        20 |  3.04138126514911 |      2 |      3 |   0.5 |      0.5
         5 | 0.502493781056044 |   0.05 |  -0.05 |   0.5 |      0.5
```

□□

ST_PixelWidth, ST_ScaleX, ST_ScaleY, ST_SkewX, ST_SkewY

11.4.8 ST_PixelWidth

ST_PixelWidth — ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

double precision **ST_PixelWidth**(raster rast);

Description

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

The following executable example draws the two pixel basis vectors on the same coordinate frame as a skewed raster footprint. The `i direction` vector is `(scaleX, skewY)`; the `j direction` vector is `(skewX, scaleY)`.

⊠⊠

Pixel basis vectors and footprint for a raster with non-zero skew.

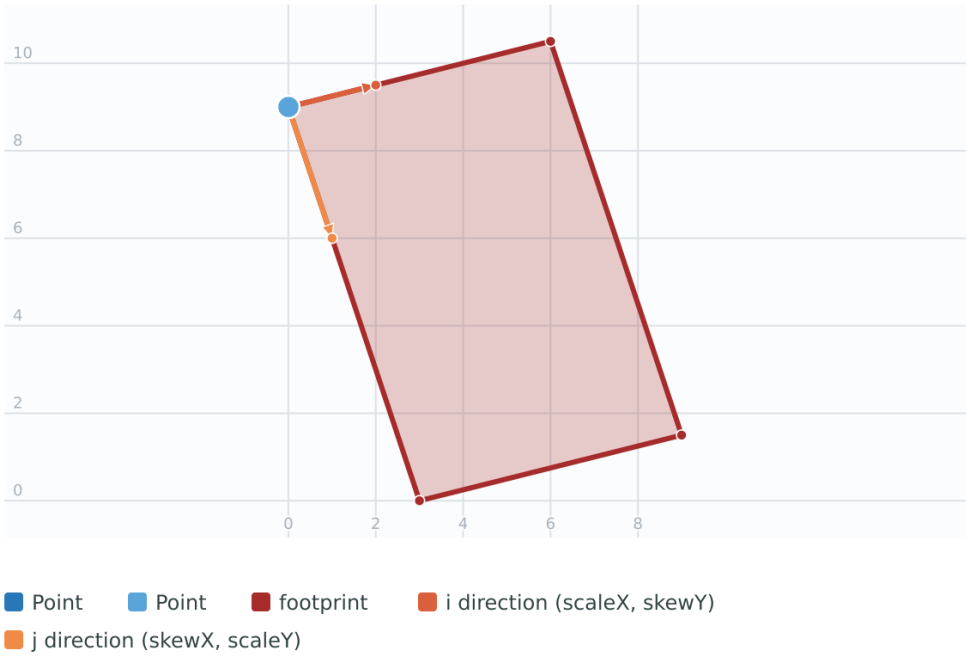
Code

```
WITH raster AS (
    SELECT ST_MakeEmptyRaster(
        3, 3, 0, 0,
        2, -3, 1, 0.5,
        0
    ) AS rast
)
SELECT
    ST_ConvexHull(rast) AS footprint,
    ST_MakeLine(
        ST_Point(0, 0),
        ST_Point(ST_ScaleX(rast), ST_SkewY(rast))
    ) AS "i direction (scaleX, skewY)",
    ST_MakeLine(
        ST_Point(0, 0),
        ST_Point(ST_SkewX(rast), ST_ScaleY(rast))
    ) AS "j direction (skewX, scaleY)"
FROM raster;
```

Output

```
POLYGON((0 0,6 1.5,9 -7.5,3 -9,0 0)) | LINESTRING(0 0,2 0.5) | LINESTRING(0 0,1 -3)
```

Figure



Rasters with no skew.

Code

```
SELECT ST_Width(rast) As rastwidth, ST_PixelWidth(rast) As pixwidth,
ST_ScaleX(rast) As scalex, ST_ScaleY(rast) As scaley, ST_SkewX(rast) As skewx,
ST_SkewY(rast) As skewy
FROM dummy_rast;
```

Output

```
rastwidth | pixwidth | scalex | scaley | skewx | skewy
----------+----------+--------+--------+-------+----------
10 |        2 |      2 |      3 |     0 |        0
 5 |     0.05 |   0.05 |  -0.05 |     0 |        0
```

Rasters with skew different than 0.

Code

```
SELECT ST_Width(rast) As rastwidth, ST_PixelWidth(rast) As pixwidth,
ST_ScaleX(rast) As scalex, ST_ScaleY(rast) As scaley, ST_SkewX(rast) As skewx,
ST_SkewY(rast) As skewy
FROM (SELECT ST_SetSkew(rast, 0.5, 0.5) As rast
FROM dummy_rast) As skewed;
```

Output

```
rastwidth |     pixwidth      | scalex | scaley | skewx | skewy
----------+-------------------+--------+--------+-------+----------
10 |  2.06155281280883 |      2 |      3 |   0.5 |      0.5
 5 | 0.502493781056044 |   0.05 |  -0.05 |   0.5 |      0.5
```

☒☒

ST_PixelHeight, ST_ScaleX, ST_ScaleY, ST_SkewX, ST_SkewY

11.4.9 ST_ScaleX

ST_ScaleX — ⊠⊠⊠⊠⊠ X ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

float8 **ST_ScaleX**(raster rast);

Description

⊠⊠⊠⊠⊠ X ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠.

⊠⊠⊠⊠: 2.0.0 ⊠⊠ WKTRaster ⊠⊠⊠⊠⊠ ST_PixelSizeX ⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

Code

```
SELECT rid, ST_ScaleX(rast) As rastpixwidth
FROM dummy_rast;
```

Output

```
rid | rastpixwidth
-----+--------------
   1 |            2
   2 |         0.05
```

⊠⊠

ST_Width

11.4.10 ST_ScaleY

ST_ScaleY — ⊠⊠⊠⊠⊠ Y ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

float8 **ST_ScaleY**(raster rast);

Description

⊠⊠⊠⊠⊠ Y ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠.

⊠⊠⊠⊠: 2.0.0 ⊠⊠ WKTRaster ⊠⊠⊠⊠⊠ ST_PixelSizeY ⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

Code

```
SELECT rid, ST_ScaleY(rast) As rastpixheight
FROM dummy_rast;
```

Output

```
rid | rastpixheight
-----+---------------
   1 |             3
   2 |         -0.05
```

⊠⊠

ST_Height

11.4.11 ST_RasterToWorldCoord

ST_RasterToWorldCoord — ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ X, Y(⊠⊠, ⊠⊠) ⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠ 1 ⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

record **ST_RasterToWorldCoord**(raster rast, integer xcolumn, integer yrow);

Description

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ X, Y(⊠⊠, ⊠⊠) ⊠⊠⊠⊠⊠⊠. ⊠⊠⊠ X, Y ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠ 1 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ 0, ⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠.

2.1.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

This example uses a non-skewed raster.

Code

```
SELECT
    rid,
    (ST_RasterToWorldCoord(rast, 1, 1)).*,
    (ST_RasterToWorldCoord(rast, 2, 2)).*
FROM dummy_rast
```

Output

```
rid | longitude  | latitude | longitude |  latitude
-----+------------+----------+-----------+------------
   1 |        0.5 |      0.5 |       2.5 |        3.5
   2 | 3427927.75 |  5793244 | 3427927.8 | 5793243.95
```

This example uses a skewed raster.

Code

```
SELECT
    rid,
    (ST_RasterToWorldCoord(rast, 1, 1)).*,
    (ST_RasterToWorldCoord(rast, 2, 3)).*
FROM (
    SELECT
        rid,
        ST_SetSkew(rast, 100.5, 0) As rast
    FROM dummy_rast
) As foo
```

Output

```
rid | longitude  | latitude | longitude | latitude
-----+------------+----------+-----------+------------
   1 |        0.5 |      0.5 |     203.5 |       6.5
   2 | 3427927.75 |  5793244 | 3428128.8 | 5793243.9
```

⊠⊠

ST_RasterToWorldCoordX, ST_RasterToWorldCoordY, ST_SetSkew

11.4.12 ST_RasterToWorldCoordX

ST_RasterToWorldCoordX — ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ X ⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠ 1 ⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

float8 **ST_RasterToWorldCoordX**(raster rast, integer xcolumn);
float8 **ST_RasterToWorldCoordX**(raster rast, integer xcolumn, integer yrow);

Description

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ X ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠ 1 ⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

> **⊠⊠**
> ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, X ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠ ST_ScaleX, ST_SkewX, ⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠ X ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠.

⊠⊠⊠⊠: 2.1.0 ⊠⊠⊠⊠⊠⊠⊠⊠ ST_Raster2WorldCoordX ⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

This example shows that providing a column is sufficient for a non-skewed raster.

Code

```
SELECT rid, ST_RasterToWorldCoordX(rast, 1) As x1coord,
    ST_RasterToWorldCoordX(rast, 2) As x2coord,
    ST_ScaleX(rast) As pixelx
FROM dummy_rast;
```

Output

```
rid |  x1coord   |  x2coord  | pixelx
-----+------------+-----------+--------
   1 |        0.5 |       2.5 |      2
   2 | 3427927.75 | 3427927.8 |   0.05
```

For fun let's skew it.

Code

```
SELECT rid, ST_RasterToWorldCoordX(rast, 1, 1) As x1coord,
    ST_RasterToWorldCoordX(rast, 2, 3) As x2coord,
    ST_ScaleX(rast) As pixelx
FROM (SELECT rid, ST_SetSkew(rast, 100.5, 0) As rast FROM dummy_rast) As foo;
```

Output

```
rid |  x1coord   |  x2coord  | pixelx
-----+------------+-----------+--------
   1 |        0.5 |     203.5 |      2
   2 | 3427927.75 | 3428128.8 |   0.05
```

⊠⊠

ST_ScaleX, ST_RasterToWorldCoordY, ST_SetSkew, ST_SkewX

11.4.13 ST_RasterToWorldCoordY

ST_RasterToWorldCoordY — ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ Y ⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠ 1 ⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

float8 **ST_RasterToWorldCoordY**(raster rast, integer yrow);
float8 **ST_RasterToWorldCoordY**(raster rast, integer xcolumn, integer yrow);

Description

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ Y ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠ 1 ⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠/⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

> **⊠⊠**
> ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, Y ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠ ST_ScaleY, ST_SkewY, ⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠ Y ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠.

⊠⊠⊠⊠: 2.1.0 ⊠⊠⊠⊠⊠⊠⊠ ST_Raster2WorldCoordY ⊠⊠⊠⊠⊠⊠⊠⊠⊠.

□□

Non-skewed raster providing row is sufficient.

Code

```
SELECT rid, ST_RasterToWorldCoordY(rast, 1) As y1coord,
    ST_RasterToWorldCoordY(rast, 3) As y2coord,
    ST_ScaleY(rast) As pixely
FROM dummy_rast;
```

Output

```
rid | y1coord |  y2coord  | pixely
-----+---------+-----------+--------
   1 |     0.5 |       6.5 |      3
   2 | 5793244 | 5793243.9 |  -0.05
```

For fun let's skew it.

Code

```
SELECT rid, ST_RasterToWorldCoordY(rast, 1, 1) As y1coord,
    ST_RasterToWorldCoordY(rast, 2, 3) As y2coord,
    ST_ScaleY(rast) As pixely
FROM (SELECT rid, ST_SetSkew(rast, 0, 100.5) As rast FROM dummy_rast) As foo;
```

Output

```
rid | y1coord |  y2coord  | pixely
-----+---------+-----------+--------
   1 |     0.5 |       107 |      3
   2 | 5793244 | 5793344.4 |  -0.05
```

□□

ST_ScaleY, ST_RasterToWorldCoordX, ST_SetSkew, ST_SkewY

11.4.14 ST_Rotation

ST_Rotation — □□□□□□□□□□□□□□□□□□□□□□□□.

□□

float8 **ST_Rotation**(raster rast);

Description

□□□□□□□□□□□□□□□□□□□□□□□□. □□□□□□□□□□□□□□□□□□□□□□□□, NaN □□□□□□. □□□□□□□ □□□□ □□□□□□□□□.

□□

Code

```
SELECT rid, ST_Rotation(ST_SetScale(ST_SetSkew(rast, sqrt(2)), sqrt(2))) AS rot FROM ↩
    dummy_rast;
```

Output

```
rid |        rot
----+-------------------
  1 | 0.785398163397448
  2 | 0.785398163397448
```

□□

ST_SetRotation, ST_SetScale, ST_SetSkew

11.4.15 ST_SkewX

ST_SkewX — □□□□ X □□□ (skew)(□□□□□□□□□) □□□□□□.

□□

float8 **ST_SkewX**(raster rast);

Description

□□□□ X □□□ (□□□□□□□□□) □□□□□□. □□□□□□ □□□□ □□□□□□□□.

□□

Code

```
WITH refs AS (
    SELECT
        rid,
        format('(%s, %s)', ST_SkewX(rast), ST_SkewY(rast)) AS skew,
        string_to_array(ST_GeoReference(rast), E'\n') AS georef
    FROM dummy_rast
)
SELECT
    rid,
    skew,
    term,
    georef[idx] AS value
FROM refs
CROSS JOIN (VALUES
    (1, 'scaleX'),
    (2, 'skewY'),
    (3, 'skewX'),
    (4, 'scaleY'),
    (5, 'originX'),
    (6, 'originY')
) AS parts(idx, term)
ORDER BY rid, idx;
```

Output

```
rid |  skew  |  term   |       value
-----+--------+---------+--------------------
   1 | (0, 0) | scaleX  | 2.0000000000
   1 | (0, 0) | skewY   | 0.0000000000
   1 | (0, 0) | skewX   | 0.0000000000
   1 | (0, 0) | scaleY  | 3.0000000000
   1 | (0, 0) | originX | 0.5000000000
   1 | (0, 0) | originY | 0.5000000000
   2 | (0, 0) | scaleX  | 0.0500000000
   2 | (0, 0) | skewY   | 0.0000000000
   2 | (0, 0) | skewX   | 0.0000000000
   2 | (0, 0) | scaleY  | -0.0500000000
   2 | (0, 0) | originX | 3427927.7500000000
   2 | (0, 0) | originY | 5793244.0000000000
(12 rows)
```

□□

ST_GeoReference, ST_SkewY, ST_SetSkew

11.4.16 ST_SkewY

ST_SkewY — □□□□ Y □□□ (□□□□□□□□□□) □□□□□□.

□□

float8 **ST_SkewY**(raster rast);

Description

□□□□ Y □□□ (□□□□□□□□□□) □□□□□□. □□□□□□ □□□□ □□□□□□□□.

□□

Code

```
WITH refs AS (
    SELECT
        rid,
        format('(%s, %s)', ST_SkewX(rast), ST_SkewY(rast)) AS skew,
        string_to_array(ST_GeoReference(rast), E'\n') AS georef
    FROM dummy_rast
)
SELECT
    rid,
    skew,
    term,
    georef[idx] AS value
FROM refs
CROSS JOIN (VALUES
    (1, 'scaleX'),
    (2, 'skewY'),
    (3, 'skewX'),
    (4, 'scaleY'),
    (5, 'originX'),
    (6, 'originY')
```

```
) AS parts(idx, term)
ORDER BY rid, idx;
```

Output

```
rid |  skew  |  term   |       value
-----+--------+---------+--------------------
   1 | (0, 0) | scaleX  | 2.0000000000
   1 | (0, 0) | skewY   | 0.0000000000
   1 | (0, 0) | skewX   | 0.0000000000
   1 | (0, 0) | scaleY  | 3.0000000000
   1 | (0, 0) | originX | 0.5000000000
   1 | (0, 0) | originY | 0.5000000000
   2 | (0, 0) | scaleX  | 0.0500000000
   2 | (0, 0) | skewY   | 0.0000000000
   2 | (0, 0) | skewX   | 0.0000000000
   2 | (0, 0) | scaleY  | -0.0500000000
   2 | (0, 0) | originX | 3427927.7500000000
   2 | (0, 0) | originY | 5793244.0000000000
(12 rows)
```

□□

ST_GeoReference, ST_SkewX, ST_SetSkew

11.4.17 ST_SRID

ST_SRID — spatial_ref_sys □□□□□□□□□□, □□□□□□□□□□□□□□□□□□□□.

□□

integer **ST_SRID**(raster rast);

Description

spatial_ref_sys □□□□□□□□□□, □□□□□□□□□□□□□□□□□□□□.



□□
Non-georeferenced rasters and geometries use SRID 0.

□□

Code

```
SELECT ST_SRID(rast) As srid
FROM dummy_rast WHERE rid=1;
```

Output

```
0
```

□□

4.5□, ST_SRID

11.4.18 ST_Summary

ST_Summary — □□□□□□□□□□□□□□□□□□□□□□□□□□□□□.

□□

text **ST_Summary**(raster rast);

Description

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□.
2.1.0 □□□□□□□□□□□□□□□□□□□□□□□□□□□□□.

□□

Code

```
SELECT ST_Summary(ST_AddBand(ST_AddBand(ST_AddBand(ST_MakeEmptyRaster(10, 10, 0, 0, 1, -1, ↩
    0, 0, 0),
                1, '8BUI', 1, 0
            ),
            2, '32BF', 0, -9999
        ),
        3, '16BSI', 0, NULL
    )
);
```

Output

```
Raster of 10x10 pixels has 3 bands and extent of BOX(0 -10,10 0)+
     band 1 of pixtype 8BUI is in-db with NODATA value of 0      +
     band 2 of pixtype 32BF is in-db with NODATA value of -9999  +
     band 3 of pixtype 16BSI is in-db with no NODATA value
(1 row)
```

□□

ST_MetaData, ST_BandMetaData, ST_Summary ST_Extent

11.4.19 ST_UpperLeftX

ST_UpperLeftX — □□□□□□□□ X □□□□□□□□□□□□□□□□□□□□□□□□□□□□□.

□□

float8 **ST_UpperLeftX**(raster rast);

Description

□□□□□□□ X □□□□□□□□□□□□□□□□□□□□.

□□

Code

```
SELECt rid, ST_UpperLeftX(rast) As ulx
FROM dummy_rast;
```

Output

```
rid |    ulx
-----+------------
   1 |        0.5
   2 | 3427927.75
```

□□

ST_UpperLeftY, ST_GeoReference, Box3D

11.4.20 ST_UpperLeftY

ST_UpperLeftY — □□□□□□□ Y □□□□□□□□□□□□□□□□□□□□.

□□

float8 **ST_UpperLeftY**(raster rast);

Description

□□□□□□□ Y □□□□□□□□□□□□□□□□□□□□.

□□

Code

```
SELECT rid, ST_UpperLeftY(rast) As uly
FROM dummy_rast;
```

Output

```
rid |   uly
-----+---------
   1 |     0.5
   2 | 5793244
```

□□

ST_UpperLeftX, ST_GeoReference, Box3D

11.4.21 ST_Width

ST_Width — ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

integer **ST_Width**(raster rast);

Description

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

Code

```
SELECT ST_Width(rast) As rastwidth
FROM dummy_rast WHERE rid=1;
```

Output

```
10
```

⊠⊠

ST_Height

11.4.22 ST_WorldToRasterCoord

ST_WorldToRasterCoord — ⊠⊠⊠⊠⊠⊠⊠⊠ X, Y(⊠⊠, ⊠⊠) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

record **ST_WorldToRasterCoord**(raster rast, geometry pt);
record **ST_WorldToRasterCoord**(raster rast, double precision longitude, double precision latitude);

Description

⊠⊠⊠⊠⊠⊠⊠⊠ X, Y(⊠⊠, ⊠⊠) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠ X, Y ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠ X, Y ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

2.1.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

Code

```
SELECT
    rid,
    (ST_WorldToRasterCoord(rast, 3427927.8, 20.5)).*,
    (ST_WorldToRasterCoord(rast, ST_GeomFromText('POINT(3427927.8 20.5)', ST_SRID(rast)))) ↩
    .*
FROM dummy_rast;
```

Output

```
rid | columnx |   rowy    | columnx |   rowy
-----+---------+-----------+---------+-----------
   1 | 1713964 |         7 | 1713964 |         7
   2 |       2 | 115864471 |       2 | 115864471
```

□□

ST_WorldToRasterCoordX, ST_WorldToRasterCoordY, ST_RasterToWorldCoordX, ST_RasterToWorldCoordY, ST_SRID

11.4.23 ST_WorldToRasterCoordX

ST_WorldToRasterCoordX — □□□□□ (pt) □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ X, Y □□□□□ (xw, yw) □□□□□□□.

□□

integer **ST_WorldToRasterCoordX**(raster rast, geometry pt);
integer **ST_WorldToRasterCoordX**(raster rast, double precision xw);
integer **ST_WorldToRasterCoordX**(raster rast, double precision xw, double precision yw);

Description

□□□□□□ (pt) □□□□□□□□□□□ X, Y □□□□□ (xw, yw) □□□□□□□. □□□□□□□□□□□ (□□□□□□□□ □□□□□□ xw □ yw □□□□□□□□□□□□□□□□□). □□□□□□□□□□□□□□□ xw □□□□□□□□□□□□ □. □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□.

□□□□: 2.1.0 □□□□□□□□ ST_World2RasterCoordX □□□□□□□□□□.

□□

Code

```
SELECT rid, ST_WorldToRasterCoordX(rast, 3427927.8) As xcoord,
        ST_WorldToRasterCoordX(rast, 3427927.8, 20.5) As xcoord_xwyw,
        ST_WorldToRasterCoordX(rast, ST_GeomFromText('POINT(3427927.8 20.5)', ST_SRID(rast) ↩
    )) As ptxcoord
FROM dummy_rast;
```

Output

```
rid | xcoord  |  xcoord_xwyw   | ptxcoord
-----+---------+---------+----------
   1 | 1713964 | 1713964 |  1713964
   2 |       1 |       1 |        1
```

⊠⊠

ST_RasterToWorldCoordX, ST_RasterToWorldCoordY, ST_SRID

11.4.24 ST_WorldToRasterCoordY

ST_WorldToRasterCoordY — ⊠⊠⊠⊠⊠ (pt) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ X, Y ⊠⊠⊠⊠⊠ (xw, yw) ⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

integer **ST_WorldToRasterCoordY**(raster rast, geometry pt);
integer **ST_WorldToRasterCoordY**(raster rast, double precision xw);
integer **ST_WorldToRasterCoordY**(raster rast, double precision xw, double precision yw);

Description

⊠⊠⊠⊠⊠ (pt) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ X, Y ⊠⊠⊠⊠⊠ (xw, yw) ⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ xw ⊠ yw ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠). ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ xw ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠⊠⊠: 2.1.0 ⊠⊠⊠⊠⊠⊠⊠⊠ ST_World2RasterCoordY ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

Code

```
SELECT rid, ST_WorldToRasterCoordY(rast, 20.5) As ycoord,
        ST_WorldToRasterCoordY(rast, 3427927.8, 20.5) As ycoord_xwyw,
        ST_WorldToRasterCoordY(rast, ST_GeomFromText('POINT(3427927.8 20.5)', ST_SRID(rast) ↩
    )) As ptycoord
FROM dummy_rast;
```

Output

```
rid |  ycoord   | ycoord_xwyw | ptycoord
-----+-----------+-------------+-----------
   1 |         7 |           7 |         7
   2 | 115864471 |   115864471 | 115864471
```

⊠⊠

ST_RasterToWorldCoordX, ST_RasterToWorldCoordY, ST_SRID

11.5 ⊠⊠⊠⊠⊠⊠⊠⊠

11.5.1 ST_BandMetaData

ST_BandMetaData — ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ 1 ⊠⊠ ⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

table(pixeltype text, nodatavalue double precision, isoutdb boolean, path text, outdbbandnum integer, filesize bigint, filetimestamp bigint) **ST_BandMetaData**(raster rast, integer band=1);
table(bandnum integer, pixeltype text, nodatavalue double precision, isoutdb boolean, path text, outdbbandnum integer, filesize bigint, filetimestamp bigint) **ST_BandMetaData**(raster rast, integer[] band);

Description

Returns basic meta data about a raster band. Columns returned: pixeltype, nodatavalue, isoutdb, path, outdbbandnum, filesize, filetimestamp.

> ⊠⊠
> ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

> ⊠⊠
> If band has no NODATA value, nodatavalue are NULL.

> ⊠⊠
> If isoutdb is False, path, outdbbandnum, filesize and filetimestamp are NULL. If outdb access is disabled, filesize and filetimestamp will also be NULL.

Enhanced: 2.5.0 to include *outdbbandnum*, *filesize* and *filetimestamp* for outdb rasters.

⊠⊠

Variant 1.

Code

```
SELECT
    rid,
    (foo.md).*
FROM (
    SELECT
        rid,
        ST_BandMetaData(rast, 1) AS md
    FROM dummy_rast
    WHERE rid=2
) As foo;
```

Output

```
rid | pixeltype | nodatavalue | isoutdb | path | outdbbandnum
-----+-----------+-------------+---------+------+--------------
   2 | 8BUI      |           0 | f       |      |
(1 row)
```

Variant 2.

Code

```
WITH foo AS (
    SELECT
        ST_AddBand(NULL::raster, '/home/pele/devel/geo/postgis-git/raster/test/regress/ ↩
    loader/Projected.tif', NULL::int[]) AS rast
)
SELECT
    bandnum,
    pixeltype,
    isoutdb,
    regexp_replace(path, '^.*/', '') AS file,
    outdbbandnum
FROM ST_BandMetaData((SELECT rast FROM foo),
    ARRAY[1, 3, 2]::int[]
);
```

Output

```
bandnum | pixeltype | isoutdb |     file      | outdbbandnum
---------+-----------+---------+---------------+--------------
       1 | 8BUI      | t       | Projected.tif |            1
       3 | 8BUI      | t       | Projected.tif |            3
       2 | 8BUI      | t       | Projected.tif |            2
```

⊠⊠

ST_MetaData, ST_BandPixelType

11.5.2 ST_BandNoDataValue

ST_BandNoDataValue — ⊠⊠⊠⊠⊠⊠ NODATA ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠ 1 ⊠⊠⊠⊠⊠⊠.

⊠⊠

double precision **ST_BandNoDataValue**(raster rast, integer bandnum=1);

Description

⊠⊠⊠⊠ NODATA ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

Code

```
SELECT ST_BandNoDataValue(rast, 1) As bnval1,
    ST_BandNoDataValue(rast, 2) As bnval2, ST_BandNoDataValue(rast, 3) As bnval3
FROM dummy_rast
WHERE rid = 2;
```

Output

```
bnval1 | bnval2 | bnval3
--------+--------+--------
      0 |      0 |      0
```

□□

ST_NumBands

11.5.3 ST_BandIsNoData

ST_BandIsNoData — □□□ NODATA □□□□□□□□□□□□□□□□□□□□□.

□□

boolean **ST_BandIsNoData**(raster rast, integer band, boolean forceChecking=false);
boolean **ST_BandIsNoData**(raster rast, boolean forceChecking=false);

Description

□□□ NODATA □□□□□□□□□□□□□□□□□□□□. □□□□□□□□□□□□□ 1 □□□□□□□. □□□□ □□ TRUE □□□□, □□□□□□□□□□□□□□□□□□□□. □□□□□□□□, □□□□□□□□□□ isnodata □□ □□□□□□□□□□□□□. □□□□□□□□□□□, □□□□□□□□□□□□□ FALSE □□□□.

2.0.0 □□□□□□□□□□□□□.



□□
If the flag is dirty (this is, the result is different using TRUE as last parameter and not using it) you should update the raster to set this flag to true, by using ST_SetBandIsNoData(), or ST_SetBandNoDataValue() with TRUE as last argument. See ST_SetBandIsNoData.

□□

This example creates a dummy table with one raster column. The raster has two bands and one pixel per band. In the first band, the NODATA value and pixel value are both 3. In the second band, the NODATA value is 13 and the pixel value is 4.

Code

```
create table dummy_rast (rid integer, rast raster);

insert into dummy_rast values(
1,
(
'01' -- little endian (uint8 ndr)
```

```
||
'0000' -- version (uint16 0)
||
'0200' -- nBands (uint16 0)
||
'17263529ED684A3F' -- scaleX (float64 0.000805965234044584)
||
'F9253529ED684ABF' -- scaleY (float64 -0.00080596523404458)
||
'1C9F33CE69E352C0' -- ipX (float64 -75.5533328537098)
||
'718F0E9A27A44840' -- ipY (float64 49.2824585505576)
||
'ED50EB853EC32B3F' -- skewX (float64 0.000211812383858707)
||
'7550EB853EC32B3F' -- skewY (float64 0.000211812383858704)
||
'E6100000' -- SRID (int32 4326)
||
'0100' -- width (uint16 1)
||
'0100' -- height (uint16 1)
||
'6' -- hasnodatavalue and isnodata value set to true.
||
'2' -- first band type (4BUI)
||
'03' -- novalue==3
||
'03' -- pixel(0,0)==3 (same that nodata)
||
'0' -- hasnodatavalue set to false
||
'5' -- second band type (16BSI)
||
'0D00' -- novalue==13
||
'0400' -- pixel(0,0)==4
)::raster
);

select ST_BandIsNoData(rast, 1) from dummy_rast where rid = 1; -- Expected true
select ST_BandIsNoData(rast, 2) from dummy_rast where rid = 1; -- Expected false
```

□□

ST_BandNoDataValue, ST_NumBands, ST_SetBandNoDataValue, ST_SetBandIsNoData

11.5.4 ST_BandPath

ST_BandPath — □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□. bandnum □□□□□□□□□□□□ 1 □□□□□□□.

□□

text **ST_BandPath**(raster rast, integer bandnum=1);

Description

□□□□□□□□□□□□□□□□□□□□□□. DB □□□□□□□□□□□□□□□□□□□□□□.

□□

Code

```
WITH foo AS (
    SELECT ST_AddBand(
        NULL::raster,
        '/home/pele/devel/geo/postgis-git/raster/test/regress/loader/Projected.tif',
        NULL::int[]
    ) AS rast
)
SELECT bandnum, ST_BandPath(rast, bandnum) AS path
FROM foo
CROSS JOIN generate_series(1, 3) AS bandnum;
```

Output

```
bandnum |                                 path
--------+-----------------------------------------------------------------------
      1 | /home/pele/devel/geo/postgis-git/raster/test/regress/loader/Projected.tif
      2 | /home/pele/devel/geo/postgis-git/raster/test/regress/loader/Projected.tif
      3 | /home/pele/devel/geo/postgis-git/raster/test/regress/loader/Projected.tif
```

□□

ST_BandMetaData, ST_SetBandPath, ST_SetBandIndex

11.5.5 ST_BandFileSize

ST_BandFileSize — Returns the file size of a band stored in file system. If no bandnum specified, 1 is assumed.

□□

bigint **ST_BandFileSize**(raster rast, integer bandnum=1);

Description

Returns the file size of a band stored in file system. Throws an error if called with an in db band, or if outdb access is not enabled.

This function is typically used in conjunction with ST_BandPath() and ST_BandFileTimestamp() so a client can determine if the filename of a outdb raster as seen by it is the same as the one seen by the server.

Availability: 2.5.0

□□

Code

```
SELECT ST_BandFileSize(rast, 1) FROM dummy_rast WHERE rid = 1;
```

Output

```
240574
```

11.5.6 ST_BandFileTimestamp

ST_BandFileTimestamp — Returns the file timestamp of a band stored in file system. If no bandnum specified, 1 is assumed.

□□

bigint **ST_BandFileTimestamp**(raster rast, integer bandnum=1);

Description

Returns the file timestamp (number of seconds since Jan 1st 1970 00:00:00 UTC) of a band stored in file system. Throws an error if called with an in db band, or if outdb access is not enabled.

This function is typically used in conjunction with ST_BandPath() and ST_BandFileSize() so a client can determine if the filename of a outdb raster as seen by it is the same as the one seen by the server.

Availability: 2.5.0

□□

Code

```
SELECT ST_BandFileTimestamp(rast, 1) FROM dummy_rast WHERE rid = 1;
```

Output

```
1521807257
```

11.5.7 ST_BandPixelType

ST_BandPixelType — □□□□□□□□□□□□□□□□. bandnum □□□□□□□□□□ 1 □□□□□□.

□□

text **ST_BandPixelType**(raster rast, integer bandnum=1);

Description

Returns name describing data type and size of values stored in each cell of given band.

11 ☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒:

- 1BB - 1 ☒☒☒☒
- 2BUI - ☒☒☒☒ 2 ☒☒☒☒☒
- 4BUI - ☒☒☒☒ 4 ☒☒☒☒☒
- 8BSI - ☒☒☒☒ 8 ☒☒☒☒☒
- 8BUI - ☒☒☒☒ 8 ☒☒☒☒☒
- 16BSI - ☒☒☒☒ 16 ☒☒☒☒☒
- 16BUI - ☒☒☒☒ 16 ☒☒☒☒☒
- 32BSI - ☒☒☒☒ 32 ☒☒☒☒☒
- 32BUI - ☒☒☒☒ 32 ☒☒☒☒☒
- 32BF - 32 ☒☒☒☒☒☒☒☒☒☒
- 64BF - 64 ☒☒☒☒☒☒☒☒☒☒

☒☒

Code

```
SELECT ST_BandPixelType(rast, 1) As btype1,
    ST_BandPixelType(rast, 2) As btype2, ST_BandPixelType(rast, 3) As btype3
FROM dummy_rast
WHERE rid = 2;
```

Output

```
btype1 | btype2 | btype3
--------+--------+--------
 8BUI   | 8BUI   | 8BUI
```

☒☒

ST_NumBands

11.5.8 ST_MinPossibleValue

ST_MinPossibleValue — ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.

☒☒

integer **ST_MinPossibleValue**(text pixeltype);

Description

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

Code

```
SELECT ST_MinPossibleValue('16BSI');
```

Output

```
-32768
```

Code

```
SELECT ST_MinPossibleValue('8BUI');
```

Output

```
0
```

⊠⊠

ST_BandPixelType

11.5.9 ST_HasNoBand

ST_HasNoBand — ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠ 1 ⊠⊠⊠⊠⊠⊠.

⊠⊠

boolean **ST_HasNoBand**(raster rast, integer bandnum=1);

Description

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠ 1 ⊠⊠⊠⊠⊠⊠.

2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

Code

```
SELECT rid, ST_HasNoBand(rast) As hb1, ST_HasNoBand(rast, 2) as hb2,
ST_HasNoBand(rast, 4) as hb4, ST_NumBands(rast) As numbands
FROM dummy_rast;
```

Output

```
rid | hb1 | hb2 | hb4 | numbands
-----+-----+-----+-----+----------
1 | t   | t   | t   |        0
2 | f   | f   | t   |        3
```

□□

ST_NumBands

11.6 □□□□□□□□□□□□ (setter)

11.6.1 ST_PixelAsPolygon

ST_PixelAsPolygon — □□□□□□□□□□□□□□□□□□□□□□□□□□□□.

□□

geometry **ST_PixelAsPolygon**(raster rast, integer columnx, integer rowy);

Description

□□□□□□□□□□□□□□□□□□□□□□□□□□□□.

2.0.0 □□□□□□□□□□□□.

□□

This example returns a raster pixel polygon.

Code

```
SELECT i, j, ST_PixelAsPolygon(foo.rast, i, j) As b1pgeom
FROM dummy_rast As foo
    CROSS JOIN generate_series(1, 2) As i
    CROSS JOIN generate_series(1, 1) As j
WHERE rid=2;
```

Output

```
i | j |                                                 b1pgeom
---+---+-----------------------------------------------------------------------------
 1 | 1 | POLYGON((3427927.75 5793244,3427927.8 5793244,3427927.8 5793243.95,...
 2 | 1 | POLYGON((3427927.8 5793244,3427927.85 5793244,3427927.85 5793243.95, ..
```

□□

ST_DumpAsPolygons, ST_PixelAsPolygons, ST_PixelAsPoint, ST_PixelAsPoints, ST_PixelAsCentroid, ST_PixelAsCentroids, ST_Intersection, ST_AsText

11.6.2 ST_PixelAsPolygons

ST_PixelAsPolygons — □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ X, Y □□□□□□□ □□□□□□□.

□□

table(geom geometry, val double precision, x integer, y integer) **ST_PixelAsPolygons**(raster rast, integer band=1, boolean exclude_nodata_value=TRUE);

Description

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ (□□□□□□□□□□) □ X, Y □□□□□□ (□□□□) □□□□□□□□□□.

Return record format: *geom* geometry, *val* double precision, *x* integer, *y* integers.



□□
When *exclude_nodata_value* is TRUE, only pixels whose values are not NODATA are returned. Unlike ST_DumpAsPolygons, this function returns one polygon per pixel rather than merging adjacent pixels with the same value.

2.0.0 □□□□□□□□□□□□□.

□□□□: 2.1.0 □□□□□□□□□ exclude_nodata_value □□□□□□□□□□.

□□□□: 2.1.1 □□□□□ exclude_nodata_value □□□□□□□□□□□.

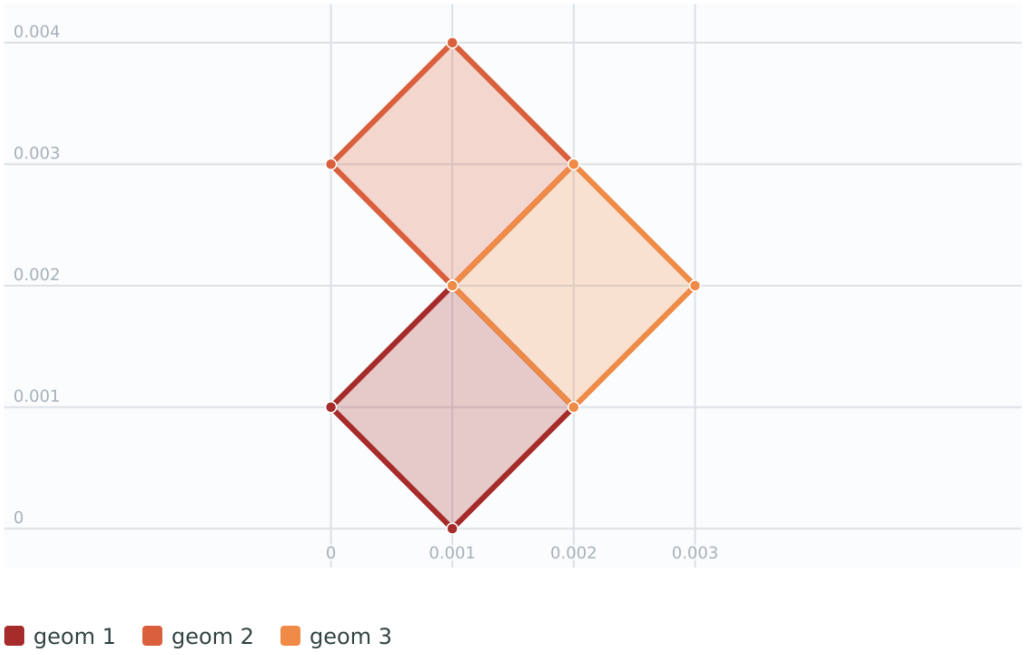
□□

This example returns a raster pixel polygon.

Code

```
WITH raster AS (
    SELECT ST_SetValue(
        ST_SetValue(
            ST_AddBand(
                ST_MakeEmptyRaster(
                    2, 2, 0, 0, 0.001, -0.001, 0.001, 0.001, 4269
                ),
                1, '8BUI', 1, 0
            ),
            2, 2, 10
        ),
        1, 1, NULL
    ) AS rast
), pixels AS (
    SELECT (ST_PixelAsPolygons(rast)).*
    FROM raster
)
SELECT x, y, val, geom AS geom
FROM pixels
ORDER BY x, y;
```

Figure



□□

ST_DumpAsPolygons, ST_PixelAsPolygon, ST_PixelAsPoint, ST_PixelAsPoints, ST_PixelAsCentroid, ST_Pixel
ST_AsText

11.6.3 ST_PixelAsPoint

ST_PixelAsPoint — □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□.

□□

geometry **ST_PixelAsPoint**(raster rast, integer columnx, integer rowy);

Description

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□.
2.1.0 □□□□□□□□□□□□□□.

□□

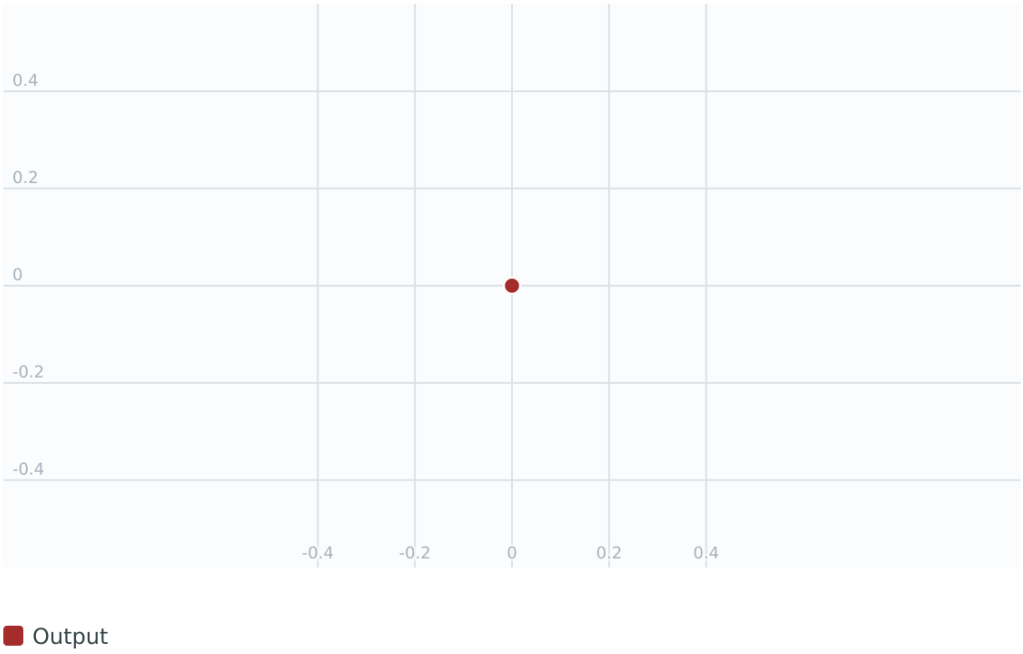
Code

```
SELECT ST_PixelAsPoint(rast, 1, 1) FROM dummy_rast WHERE rid = 1;
```

Output

```
POINT(0.5 0.5)
```

Figure



□□

ST_DumpAsPolygons, ST_PixelAsPolygon, ST_PixelAsPolygons, ST_PixelAsPoints, ST_PixelAsCentroid, ST_PixelAsCentroids

11.6.4 ST_PixelAsPoints

ST_PixelAsPoints — □□□□□□□□□□□□□□□□□□□□□□□□□□□ X, Y □□□□□□□□□□□□□□. □□□□□□□□□□□□□□□□□□□□□□□□□.

□□

table(geom geometry, val double precision, x integer, y integer) **ST_PixelAsPoints**(raster rast, integer band=1, boolean exclude_nodata_value=TRUE);

Description

□□□□□□□□□□□□□□□□□□□□□□□□□□□ X, Y □□□□□□□□□□□□□□. □□□□□□□□□□□□ □□□□□□□□□□□□□.

Return record format: *geom* geometry, *val* double precision, *x* integer, *y* integers.



□□
When *exclude_nodata_value* = TRUE, only those pixels whose values are not NODATA are returned as points.

2.1.0 □□□□□□□□□□□□.

□□□□: 2.1.1 □□□□ exclude_nodata_value □□□□□□□□□□.

□□

This example returns point features together with a derived marker size that a client can use for value-scaled point rendering.

Code

```
SELECT
    x,
    y,
    val,
    round((val / 32.0)::numeric, 2) AS marker_scale,geom AS geom
FROM (
    SELECT (ST_PixelAsPoints(rast, 1)).*
    FROM dummy_rast
    WHERE rid = 2
) AS foo
ORDER BY y, x;
```

Output

```
x | y | val | marker_scale |          geom
---+---+-----+--------------+----------------------------
 1 | 1 | 253 |         7.91 | POINT(3427927.75 5793244)
 2 | 1 | 254 |         7.94 | POINT(3427927.8 5793244)
 3 | 1 | 253 |         7.91 | POINT(3427927.85 5793244)
 4 | 1 | 254 |         7.94 | POINT(3427927.9 5793244)
 5 | 1 | 254 |         7.94 | POINT(3427927.95 5793244)
 1 | 2 | 253 |         7.91 | POINT(3427927.75 5793243.95)
 2 | 2 | 254 |         7.94 | POINT(3427927.8 5793243.95)
 3 | 2 | 254 |         7.94 | POINT(3427927.85 5793243.95)
 4 | 2 | 253 |         7.91 | POINT(3427927.9 5793243.95)
 5 | 2 | 249 |         7.78 | POINT(3427927.95 5793243.95)
 1 | 3 | 250 |         7.81 | POINT(3427927.75 5793243.9)
 2 | 3 | 254 |         7.94 | POINT(3427927.8 5793243.9)
 3 | 3 | 254 |         7.94 | POINT(3427927.85 5793243.9)
 4 | 3 | 252 |         7.88 | POINT(3427927.9 5793243.9)
 5 | 3 | 249 |         7.78 | POINT(3427927.95 5793243.9)
 1 | 4 | 251 |         7.84 | POINT(3427927.75 5793243.85)
 2 | 4 | 253 |         7.91 | POINT(3427927.8 5793243.85)
 3 | 4 | 254 |         7.94 | POINT(3427927.85 5793243.85)
 4 | 4 | 254 |         7.94 | POINT(3427927.9 5793243.85)
 5 | 4 | 253 |         7.91 | POINT(3427927.95 5793243.85)
 1 | 5 | 252 |         7.88 | POINT(3427927.75 5793243.8)
 2 | 5 | 250 |         7.81 | POINT(3427927.8 5793243.8)
 3 | 5 | 254 |         7.94 | POINT(3427927.85 5793243.8)
 4 | 5 | 254 |         7.94 | POINT(3427927.9 5793243.8)
 5 | 5 | 254 |         7.94 | POINT(3427927.95 5793243.8)
```

Figure



□□

ST_DumpAsPolygons, ST_PixelAsPolygon, ST_PixelAsPolygons, ST_PixelAsPoint, ST_PixelAsCentroid, ST_PixelAsCentroids

11.6.5 ST_PixelAsCentroid

ST_PixelAsCentroid — □□□□□□□□□□□□□□□□ (□□□□□□) □□□□□□□.

□□

geometry **ST_PixelAsCentroid**(raster rast, integer x, integer y);

Description

□□□□□□□□□□□□□□□□ (□□□□□□) □□□□□□□.
□□□□: 2.1.0 □□□□□ C □□□□□□□□□□□□.
2.1.0 □□□□□□□□□□□□□□.

□□

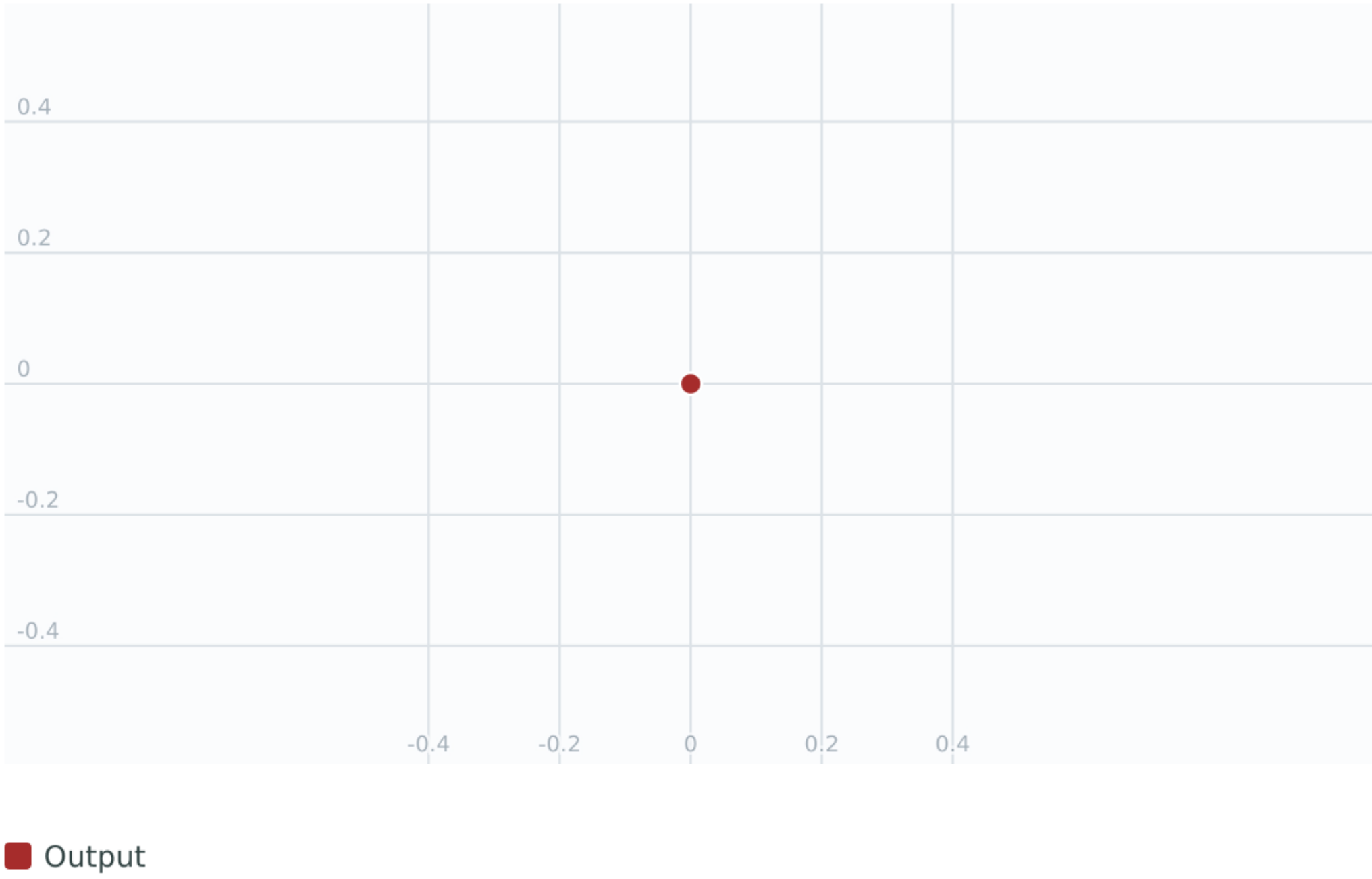
Code

```
SELECT ST_PixelAsCentroid(rast, 1, 1) FROM dummy_rast WHERE rid = 1;
```

Output

```
POINT(1.5 2)
```

Figure



⊠⊠

ST_DumpAsPolygons, ST_PixelAsPolygon, ST_PixelAsPolygons, ST_PixelAsPoint, ST_PixelAsPoints, ST_Pixel

11.6.6 ST_PixelAsCentroids

ST_PixelAsCentroids — ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (⊠⊠⊠⊠⊠⊠) ⊠⊠⊠⊠⊠⊠⊠⊠ X, Y ⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

table(geom geometry, val double precision, x integer, y integer) **ST_PixelAsCentroids**(raster rast, integer band=1, boolean exclude_nodata_value=TRUE);

Description

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (⊠⊠⊠⊠⊠⊠) ⊠⊠⊠⊠⊠⊠⊠⊠ X, Y ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

Return record format: *geom* geometry, *val* double precision, *x* integer, *y* integers.

> **Note!** ⊠⊠
> When *exclude_nodata_value* = TRUE, only those pixels whose values are not NODATA are returned as points.

□□□□: 2.1.0 □□□□ C □□□□□□□□□□□.

□□□□: 2.1.1 □□□□ exclude_nodata_value □□□□□□□□□□.

2.1.0 □□□□□□□□□□□□□.

□□

This example uses LATERAL syntax.

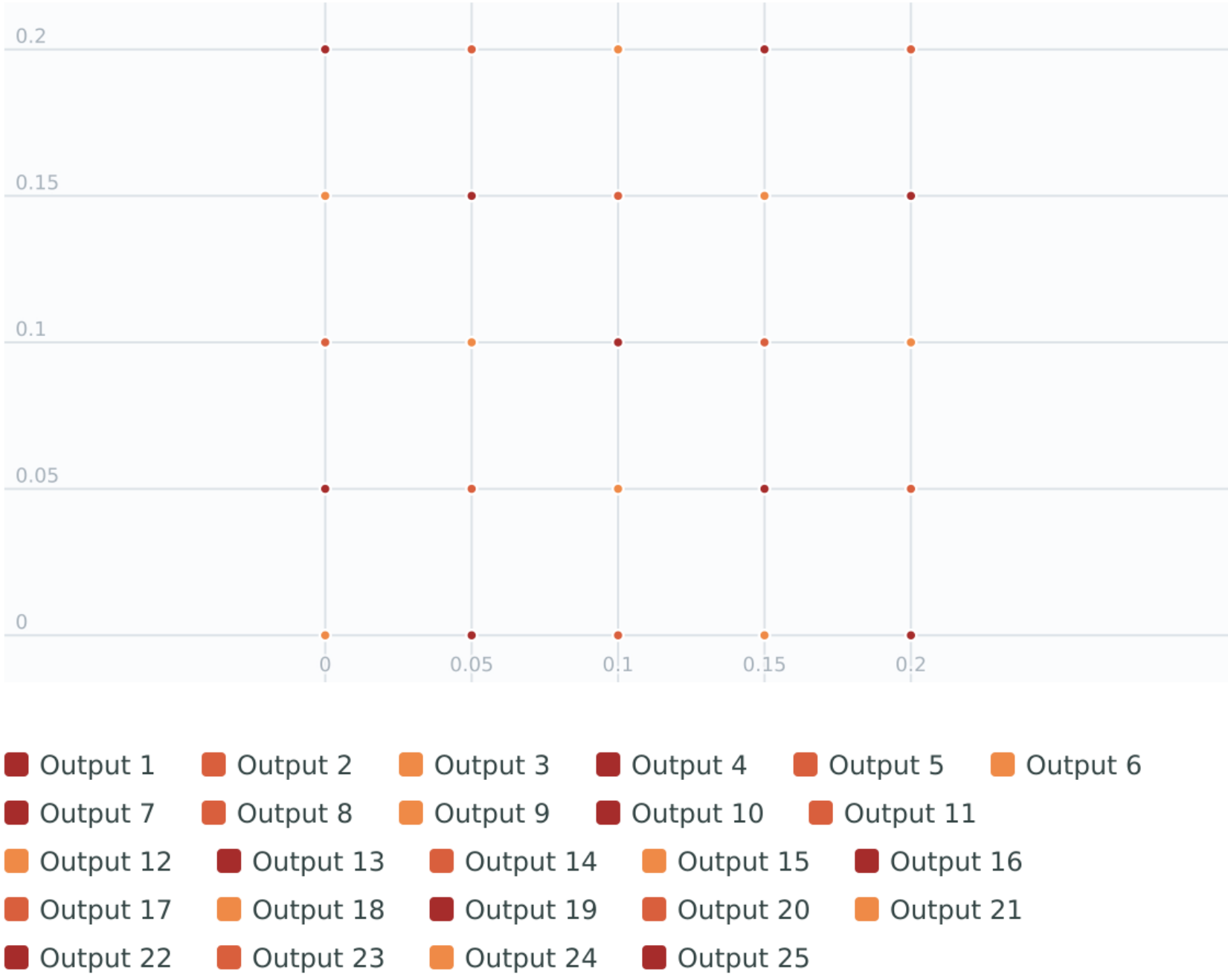
Code

```
SELECT x, y, val,geom
    FROM (SELECT dp.* FROM dummy_rast, LATERAL ST_PixelAsCentroids(rast, 1) AS dp WHERE rid ↩
     = 2) foo;
```

Output

```
x | y | val |           st_astext
---+---+-----+-------------------------------
 1 | 1 | 253 | POINT(3427927.775 5793243.975)
 2 | 1 | 254 | POINT(3427927.825 5793243.975)
 3 | 1 | 253 | POINT(3427927.875 5793243.975)
 4 | 1 | 254 | POINT(3427927.925 5793243.975)
 5 | 1 | 254 | POINT(3427927.975 5793243.975)
 1 | 2 | 253 | POINT(3427927.775 5793243.925)
 2 | 2 | 254 | POINT(3427927.825 5793243.925)
 3 | 2 | 254 | POINT(3427927.875 5793243.925)
 4 | 2 | 253 | POINT(3427927.925 5793243.925)
 5 | 2 | 249 | POINT(3427927.975 5793243.925)
 1 | 3 | 250 | POINT(3427927.775 5793243.875)
 2 | 3 | 254 | POINT(3427927.825 5793243.875)
 3 | 3 | 254 | POINT(3427927.875 5793243.875)
 4 | 3 | 252 | POINT(3427927.925 5793243.875)
 5 | 3 | 249 | POINT(3427927.975 5793243.875)
 1 | 4 | 251 | POINT(3427927.775 5793243.825)
 2 | 4 | 253 | POINT(3427927.825 5793243.825)
 3 | 4 | 254 | POINT(3427927.875 5793243.825)
 4 | 4 | 254 | POINT(3427927.925 5793243.825)
 5 | 4 | 253 | POINT(3427927.975 5793243.825)
 1 | 5 | 252 | POINT(3427927.775 5793243.775)
 2 | 5 | 250 | POINT(3427927.825 5793243.775)
 3 | 5 | 254 | POINT(3427927.875 5793243.775)
 4 | 5 | 254 | POINT(3427927.925 5793243.775)
 5 | 5 | 254 | POINT(3427927.975 5793243.775)
```

Figure



⊠⊠

ST_DumpAsPolygons, ST_PixelAsPolygon, ST_PixelAsPolygons, ST_PixelAsPoint, ST_PixelAsPoints, ST_Pixel

11.6.7 ST_Value

ST_Value — ⊠⊠ columnx, rowy ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠ 1 ⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ 1 ⊠⊠⊠⊠⊠⊠⊠. `exclude_nodata_value` ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, `nodata` ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. `exclude_nodata_value` ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

double precision **ST_Value**(raster rast, geometry pt, boolean exclude_nodata_value=true);
double precision **ST_Value**(raster rast, integer band, geometry pt, boolean exclude_nodata_value=true, text resample='nearest');
double precision **ST_Value**(raster rast, integer x, integer y, boolean exclude_nodata_value=true);
double precision **ST_Value**(raster rast, integer band, integer x, integer y, boolean exclude_nodata_value=tru

Description

⊠⊠⊠ columnx, rowy ⊠⊠⊠. ⊠⊠⊠⊠⊠ 1 ⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ 1 ⊠⊠⊠⊠⊠⊠⊠.

`exclude_nodata_value` ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠ `nodata` ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ `exclude_nodata_value` ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

The allowed values of the `resample` parameter are "nearest" which performs the default nearest-neighbor resampling, "bilinear" which performs a bilinear interpolation to estimate the value between

pixel centers, and the nearest-neighbor boundary options "nearest-UL", "nearest-UR", "nearest-LL", and "nearest-LR". Boundary options choose which pixel is returned when the point lies on a horizontal or vertical pixel boundary, including corner intersections.

Enhanced: 3.7.0 resample accepts nearest-neighbor boundary options "nearest-UL", "nearest-UR", "nearest-LL", and "nearest-LR".

□□□□: 2.1.0 □□□□□□□□ exclude_nodata_value □□□□□□□□.

□□□□: 2.0.0 □□□□□□□□ exclude_nodata_value □□□□□□□□.

□□

Get raster values at particular postgis geometry points. The srid of your geometry should be same as for your raster.

Code

```
SELECT rid, ST_Value(rast, foo.pt_geom) As b1pval, ST_Value(rast, 2, foo.pt_geom) As b2pval
FROM dummy_rast CROSS JOIN (SELECT ST_SetSRID(ST_Point(3427927.77, 5793243.76), 0) As ↩
    pt_geom) As foo
WHERE rid=2;
```

Output

```
rid | b1pval | b2pval
-----+--------+--------
   2 |    252 |     79
```

General fictitious example using a real table.

Code

```
SELECT rid, ST_Value(rast, 3, sometable.geom) As b3pval
FROM sometable
WHERE ST_Intersects(rast, sometable.geom);
```

Code

```
SELECT rid, ST_Value(rast, 1, 1, 1) As b1pval,
    ST_Value(rast, 2, 1, 1) As b2pval, ST_Value(rast, 3, 1, 1) As b3pval
FROM dummy_rast
WHERE rid=2;
```

Output

```
rid | b1pval | b2pval | b3pval
-----+--------+--------+--------
   2 |    253 |     78 |     70
```

Get all values in bands 1,2,3 of each pixel.

Code

```
SELECT x, y, ST_Value(rast, 1, x, y) As b1val,
    ST_Value(rast, 2, x, y) As b2val, ST_Value(rast, 3, x, y) As b3val
FROM dummy_rast CROSS JOIN
generate_series(1, 1000) As x CROSS JOIN generate_series(1, 1000) As y
WHERE rid =  2 AND x <= ST_Width(rast) AND y <= ST_Height(rast);
```

Output

```
 x | y | b1val | b2val | b3val
---+---+-------+-------+-------
 1 | 1 |   253 |    78 |    70
 1 | 2 |   253 |    96 |    80
 1 | 3 |   250 |    99 |    90
 1 | 4 |   251 |    89 |    77
 1 | 5 |   252 |    79 |    62
 2 | 1 |   254 |    98 |    86
 2 | 2 |   254 |   118 |   108
 :
 :
```

Get all values in bands 1,2,3 of each pixel same as above but returning the upper left point point of each pixel.

Code

```
SELECT ST_SetSRID(ST_Point(ST_UpperLeftX(rast) + ST_ScaleX(rast)*x,
        ST_UpperLeftY(rast) + ST_ScaleY(rast)*y),
        ST_SRID(rast)) As uplpt,
    ST_Value(rast, 1, x, y) As b1val,
    ST_Value(rast, 2, x, y) As b2val, ST_Value(rast, 3, x, y) As b3val
FROM dummy_rast CROSS JOIN
generate_series(1,1000) As x CROSS JOIN generate_series(1,1000) As y
WHERE rid =  2 AND x <= ST_Width(rast) AND y <= ST_Height(rast);
```

Output

```
            uplpt            | b1val | b2val | b3val
-----------------------------+-------+-------+-------
 POINT(3427929.25 5793245.5) |   253 |    78 |    70
 POINT(3427929.25 5793247)   |   253 |    96 |    80
 POINT(3427929.25 5793248.5) |   250 |    99 |    90
:
```

Get a polygon formed by union of all pixels that fall in a particular value range and intersect a particular polygon.

Code

```
SELECT ST_Union(pixpolyg) As shadow
FROM (SELECT ST_Translate(ST_MakeEnvelope(ST_UpperLeftX(rast), ST_UpperLeftY(rast),
            ST_UpperLeftX(rast) + ST_ScaleX(rast),
            ST_UpperLeftY(rast) + ST_ScaleY(rast), 0
            ), ST_ScaleX(rast)*x, ST_ScaleY(rast)*y
        ) As pixpolyg, ST_Value(rast, 2, x, y) As b2val
    FROM dummy_rast CROSS JOIN
generate_series(1,1000) As x CROSS JOIN generate_series(1,1000) As y
WHERE rid =  2
    AND x <= ST_Width(rast) AND y <= ST_Height(rast)) As foo
WHERE
    ST_Intersects(
        pixpolyg,
        ST_GeomFromText('POLYGON((3427928 5793244,3427927.75 5793243.75,3427928 ↩
    5793243.75,3427928 5793244))',0)
        ) AND b2val != 254;
```

Output

```
       shadow
-------------------------------------------------------------------------------
 MULTIPOLYGON(((3427928 5793243.9,3427928 5793243.85,3427927.95 5793243.85,3427927.95 5793243.9,
 3427927.95 5793243.95,3427928 5793243.95,3427928.05 5793243.95,3427928.05 5793243.9,3427928 5793243.9)),((3427927.95 ↩
    5793243.9,3427927.95 579324
```

```
3.85,3427927.9 5793243.85,3427927.85 5793243.85,3427927.85 5793243.9,3427927.9 5793243.9,3427927.9 5793243.95,
3427927.95 5793243.95,3427927.95 5793243.9)),((3427927.85 5793243.75,3427927.85 5793243.7,3427927.8 ↩
    5793243.7,3427927.8 5793243.75,
3427927.8 5793243.8,3427927.8 5793243.85,3427927.85 5793243.85,3427927.85 5793243.8,3427927.85 5793243.75)),
((3427928.05 5793243.75,3427928.05 5793243.7,3427928 5793243.7,3427927.95 5793243.7,3427927.95 5793243.75,3427927.95 ↩
    5793243.8,3427
927.95 5793243.85,3427928 5793243.85,3427928 5793243.8,3427928.05 5793243.8,
3427928.05 5793243.75)),((3427927.95 5793243.75,3427927.95 5793243.7,3427927.9 5793243.7,3427927.85 5793243.7,
3427927.85 5793243.75,3427927.85 5793243.8,3427927.85 5793243.85,3427927.9 5793243.85,
3427927.95 5793243.85,3427927.95 5793243.8,3427927.95 5793243.75)))
```

Checking all the pixels of a large raster tile can take a long time. You can dramatically improve speed at some lose of precision by orders of magnitude. By sampling pixels using the step optional parameter of generate_series. This next example does the same as previous but by checking 1 for every 4 (2x2) pixels and putting in the last checked. Putting in the checked pixel as the value for subsequent 4.

Code

```
SELECT ST_Union(pixpolyg) As shadow
FROM (SELECT ST_Translate(ST_MakeEnvelope(ST_UpperLeftX(rast), ST_UpperLeftY(rast),
            ST_UpperLeftX(rast) + ST_ScaleX(rast)*2,
            ST_UpperLeftY(rast) + ST_ScaleY(rast)*2, 0
            ), ST_ScaleX(rast)*x, ST_ScaleY(rast)*y
        ) As pixpolyg, ST_Value(rast, 2, x, y) As b2val
    FROM dummy_rast CROSS JOIN
generate_series(1,1000,2) As x CROSS JOIN generate_series(1,1000,2) As y
WHERE rid =  2
    AND x <= ST_Width(rast)  AND y <= ST_Height(rast)  ) As foo
WHERE
    ST_Intersects(
        pixpolyg,
        ST_GeomFromText('POLYGON((3427928 5793244,3427927.75 5793243.75,3427928 ↩
    5793243.75,3427928 5793244))',0)
        ) AND b2val != 254;
```

Output

```
        shadow
------------------------------------------------------------------------------------
 MULTIPOLYGON(((3427927.9 5793243.85,3427927.8 5793243.85,3427927.8 5793243.95,
 3427927.9 5793243.95,3427928 5793243.95,3427928.1 5793243.95,3427928.1 5793243.85,3427928 ↩
    5793243.85,3427927.9 5793243.85)),
 ((3427927.9 5793243.65,3427927.8 5793243.65,3427927.8 5793243.75,3427927.8 ↩
    5793243.85,3427927.9 5793243.85,
 3427928 5793243.85,3427928 5793243.75,3427928.1 5793243.75,3427928.1 5793243.65,3427928 ↩
    5793243.65,3427927.9 5793243.65)))
```

□□

ST_DumpValues, ST_SetValue, ST_DumpAsPolygons, ST_NumBands, ST_PixelAsPolygon, ST_ScaleX, ST_ScaleY, ST_UpperLeftX, ST_UpperLeftY, ST_SRID, ST_AsText, ST_Point, ST_MakeEnvelope, ST_Intersec ST_Intersection

11.6.8 ST_NearestValue

ST_NearestValue — columnx □ rowy, □□□ NODATA □□□□□□□□□□.

□□

double precision **ST_NearestValue**(raster rast, integer bandnum, geometry pt, boolean exclude_nodata_val
double precision **ST_NearestValue**(raster rast, geometry pt, boolean exclude_nodata_value=true);
double precision **ST_NearestValue**(raster rast, integer bandnum, integer columnx, integer rowy, boolean exclude_nodata_value=true);
double precision **ST_NearestValue**(raster rast, integer columnx, integer rowy, boolean exclude_nodata_valu

Description

□□□ columnx, rowy □□, □□□□□□□□□□□□□□□□□□□□□□□□□□□ NODATA □□□□□□□□ □□□□. columnx, rowy □□□□□□□□□□□□□□□□□□□□□ NODATA □□□□, □□□□ columnx, rowy □ □□□□□□□□□□□□□□□□□□□□□□□□ NODATA □□□□□□□□□□□□□.

□□□□□ 1 □□□□□□□, □□□□□□□□□□□ bandnum □ 1 □□□□□□□. exclude_nodata_value □□ □□□□□□□□□, nodata □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□. exclude_nodata_value □□□□□□□□□□, □□□□□□□□□□□□□□□□□□□□□□.

2.1.0 □□□□□□□□□□□□.

Note! □□
ST_NearestValue □□□□□□□□ ST_Value □□□□□□□□□□.

□□

This example reads a value from pixel 2x2.

Code

```
WITH source AS (
    SELECT ST_SetValues(
        ST_AddBand(
            ST_MakeEmptyRaster(5, 5, -2, 2, 1, -1, 0, 0, 0),
            1, '8BUI', 1, 0
        ),
        1, 1, 1,
        ARRAY[
            [0, 1, 1, 1, 1],
            [1, 1, 1, 0, 1],
            [1, 0, 1, 1, 1],
            [1, 1, 1, 1, 0],
            [1, 1, 0, 1, 1]
        ]::double precision[][]
    ) AS rast
)
SELECT
    ST_Value(rast, 2, 2) AS value,
    ST_NearestValue(rast, 2, 2) AS nearestvalue
FROM source
```

Output

```
value | nearestvalue
-------+--------------
    1 |            1
```

This example reads a NODATA value from pixel 2x3.

Code

```
WITH source AS (
    SELECT ST_SetValues(
        ST_AddBand(
            ST_MakeEmptyRaster(5, 5, -2, 2, 1, -1, 0, 0, 0),
            1, '8BUI', 1, 0
        ),
        1, 1, 1,
        ARRAY[
            [0, 1, 1, 1, 1],
            [1, 1, 1, 0, 1],
            [1, 0, 1, 1, 1],
            [1, 1, 1, 1, 0],
            [1, 1, 0, 1, 1]
        ]::double precision[][]
    ) AS rast
)
SELECT
    ST_Value(rast, 2, 3) AS value,
    ST_NearestValue(rast, 2, 3) AS nearestvalue
FROM source
```

Output

```
value | nearestvalue
------+--------------
      |            1
```

□□

ST_Neighborhood, ST_Value

11.6.9 ST_SetZ

ST_SetZ — Returns a geometry with the same X/Y coordinates as the input geometry, and values from the raster copied into the Z dimension using the requested resample algorithm.

□□

geometry **ST_SetZ**(raster rast, geometry geom, text resample=nearest, integer band=1);

Description

Returns a geometry with the same X/Y coordinates as the input geometry, and values from the raster copied into the Z dimensions using the requested resample algorithm.

The `resample` parameter can be set to ”nearest” to copy the values from the cell each vertex falls within, or ”bilinear” to use bilinear interpolation to calculate a value that takes neighboring cells into account also.

Availability: 3.2.0

□□

2x2 test raster with values. 10 50. 40 20.

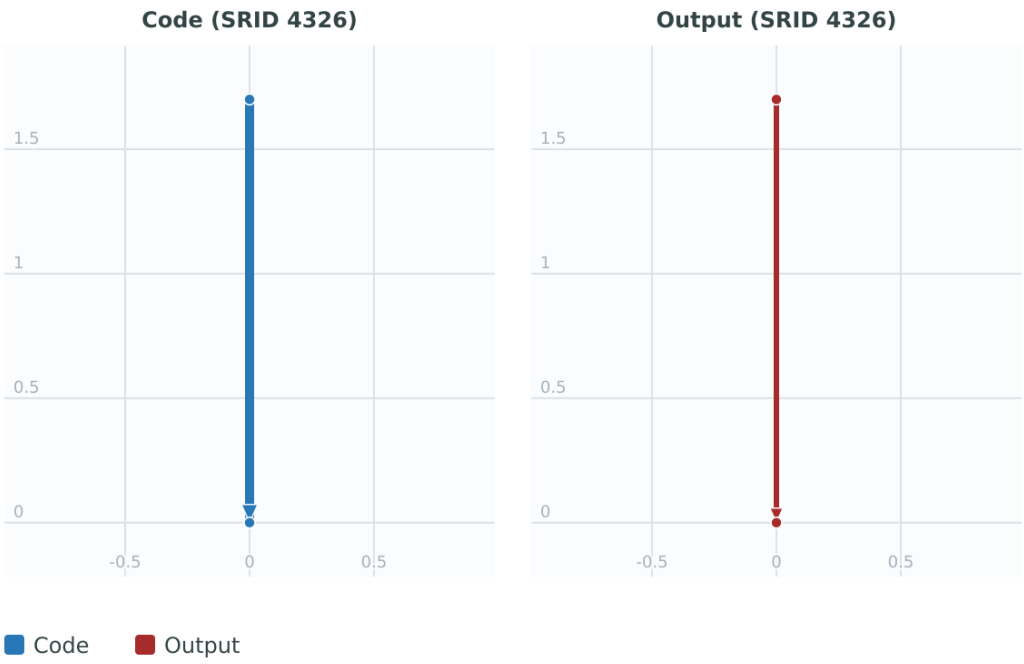
Code

```
WITH test_raster AS (
SELECT
ST_SetValues(
ST_AddBand(
ST_MakeEmptyRaster(
      width => 2,
      height => 2,
      upperleftx => 0,
      upperlefty => 2,
      scalex => 1.0,
      scaley => -1.0,
      skewx => 0,
      skewy => 0,
      srid => 4326),
    index => 1,
    pixeltype => '16BSI',
    initialvalue => 0,
    nodataval => -999
),
  1,
  1,
  1,
  newvalueset => ARRAY[ARRAY[10.0::float8, 50.0::float8], ARRAY[40.0::float8, 20.0::float8 ↩
    ]]) AS rast
)
SELECT
ST_SetZ(
    rast,
    band => 1,
    geom => 'SRID=4326;LINESTRING(1.0 1.9,1.0 0.2)'::geometry,
    resample => 'bilinear'
)
FROM test_raster
```

Output

```
LINESTRING Z (1 1.9 38,1 0.2 27)
```

Figure



□□

ST_Value, ST_SetSRID

11.6.10 ST_SetM

ST_SetM — Returns a geometry with the same X/Y coordinates as the input geometry, and values from the raster copied into the M dimension using the requested resample algorithm.

□□

geometry **ST_SetM**(raster rast, geometry geom, text resample=nearest, integer band=1);

Description

Returns a geometry with the same X/Y coordinates as the input geometry, and values from the raster copied into the M dimensions using the requested resample algorithm.

The `resample` parameter can be set to ”nearest” to copy the values from the cell each vertex falls within, or ”bilinear” to use bilinear interpolation to calculate a value that takes neighboring cells into account also.

Availability: 3.2.0

□□

2x2 test raster with values. 10 50. 40 20.

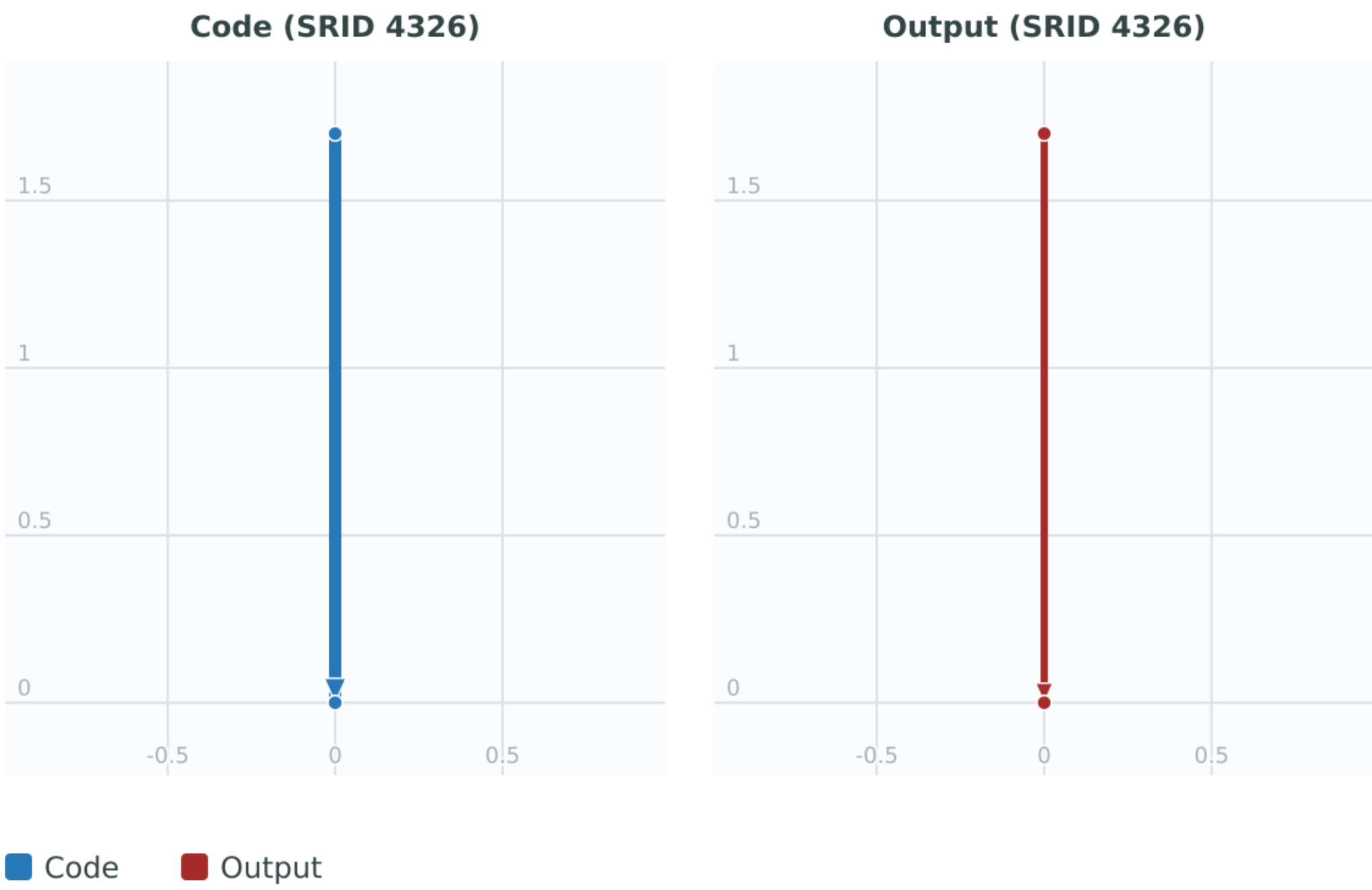
Code

```
WITH test_raster AS (
SELECT
ST_SetValues(
ST_AddBand(
ST_MakeEmptyRaster(
      width => 2,
      height => 2,
      upperleftx => 0,
      upperlefty => 2,
      scalex => 1.0,
      scaley => -1.0,
      skewx => 0,
      skewy => 0,
      srid => 4326),
    index => 1,
    pixeltype => '16BSI',
    initialvalue => 0,
    nodataval => -999
),
  1,
  1,
  1,
  newvalueset => ARRAY[ARRAY[10.0::float8, 50.0::float8], ARRAY[40.0::float8, 20.0::float8 ↩
    ]]) AS rast
)
SELECT
ST_SetM(
    rast,
    band => 1,
    geom => 'SRID=4326;LINESTRING(1.0 1.9,1.0 0.2)'::geometry,
    resample => 'bilinear'
)
FROM test_raster
```

Output

```
LINESTRING M (1 1.9 38,1 0.2 27)
```

Figure



☒☒

ST_Value, ST_SetSRID

11.6.11 ST_Neighborhood

ST_Neighborhood — columnx ☒ rowy, ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒ ☒☒☒☒☒☒☒☒☒☒☒☒☒☒ NODATA ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒ 2 ☒☒☒☒☒☒☒☒☒☒.

☒☒

double precision[][] **ST_Neighborhood**(raster rast, integer bandnum, integer columnX, integer rowY, integer distanceX, integer distanceY, boolean exclude_nodata_value=true);
double precision[][] **ST_Neighborhood**(raster rast, integer columnX, integer rowY, integer distanceX, integer distanceY, boolean exclude_nodata_value=true);
double precision[][] **ST_Neighborhood**(raster rast, integer bandnum, geometry pt, integer distanceX, integer distanceY, boolean exclude_nodata_value=true);
double precision[][] **ST_Neighborhood**(raster rast, geometry pt, integer distanceX, integer distanceY, boolean exclude_nodata_value=true);

Description

columnx ☒ rowy, ☒☒ ☒☒☒ NODATA ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒ 2 ☒☒☒☒☒☒☒☒☒☒. `distanceX` ☒ `distanceY` ☒☒☒☒ ☒☒☒☒☒☒☒☒☒☒☒☒☒☒ X ☒ Y ☒☒☒☒☒☒☒☒. ☒☒☒☒☒, ☒☒☒☒☒☒☒☒☒☒ X ☒☒☒☒ 3 ☒☒☒ ☒☒☒☒☒☒ Y ☒☒☒☒ 2 ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒. 2 ☒☒☒☒☒☒☒☒☒ columnx ☒ rowy ☒ ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.

⊠⊠⊠⊠⊠ 1 ⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ bandnum ⊠ 1 ⊠⊠⊠⊠⊠⊠. exclude_nodata_value ⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠, nodata ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. exclude_nodata_value ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

> ⊠⊠
> ⊠⊠⊠⊠ 2 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ 2 * (distanceX|distanceY) + 1 ⊠⊠⊠. ⊠⊠⊠ distanceX ⊠ distanceY ⊠⊠⊠ 1 ⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠ 3x3 ⊠⊠⊠⊠⊠⊠.

> ⊠⊠
> ST_Min4ma, ST_Sum4ma, ST_Mean4ma ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ 2 ⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

2.1.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

This example reads a value from pixel 2x2.

Code

```
SELECT
    ST_Neighborhood(rast, 2, 2, 1, 1)
FROM (
    SELECT
        ST_SetValues(ST_AddBand(ST_MakeEmptyRaster(5, 5, -2, 2, 1, -1, 0, 0, 0),
                '8BUI'::text, 1, 0
            ),
            1, 1, 1, ARRAY[
                [0, 1, 1, 1, 1],
                [1, 1, 1, 0, 1],
                [1, 0, 1, 1, 1],
                [1, 1, 1, 1, 0],
                [1, 1, 0, 1, 1]
            ]::double precision[],
            1
        ) AS rast
) AS foo
```

Output

```
{{NULL,1,1},{1,1,1},{1,NULL,1}}
```

This example reads a NODATA value from pixel 2x3.

Code

```
SELECT
    ST_Neighborhood(rast, 2, 3, 1, 1)
FROM (
    SELECT
        ST_SetValues(ST_AddBand(ST_MakeEmptyRaster(5, 5, -2, 2, 1, -1, 0, 0, 0),
                '8BUI'::text, 1, 0
            ),
            1, 1, 1, ARRAY[
                [0, 1, 1, 1, 1],
```

```
                [1, 1, 1, 0, 1],
                [1, 0, 1, 1, 1],
                [1, 1, 1, 1, 0],
                [1, 1, 0, 1, 1]
            ]::double precision[],
            1
        ) AS rast
) AS foo
```

Output

```
{{1,1,1},{1,NULL,1},{1,1,1}}
```

Pixel 3x3 has value. Exclude_nodata_value = FALSE.

Code

```
SELECT
    ST_Neighborhood(rast, 3, 3, 1, 1, false)
FROM ST_SetValues(ST_AddBand(ST_MakeEmptyRaster(5, 5, -2, 2, 1, -1, 0, 0, 0),
                '8BUI'::text, 1, 0
            ),
            1, 1, 1, ARRAY[
                [0, 1, 1, 1, 1],
                [1, 1, 1, 0, 1],
                [1, 0, 1, 1, 1],
                [1, 1, 1, 1, 0],
                [1, 1, 0, 1, 1]
            ]::double precision[],
            1
        ) AS rast
```

Output

```
{{1,1,0},{0,1,1},{1,1,1}}
```

☒☒

ST_NearestValue, ST_Min4ma, ST_Max4ma, ST_Sum4ma, ST_Mean4ma, ST_Range4ma, ST_Distinct4ma, ST_StdDev4ma

11.6.12 ST_SetValue

ST_SetValue — ☒☒☒ columnx, rowy ☒☒☒☒☒☒

Description

Returns modified raster resulting from setting the specified pixels' values to new value for the designated band given the raster's row and column or a geometry. If no band is specified, then band 1 is assumed.

□□□□: 2.1.0 □□□□ ST_SetValue() □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□. □□ □□□□ ST_SetValues() □ geomval[] □□□□□□□□□ (wrapper) □□□□.

□□

This example uses geometry input.

Code

```
               SELECT (foo.geomval).val, ST_Union((foo.geomval).geom)
FROM (SELECT ST_DumpAsPolygons(ST_SetValue(
            rast,
            1,
            ST_Point(3427927.75, 5793243.95),
            50)
            ) As geomval
FROM dummy_rast
where rid = 2) As foo
WHERE (foo.geomval).val < 250
GROUP BY (foo.geomval).val;
```

Output

```
 val |                                                        st_astext
-----+---------------------------------------------------------------------
  50 | POLYGON((3427927.75 5793244,3427927.75 5793243.95,3427927.8 579324 ...
 249 | POLYGON((3427927.95 5793243.95,3427927.95 5793243.85,3427928 57932 ...
```

This example stores the changed raster.

Code

```
UPDATE dummy_rast SET rast = ST_SetValue(rast, 1, ST_Point(3427927.75, 5793243.95), 100)
    WHERE rid = 2   ;
```

□□

ST_Value, ST_DumpAsPolygons

11.6.13 ST_SetValues

ST_SetValues — □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□.

⌧⌧

raster **ST_SetValues**(raster rast, integer nband, integer columnx, integer rowy, double precision[][] newvalueset, boolean[][] noset=NULL, boolean keepnodata=FALSE);
raster **ST_SetValues**(raster rast, integer nband, integer columnx, integer rowy, double precision[][] newvalueset, double precision nosetvalue, boolean keepnodata=FALSE);
raster **ST_SetValues**(raster rast, integer nband, integer columnx, integer rowy, integer width, integer height, double precision newvalue, boolean keepnodata=FALSE);
raster **ST_SetValues**(raster rast, integer columnx, integer rowy, integer width, integer height, double precision newvalue, boolean keepnodata=FALSE);
raster **ST_SetValues**(raster rast, integer nband, geomval[] geomvalset, boolean keepnodata=FALSE);

Description

⌧⌧⌧⌧⌧⌧⌧⌧, ⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧ (⌧) ⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧.

`keepnodata` ⌧ TRUE ⌧⌧⌧, NODATA ⌧⌧⌧⌧⌧⌧ `newvalueset` ⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧.

⌧⌧ 1 ⌧⌧⌧, `columnx`, `rowy` ⌧⌧⌧⌧⌧ `newvalueset` ⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧. ⌧⌧ `newvalueset` ⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧ `noset` ⌧⌧⌧⌧⌧⌧⌧⌧⌧ (PostgreSQL ⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧). ⌧⌧ 1 ⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧.

⌧⌧ 2 ⌧⌧⌧ 1 ⌧⌧⌧⌧⌧⌧, ⌧⌧⌧ `noset` ⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧ `nosetvalue` ⌧⌧⌧⌧⌧⌧⌧. `newvalueset` ⌧⌧⌧ `nosetvalue` ⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧. ⌧⌧ 2 ⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧.

⌧⌧ 3 ⌧⌧⌧, `columnx`, `rowy` ⌧⌧⌧⌧, `width` ⌧ `height` ⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧. ⌧⌧ 3 ⌧⌧⌧⌧⌧⌧⌧⌧⌧.

⌧⌧ 4 ⌧ `rast` ⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧ 3 ⌧⌧⌧⌧⌧⌧.

⌧⌧ 5 ⌧⌧⌧, ⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧ geomval ⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧. ⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧ POINT ⌧⌧ MULTIPOINT ⌧⌧⌧, ⌧⌧. ⌧⌧⌧⌧⌧⌧⌧, ⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧. ⌧⌧ 5 ⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧.

2.1.0 ⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧.

⌧⌧

Variant 1.

These examples pass `FALSE` for *exclude_nodata_value* so that every raster cell remains visible. NODATA cells display the band’s NODATA value, which is `0` here.

Set the lower-right 2 by 2 block of a 3 by 3 raster to 9.

Code

```
SELECT
    (poly).x,
    (poly).y,
    (poly).val
FROM (
SELECT
    ST_PixelAsPolygons(ST_SetValues(ST_AddBand(ST_MakeEmptyRaster(3, 3, 0, 0, 1, -1, 0, 0, ↩
    0),
                1, '8BUI', 1, 0
            ),
            1, 2, 2, ARRAY[[9, 9], [9, 9]]::double precision[][]
        ),
        1, FALSE
```

```
    ) AS poly
) foo
ORDER BY 1, 2;
```

Output

```
 x | y | val
---+---+-----
 1 | 1 |   1
 1 | 2 |   1
 1 | 3 |   1
 2 | 1 |   1
 2 | 2 |   9
 2 | 3 |   9
 3 | 1 |   1
 3 | 2 |   9
 3 | 3 |   9
```

Replace the complete raster from an array; the central `NULL` becomes NODATA.

Code

```
SELECT
    (poly).x,
    (poly).y,
    (poly).val
FROM (
SELECT
    ST_PixelAsPolygons(ST_SetValues(ST_AddBand(ST_MakeEmptyRaster(3, 3, 0, 0, 1, -1, 0, 0,  ↩
    0),
                1, '8BUI', 1, 0
            ),
            1, 1, 1, ARRAY[[9, 9, 9], [9, NULL, 9], [9, 9, 9]]::double precision[][]
        ),
        1, FALSE
    ) AS poly
) foo
ORDER BY 1, 2;
```

Output

```
 x | y | val
---+---+-----
 1 | 1 |   9
 1 | 2 |   9
 1 | 3 |   9
 2 | 1 |   9
 2 | 2 |   0
 2 | 3 |   9
 3 | 1 |   9
 3 | 2 |   9
 3 | 3 |   9
```

Use the *noset* mask to retain the left-middle source pixel while replacing the other cells.

Code

```
SELECT
    (poly).x,
    (poly).y,
    (poly).val
FROM (
SELECT
    ST_PixelAsPolygons(ST_SetValues(ST_AddBand(ST_MakeEmptyRaster(3, 3, 0, 0, 1, -1, 0, 0,  ↩
    0),
                1, '8BUI', 1, 0
            ),
```

```
            1, 1, 1,
                ARRAY[[9, 9, 9], [9, NULL, 9], [9, 9, 9]]::double precision[][],
                ARRAY[[false], [true]]::boolean[][]
        ),
        1, FALSE
    ) AS poly
) foo
ORDER BY 1, 2;
```

Output

```
x | y | val
---+---+-----
 1 | 1 |   9
 1 | 2 |   1
 1 | 3 |   9
 2 | 1 |   9
 2 | 2 |   0
 2 | 3 |   9
 3 | 1 |   9
 3 | 2 |   9
 3 | 3 |   9
```

With *keepnodata* enabled, an existing NODATA pixel is not overwritten.

Code

```
SELECT
    (poly).x,
    (poly).y,
    (poly).val
FROM (
SELECT
    ST_PixelAsPolygons(ST_SetValues(ST_SetValue(ST_AddBand(ST_MakeEmptyRaster(3, 3, 0, 0,  ↩
    1, -1, 0, 0, 0),
                    1, '8BUI', 1, 0
                ),
                1, 1, 1, NULL
            ),
            1, 1, 1,
                ARRAY[[9, 9, 9], [9, NULL, 9], [9, 9, 9]]::double precision[][],
                ARRAY[[false], [true]]::boolean[][],
                TRUE
        ),
        1, FALSE
    ) AS poly
) foo
ORDER BY 1, 2;
```

Output

```
x | y | val
---+---+-----
 1 | 1 |   0
 1 | 2 |   1
 1 | 3 |   9
 2 | 1 |   9
 2 | 2 |   0
 2 | 3 |   9
 3 | 1 |   9
 3 | 2 |   9
 3 | 3 |   9
```

Variant 2.

Use -1 as the value that should not be written; only the lower-right 2 by 2 block changes.

Code

```
SELECT
    (poly).x,
    (poly).y,
    (poly).val
FROM (
SELECT
    ST_PixelAsPolygons(ST_SetValues(ST_AddBand(ST_MakeEmptyRaster(3, 3, 0, 0, 1, -1, 0, 0,  ↩
    0),
                1, '8BUI', 1, 0
            ),
            1, 1, 1, ARRAY[[-1, -1, -1], [-1, 9, 9], [-1, 9, 9]]::double precision[][], -1
        ),
        1, FALSE
    ) AS poly
) foo
ORDER BY 1, 2;
```

Output

```
x | y | val
---+---+-----
 1 | 1 |   1
 1 | 2 |   1
 1 | 3 |   1
 2 | 1 |   1
 2 | 2 |   9
 2 | 3 |   9
 3 | 1 |   1
 3 | 2 |   9
 3 | 3 |   9
```

The same update can use NULL as the value that should not be written.

Code

```
SELECT
    (poly).x,
    (poly).y,
    (poly).val
FROM (
SELECT
    ST_PixelAsPolygons(ST_SetValues(ST_AddBand(ST_MakeEmptyRaster(3, 3, 0, 0, 1, -1, 0, 0,  ↩
    0),
                1, '8BUI', 1, 0
            ),
            1, 1, 1, ARRAY[[NULL, NULL, NULL], [NULL, 9, 9], [NULL, 9, 9]]::double  ↩
    precision[][], NULL::double precision
        ),
        1, FALSE
    ) AS poly
) foo
ORDER BY 1, 2;
```

Output

```
x | y | val
---+---+-----
 1 | 1 |   1
 1 | 2 |   1
 1 | 3 |   1
 2 | 1 |   1
 2 | 2 |   9
 2 | 3 |   9
 3 | 1 |   1
 3 | 2 |   9
 3 | 3 |   9
```

Variant 3.

Set a 2 by 2 rectangle starting at column 2, row 2.

Code

```
SELECT
    (poly).x,
    (poly).y,
    (poly).val
FROM (
SELECT
    ST_PixelAsPolygons(ST_SetValues(ST_AddBand(ST_MakeEmptyRaster(3, 3, 0, 0, 1, -1, 0, 0,  ↩
    0),
                1, '8BUI', 1, 0
            ),
            1, 2, 2, 2, 2, 9
        ),
        1, FALSE
    ) AS poly
) foo
ORDER BY 1, 2;
```

Output

```
x | y | val
---+---+-----
 1 | 1 |   1
 1 | 2 |   1
 1 | 3 |   1
 2 | 1 |   1
 2 | 2 |   9
 2 | 3 |   9
 3 | 1 |   1
 3 | 2 |   9
 3 | 3 |   9
```

When *keepnodata* is true, the NODATA cell inside the rectangle remains unchanged.

Code

```
SELECT
    (poly).x,
    (poly).y,
    (poly).val
FROM (
SELECT
    ST_PixelAsPolygons(ST_SetValues(ST_SetValue(ST_AddBand(ST_MakeEmptyRaster(3, 3, 0, 0,  ↩
    1, -1, 0, 0, 0),
                    1, '8BUI', 1, 0
                ),
                1, 2, 2, NULL
            ),
            1, 2, 2, 2, 2, 9, TRUE
        ),
        1, FALSE
    ) AS poly
) foo
ORDER BY 1, 2;
```

Output

```
x | y | val
---+---+-----
 1 | 1 |   1
 1 | 2 |   1
 1 | 3 |   1
```

```
 2 | 1 |   1
 2 | 2 |   0
 2 | 3 |   9
 3 | 1 |   1
 3 | 2 |   9
 3 | 3 |   9
```

Variant 5.

Code

```
WITH foo AS (
    SELECT 1 AS rid, ST_AddBand(ST_MakeEmptyRaster(5, 5, 0, 0, 1, -1, 0, 0, 0), 1, '8BUI',  ↩
    0, 0) AS rast
), bar AS (
    SELECT 1 AS gid, 'SRID=0;POINT(2.5 -2.5)'::geometry geom UNION ALL
    SELECT 2 AS gid, 'SRID=0;POLYGON((1 -1,4 -1,4 -4,1 -4,1 -1))'::geometry geom UNION ALL
    SELECT 3 AS gid, 'SRID=0;POLYGON((0 0,5 0,5 -1,1 -1,1 -4,0 -4,0 0))'::geometry geom  ↩
    UNION ALL
    SELECT 4 AS gid, 'SRID=0;MULTIPOINT(0 0,4 4,4 -4)'::geometry
), changed AS (
    SELECT rid, gid, ST_SetValue(rast, 1, geom, gid) AS rast
    FROM foo
    CROSS JOIN bar
)
SELECT
    rid,
    gid,
    y,
    array_agg(ST_Value(rast, 1, x, y) ORDER BY x) AS values
FROM changed
CROSS JOIN generate_series(1, 5) AS x
CROSS JOIN generate_series(1, 5) AS y
GROUP BY rid, gid, y
ORDER BY rid, gid, y;
```

Output

```
rid | gid | y |           values
-----+-----+---+---------------------------
   1 |   1 | 1 | {NULL,NULL,NULL,NULL,NULL}
   1 |   1 | 2 | {NULL,NULL,NULL,NULL,NULL}
   1 |   1 | 3 | {NULL,NULL,1,NULL,NULL}
   1 |   1 | 4 | {NULL,NULL,NULL,NULL,NULL}
   1 |   1 | 5 | {NULL,NULL,NULL,NULL,NULL}
   1 |   2 | 1 | {NULL,NULL,NULL,NULL,NULL}
   1 |   2 | 2 | {NULL,2,2,2,NULL}
   1 |   2 | 3 | {NULL,2,2,2,NULL}
   1 |   2 | 4 | {NULL,2,2,2,NULL}
   1 |   2 | 5 | {NULL,NULL,NULL,NULL,NULL}
   1 |   3 | 1 | {3,3,3,3,3}
   1 |   3 | 2 | {3,NULL,NULL,NULL,NULL}
   1 |   3 | 3 | {3,NULL,NULL,NULL,NULL}
   1 |   3 | 4 | {3,NULL,NULL,NULL,NULL}
   1 |   3 | 5 | {NULL,NULL,NULL,NULL,NULL}
   1 |   4 | 1 | {4,NULL,NULL,NULL,NULL}
   1 |   4 | 2 | {NULL,NULL,NULL,NULL,NULL}
   1 |   4 | 3 | {NULL,NULL,NULL,NULL,NULL}
   1 |   4 | 4 | {NULL,NULL,NULL,NULL,NULL}
   1 |   4 | 5 | {NULL,NULL,NULL,NULL,4}
(20 rows)
```

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ geomvals ⊠⊠⊠⊠ geomvals ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

Code

```
WITH foo AS (
    SELECT 1 AS rid, ST_AddBand(ST_MakeEmptyRaster(5, 5, 0, 0, 1, -1, 0, 0, 0), 1, '8BUI',  ↩
    0, 0) AS rast
), bar AS (
```

```
    SELECT 1 AS gid, 'SRID=0;POINT(2.5 -2.5)'::geometry geom UNION ALL
    SELECT 2 AS gid, 'SRID=0;POLYGON((1 -1,4 -1,4 -4,1 -4,1 -1))'::geometry geom UNION ALL
    SELECT 3 AS gid, 'SRID=0;POLYGON((0 0,5 0,5 -1,1 -1,1 -4,0 -4,0 0))'::geometry geom
    UNION ALL
    SELECT 4 AS gid, 'SRID=0;MULTIPOINT(0 0,4 4,4 -4)'::geometry
), changed AS (
    SELECT ST_SetValues(
        rast,
        1,
        ARRAY[
            ROW((SELECT geom FROM bar WHERE gid = 1), 1),
            ROW((SELECT geom FROM bar WHERE gid = 2), 2)
        ]::geomval[]
    ) AS rast
    FROM foo
)
SELECT
    y,
    array_agg(ST_Value(rast, 1, x, y) ORDER BY x) AS values
FROM changed
CROSS JOIN generate_series(1, 5) AS x
CROSS JOIN generate_series(1, 5) AS y
GROUP BY y
ORDER BY y;
```

Output

```
y |          values
---+--------------------------
 1 | {NULL,NULL,NULL,NULL,NULL}
 2 | {NULL,2,2,2,NULL}
 3 | {NULL,2,2,2,NULL}
 4 | {NULL,2,2,2,NULL}
 5 | {NULL,NULL,NULL,NULL,NULL}
(5 rows)
```

☒☒☒☒☒☒

```
CROSS JOIN generate_series(1, 5) AS x
CROSS JOIN generate_series(1, 5) AS y
GROUP BY y
ORDER BY y;
```

Output

```
y |           values
---+---------------------------
 1 | {NULL,NULL,NULL,NULL,NULL}
 2 | {NULL,2,2,2,NULL}
 3 | {NULL,2,1,2,NULL}
 4 | {NULL,2,2,2,NULL}
 5 | {NULL,NULL,NULL,NULL,NULL}
(5 rows)
```

□□

ST_Value, ST_SetValue, ST_PixelAsPolygons

11.6.14 ST_DumpValues

ST_DumpValues — □□□□□□□□□□ 2 □□□□□□□□□□□.

□□

table(nband integer, valarray double precision[][]) **ST_DumpValues**(raster rast , integer[] nband=NULL , boolean exclude_nodata_value=true);
double precision[][] **ST_DumpValues**(raster rast , integer nband , boolean exclude_nodata_value=true);

Description

□□□□□□□□□□ 2 □□□□□□□□□□□ (□□□□□□□□□, □□□□□□□□□□□□). nband □ NULL □ □□□□□□□□□□□, □□□□□□□□□□□□□□□□.

2.1.0 □□□□□□□□□□□□.

□□

Code

```
WITH foo AS (
    SELECT ST_AddBand(ST_AddBand(ST_AddBand(ST_MakeEmptyRaster(3, 3, 0, 0, 1, -1, 0, 0, 0), ↩
     1, '8BUI'::text, 1, 0), 2, '32BF'::text, 3, -9999), 3, '16BSI', 0, 0) AS rast
)
SELECT
    (ST_DumpValues(rast)).*
FROM foo;
```

Output

```
nband |                       valarray
-------+-----------------------------------------------------
     1 | {{1,1,1},{1,1,1},{1,1,1}}
     2 | {{3,3,3},{3,3,3},{3,3,3}}
     3 | {{NULL,NULL,NULL},{NULL,NULL,NULL},{NULL,NULL,NULL}}
(3 rows)
```

Code

```
WITH foo AS (
    SELECT ST_AddBand(ST_AddBand(ST_AddBand(ST_MakeEmptyRaster(3, 3, 0, 0, 1, -1, 0, 0, 0), ↩
     1, '8BUI'::text, 1, 0), 2, '32BF'::text, 3, -9999), 3, '16BSI', 0, 0) AS rast
)
SELECT
    (ST_DumpValues(rast, ARRAY[3, 1])).*
FROM foo;
```

Output

```
nband |                       valarray
-------+-----------------------------------------------------
     3 | {{NULL,NULL,NULL},{NULL,NULL,NULL},{NULL,NULL,NULL}}
     1 | {{1,1,1},{1,1,1},{1,1,1}}
(2 rows)
```

Code

```
WITH foo AS (
    SELECT ST_SetValue(ST_AddBand(ST_MakeEmptyRaster(3, 3, 0, 0, 1, -1, 0, 0, 0), 1, '8BUI' ↩
    , 1, 0), 1, 2, 5) AS rast
)
SELECT
    (ST_DumpValues(rast, 1))[2][1]
FROM foo;
```

Output

```
5
```

☒☒

ST_Value, ST_SetValue, ST_SetValues

11.6.15 ST_PixelOfValue

ST_PixelOfValue — ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒ columnx, rowy ☒☒☒☒☒☒☒☒.

☒☒

table(val double precision, x integer, y integer) **ST_PixelOfValue**(raster rast , integer nband , double precision[] search , boolean exclude_nodata_value=true);
table(val double precision, x integer, y integer) **ST_PixelOfValue**(raster rast , double precision[] search , boolean exclude_nodata_value=true);
table(x integer, y integer) **ST_PixelOfValue**(raster rast , integer nband , double precision search , boolean exclude_nodata_value=true);
table(x integer, y integer) **ST_PixelOfValue**(raster rast , double precision search , boolean exclude_nodata_v
);

Description

□□□□□□□□□□□□□□□ columnx, rowy □□□□□□□□. □□□□□□□□□□□□□, □□ 1 □□□□□□.

2.1.0 □□□□□□□□□□□□.

□□

Code

```
WITH source AS (
    SELECT ST_SetValues(
        ST_AddBand(
            ST_MakeEmptyRaster(5, 5, -2, 2, 1, -1, 0, 0, 0),
            1, '8BUI', 1, 0
        ),
        1, 1, 1,
        ARRAY[
            [0, 1, 1, 1, 1],
            [1, 1, 1, 0, 1],
            [1, 0, 1, 1, 1],
            [1, 1, 1, 1, 255],
            [1, 1, 0, 1, 1]
        ]::double precision[][]
    ) AS rast
)
SELECT (ST_PixelOfValue(rast, 1, ARRAY[1, 255])).*
FROM source
```

Output

```
val | x | y
-----+---+---
   1 | 1 | 2
   1 | 1 | 3
   1 | 1 | 4
   1 | 1 | 5
   1 | 2 | 1
   1 | 2 | 2
   1 | 2 | 4
   1 | 2 | 5
   1 | 3 | 1
   1 | 3 | 2
   1 | 3 | 3
   1 | 3 | 4
   1 | 4 | 1
   1 | 4 | 3
   1 | 4 | 4
   1 | 4 | 5
   1 | 5 | 1
   1 | 5 | 2
   1 | 5 | 3
 255 | 5 | 4
   1 | 5 | 5
```

11.7 □□□□□□

11.7.1 ST_SetGeoReference

ST_SetGeoReference — □□□□□□□□□□□□□□ 6 □□□□□□□. □□□□□□□□□□□□□□□. GDAL □□ ESRI □□□□□□□□□□□□□□. □□□□ GDAL □□□.

⊠⊠

raster **ST_SetGeoReference**(raster rast, text georefcoords, text format=GDAL);
raster **ST_SetGeoReference**(raster rast, double precision upperleftx, double precision upperlefty, double precision scalex, double precision scaley, double precision skewx, double precision skewy);

Description

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ 6 ⊠⊠⊠⊠⊠⊠⊠. ’GDAL’ ⊠⊠’ESRI’ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠ GDAL ⊠⊠⊠. 6 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ NULL ⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠:

GDAL uses the value order shown below.

Code

```
scalex skewy skewx scaley upperleftx upperlefty
```

ESRI uses the value order shown below.

Code

```
scalex skewy skewx scaley upperleftx + scalex*0.5 upperlefty + scaley*0.5
```

> **⊠⊠**
> ⊠⊠⊠⊠ DB ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠⊠⊠: 2.1.0 ⊠⊠⊠⊠ ST_SetGeoReference(raster, double precision, ...) ⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

Code

```
WITH source AS (
    SELECT ST_MakeEmptyRaster(5, 5, 0, 0, 1, -1, 0, 0, 0) AS rast
), variants AS (
    SELECT 0 AS rid, rast FROM source
    UNION ALL
    SELECT 1, ST_SetGeoReference(rast, '10 0 0 -10 0.1 0.1', 'GDAL')
    FROM source
    UNION ALL
    SELECT 2, ST_SetGeoReference(rast, '10 0 0 -10 5.1 -4.9', 'ESRI')
    FROM source
    UNION ALL
    SELECT 3, ST_SetGeoReference(rast, 1, 1, 10, -10, 0.001, 0.001)
    FROM source
)
SELECT
    rid,
    round(ST_UpperLeftX(rast)::numeric, 3) AS upperleftx,
    round(ST_UpperLeftY(rast)::numeric, 3) AS upperlefty,
    ST_ScaleX(rast) AS scalex,
    ST_ScaleY(rast) AS scaley,
    ST_SkewX(rast) AS skewx,
```

```
    ST_SkewY(rast) AS skewy
FROM variants
ORDER BY rid;
```

Output

```
rid | upperleftx | upperlefty | scalex | scaley | skewx | skewy
----+------------+------------+--------+--------+-------+-------
  0 |      0.000 |      0.000 |      1 |     -1 |     0 |     0
  1 |      0.100 |      0.100 |     10 |    -10 |     0 |     0
  2 |      0.100 |      0.100 |     10 |    -10 |     0 |     0
  3 |      1.000 |      1.000 |     10 |    -10 | 0.001 | 0.001
(4 rows)
```

□□

ST_GeoReference, ST_ScaleX, ST_ScaleY, ST_UpperLeftX, ST_UpperLeftY

11.7.2 ST_SetRotation

ST_SetRotation — □□□□□□□□□□□□□□□□□□□□□.

□□

raster **ST_SetRotation**(raster rast, float8 rotation);

Description

□□□□□□□□□□□□□□□. □□□□□□□□□□□□□. □□□□□□□ □□□□ □□□□□□□□.

□□

Code

```
WITH variants AS (
    SELECT rid, 'rotated' AS kind, ST_SetRotation(rast, 15) AS rast
    FROM dummy_rast
    UNION ALL
    SELECT rid, 'original', rast
    FROM dummy_rast
)
SELECT
    rid,
    kind,
    round(ST_ScaleX(rast)::numeric, 4) AS scale_x,
    round(ST_ScaleY(rast)::numeric, 4) AS scale_y,
    round(ST_SkewX(rast)::numeric, 4) AS skew_x,
    round(ST_SkewY(rast)::numeric, 4) AS skew_y
FROM variants
ORDER BY rid, kind;
```

Output

```
rid |   kind   | scale_x | scale_y | skew_x | skew_y
-----+----------+---------+---------+--------+--------
   1 | original |  2.0000 |  3.0000 | 0.0000 | 0.0000
   1 | rotated  | -1.5194 | -2.2791 | 1.9509 | 1.3006
   2 | original |  0.0500 | -0.0500 | 0.0000 | 0.0000
   2 | rotated  | -0.0380 | -0.0380 | 0.0325 | 0.0325
(4 rows)
```

□□

ST_Rotation, ST_ScaleX, ST_ScaleY, ST_SkewX, ST_SkewY

11.7.3 ST_SetScale

ST_SetScale — X □ Y □□□□□□□□□□□□□□□□□□□□□. □□/□□□□/□□□□□□□□□.

□□

raster **ST_SetScale**(raster rast, float8 xy);
raster **ST_SetScale**(raster rast, float8 x, float8 y);

Description

X □ Y □□□□□□□□□□□□□□□□□□□□□. □□/□□□□/□□□□□□□□□□□□□□□. □□□ □□□□□□□□, X □ Y □□□□□□□□□□□□□.

> □□
> ST_SetScale □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ ST_Rescale □□□ □□□. □□□□□□□□□□□□□□□□□□□□□□□□□□□□ (□□□□□□□) □□□□□□□ □. ST_Rescale □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□. ST_SetScale □□□□□□□□□□□□□□□□□□□□□□□.

□□□□: 2.0.0 □□ WKTRaster □□□□□ ST_SetPixelSize □□□□□□□□□. 2.0.0 □□□□□□□□ □□□□□□□□.

□□

Code

```
UPDATE dummy_rast
    SET rast = ST_SetScale(rast, 1.5)
WHERE rid = 2;

SELECT ST_ScaleX(rast) As pixx, ST_ScaleY(rast) As pixy, Box3D(rast) As newbox
FROM dummy_rast
WHERE rid = 2;
```

Output

```
pixx | pixy |                    newbox
------+------+-----------------------------------------------
  1.5 |  1.5 | BOX(3427927.75 5793244 0, 3427935.25 5793251.5 0)
```

Code

```
UPDATE dummy_rast
    SET rast = ST_SetScale(rast, 1.5, 0.55)
WHERE rid = 2;

SELECT ST_ScaleX(rast) As pixx, ST_ScaleY(rast) As pixy, Box3D(rast) As newbox
FROM dummy_rast
WHERE rid = 2;
```

Output

```
pixx | pixy |                   newbox
------+------+-----------------------------------------------
  1.5 | 0.55 | BOX(3427927.75 5793244 0,3427935.25 5793247 0)
```

□□

ST_ScaleX, ST_ScaleY, Box3D

11.7.4 ST_SetSkew

ST_SetSkew — □□□□ X □ Y □□□ (skew)(□□□□□□□□□) □□□□□□. □□□□□□□□□, X □ Y □□□□□□□□□□□□□.

□□

raster **ST_SetSkew**(raster rast, float8 skewxy);
raster **ST_SetSkew**(raster rast, float8 skewx, float8 skewy);

Description

□□□□ X □ Y □□□ (□□□□□□□□□) □□□□□□. □□□□□□□□□, X □ Y □□□□□□□□□□□□□□. □□□□□□ □□□□ □□□□□□□□.

□□

This is the first example.

Code

```
UPDATE dummy_rast SET rast = ST_SetSkew(rast, 1, 2) WHERE rid = 1;
SELECT
    rid,
    format('(%s, %s)', skewx, skewy) AS skew,
    term,
    georef[idx] AS value
FROM (
    SELECT
```

```
        rid,
        ST_SkewX(rast) AS skewx,
        ST_SkewY(rast) AS skewy,
        string_to_array(ST_GeoReference(rast), E'\n') AS georef
    FROM dummy_rast
    WHERE rid = 1
) AS refs
CROSS JOIN (VALUES
    (1, 'scaleX'),
    (2, 'skewY'),
    (3, 'skewX'),
    (4, 'scaleY'),
    (5, 'originX'),
    (6, 'originY')
) AS parts(idx, term)
ORDER BY idx;
```

Output

```
rid |  skew  |  term   |    value
-----+--------+---------+-------------
   1 | (1, 2) | scaleX  | 2.0000000000
   1 | (1, 2) | skewY   | 2.0000000000
   1 | (1, 2) | skewX   | 1.0000000000
   1 | (1, 2) | scaleY  | 3.0000000000
   1 | (1, 2) | originX | 0.5000000000
   1 | (1, 2) | originY | 0.5000000000
(6 rows)
```

This second example sets both values to the same number.

Code

```
UPDATE dummy_rast SET rast = ST_SetSkew(rast, 0) WHERE rid = 1;
SELECT
    rid,
    format('(%s, %s)', skewx, skewy) AS skew,
    term,
    georef[idx] AS value
FROM (
    SELECT
        rid,
        ST_SkewX(rast) AS skewx,
        ST_SkewY(rast) AS skewy,
        string_to_array(ST_GeoReference(rast), E'\n') AS georef
    FROM dummy_rast
    WHERE rid = 1
) AS refs
CROSS JOIN (VALUES
    (1, 'scaleX'),
    (2, 'skewY'),
    (3, 'skewX'),
    (4, 'scaleY'),
    (5, 'originX'),
    (6, 'originY')
) AS parts(idx, term)
ORDER BY idx;
```

Output

```
rid |  skew  |  term   |    value
-----+--------+---------+-------------
   1 | (0, 0) | scaleX  | 2.0000000000
   1 | (0, 0) | skewY   | 0.0000000000
   1 | (0, 0) | skewX   | 0.0000000000
```

```
   1 | (0, 0) | scaleY  | 3.0000000000
   1 | (0, 0) | originX | 0.5000000000
   1 | (0, 0) | originY | 0.5000000000
(6 rows)
```

⊠⊠

ST_GeoReference, ST_SetGeoReference, ST_SkewX, ST_SkewY

11.7.5 ST_SetSRID

ST_SetSRID — ⊠⊠⊠⊠ SRID ⊠ spatial_ref_sys ⊠⊠⊠⊠⊠⊠⊠⊠⊠ SRID ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

raster **ST_SetSRID**(raster rast, integer srid);

Description

⊠⊠⊠⊠ SRID ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

> **Note!** ⊠⊠
> ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

4.5⊠, ST_SRID

11.7.6 ST_SetUpperLeft

ST_SetUpperLeft — Sets the value of the upper left corner of the pixel of the raster to projected X and Y coordinates.

⊠⊠

raster **ST_SetUpperLeft**(raster rast, double precision x, double precision y);

Description

Set the value of the upper left corner of raster to the projected X and Y coordinates

⊠⊠

Code

```
SELECT ST_SetUpperLeft(rast, -71.01, 42.37)
FROM dummy_rast
WHERE rid = 2;
```

􏰀􏰀

ST_UpperLeftX, ST_UpperLeftY

11.7.7 ST_Resample

ST_Resample — 􏰀􏰀􏰀􏰀􏰀􏰀􏰀􏰀􏰀􏰀, 􏰀􏰀􏰀􏰀􏰀, 􏰀􏰀􏰀􏰀􏰀􏰀􏰀􏰀􏰀, 􏰀􏰀􏰀􏰀􏰀􏰀􏰀􏰀􏰀􏰀􏰀􏰀􏰀􏰀􏰀􏰀􏰀 􏰀􏰀􏰀􏰀􏰀􏰀􏰀􏰀􏰀􏰀􏰀􏰀􏰀􏰀􏰀􏰀􏰀􏰀􏰀􏰀􏰀􏰀􏰀􏰀􏰀􏰀􏰀􏰀􏰀􏰀􏰀􏰀􏰀.

􏰀􏰀

raster **ST_Resample**(raster rast, integer width, integer height, double precision gridx=NULL, double precision gridy=NULL, double precision skewx=0, double precision skewy=0, text algorithm=NearestNeighbor, double precision maxerr=0.125);
raster **ST_Resample**(raster rast, double precision scalex=0, double precision scaley=0, double precision gridx=NULL, double precision gridy=NULL, double precision skewx=0, double precision skewy=0, text algorithm=NearestNeighbor, double precision maxerr=0.125);
raster **ST_Resample**(raster rast, raster ref, text algorithm=NearestNeighbor, double precision maxerr=0.125, boolean usescale=true);
raster **ST_Resample**(raster rast, raster ref, boolean usescale, text algorithm=NearestNeighbor, double precision maxerr=0.125);

Description

􏰀􏰀􏰀􏰀􏰀􏰀􏰀􏰀􏰀􏰀, 􏰀􏰀􏰀􏰀􏰀 (width & height), 􏰀􏰀􏰀􏰀􏰀􏰀 (gridx & gridy), 􏰀􏰀􏰀􏰀􏰀􏰀􏰀􏰀􏰀􏰀􏰀 􏰀􏰀􏰀􏰀􏰀􏰀􏰀􏰀􏰀􏰀􏰀􏰀􏰀􏰀􏰀􏰀􏰀􏰀 (scalex, scaley, skewx & skewy) 􏰀􏰀􏰀􏰀􏰀􏰀􏰀􏰀􏰀􏰀􏰀􏰀􏰀􏰀􏰀 􏰀􏰀􏰀. 􏰀􏰀􏰀􏰀􏰀􏰀􏰀􏰀􏰀􏰀􏰀, 􏰀􏰀􏰀􏰀􏰀􏰀􏰀􏰀 SRID 􏰀􏰀􏰀􏰀􏰀􏰀􏰀􏰀􏰀􏰀.

New pixel values are computed using one of the following resampling algorithms:

- NearestNeighbor (english or american spelling)
- Bilinear
- Cubic
- CubicSpline
- Lanczos
- Max
- Min

The default is NearestNeighbor which is the fastest but results in the worst interpolation.

`maxerr` 􏰀􏰀􏰀􏰀􏰀􏰀􏰀􏰀􏰀􏰀􏰀􏰀􏰀􏰀􏰀􏰀 0.125 􏰀􏰀􏰀􏰀􏰀􏰀.



□□
□□□□□□ GDAL Warp resampling methods □□□□□□□.

2.0.0 □□□□□□□□□□□□. GDAL 1.6.1 □□□□□□□□□□.

Enhanced: 3.4.0 max and min resampling options added

□□

Code

```
SELECT
    ST_Width(orig) AS orig_width,
    ST_Width(reduce_100) AS new_width
FROM (
    SELECT
        rast AS orig,
        ST_Resample(rast, 100, 100) AS reduce_100
    FROM aerials.boston
    WHERE ST_Intersects(
        rast,
        ST_Transform(ST_MakeEnvelope(-71.128, 42.2392, -71.1277, 42.2397, 4326),
            26986)
    )
    LIMIT 1
) AS foo;
```

Output

```
orig_width | new_width
-----------+-------------
        200 |         100
```

□□

ST_Rescale, ST_Resize, ST_Transform

11.7.8 ST_Rescale

ST_Rescale — Resample a raster by adjusting only its scale (or pixel size). New pixel values are computed using the NearestNeighbor (english or american spelling), Bilinear, Cubic, CubicSpline, Lanczos, Max or Min resampling algorithm. Default is NearestNeighbor.

□□

raster **ST_Rescale**(raster rast, double precision scalexy, text algorithm=NearestNeighbor, double precision maxerr=0.125);
raster **ST_Rescale**(raster rast, double precision scalex, double precision scaley, text algorithm=NearestNeig
double precision maxerr=0.125);

Description

Resample a raster by adjusting only its scale (or pixel size). New pixel values are computed using one of the following resampling algorithms:

- NearestNeighbor (english or american spelling)
- Bilinear
- Cubic
- CubicSpline
- Lanczos
- Max
- Min

The default is NearestNeighbor which is the fastest but results in the worst interpolation.

`scalex` and `scaley` define the new pixel size. scaley must often be negative to get well oriented raster.

⊠ scalex ⊠⊠ scaley ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ST_Resize ⊠⊠⊠⊠⊠⊠⊠.

`maxerr` is the threshold for transformation approximation by the resampling algorithm (in pixel units). A default of 0.125 is used if no `maxerr` is specified, which is the same value used in GDAL gdalwarp utility. If set to zero, no approximation takes place.

> **⊠⊠**
> ⊠⊠⊠⊠⊠⊠ GDAL Warp resampling methods ⊠⊠⊠⊠⊠⊠⊠.

> **⊠⊠**
> ST_Rescale ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ST_SetScale ⊠⊠⊠⊠⊠⊠. ST_SetScale ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (⊠⊠⊠⊠⊠⊠) ⊠⊠⊠⊠⊠ ⊠⊠⊠. ST_Rescale ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠. ST_SetScale ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. GDAL 1.6.1 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

Enhanced: 3.4.0 max and min resampling options added

⊠⊠⊠⊠: 2.1.0 ⊠⊠⊠⊠ SRID ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

⊠⊠⊠⊠ 0.001 ⊠⊠⊠⊠⊠⊠⊠ 0.0015 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

This example returns the original raster pixel size.

Code

```
SELECT ST_PixelWidth(ST_AddBand(ST_MakeEmptyRaster(100, 100, 0, 0, 0.001, -0.001, 0, 0, ↩
    4269), '8BUI'::text, 1, 0)) width
```

Output

```
0.001
```

This example returns the pixel size of the rescaled raster.

Code

```
SELECT ST_PixelWidth(ST_Rescale(ST_AddBand(ST_MakeEmptyRaster(100, 100, 0, 0, 0.001, ↩
    -0.001, 0, 0, 4269), '8BUI'::text, 1, 0), 0.0015)) width
```

Output

```
0.0015
```

⊠⊠

ST_Resize, ST_Resample, ST_SetScale, ST_ScaleX, ST_ScaleY, ST_Transform

11.7.9 ST_Reskew

ST_Reskew — ⊠⊠⊠ (⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. NearestNeighbor(⊠⊠⊠⊠⊠⊠⊠⊠⊠), Bilinear, Cubic, CubicSpline ⊠⊠ Lanczos ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠ NearestNeighbor ⊠⊠⊠.

⊠⊠

raster **ST_Reskew**(raster rast, double precision skewxy, text algorithm=NearestNeighbor, double precision maxerr=0.125);
raster **ST_Reskew**(raster rast, double precision skewx, double precision skewy, text algorithm=NearestNeig
double precision maxerr=0.125);

Description

⊠⊠⊠ (⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. NearestNeighbor(⊠⊠⊠⊠⊠⊠⊠⊠⊠), Bilinear, Cubic, CubicSpline ⊠⊠ Lanczos ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ NearestNeighbor ⊠⊠⊠.

skewx ⊠ skewy ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

maxerr ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ 0.125 ⊠⊠⊠⊠⊠⊠.

> **⊠⊠**
> ⊠⊠⊠⊠⊠⊠ GDAL Warp resampling methods ⊠⊠⊠⊠⊠⊠⊠.

> **⊠⊠**
> ST_Reskew ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ST_SetSkew ⊠⊠⊠⊠⊠⊠⊠. ST_SetSkew ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (⊠⊠⊠⊠⊠⊠⊠) ⊠⊠⊠⊠⊠⊠⊠⊠⊠. ST_Reskew ⊠⊠ ⊠⊠. ST_SetSkew ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. GDAL 1.6.1 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠⊠⊠: 2.1.0 ⊠⊠⊠⊠ SRID ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

□□

□□□ 0.0 □□□□□□ 0.0015 □□□□□□□□□□□□□□□□□□□□□□□□□□□.

This example uses the original non-rotated raster.

Code

```
SELECT ST_Rotation(ST_AddBand(ST_MakeEmptyRaster(100, 100, 0, 0, 0.001, -0.001, 0, 0, 4269) ↩
    , '8BUI'::text, 1, 0));
```

Output

```
st_rotation
-------------
           0
(1 row)
```

The reskewed raster rotation.

Code

```
SELECT ST_Rotation(ST_Reskew(ST_AddBand(ST_MakeEmptyRaster(100, 100, 0, 0, 0.001, -0.001,  ↩
    0, 0, 4269), '8BUI'::text, 1, 0), 0.0015));
```

Output

```
st_rotation
------------------
 -0.982793723247329
(1 row)
```

□□

ST_Resample, ST_Rescale, ST_SetSkew, ST_SetRotation, ST_SkewX, ST_SkewY, ST_Transform

11.7.10 ST_SnapToGrid

ST_SnapToGrid — □□□□□□□□□□□□□□□□□□□□□□□□□□□□. NearestNeighbor(□□□□□□□□), Bilinear, Cubic, CubicSpline □□ Lanczos □□□□□□□□□□□□□□□□□□□□□□□□□□□□. □□□□ NearestNeighbor □□□.

□□

raster **ST_SnapToGrid**(raster rast, double precision gridx, double precision gridy, text algorithm=NearestN
double precision maxerr=0.125, double precision scalex=DEFAULT 0, double precision scaley=DEFAULT
0);
raster **ST_SnapToGrid**(raster rast, double precision gridx, double precision gridy, double precision
scalex, double precision scaley, text algorithm=NearestNeighbor, double precision maxerr=0.125);
raster **ST_SnapToGrid**(raster rast, double precision gridx, double precision gridy, double precision
scalexy, text algorithm=NearestNeighbor, double precision maxerr=0.125);

Description

⊠⊠⊠⊠⊠⊠⊠⊠ (gridx & gridy) ⊠⊠⊠⊠⊠⊠⊠⊠⊠ (scalex & scaley) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. NearestNeighbor(⊠⊠⊠⊠⊠⊠⊠⊠), Bilinear, Cubic, CubicSpline ⊠ ⊠ Lanczos ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ NearestNeighbor ⊠⊠⊠.

`gridx` ⊠ `gridy` ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

You can optionally define the pixel size of the new grid with `scalex` and `scaley`.

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

`maxerr` ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ 0.125 ⊠⊠⊠⊠⊠⊠.

> **Note!**
>
> ⊠⊠
>
> ⊠⊠⊠⊠⊠⊠ GDAL Warp resampling methods ⊠⊠⊠⊠⊠⊠⊠.

> **Note!**
>
> ⊠⊠
>
> ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ST_Resample ⊠⊠⊠⊠⊠⊠⊠⊠⊠.

2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. GDAL 1.6.1 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠⊠⊠: 2.1.0 ⊠⊠⊠⊠ SRID ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

This example returns the upper-left X coordinate of the original raster.

Code

```
SELECT ST_UpperLeftX(ST_AddBand(ST_MakeEmptyRaster(10, 10, 0, 0, 0.001, -0.001, 0, 0, 4269) ↩
    , '8BUI'::text, 1, 0));
```

Output

```
st_upperleftx
---------------
             0
(1 row)
```

The upper left of raster after snapping.

Code

```
SELECT ST_UpperLeftX(ST_SnapToGrid(ST_AddBand(ST_MakeEmptyRaster(10, 10, 0, 0, 0.001,  ↩
    -0.001, 0, 0, 4269), '8BUI'::text, 1, 0), 0.0002, 0.0002));
```

Output

```
st_upperleftx
---------------
       -0.0008
(1 row)
```

☒☒

ST_Resample, ST_Rescale, ST_UpperLeftX, ST_UpperLeftY

11.7.11 ST_Resize

ST_Resize — ☒☒☒☒☒☒☒☒☒☒/☒☒☒☒☒☒☒☒.

☒☒

raster **ST_Resize**(raster rast, integer width, integer height, text algorithm=NearestNeighbor, double precision maxerr=0.125);
raster **ST_Resize**(raster rast, double precision percentwidth, double precision percentheight, text algorithm=NearestNeighbor, double precision maxerr=0.125);
raster **ST_Resize**(raster rast, text width, text height, text algorithm=NearestNeighbor, double precision maxerr=0.125);

Description

☒☒☒☒☒☒☒☒☒☒/☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒/☒☒☒☒☒☒☒☒☒☒/☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.

NearestNeighbor(☒☒☒☒☒☒☒☒☒), Bilinear, Cubic, CubicSpline ☒☒ Lanczos ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒ NearestNeighbor ☒☒☒.

☒☒ 1 ☒☒☒☒☒☒☒☒☒☒☒/☒☒☒☒☒☒☒☒☒.

☒☒ 2 ☒☒☒☒☒☒☒☒☒/☒☒☒☒☒☒☒☒☒☒☒ 0 ☒ 1 ☒☒☒☒☒☒☒☒☒☒☒☒☒.

☒☒ 3 ☒☒☒☒☒☒☒☒☒☒☒/☒☒☒☒☒☒☒☒☒☒☒☒/☒☒☒☒☒☒☒☒☒☒☒☒☒☒ ("20%") ☒☒☒☒☒☒☒.

2.1.0 ☒☒☒☒☒☒☒☒☒☒☒☒. GDAL 1.6.1 ☒☒☒☒☒☒☒☒☒☒.

☒☒

Code

```
WITH foo AS(
SELECT
    1 AS rid,
    ST_Resize(ST_AddBand(ST_MakeEmptyRaster(1000, 1000, 0, 0, 1, -1, 0, 0, 0),
            1, '8BUI', 255, 0
        ),
    '50%', '500') AS rast
UNION ALL
SELECT
    2 AS rid,
    ST_Resize(ST_AddBand(ST_MakeEmptyRaster(1000, 1000, 0, 0, 1, -1, 0, 0, 0),
```

```
            1, '8BUI', 255, 0
        ),
    500, 100) AS rast
UNION ALL
SELECT
    3 AS rid,
    ST_Resize(ST_AddBand(ST_MakeEmptyRaster(1000, 1000, 0, 0, 1, -1, 0, 0, 0),
            1, '8BUI', 255, 0
        ),
    0.25, 0.9) AS rast
)
SELECT
    rid,
    ST_Width(rast) AS width,
    ST_Height(rast) AS height,
    ST_ScaleX(rast) AS scale_x,
    round(ST_ScaleY(rast)::numeric, 4) AS scale_y
FROM foo
ORDER BY rid
```

Output

```
rid | width | height | scale_x | scale_y
-----+-------+--------+---------+---------
   1 |   500 |    500 |       2 | -2.0000
   2 |   500 |    100 |       2 | -10.0000
   3 |   250 |    900 |       4 | -1.1111
(3 rows)
```

⊠⊠

ST_Resample, ST_Rescale, ST_Reskew, ST_SnapToGrid

11.7.12 ST_Transform

ST_Transform — ⊠⊠. NearestNeighbor, Bilinear, Cubic, CubicSpline, Lanczos ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠ NearestNeighbor ⊠⊠⊠.

⊠⊠

raster **ST_Transform**(raster rast, integer srid, text algorithm=NearestNeighbor, double precision maxerr=0.125, double precision scalex, double precision scaley);
raster **ST_Transform**(raster rast, integer srid, double precision scalex, double precision scaley, text algorithm=NearestNeighbor, double precision maxerr=0.125);
raster **ST_Transform**(raster rast, raster alignto, text algorithm=NearestNeighbor, double precision maxerr=0.125);

Description

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (pixel warp) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ NearestNeighbor ⊠⊠, maxerror ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ 0.125 ⊠⊠⊠.

⊠⊠⊠⊠⊠⊠⊠⊠'NearestNeighbor', 'Bilinear', 'Cubic', 'CubicSpline', ⊠⊠⊠'Lanczos' ⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠ GDAL Warp resampling methods ⊠⊠⊠⊠⊠⊠⊠.

ST_Transform ⊠⊠⊠⊠⊠ ST_SetSRID() ⊠⊠⊠⊠⊠⊠⊠. ST_Transform ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠) ⊠⊠, ST_SetSRID() ⊠⊠⊠⊠ ⊠ SRID ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠⊠⊠⊠⊠⊠, ⊠⊠ 3 ⊠ `alignto` ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠ (SRID) ⊠⊠⊠⊠⊠⊠⊠⊠, (ST_SameAlignment = TRUE ⊠⊠⊠) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

> ⊠⊠
> If you find your transformation support is not working right, you may need to set the environment variable PROJSO to the projection library your PostGIS build is using. This just needs to have the name of the file. So for example on windows, you would in Control Panel -> System -> Environment Variables add a system variable called `PROJSO` and set it to `libproj.dll`. You'll have to restart your PostgreSQL service/daemon after this change.

> ⊠⊠
> When transforming a coverage of tiles, you almost always want to use a reference raster to insure same alignment and no gaps in your tiles as demonstrated in example: Variant 3.

2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. GDAL 1.6.1 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠⊠⊠: 2.1.0 ⊠⊠⊠⊠ ST_Transform(rast, alignto) ⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

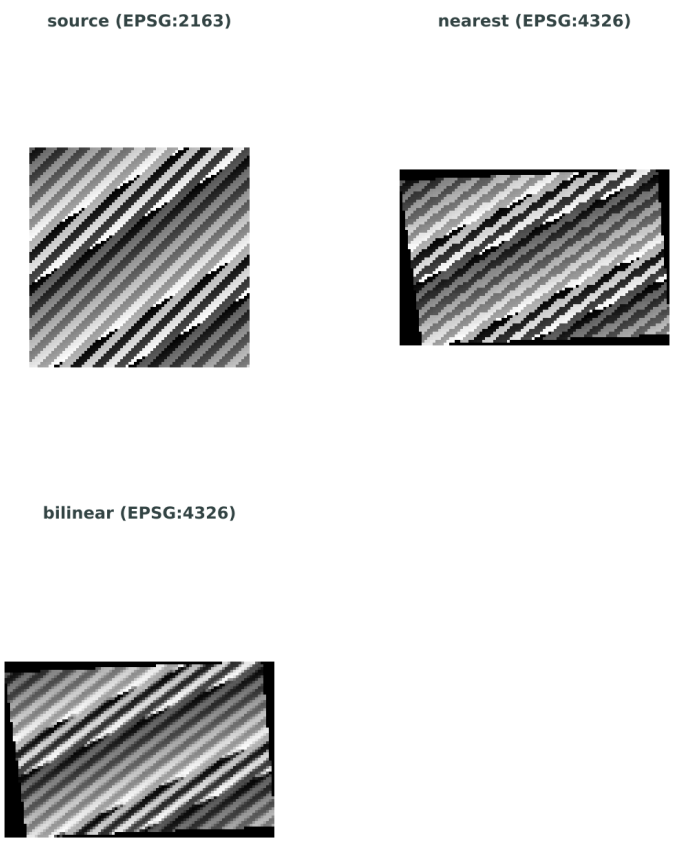
Compare a source raster with nearest-neighbor and bilinear reprojections. The diagonal bands make the resampling behavior visible: nearest-neighbor keeps hard pixel steps, while bilinear smooths values between pixels. All three figures are generated from this query at manual build time.

Code

```
WITH rows AS (
    SELECT
        y,
        array_agg(
            ((
                    x * 3 + y * 5
                    + CASE WHEN (x + y) % 8 < 4 THEN 80 ELSE 0 END
                ) % 255)::double precision
            ORDER BY x
        ) AS row_values
    FROM generate_series(1, 80) AS y
    CROSS JOIN generate_series(1, 80) AS x
    GROUP BY y
), source AS (
    SELECT ST_SetValues(
        ST_AddBand(
            ST_MakeEmptyRaster(
                80, 80,
                -500000, 600000,
                2500, -2500,
                0, 0,
                2163
            ),
            1, '8BUI', 0, NULL
        ),
```

```
        1, 1, 1,
        array_agg(row_values ORDER BY y)::double precision[][]
    ) AS rast
    FROM rows
), transformed AS (
    SELECT
        rast,
        ST_Transform(rast, 4326, 'NearestNeighbor') AS nearest,
        ST_Transform(rast, 4326, 'Bilinear') AS bilinear
    FROM source
)
SELECT
    ST_AsPNG(rast) AS "source (EPSG:2163)",
    ST_AsPNG(nearest) AS "nearest (EPSG:4326)",
    ST_AsPNG(bilinear) AS "bilinear (EPSG:4326)"
FROM transformed;
```

Figure



Raster reprojection and resampling.



Massachusetts State Plane source



WGS 84 nearest-neighbor



WGS 84 bilinear

Variant 3.

☒☒☒ ST_Transform(raster, srid) ☒ ST_Transform(raster, alignto) ☒☒☒☒☒☒☒☒☒☒☒☒☒☒.

Code

```
WITH foo AS (
    SELECT 0 AS rid, ST_AddBand(ST_MakeEmptyRaster(2, 2, -500000, 600000, 100, -100, 0, 0,  ↩
    2163), 1, '16BUI', 1, 0) AS rast UNION ALL
    SELECT 1, ST_AddBand(ST_MakeEmptyRaster(2, 2, -499800, 600000, 100, -100, 0, 0, 2163),  ↩
    1, '16BUI', 2, 0) AS rast UNION ALL
    SELECT 2, ST_AddBand(ST_MakeEmptyRaster(2, 2, -499600, 600000, 100, -100, 0, 0, 2163),  ↩
    1, '16BUI', 3, 0) AS rast UNION ALL

    SELECT 3, ST_AddBand(ST_MakeEmptyRaster(2, 2, -500000, 599800, 100, -100, 0, 0, 2163),  ↩
    1, '16BUI', 10, 0) AS rast UNION ALL
    SELECT 4, ST_AddBand(ST_MakeEmptyRaster(2, 2, -499800, 599800, 100, -100, 0, 0, 2163),  ↩
    1, '16BUI', 20, 0) AS rast UNION ALL
    SELECT 5, ST_AddBand(ST_MakeEmptyRaster(2, 2, -499600, 599800, 100, -100, 0, 0, 2163),  ↩
    1, '16BUI', 30, 0) AS rast UNION ALL

    SELECT 6, ST_AddBand(ST_MakeEmptyRaster(2, 2, -500000, 599600, 100, -100, 0, 0, 2163),  ↩
    1, '16BUI', 100, 0) AS rast UNION ALL
    SELECT 7, ST_AddBand(ST_MakeEmptyRaster(2, 2, -499800, 599600, 100, -100, 0, 0, 2163),  ↩
    1, '16BUI', 200, 0) AS rast UNION ALL
    SELECT 8, ST_AddBand(ST_MakeEmptyRaster(2, 2, -499600, 599600, 100, -100, 0, 0, 2163),  ↩
    1, '16BUI', 300, 0) AS rast
), bar AS (
    SELECT
        ST_Transform(rast, 4269) AS alignto
    FROM foo
    LIMIT 1
), baz AS (
    SELECT
        rid,
        rast,
        ST_Transform(rast, 4269) AS not_aligned,
        ST_Transform(rast, alignto) AS aligned
    FROM foo
    CROSS JOIN bar
)
SELECT
    ST_Collect(ST_ConvexHull(not_aligned) ORDER BY rid) AS not_aligned,
```

```
    ST_Collect(ST_ConvexHull(aligned) ORDER BY rid) AS aligned
FROM baz
```

Output

```
MULTIPOLYGON(((-107.007 50.2,-107.005 50.2,-107.005 50.198,-107.007 50.198,-107.007 50.2)),((-107.004 50.201,-107.002 ↩
    50.201,-107.002 50.198,-107.004 50.198,-107.004 50.201)),((-107.001 50.201,-106.999 50.201,-106.999 ↩
    50.198,-107.001 50.198,-107.001 50.201)),((-107.007 50.199,-107.004 50.199,-107.004 50.196,-107.007 ↩
    50.196,-107.007 50.199)),((-107.004 50.199,-107.001 50.199,-107.001 50.196,-107.004 50.196,-107.004 50.199)) ↩
    ,((-107.001 50.199,-106.999 50.199,-106.999 50.196,-107.001 50.196,-107.001 50.199)),((-107.006 50.197,-107.004 ↩
    50.197,-107.004 50.194,-107.006 50.194,-107.006 50.197)),((-107.004 50.197,-107.001 50.197,-107.001 ↩
    50.194,-107.004 50.194,-107.004 50.197)),((-107.001 50.197,-106.998 50.197,-106.998 50.195,-107.001 ↩
    50.195,-107.001 50.197))) | MULTIPOLYGON(((-107.007 50.2,-107.005 50.2,-107.005 50.198,-107.007 50.198,-107.007 ↩
    50.2)),((-107.005 50.202,-107.001 50.202,-107.001 50.198,-107.005 50.198,-107.005 50.202)),((-107.002 ↩
    50.202,-106.998 50.202,-106.998 50.198,-107.002 50.198,-107.002 50.202)),((-107.007 50.199,-107.003 ↩
    50.199,-107.003 50.195,-107.007 50.195,-107.007 50.199)),((-107.005 50.199,-107.001 50.199,-107.001 ↩
    50.195,-107.005 50.195,-107.005 50.199)),((-107.002 50.199,-106.998 50.199,-106.998 50.195,-107.002 ↩
    50.195,-107.002 50.199)),((-107.007 50.198,-107.003 50.198,-107.003 50.194,-107.007 50.194,-107.007 50.198)) ↩
    ,((-107.005 50.198,-107.001 50.198,-107.001 50.194,-107.005 50.194,-107.005 50.198)),((-107.002 50.198,-106.998 ↩
    50.198,-106.998 50.194,-107.002 50.194,-107.002 50.198)))
```

Figure



□□

ST_Transform, ST_SetSRID

11.8 □□□□□□□□

11.8.1 ST_SetBandNoDataValue

ST_SetBandNoDataValue — NODATA □□□□□□□□□□□□□□□□□□□□. □□□□□□□□□□□□□□□□ 1 □□□□□□□. □□□ NODATA □□□□□□□□□□, nodata value = NULL □□□□□□□□□□.

□□

raster **ST_SetBandNoDataValue**(raster rast, double precision nodatavalue);
raster **ST_SetBandNoDataValue**(raster rast, integer band, double precision nodatavalue, boolean forcechecking=false);

Description

□□□□ NODATA □□□□□□□□□□□□□. □□□□□□□□□□□□□□□ 1 □□□□□□□. □□□□ ST_Polygon, ST_DumpAsPolygons, □□□ ST_PixelAs...() □□□□□□□□□□□□□□□□.

□□

This example changes only the first band NODATA value.

Code

```
UPDATE dummy_rast
    SET rast = ST_SetBandNoDataValue(rast, 1, 254)
WHERE rid = 2;
```

This example changes the NODATA value for bands 1, 2, and 3.

Code

```
UPDATE dummy_rast
    SET rast =
        ST_SetBandNoDataValue(ST_SetBandNoDataValue(ST_SetBandNoDataValue(rast, 1, 254),
                2, 99),
                3, 108)
        WHERE rid = 2;
```

This example removes the NODATA value so that all pixels are considered by processing functions.

Code

```
UPDATE dummy_rast
    SET rast = ST_SetBandNoDataValue(rast, 1, NULL)
WHERE rid = 2;
```

□□

ST_BandNoDataValue, ST_NumBands

11.8.2 ST_SetBandIsNoData

ST_SetBandIsNoData — □□□ isnodata □□□□□□□□□□□□.

□□

raster **ST_SetBandIsNoData**(raster rast, integer band=1);

Description

□□□ isnodata □□□□□□□□□□□□. □□□□□□□□□□□□□□□ 1 □□□□□□□. □□□□□ □□□□□□□□□□□□□□□□□□□□□□□□□□□□□. □, □□□□□□□□□□□□□□□□□□□□□□□□□□ ST_BandIsNoData □□□□□□□□□□□□□□□□□□□□□□□□.

2.0.0 □□□□□□□□□□□□.

□□

This example creates a dummy table with one raster column. The raster has two bands and one pixel per band. In the first band, the NODATA value and pixel value are both 3. In the second band, the NODATA value is 13 and the pixel value is 4.

Code

```
create table dummy_rast (rid integer, rast raster);

insert into dummy_rast values(
1,
(
'01' -- little endian (uint8 ndr)
||
'0000' -- version (uint16 0)
||
'0200' -- nBands (uint16 0)
||
'17263529ED684A3F' -- scaleX (float64 0.000805965234044584)
||
'F9253529ED684ABF' -- scaleY (float64 -0.00080596523404458)
||
'1C9F33CE69E352C0' -- ipX (float64 -75.5533328537098)
||
'718F0E9A27A44840' -- ipY (float64 49.2824585505576)
||
'ED50EB853EC32B3F' -- skewX (float64 0.000211812383858707)
||
'7550EB853EC32B3F' -- skewY (float64 0.000211812383858704)
||
'E6100000' -- SRID (int32 4326)
||
'0100' -- width (uint16 1)
||
'0100' -- height (uint16 1)
||
'4' -- hasnodatavalue set to true, isnodata value set to false (when it should be true)
||
'2' -- first band type (4BUI)
||
'03' -- novalue==3
||
'03' -- pixel(0,0)==3 (same that nodata)
||
'0' -- hasnodatavalue set to false
||
'5' -- second band type (16BSI)
||
'0D00' -- novalue==13
||
'0400' -- pixel(0,0)==4
)::raster
```

```
);

select ST_BandIsNoData(rast, 1) from dummy_rast where rid = 1; -- Expected false
select ST_BandIsNoData(rast, 1, TRUE) from dummy_rast where rid = 1; -- Expected true
```

The `isnodata` flag is dirty, so set it to true.

Code

```
update dummy_rast set rast = ST_SetBandIsNoData(rast, 1) where rid = 1;


select ST_BandIsNoData(rast, 1) from dummy_rast where rid = 1; -- Expected true
```

□□

ST_BandNoDataValue, ST_NumBands, ST_SetBandNoDataValue, ST_BandIsNoData

11.8.3 ST_SetBandPath

ST_SetBandPath — Update the external path and band number of an out-db band

□□

raster **ST_SetBandPath**(raster rast, integer band, text outdbpath, integer outdbindex, boolean force=false)

Description

Updates an out-db band's external raster file path and external band number.



□□
If `force` is set to true, no tests are done to ensure compatibility (e.g. alignment, pixel support) between the external raster file and the PostGIS raster. This mode is intended for file system changes where the external raster resides.

Availability: 2.5.0

□□

Code

```
WITH foo AS (
    SELECT ST_AddBand(
        NULL::raster,
        '/home/pele/devel/geo/postgis-git/raster/test/regress/loader/Projected.tif',
        NULL::int[]
    ) AS rast
)
```

```
SELECT
    1 AS query,
    bandnum,
    regexp_replace(path, '^.*/', '') AS path,
    outdbbandnum
FROM ST_BandMetaData(
    (SELECT rast FROM foo),
    ARRAY[1, 3, 2]::int[]
) AS metadata
UNION ALL
SELECT
    2, bandnum,
    regexp_replace(path, '^.*/', ''),
    outdbbandnum
FROM ST_BandMetaData((
        SELECT
            ST_SetBandPath(rast,
                2,
                '/home/pele/devel/geo/postgis-git/raster/test/regress/loader/Projected2.tif ↩
    ',
                1
            ) AS rast
        FROM foo
    ),
    ARRAY[1, 3, 2]::int[]
) AS metadata
ORDER BY 1, 2;
```

In the second result set, band 2 now points at `Projected2.tif` and uses out-db band 1.

Output

```
query | bandnum | path           | outdbbandnum
-------+---------+----------------+--------------
     1 |       1 | Projected.tif  |            1
     1 |       2 | Projected.tif  |            2
     1 |       3 | Projected.tif  |            3
     2 |       1 | Projected.tif  |            1
     2 |       2 | Projected2.tif |            1
     2 |       3 | Projected.tif  |            3
```

☒☒

ST_BandMetaData, ST_SetBandIndex

11.8.4 ST_SetBandIndex

ST_SetBandIndex — Update the external band number of an out-db band

☒☒

raster **ST_SetBandIndex**(raster rast, integer band, integer outdbindex, boolean force=false);

Description

Updates an out-db band's external band number. This does not touch the external raster file associated with the out-db band



□□
If `force` is set to true, no tests are done to ensure compatibility (e.g. alignment, pixel support) between the external raster file and the PostGIS raster. This mode is intended for where bands are moved around in the external raster file.



□□
Internally, this method replaces the PostGIS raster's band at index band with a new band instead of updating the existing path information.

Availability: 2.5.0

□□

Code

```
WITH foo AS (
    SELECT ST_AddBand(
        NULL::raster,
        '/home/pele/devel/geo/postgis-git/raster/test/regress/loader/Projected.tif',
        NULL::int[]
    ) AS rast
)
SELECT
    1 AS query,
    bandnum,
    regexp_replace(path, '^.*/', '') AS path,
    outdbbandnum
FROM ST_BandMetaData(
    (SELECT rast FROM foo),
    ARRAY[1, 3, 2]::int[]
) AS metadata
UNION ALL
SELECT
    2, bandnum,
    regexp_replace(path, '^.*/', ''),
    outdbbandnum
FROM ST_BandMetaData((
        SELECT
            ST_SetBandIndex(rast,
                2,
                1
            ) AS rast
        FROM foo
    ),
    ARRAY[1, 3, 2]::int[]
) AS metadata
ORDER BY 1, 2;
```

In the second result set, band 2 still points at the same file but now reads external band 1.

Output

```
query | bandnum | path          | outdbbandnum
------+---------+---------------+--------------
    1 |       1 | Projected.tif |            1
    1 |       2 | Projected.tif |            2
    1 |       3 | Projected.tif |            3
```

```
     2 |       1 | Projected.tif |            1
     2 |       2 | Projected.tif |            1
     2 |       3 | Projected.tif |            3
```

⊠⊠

ST_BandMetaData, ST_SetBandPath

11.9 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠

11.9.1 ST_Count

ST_Count — ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠ 1 ⊠⊠⊠. exclude_nodata_value ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, NODATA ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠.

⊠⊠

bigint **ST_Count**(raster rast, integer nband=1, boolean exclude_nodata_value=true);
bigint **ST_Count**(raster rast, boolean exclude_nodata_value);

Description

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ nband ⊠⊠⊠ ⊠⊠ 1 ⊠⊠⊠.

> **Note!**
> ⊠⊠
> exclude_nodata_value ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠ nodata ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ exclude_nodata_value ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

2.2.0 ⊠⊠⊠⊠⊠⊠⊠ ST_Count(rastertable, rastercolumn, ...) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠ ST_CountAgg ⊠⊠⊠⊠⊠⊠⊠⊠⊠.

2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

Example will count all pixels not 249 and one will count all pixels.

Code

```
SELECT rid, ST_Count(ST_SetBandNoDataValue(rast, 249)) As exclude_nodata,
        ST_Count(ST_SetBandNoDataValue(rast, 249), false) As include_nodata
    FROM dummy_rast WHERE rid=2;
```

Output

```
rid | exclude_nodata | include_nodata
-----+----------------+----------------
   2 |             23 |             25
```

□□

ST_CountAgg, ST_SummaryStats, ST_SetBandNoDataValue

11.9.2 ST_CountAgg

ST_CountAgg — □□□□□□□□. □□□□□□□□□□□□□□□□□□□□□□□□□□□. □□□□□□□□□□□□□□□□□□□□□ 1 □□□. exclude_nodata_value □□□□□□□□□□, NODATA □□□□□□□□□□□□□□□□□□□.

□□

bigint **ST_CountAgg**(raster rast, integer nband, boolean exclude_nodata_value, double precision sample_percent);
bigint **ST_CountAgg**(raster rast, integer nband, boolean exclude_nodata_value);
bigint **ST_CountAgg**(raster rast, boolean exclude_nodata_value);

Description

□□□□□□□□□□□□□□□□□□□□□□□□□□□. □□□□□□□□□□□□□□ nband □□□□□ 1 □□□.

exclude_nodata_value □□□□□□□□□□, □□□□ nodata □□□□□□□□□□□□□□□□□□□□. □□□□□□□□□□□□□ exclude_nodata_value □□□□□□□□□□□□.

□□□□□□□□□□□□□□□□□□□□□. □□□□□□□□□□□, sample_percent □ 0 □ 1 □□□□□□□□□□□□□.

2.2.0 □□□□□□□□□□□□□□□.

□□

Code

```
WITH foo AS (
    SELECT
        rast.rast
    FROM (
        SELECT ST_SetValue(ST_SetValue(ST_SetValue(ST_AddBand(ST_MakeEmptyRaster(10, 10,  ↩
    10, 10, 2, 2, 0, 0, 0),
                        1, '64BF', 0, 0
                    ),
                    1, 1, 1, -10
                ),
                1, 5, 4, 0
            ),
            1, 5, 5, 3.14159
        ) AS rast
    ) AS rast
    FULL JOIN (
```

```
        SELECT generate_series(1, 10) AS id
    ) AS id
        ON 1 = 1
)
SELECT
    ST_CountAgg(rast, 1, TRUE) AS non_nodata_pixels
FROM foo;
```

Output

```
non_nodata_pixels
-------------------
                20
(1 row)
```

☒☒

ST_Count, ST_SummaryStats, ST_SetBandNoDataValue

11.9.3 ST_Histogram

ST_Histogram — ☒ (bin; ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒) ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒ ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.

☒☒

SETOF record **ST_Histogram**(raster rast, integer nband=1, boolean exclude_nodata_value=true, integer bins=autocomputed, double precision[] width=NULL, boolean right=false);
SETOF record **ST_Histogram**(raster rast, integer nband, integer bins, double precision[] width=NULL, boolean right=false);
SETOF record **ST_Histogram**(raster rast, integer nband, boolean exclude_nodata_value, integer bins, boolean right);
SETOF record **ST_Histogram**(raster rast, integer nband, integer bins, boolean right);

Description

☒☒☒☒☒☒☒☒☒☒☒☒☒ min, max, count, percent ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒ ☒☒☒☒☒ nband ☒☒☒☒☒ 1 ☒☒☒.

☒☒
☒ ☒ ☒ ☒ ☒ nodata ☒ ☒ ☒ ☒ ☒ ☒ ☒ ☒ ☒ ☒ ☒ ☒ ☒. ☒ ☒ ☒ ☒ ☒ ☒ ☒ ☒ ☒ ☒ ☒ ☒ exclude_nodata_value ☒☒☒☒☒☒☒☒☒☒☒.

---

width width: ☒☒☒☒☒/☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒ width ☒☒☒☒☒☒☒, width ☒☒☒☒☒ ☒.

☒☒: ☒ 9 ☒, width [a, b, c] ☒ [a, b, c, a, b, c, a, b, c] ☒☒☒☒☒☒☒☒.

bins ☒☒☒☒☒ (breakout) ☒☒☒: ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒ ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.

right □□□□□□□□□□□□□□□□□□□□ (□□□) □□□□□□. X □□□□□□□□□□ [a, b) □□ (a, b] □ □□□□□□.

Changed: 3.1.0 Removed ST_Histogram(table_name, column_name) variant.

2.0.0 □□□□□□□□□□□□.

□□

Single raster tile - compute histograms for bands 1, 2, 3 and autocompute bins.

Code

```
SELECT band, (stats).*
FROM (SELECT rid, band, ST_Histogram(rast, band) As stats
    FROM dummy_rast CROSS JOIN generate_series(1, 3) As band
     WHERE rid=2) As foo;
```

Output

```
band |  min  |  max  | count | percent
------+-------+-------+-------+---------
   1 |   249 |   250 |     2 |    0.08
   1 |   250 |   251 |     2 |    0.08
   1 |   251 |   252 |     1 |    0.04
   1 |   252 |   253 |     2 |    0.08
   1 |   253 |   254 |    18 |    0.72
   2 |    78 | 113.2 |    11 |    0.44
   2 | 113.2 | 148.4 |     4 |    0.16
   2 | 148.4 | 183.6 |     4 |    0.16
   2 | 183.6 | 218.8 |     1 |    0.04
   2 | 218.8 |   254 |     5 |     0.2
   3 |    62 | 100.4 |    11 |    0.44
   3 | 100.4 | 138.8 |     5 |     0.2
   3 | 138.8 | 177.2 |     4 |    0.16
   3 | 177.2 | 215.6 |     1 |    0.04
   3 | 215.6 |   254 |     4 |    0.16
```

Just band 2 but for 6 bins.

Code

```
SELECT (stats).*
FROM (SELECT rid, ST_Histogram(rast, 2, 6) As stats
    FROM dummy_rast
     WHERE rid=2) As foo;
```

Output

```
min     |    max     | count | percent
------------+------------+-------+---------
         78 | 107.333333 |     9 |    0.36
 107.333333 | 136.666667 |     6 |    0.24
 136.666667 |        166 |     0 |       0
        166 | 195.333333 |     4 |    0.16
 195.333333 | 224.666667 |     1 |    0.04
 224.666667 |        254 |     5 |     0.2
(6 rows)
```

Same as previous but we explicitly control the pixel value range of each bin.

Code

```
SELECT (stats).*
FROM (SELECT rid, ST_Histogram(rast, 2, 6, ARRAY[0.5, 1, 4, 100, 5]) As stats
    FROM dummy_rast
     WHERE rid=2) As foo;
```

Output

```
min  |  max  | count | percent
-------+-------+-------+----------
    78 |  78.5 |     1 |     0.08
  78.5 |  79.5 |     1 |     0.04
  79.5 |  83.5 |     0 |        0
  83.5 | 183.5 |    17 |   0.0068
 183.5 | 188.5 |     0 |        0
 188.5 |   254 |     6 | 0.003664
(6 rows)
```

□□

ST_Count, ST_SummaryStats, ST_SummaryStatsAgg

11.9.4 ST_Quantile

ST_Quantile — □□□□□□□□ (population) □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ (quantile) □□□□□□□. □□□□, □□□□□ 25%, 50%, 75% □□□□ (percentile) □□□□□□□□□□□□□□□□.

□□

SETOF record **ST_Quantile**(raster rast, integer nband=1, boolean exclude_nodata_value=true, double precision[] quantiles=NULL);
SETOF record **ST_Quantile**(raster rast, double precision[] quantiles);
SETOF record **ST_Quantile**(raster rast, integer nband, double precision[] quantiles);
double precision **ST_Quantile**(raster rast, double precision quantile);
double precision **ST_Quantile**(raster rast, boolean exclude_nodata_value, double precision quantile=NULL);
double precision **ST_Quantile**(raster rast, integer nband, double precision quantile);
double precision **ST_Quantile**(raster rast, integer nband, boolean exclude_nodata_value, double precision quantile);
double precision **ST_Quantile**(raster rast, integer nband, double precision quantile);

Description

□□□□□□□□□ (population) □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ (quantile) □□□□□□. □□□□, □□□□□ 25%, 50%, 75% □□□□ (percentile) □□□□□□□□□□□□□□□□.



□□
exclude_nodata_value □□□□□□□□□□, NODATA □□□□□□□□□□□□□□.

Changed: 3.1.0 Removed ST_Quantile(table_name, column_name) variant.

2.0.0 □□□□□□□□□□□□□□.

□□

Example will consider only pixels of band 1 that are not 249 and in named quantiles.

Code

```
UPDATE dummy_rast SET rast = ST_SetBandNoDataValue(rast, 249) WHERE rid=2;

SELECT (pvq).*
FROM (SELECT ST_Quantile(rast, ARRAY[0.25, 0.75]) As pvq
    FROM dummy_rast WHERE rid=2) As foo
    ORDER BY (pvq).quantile;
```

Output

```
quantile | value
----------+-------
    0.25 |   253
    0.75 |   254
```

Code

```
SELECT ST_Quantile(rast, 0.75) As value
FROM dummy_rast WHERE rid=2;
```

Output

```
254
```

Real live example. Quantile of all pixels in band 2 intersecting a geometry.

Code

```
SELECT rid, (ST_Quantile(rast, 2)).* As pvc
    FROM o_4_boston
        WHERE ST_Intersects(
            rast,
            ST_GeomFromText('POLYGON((224486 892151,224486 892200,224706 892200,224706 ↩
    892151,224486 892151))', 26986)
            )
ORDER BY value, quantile, rid
;
```

Output

```
rid | quantile | value
-----+----------+-------
   1 |        0 |     0
   2 |        0 |     0
  14 |        0 |     1
  15 |        0 |     2
  14 |     0.25 |    37
   1 |     0.25 |    42
  15 |     0.25 |    47
   2 |     0.25 |    50
  14 |      0.5 |    56
   1 |      0.5 |    64
  15 |      0.5 |    66
   2 |      0.5 |    77
  14 |     0.75 |    81
  15 |     0.75 |    87
   1 |     0.75 |    94
   2 |     0.75 |   106
  14 |        1 |   199
   1 |        1 |   244
   2 |        1 |   255
  15 |        1 |   255
```

☒☒

ST_Count, ST_SummaryStats, ST_SummaryStatsAgg, ST_SetBandNoDataValue

11.9.5 ST_SummaryStats

ST_SummaryStats — Returns summarystats consisting of count, sum, mean, stddev, min, max for a given raster band of a raster or raster coverage. Band 1 is assumed if no band is specified.

□□

summarystats **ST_SummaryStats**(raster rast, boolean exclude_nodata_value);
summarystats **ST_SummaryStats**(raster rast, integer nband, boolean exclude_nodata_value);

Description

□□□□□□□□□□□□□□□□□□□□□□□□□ count, sum, mean, stddev, min, max □□□□□ summarystats □□□□□□□. □□□□□□□□□□□□□□□ nband □□□□□□ 1 □□□□.



□□
□ □ □ □ □ nodata □ □ □ □ □ □ □ □ □ □ □ □ □. □ □ □ □ □ □ □ □ □ □ □ □ □ exclude_nodata_value □□□□□□□□□□□□.



□□
□□□□□□□□□□□□□□□□□□□□□. □□□□□□□□□□□, sample_percent □ 1 □□□□□□□ □□□□□□□□.

2.2.0 □□□□□□□ ST_SummaryStats(rastertable, rastercolumn, ...) □□□□□□□□□□□□□□. □□ ST_SummaryStatsAgg □□□□□□□□□□.

2.0.0 □□□□□□□□□□□□.

□□

Single raster tile.

Code

```
SELECT rid, band, (stats).*
FROM (SELECT rid, band, ST_SummaryStats(rast, band) As stats
    FROM dummy_rast CROSS JOIN generate_series(1, 3) As band
     WHERE rid=2) As foo;
```

Output

```
rid | band | count | sum  |    mean    |  stddev   | min | max
----+------+-------+------+------------+-----------+-----+-----
  2 |    1 |    23 | 5821 | 253.086957 |  1.248061 | 250 | 254
  2 |    2 |    25 | 3682 |     147.28 | 59.862188 |  78 | 254
  2 |    3 |    25 | 3290 |      131.6 | 61.647384 |  62 | 254
```

Summarize pixels that intersect buildings of interest.

This example selects the target buildings, clips band 2 of the raster tiles to each building boundary, computes statistics for each clipped region, and then summarizes those statistics by building. It took 574ms on PostGIS Windows 64-bit with all of Boston Buildings and aerial Tiles (tiles each 150x150 pixels ~ 134,000 tiles), ~102,000 building records.

Code

```
WITH
-- our features of interest
feat AS (
    SELECT gid AS building_id, geom_26986 AS geom
    FROM buildings AS b
    WHERE gid IN (100, 103, 150)
),
-- clip band 2 of raster tiles to boundaries of builds
-- then get stats for these clipped regions
b_stats AS (
    SELECT building_id, (stats).*
    FROM (
        SELECT
            building_id,
            ST_SummaryStats(ST_Clip(rast, 2, geom)) AS stats
        FROM aerials.boston
        INNER JOIN feat
            ON ST_Intersects(feat.geom, rast)
    ) AS foo
)
-- finally summarize stats
SELECT
    building_id,
    SUM(count) AS num_pixels,
    MIN(min) AS min_pval,
    MAX(max) AS max_pval,
    SUM(mean * count) / SUM(count) AS avg_pval
FROM b_stats
WHERE count
> 0
GROUP BY building_id
ORDER BY building_id;
```

Output

```
building_id | num_pixels | min_pval | max_pval |     avg_pval
------------+------------+----------+----------+------------------
        100 |       1090 |        1 |      255 | 61.0697247706422
        103 |        655 |        7 |      182 | 70.5038167938931
        150 |        895 |        2 |      252 | 185.642458100559
```

Raster coverage.

This example returns statistics for each band.

Code

```
SELECT band, (stats).*
FROM (SELECT band, ST_SummaryStats('o_4_boston', 'rast', band) As stats
    FROM generate_series(1, 3) As band) As foo;
```

Output

```
band |  count  |  sum   |       mean       |      stddev      | min | max
-----+---------+--------+------------------+------------------+-----+-----
   1 | 8450000 | 725799 | 82.7064349112426 | 45.6800222638537 |   0 | 255
   2 | 8450000 | 700487 | 81.4197705325444 | 44.2161184161765 |   0 | 255
   3 | 8450000 | 575943 |  74.682739408284 | 44.2143885481407 |   0 | 255
```

For a table, sampling less than 100% can improve speed. Here the sampling is set to 25%.

Code

```
SELECT band, (stats).*
FROM (SELECT band, ST_SummaryStats('o_4_boston', 'rast', band, true, 0.25) As stats
    FROM generate_series(1, 3) As band) As foo;
```

Output

```
band |  count  |  sum   |       mean       |      stddev      | min | max
------+---------+--------+------------------+------------------+-----+-----
    1 | 2112500 | 180686 | 82.6890480473373 | 45.6961043857248 |   0 | 255
    2 | 2112500 | 174571 |  81.448503668639 | 44.2252623171821 |   0 | 255
    3 | 2112500 | 144364 | 74.6765884023669 | 44.2014869384578 |   0 | 255
```

□□

summarystats, ST_SummaryStatsAgg, ST_Count, ST_Clip

11.9.6 ST_SummaryStatsAgg

ST_SummaryStatsAgg — Aggregate. Returns summarystats consisting of count, sum, mean, stddev, min, max for a given raster band of a set of raster. Band 1 is assumed if no band is specified.

□□

summarystats **ST_SummaryStatsAgg**(raster rast, integer nband, boolean exclude_nodata_value, double precision sample_percent);
summarystats **ST_SummaryStatsAgg**(raster rast, boolean exclude_nodata_value, double precision sample_percent);
summarystats **ST_SummaryStatsAgg**(raster rast, integer nband, boolean exclude_nodata_value);

Description

□□□□□□□□□□□□□□□□□□□□□□□□□ count, sum, mean, stddev, min, max □□□□□ summarystats □□□□□□□. □□□□□□□□□□□□□□□ nband □□□□□□ 1 □□□□.



□□
□ □ □ □ □ nodata □ □ □ □ □ □ □ □ □ □ □ □ □ □. □ □ □ □ □ □ □ □ □ □ □ □ □ □ exclude_nodata_value □□□□□□□□□□□□.



□□
□□□□□□□□□□□□□□□□□□□□□. □□□□□□□□□□□, sample_percent □ 0 □ 1 □□□□ □□□□□□□□□.

2.2.0 □□□□□□□□□□□□.

□□

Code

```
WITH foo AS (
    SELECT
        rast.rast
    FROM (
        SELECT ST_SetValue(ST_SetValue(ST_SetValue(ST_AddBand(ST_MakeEmptyRaster(10, 10, ↩
    10, 10, 2, 2, 0, 0, 0),
                        1, '64BF', 0, 0
                    ),
                    1, 1, 1, -10
                ),
                1, 5, 4, 0
            ),
            1, 5, 5, 3.14159
        ) AS rast
    ) AS rast
    FULL JOIN (
        SELECT generate_series(1, 10) AS id
    ) AS id
        ON 1 = 1
)
SELECT
    (stats).count AS count,
    round((stats).sum::numeric, 3) AS sum,
    round((stats).mean::numeric, 3) AS mean,
    round((stats).stddev::numeric, 3) AS stddev,
    round((stats).min::numeric, 3) AS min,
    round((stats).max::numeric, 3) AS max
FROM (
    SELECT
        ST_SummaryStatsAgg(rast, 1, TRUE, 1) AS stats
    FROM foo
) bar;
```

Output

```
count |   sum    |  mean  | stddev |   min    |  max
-------+---------+--------+--------+---------+-------
    20 | -68.584 | -3.429 |  6.571 | -10.000 | 3.142
(1 row)
```

☒☒

summarystats, ST_SummaryStats, ST_Count, ST_Clip

11.9.7 ST_ValueCount

ST_ValueCount — ☒☒☒☒☒☒☒☒☒☒☒☒☒☒ (☒☒☒☒☒☒☒☒☒☒) ☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒ 1 ☒☒☒. ☒☒☒☒☒☒ NODATA ☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒, ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.

☒☒

SETOF record **ST_ValueCount**(raster rast, integer nband=1, boolean exclude_nodata_value=true, double precision[] searchvalues=NULL, double precision roundto=0, double precision OUT value, integer OUT count);

SETOF record **ST_ValueCount**(raster rast, integer nband, double precision[] searchvalues, double precision roundto=0, double precision OUT value, integer OUT count);
SETOF record **ST_ValueCount**(raster rast, double precision[] searchvalues, double precision roundto=0, double precision OUT value, integer OUT count);
bigint **ST_ValueCount**(raster rast, double precision searchvalue, double precision roundto=0);
bigint **ST_ValueCount**(raster rast, integer nband, boolean exclude_nodata_value, double precision searchvalue, double precision roundto=0);
bigint **ST_ValueCount**(raster rast, integer nband, double precision searchvalue, double precision roundto=0);
SETOF record **ST_ValueCount**(text rastertable, text rastercolumn, integer nband=1, boolean exclude_nodata_value=true, double precision[] searchvalues=NULL, double precision roundto=0, double precision OUT value, integer OUT count);
SETOF record **ST_ValueCount**(text rastertable, text rastercolumn, double precision[] searchvalues, double precision roundto=0, double precision OUT value, integer OUT count);
SETOF record **ST_ValueCount**(text rastertable, text rastercolumn, integer nband, double precision[] searchvalues, double precision roundto=0, double precision OUT value, integer OUT count);
bigint**ST_ValueCount**(text rastertable, text rastercolumn, integer nband, boolean exclude_nodata_value, double precision searchvalue, double precision roundto=0);
bigint **ST_ValueCount**(text rastertable, text rastercolumn, double precision searchvalue, double precision roundto=0);
bigint **ST_ValueCount**(text rastertable, text rastercolumn, integer nband, double precision searchvalue, double precision roundto=0);

Description

□□□ value, count □□□□□□□□□□□□□□□□.

□□□□□□□□□□□□□□□ nband □□□□□□ 1 □□□. searchvalues □□□□□□□□□□, □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□. searchvalues □□□□□□□□□, □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□.

> **Note**
>
> □□
> exclude_nodata_value □□□□□□□□□□, NODATA □□□□□□□□□□□□□.

2.0.0 □□□□□□□□□□□□□.

□□

Example will count only pixels of band 1 that are not 249.

Code

```
UPDATE dummy_rast SET rast = ST_SetBandNoDataValue(rast, 249) WHERE rid=2;

SELECT (pvc).*
FROM (SELECT ST_ValueCount(rast) As pvc
    FROM dummy_rast WHERE rid=2) As foo
    ORDER BY (pvc).value;
```

Output

```
value | count
-------+-------
   250 |     2
   251 |     1
   252 |     2
   253 |     6
   254 |    12
```

Example will coount all pixels of band 1 including 249.

Code

```
SELECT (pvc).*
FROM (SELECT ST_ValueCount(rast, 1, false) As pvc
    FROM dummy_rast WHERE rid=2) As foo
    ORDER BY (pvc).value;
```

Output

```
value | count
-------+-------
   249 |     2
   250 |     2
   251 |     1
   252 |     2
   253 |     6
   254 |    12
```

Example will count only non-nodata value pixels of band 2.

Code

```
SELECT (pvc).*
FROM (SELECT ST_ValueCount(rast, 2) As pvc
    FROM dummy_rast WHERE rid=2) As foo
    ORDER BY (pvc).value;
```

Output

```
value | count
-------+-------
    78 |     1
    79 |     1
    88 |     1
    89 |     1
    96 |     1
    97 |     1
    98 |     1
    99 |     2
   112 |     2
:
```

Real live example. Count all the pixels in an aerial raster tile band 2 intersecting a geometry. And return only the pixel band values that have a count > 500.

Code

```
SELECT (pvc).value, SUM((pvc).count) As total
FROM (SELECT ST_ValueCount(rast, 2) As pvc
    FROM o_4_boston
        WHERE ST_Intersects(
            rast,
            ST_GeomFromText('POLYGON((224486 892151,224486 892200,224706 892200,224706 ↩
    892151,224486 892151))', 26986)
             )
        ) As foo
```

```
    GROUP BY (pvc).value
    HAVING SUM((pvc).count) > 500
    ORDER BY (pvc).value;
```

Output

```
value | total
-------+-----
    51 | 502
    54 | 521
```

Just return count of pixels in each raster tile that have value of 100 of tiles that intersect a specific geometry.

Code

```
SELECT rid, ST_ValueCount(rast, 2, 100) As count
    FROM o_4_boston
        WHERE ST_Intersects(
            rast,
            ST_GeomFromText('POLYGON((224486 892151,224486 892200,224706 892200,224706 ↩
    892151,224486 892151))', 26986)
             ) ;
```

Output

```
rid | count
-----+-------
   1 |    56
   2 |    95
  14 |    37
  15 |    64
```

□□

ST_Count, ST_SetBandNoDataValue

11.10 Raster Inputs

11.10.1 ST_RastFromWKB

ST_RastFromWKB — Return a raster value from a Well-Known Binary (WKB) raster.

□□

raster **ST_RastFromWKB**(bytea wkb);

Description

Given a Well-Known Binary (WKB) raster, return a raster.

Availability: 2.5.0

□□

Code

```
SELECT (ST_MetaData(ST_RastFromWKB('\001\000\000\000\000\000\000\000\000\000\000\000@ ↩
    \000\000\000\000\000\000\010@ ↩
    \000\000\000\000\000\000\340?\000\000\000\000\000\000\340?\000\000\000\000\000\000\000\000\0
    '::bytea
    )
)).* AS metadata;
```

Output

```
upperleftx | upperlefty | width | height | scalex | scaley | skewx | skewy | srid | numbands
-----------+------------+-------+--------+--------+--------+-------+-------+------+----------
       0.5 |        0.5 |    10 |     20 |      2 |      3 |     0 |     0 |   10 |        0
```

□□

ST_MetaData, ST_RastFromHexWKB, ST_AsBinary/ST_AsWKB, ST_AsHexWKB

11.10.2 ST_RastFromHexWKB

ST_RastFromHexWKB — Return a raster value from a Hex representation of Well-Known Binary (WKB) raster.

□□

raster **ST_RastFromHexWKB**(text wkb);

Description

Given a Well-Known Binary (WKB) raster in Hex representation, return a raster.

Availability: 2.5.0

□□

Code

```
SELECT (ST_MetaData(ST_RastFromHexWKB(' ↩
    01000000000000000000000040000000000000084000000000000000 ↩
    E03F000000000000E03F0000000000000000000000000000000A0000000A001400'
    )
)).* AS metadata;
```

Output

```
upperleftx | upperlefty | width | height | scalex | scaley | skewx | skewy | srid | numbands
-----------+------------+-------+--------+--------+--------+-------+-------+------+----------
       0.5 |        0.5 |    10 |     20 |      2 |      3 |     0 |     0 |   10 |        0
```

□□

ST_MetaData, ST_RastFromWKB, ST_AsBinary/ST_AsWKB, ST_AsHexWKB

11.11 ⊠⊠⊠⊠⊠

11.11.1 ST_AsBinary/ST_AsWKB

ST_AsBinary/ST_AsWKB — Return the Well-Known Binary (WKB) representation of the raster.

⊠⊠

bytea **ST_AsBinary**(raster rast, boolean outasin=FALSE);
bytea **ST_AsWKB**(raster rast, boolean outasin=FALSE);

Description

Returns the Binary representation of the raster. If `outasin` is TRUE, out-db bands are treated as in-db. Refer to raster/doc/RFC2-WellKnownBinaryFormat located in the PostGIS source folder for details of the representation.

⊠⊠⊠.

> **⊠⊠**
> ⊠⊠⊠⊠⊠, WKB ⊠⊠⊠⊠ DB ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠ DB ⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, `outasin` ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠⊠⊠: 2.1.0 ⊠⊠⊠⊠ `outasin` ⊠⊠⊠⊠⊠⊠⊠.

Enhanced: 2.5.0 Addition of `ST_AsWKB`

⊠⊠

Code

```
SELECT ST_AsBinary(rast) AS rastbin FROM dummy_rast WHERE rid=1;
```

Output

```
\001\000\000\000\000\000\000\000\000\000\000\000@\000\000\000\000\000\000\010@ ↩
    \000\000\000\000\000\000\340?\000\000\000\000\000\000\340?\000\000\000\000\000\000\000\000\000\000\000\000\000\000\000\0
```

⊠⊠

ST_RastFromWKB, ST_AsHexWKB

11.11.2 ST_AsHexWKB

ST_AsHexWKB — Return the Well-Known Binary (WKB) in Hex representation of the raster.

⊠⊠

bytea **ST_AsHexWKB**(raster rast, boolean outasin=FALSE);

Description

Returns the Binary representation in Hex representation of the raster. If `outasin` is TRUE, out-db bands are treated as in-db. Refer to raster/doc/RFC2-WellKnownBinaryFormat located in the PostGIS source folder for details of the representation.



□□
By default, Hex WKB output contains the external file path for out-db bands. If the client does not have access to the raster file underlying an out-db band, set `outasin` to TRUE.

Availability: 2.5.0

□□

Code

```
SELECT ST_AsHexWKB(rast) AS rastbin FROM dummy_rast WHERE rid=1;
```

Output

```
01000000000000000000000040000000000000008400000000000000 ↩
     E03F000000000000E03F000000000000000000000000000000000A0000000A001400
```

□□

ST_RastFromHexWKB, ST_AsBinary/ST_AsWKB

11.11.3 ST_AsGDALRaster

ST_AsGDALRaster — Return the raster tile in the designated GDAL Raster format. Raster formats are one of those supported by your compiled library. Use ST_GDALDrivers() to get a list of formats supported by your library.

□□

bytea **ST_AsGDALRaster**(raster rast, text format, text[] options=NULL, integer srid=sameassource);

Description

□□□□□□□□□□□□□□□□□□□□. □□□□□□□□□□□□:

- `format` - □□□□□□□□□. □□□□□ LibGDAL □□□□□□□□□□□□□□□□□□□□□□□□□□□□. □□□□□□□□□□□□□□□□□'JPEG', 'GTiff', 'PNG' □□□□. □□□□□□□□□□□□□□□□□□□□□□□□□□□ ST_GDALDrivers □□□□□□□□□□.
- `options` - GDAL □□□□□□□□□□□□□. □□□□□□□□□□□□□□□□□□□. □□□□□□□ GDAL □□□□□□□□ □□□□□□□□.
- `srid` the spatial reference identifier to embed in the output raster

2.0.0 □□□□□□□□□□□□□. GDAL 1.6.0 □□□□□□□□□□□.

□□

Export multiple raster tiles as a single JPEG image with 50 percent quality.

Code

```
SELECT ST_AsGDALRaster(ST_Union(rast), 'JPEG', ARRAY['QUALITY=50']) As rastjpg
FROM dummy_rast
WHERE rast && ST_MakeEnvelope(10, 10, 11, 11);
```

One way to export raster into another format is using PostgreSQL large object export functions. The following repeats the JPEG example and exports the result. Note that this requires superuser access because it uses server-side large object functions. It will also export to path on server network. If you need export locally, use the psql equivalent lo_ functions which export to the local file system instead of the server file system.

Code

```
DROP TABLE IF EXISTS tmp_out ;

CREATE TABLE tmp_out AS
SELECT lo_from_bytea(
       0,
       ST_AsGDALRaster(ST_Union(rast), 'JPEG', ARRAY['QUALITY=50'])
        ) AS loid
  FROM dummy_rast
WHERE rast && ST_MakeEnvelope(10, 10, 11, 11);

SELECT lo_export(loid, '/tmp/dummy.jpg')
   FROM tmp_out;

SELECT lo_unlink(loid)
  FROM tmp_out;
```

Export a raster as a GeoTIFF using the default creation options.

Code

```
SELECT ST_AsGDALRaster(rast, 'GTiff') AS rasttiff
FROM dummy_rast WHERE rid = 2;
```

Export a GeoTIFF using JPEG compression at 90 percent quality and embed SRID 4269.

Code

```
SELECT ST_AsGDALRaster(
  rast,
  'GTiff',
  ARRAY['COMPRESS=JPEG', 'JPEG_QUALITY=90'],
  4269) AS rasttiff
FROM dummy_rast WHERE rid = 2;
```

□□

10.4□, ST_GDALDrivers, ST_SRID

11.11.4 ST_AsJPEG

ST_AsJPEG — ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ JPEG(Joint Photographic Exports Group) ⊠⊠⊠ (⊠⊠⊠⊠⊠) ⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠ 1 ⊠⊠⊠⊠⊠ 3 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠ 3 ⊠⊠⊠⊠⊠⊠⊠⊠ 3 ⊠⊠⊠⊠⊠⊠⊠ RGB ⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

bytea **ST_AsJPEG**(raster rast, text[] options=NULL);
bytea **ST_AsJPEG**(raster rast, integer nband, integer quality);
bytea **ST_AsJPEG**(raster rast, integer nband, text[] options=NULL);
bytea **ST_AsJPEG**(raster rast, integer[] nbands, text[] options=NULL);
bytea **ST_AsJPEG**(raster rast, integer[] nbands, integer quality);

Description

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ JPEG(Joint Photographic Exports Group) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ST_AsGDALRaster ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠ 1 ⊠⊠⊠⊠⊠ 3 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠ 3 ⊠⊠⊠⊠⊠⊠⊠ 3 ⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

- `nband` - ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- `nbands` - ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (JPEG ⊠⊠⊠ 3 ⊠⊠⊠⊠⊠⊠⊠⊠). ⊠⊠⊠⊠⊠⊠⊠ RGB ⊠⊠⊠. ⊠⊠⊠⊠⊠ ARRAY[3,2,1] ⊠⊠⊠ 3 ⊠⊠⊠⊠⊠, ⊠⊠ 2 ⊠⊠⊠⊠⊠, ⊠⊠ 1 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- `quality` - 1 ⊠⊠ 100 ⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- `options` - JPEG ⊠⊠⊠⊠⊠⊠⊠ GDAL ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (ST_GDALDrivers⊠⊠ JPEG ⊠⊠⊠ create_options ⊠⊠⊠⊠⊠⊠⊠⊠). JPEG ⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠ `PROGRESSIVE` ON/OFF ⊠⊠⊠⊠⊠ 75 ⊠⊠ 0 ⊠⊠ 100 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ `QUALITY` ⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠ GDAL ⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠.

2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. GDAL 1.6.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠: ⊠⊠

Output the first three bands at 75 percent quality.

Code

```
SELECT ST_AsJPEG(rast) As rastjpg
FROM dummy_rast WHERE rid=2;
```

Output only the first band at 90 percent quality.

Code

```
SELECT ST_AsJPEG(rast, 1, 90) As rastjpg
FROM dummy_rast WHERE rid=2;
```

Output the first three bands, mapping band 2 to red, band 1 to green, and band 3 to blue, using progressive encoding at 90 percent quality.

Code

```
SELECT ST_AsJPEG(rast, ARRAY[2, 1, 3], ARRAY['QUALITY=90', 'PROGRESSIVE=ON']) As rastjpg
FROM dummy_rast WHERE rid=2;
```

☒☒

10.4☒, ST_GDALDrivers, ST_AsGDALRaster, ST_AsPNG, ST_AsTIFF

11.11.5 ST_AsPNG

ST_AsPNG — ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒ PNG(Portable Network Graphics) ☒☒☒ (☒☒☒☒☒☒) ☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒ 1 ☒, 3 ☒, ☒☒ 4 ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒ 2 ☒☒☒ 4 ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒, ☒☒ 1 ☒☒☒☒. ☒☒☒ RGB ☒☒ RGBA ☒☒☒☒☒☒☒☒☒☒☒☒.

☒☒

bytea **ST_AsPNG**(raster rast, text[] options=NULL);
bytea **ST_AsPNG**(raster rast, integer nband, integer compression);
bytea **ST_AsPNG**(raster rast, integer nband, text[] options=NULL);
bytea **ST_AsPNG**(raster rast, integer[] nbands, integer compression);
bytea **ST_AsPNG**(raster rast, integer[] nbands, text[] options=NULL);

Description

☒☒☒☒☒☒☒☒☒☒☒☒☒☒ PNG(Portable Network Graphics) ☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒ ST_AsGDALRaster ☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒, ☒☒ 3 ☒☒☒☒☒☒☒☒☒☒☒. srid ☒☒☒☒☒☒☒☒☒☒☒☒☒☒ SRID ☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒:

- nband - ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.
- nbands - ☒☒☒☒☒☒☒☒☒☒☒☒ (PNG ☒☒☒ 4 ☒☒☒☒☒☒☒☒). ☒☒☒☒☒☒☒ RGBA ☒☒☒. ☒☒☒☒☒ ARRAY[3,2,1] ☒☒☒ 3 ☒☒☒☒☒, ☒☒ 2 ☒☒☒☒☒, ☒☒ 1 ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.
- compression - 1 ☒☒ 9 ☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.
- options - PNG ☒☒☒☒☒☒☒ GDAL ☒☒☒☒☒☒☒☒☒☒☒☒ (ST_GDALDrivers☒☒ PNG ☒☒☒ create_options ☒☒☒☒☒☒☒☒). PNG ☒☒☒, ☒☒☒☒☒☒☒ ZLEVEL(☒☒☒☒☒☒☒☒☒ - ☒☒☒☒☒ 6) ☒☒☒, ☒☒☒☒ ARRAY['ZLEVEL=9'] ☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒ 2 ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒ GDAL ☒☒☒☒☒☒☒☒ ☒☒☒☒☒☒☒☒.

2.0.0 ☒☒☒☒☒☒☒☒☒☒☒☒☒☒. GDAL 1.6.0 ☒☒☒☒☒☒☒☒☒☒☒.

☒☒

Export the first 3 bands and map band 3 to red, band 1 to green, and band 2 to blue.

Code

```
SELECT ST_AsPNG(rast) As rastpng
FROM dummy_rast WHERE rid=2;
SELECT ST_AsPNG(rast, ARRAY[3, 1, 2]) As rastpng
FROM dummy_rast WHERE rid=2;
```

☒☒

ST_AsGDALRaster, ST_ColorMap, ST_GDALDrivers, 10.4☒

11.11.6 ST_AsTIFF

ST_AsTIFF — Return the raster selected bands as a single TIFF image (byte array). If no band is specified or any of specified bands does not exist in the raster, then will try to use all bands.

☒☒

bytea **ST_AsTIFF**(raster rast, text[] options=”, integer srid=sameassource);
bytea **ST_AsTIFF**(raster rast, text compression=”, integer srid=sameassource);
bytea **ST_AsTIFF**(raster rast, integer[] nbands, text compression=”, integer srid=sameassource);
bytea **ST_AsTIFF**(raster rast, integer[] nbands, text[] options, integer srid=sameassource);

Description

☒☒☒☒☒☒☒☒☒☒☒☒☒☒ TIFF(Tagged Image File Format) ☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒, ☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒ ST_AsGDALRaster ☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒ ST_AsGDALRaster ☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒ SRS ☒☒☒☒☒☒☒☒☒, ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒:

- `nbands` - ☒☒☒☒☒☒☒☒☒☒☒☒ (PNG ☒☒☒ 3 ☒☒☒☒☒☒☒☒). ☒☒☒☒☒☒☒ RGB ☒☒☒. ☒☒☒☒☒ ARRAY[3,2,1] ☒☒☒ 3 ☒☒☒☒☒, ☒☒ 2 ☒☒☒☒☒, ☒☒ 1 ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.
- `compression` - ☒☒☒☒☒: JPEG90(☒☒☒☒☒☒☒), LZW, JPEG, DEFLATE9
- `options` - GTiff ☒☒☒☒☒☒ GDAL ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒ (ST_GDALDrivers☒☒ GTiff ☒☒☒ create_options ☒☒☒☒☒☒☒☒). ☒☒☒☒☒☒☒ GDAL ☒☒☒☒☒☒☒☒ ☒☒☒☒☒☒☒☒.
- `srid` - ☒☒☒☒ spatial_ref_sys ☒ SRID ☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.

2.0.0 ☒☒☒☒☒☒☒☒☒☒☒☒. GDAL 1.6.0 ☒☒☒☒☒☒☒☒☒☒.

☒☒: JPEG ☒☒ 90%

Code

```
SELECT ST_AsTIFF(rast, 'JPEG90') As rasttiff
FROM dummy_rast WHERE rid=2;
```

☒☒

ST_GDALDrivers, ST_AsGDALRaster, ST_SRID

11.12 ☒☒☒☒☒☒☒

11.12.1 ST_Clip

ST_Clip — Returns the raster clipped by the input geometry. If band number is not specified, all bands are processed. If `crop` is not specified or TRUE, the output raster is cropped. If `touched` is set to TRUE, then touched pixels are included, otherwise only if the center of the pixel is in the geometry it is included.

☒☒

raster **ST_Clip**(raster rast, integer[] nband, geometry geom, double precision[] nodataval=NULL, boolean crop=TRUE, boolean touched=FALSE);
raster **ST_Clip**(raster rast, integer nband, geometry geom, double precision nodataval, boolean crop=TRUE, boolean touched=FALSE);
raster **ST_Clip**(raster rast, integer nband, geometry geom, boolean crop, boolean touched=FALSE);
raster **ST_Clip**(raster rast, geometry geom, double precision[] nodataval=NULL, boolean crop=TRUE, boolean touched=FALSE);
raster **ST_Clip**(raster rast, geometry geom, double precision nodataval, boolean crop=TRUE, boolean touched=FALSE);
raster **ST_Clip**(raster rast, geometry geom, boolean crop, boolean touched=FALSE);

Description

☒☒☒☒ `geom` ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒, ☒☒☒☒☒☒☒☒☒☒☒☒.

ST_Clip ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒ NODATA ☒☒☒☒☒☒☒☒☒☒☒☒. NODATA ☒☒☒ ☒☒☒☒☒☒☒☒☒☒☒☒☒☒ NODATA ☒☒☒☒☒☒☒, ☒☒☒☒☒☒☒ NODATA ☒☒ ST_MinPossibleValue(ST_BandPixelType(rast, band)) ☒☒☒☒☒☒☒. ☒☒☒☒☒☒ NODATA ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒, ☒☒☒☒☒☒☒☒ NODATA ☒☒ ☒☒☒☒☒☒ NODATA ☒☒☒☒☒☒☒. NODATA ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒, ☒☒ NODATA ☒☒☒☒☒☒☒☒. NODATA ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.

If `crop` is not specified, true is assumed meaning the output raster is cropped to the intersection of the `geom`and `rast` extents. If `crop` is set to false, the new raster gets the same extent as `rast`. If `touched` is set to true, then all pixels in the `rast` that intersect the geometry are selected.

> **Note!** ☒☒
> The default behavior is touched=false, which will only select pixels where the center of the pixel is covered by the geometry.

Enhanced: 3.5.0 - touched argument added.

2.0.0 ☒☒☒☒☒☒☒☒☒☒☒☒☒☒.

☒☒☒☒: 2.1.0 ☒☒☒☒ C ☒☒☒☒☒☒☒☒☒☒☒.

Examples here use Massachusetts aerial data available on MassGIS site MassGIS Aerial Orthos.

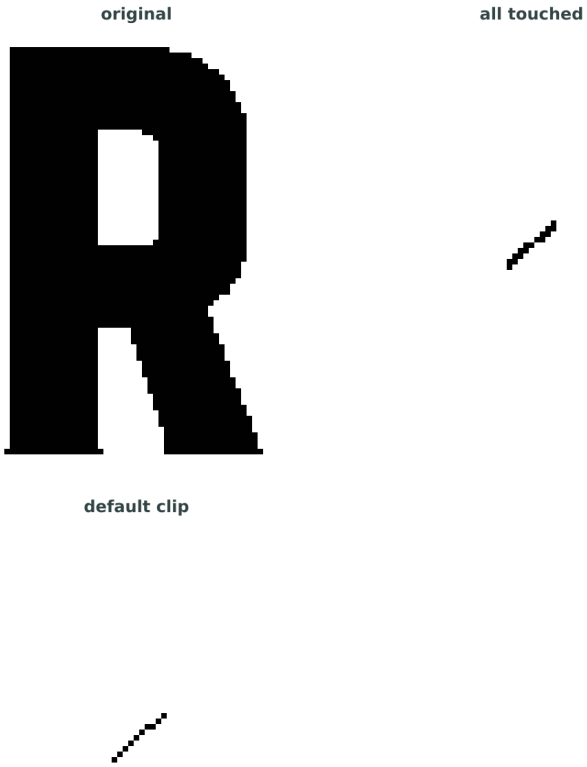
☒☒

Comparing selecting all touched vs. not all touched.

Code

```
WITH source AS (
    SELECT rast, geom
    FROM ST_AsRaster(ST_Letters('R'), scalex =
> 1.0, scaley =
> -1.0) AS r(rast)
    INNER JOIN ST_GeomFromText('LINESTRING(0 1,5 6,10 10)') AS g(geom)
        ON ST_Intersects(r.rast, g.geom)
), clipped AS (
    SELECT rast,
        ST_Clip(rast, geom, touched =
> true) AS rast_touched,
        ST_Clip(rast, geom, touched =
> false) AS rast_not_touched
    FROM source
), variants AS (
    SELECT 'original' AS title, ST_Count(rast) AS pixel_count, rast AS rendered
    FROM clipped
    UNION ALL
    SELECT 'all touched', ST_Count(rast_touched), rast_touched
    FROM clipped
    UNION ALL
    SELECT 'default clip', ST_Count(rast_not_touched), rast_not_touched
    FROM clipped
)
SELECT title, pixel_count, ST_AsPNG(rendered) AS image
FROM variants
ORDER BY CASE title
    WHEN 'original' THEN 1
    WHEN 'all touched' THEN 2
    ELSE 3
END;
```

Figure



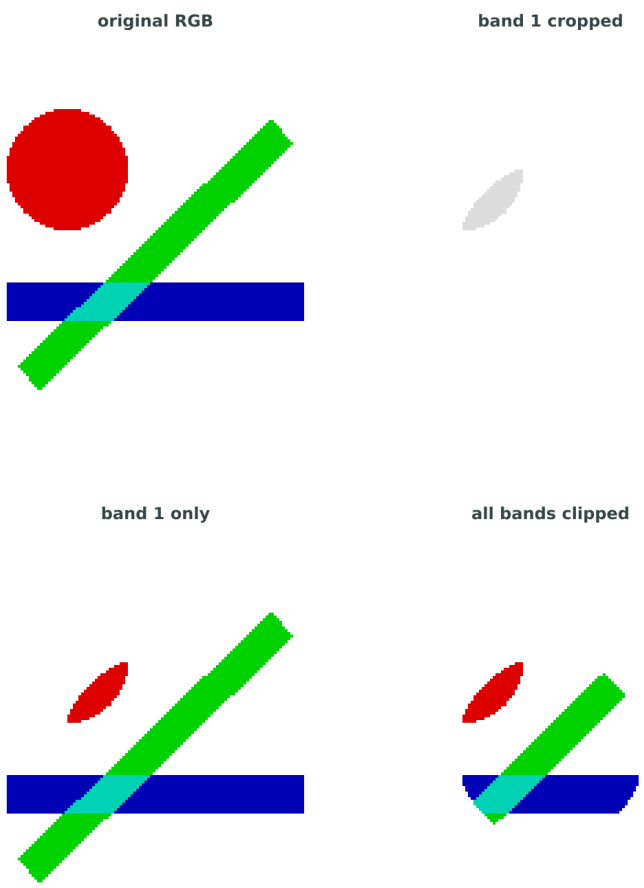
The following self-contained example uses three simple shapes as RGB bands. It compares cropping band 1, clipping band 1 without cropping and then restoring bands 2 and 3, and clipping every band without cropping.

Code

```
WITH canvas AS (
    SELECT ST_AddBand(
        ST_AddBand(
            ST_AddBand(
                ST_MakeEmptyRaster(128, 128, 0, 128, 1, -1, 0, 0, 0),
                '8BUI'::text, 0::double precision, 0::double precision
            ),
            '8BUI'::text, 0::double precision, 0::double precision
        ),
        '8BUI'::text, 0::double precision, 0::double precision
    ) AS rast
), source AS (
    SELECT ST_SetValue(
        ST_SetValue(
            ST_SetValue(rast, 1, ST_Buffer('POINT(32 96)'::geometry, 22), 220),
            2, ST_Buffer('LINESTRING(18 20,110 110)'::geometry, 6,
                         'endcap=flat join=bevel'), 210
        ),
```

```
        3, ST_Buffer('LINESTRING(10 48,118 48)'::geometry, 7,
                     'endcap=flat'), 180
    ) AS rast,
    ST_Buffer('POINT(64 64)'::geometry, 34) AS clipper
    FROM canvas
), variants AS (
    SELECT 'original RGB' AS title, rast AS rendered FROM source
    UNION ALL
    SELECT 'band 1 cropped', ST_Clip(rast, 1, clipper, true) FROM source
    UNION ALL
    SELECT
        'band 1 only',
        ST_AddBand(
            ST_Clip(rast, 1, clipper, false),
            ARRAY[ST_Band(rast, 2), ST_Band(rast, 3)]
        )
    FROM source
    UNION ALL
    SELECT 'all bands clipped', ST_Clip(rast, clipper, false) FROM source
)
SELECT title, ST_AsPNG(rendered) AS image
FROM variants
ORDER BY CASE title
    WHEN 'original RGB' THEN 1
    WHEN 'band 1 cropped' THEN 2
    WHEN 'band 1 only' THEN 3
    ELSE 4
END;
```

Figure



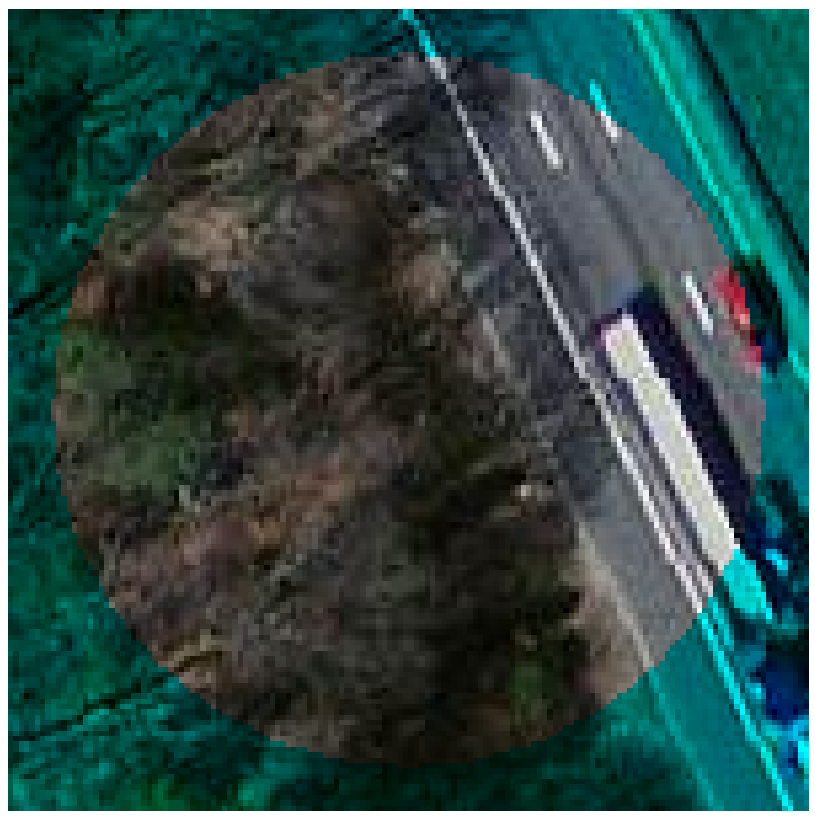
Crop, single-band clipping, and all-band clipping.



Source tile



Cropped

 *One-band source*	 *One-band clip*
 *All-band source*	 *All-band clip*

□□

ST_AddBand, ST_Count, ST_MapAlgebra (callback function version), ST_Intersection

11.12.2 ST_ColorMap

ST_ColorMap — □□□□□□□□□□□□□□ 8BUI □□ (grayscale, RGB, RGBA) □ 4 □□□□□□□□ □□□□□□□□. □□□□□□□□□□□□□□ 1 □□□□□□.

□□

raster **ST_ColorMap**(raster rast, integer nband=1, text colormap=grayscale, text method=INTERPOLATE);

raster **ST_ColorMap**(raster rast, text colormap, text method=INTERPOLATE);

Description

`rast` □ `nband` □□□□□□ `colormap` □□□□□ 8BUI □□ 4 □□□□□□□□□□□□□□□□□□□□. `colormap` □□□□□□□□□□□□□□□□□□□□□ 8BUI □□□□□□□□□□□.

nband ⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠ 1 ⊠⊠⊠⊠⊠⊠.

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ colormap ⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠⊠⊠⊠⊠⊠⊠ colormap ⊠⊠⊠:

- grayscale ⊠⊠ greyscale - 8BUI ⊠⊠⊠⊠⊠⊠⊠⊠⊠ (shades of gray) ⊠⊠⊠
- pseudocolor - 8BUI(RGBA) ⊠⊠ 4 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠
- fire - 8BUI(RGBA) ⊠⊠ 4 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠
- bluered - 8BUI(RGBA) ⊠⊠ 4 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ colormap ⊠ (⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ 5 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠, ⊠⊠, ⊠⊠⊠⊠⊠⊠⊠ (RGBA) ⊠⊠⊠ (⊠⊠⊠⊠⊠⊠⊠⊠ 0 ⊠⊠ 255 ⊠⊠⊠⊠⊠⊠⊠⊠). ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠/⊠⊠⊠⊠⊠ 0/100% ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠, ⊠, ⊠⊠, ⊠⊠⊠/⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠ NODATA ⊠⊠⊠⊠, *nv*, *null* ⊠⊠ *nodata* ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

Code

```
5 0 0 0 255
4 100:50 55 255
1 150, 100 150 255
0% 255 255 255 255
nv 0 0 0 0
```

colormap ⊠⊠⊠⊠ GDAL ⊠⊠⊠⊠⊠ (color-relief) ⊠⊠ gdaldem ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠⊠ method ⊠⊠⊠:

- INTERPOLATE - ⊠⊠.
- EXACT - ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ 0 0 0 0(RGBA) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- NEAREST - ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

> ⊠⊠
> ⊠⊠⊠⊠⊠⊠⊠⊠⊠ ColorBrewer ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

> ⊠⊠
> ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ NODATA ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. NODATA ⊠⊠⊠⊠⊠⊠⊠⊠ ST_SetBandNoDataValue ⊠⊠⊠⊠⊠⊠ NODATA ⊠⊠⊠⊠⊠⊠⊠⊠⊠.

2.1.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

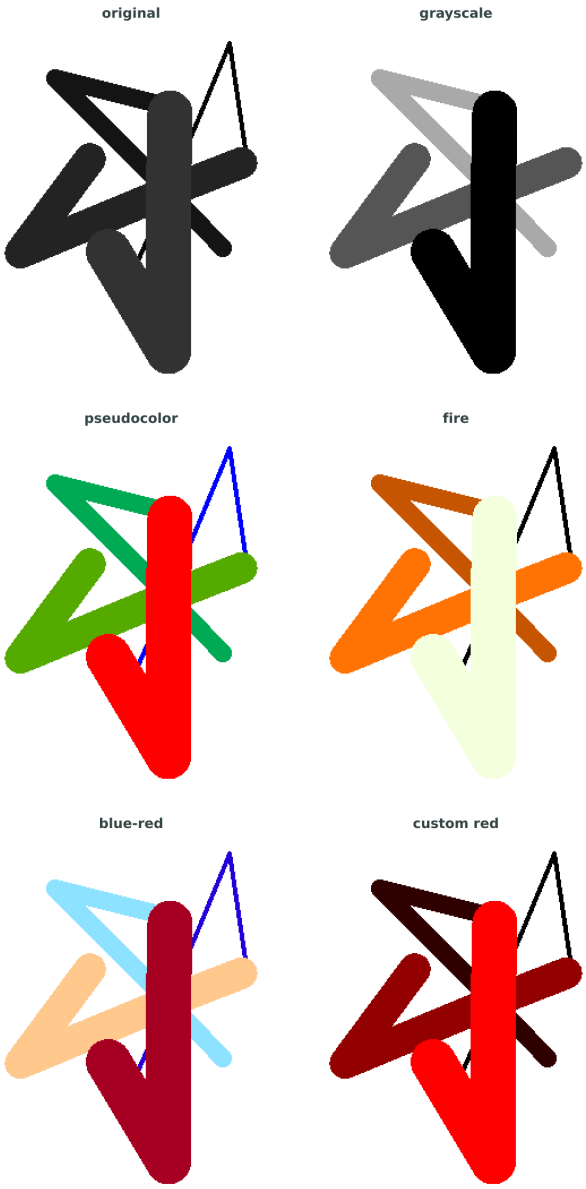
⊠⊠

This example builds a raster from rotated buffered lines and compares the original values with pre-defined and custom color maps.

Code

```
WITH ref AS (
    SELECT ST_MakeEmptyRaster(200, 200, 0, 200, 1, -1, 0, 0) AS rast
), source AS (
    SELECT ST_Union(rast) AS rast
    FROM (
    SELECT
        ST_AsRaster(ST_Rotate(ST_Buffer('LINESTRING(0 2,50 50,150 150,125 50)'::geometry,
                    i*2
                ),
                pi() * i * 0.125, ST_Point(50, 50)
            ),
            ref.rast, '8BUI'::text, i * 5
        ) AS rast
    FROM ref
    CROSS JOIN generate_series(1, 10, 3) AS i
) AS shapes
)
SELECT title, ST_AsPNG(rendered) AS image
FROM source
CROSS JOIN LATERAL (VALUES
    ('original', rast),
    ('grayscale', ST_ColorMap(rast, 1, 'greyscale')),
    ('pseudocolor', ST_ColorMap(rast, 1, 'pseudocolor')),
    ('fire', ST_ColorMap(rast, 1, 'fire')),
    ('blue-red', ST_ColorMap(rast, 1, 'bluered')),
    ('custom red', ST_ColorMap(
        rast,
        1,
        E'100% 255 0 0 255\n80% 160 0 0 255\n50% 130 0 0 255\n' ||
        E'30% 30 0 0 255\n20% 60 0 0 255\n0% 0 0 0 255\n' ||
        E'nv 255 255 255 0'
    ))
) AS variants(title, rendered);
```

Figure



□□

ST_AsPNG, ST_AsRaster ST_MapAlgebra (callback function version), ST_Grayscale ST_NumBands, ST_Reclass, ST_SetBandNoDataValue, ST_Union

11.12.3 ST_Grayscale

ST_Grayscale — Creates a new one-8BUI band raster from the source raster and specified bands representing Red, Green and Blue

□□

(1) raster **ST_Grayscale**(raster rast, integer redband=1, integer greenband=2, integer blueband=3, text extenttype=INTERSECTION);

(2) raster **ST_Grayscale**(rastbandarg[] rastbandargset, text extenttype=INTERSECTION);

Description

Create a raster with one 8BUI band given three input bands (from one or more rasters). Any input band whose pixel type is not 8BUI will be reclassified using ST_Reclass.



□□
This function is not like ST_ColorMap with the `grayscale` keyword as ST_ColorMap operates on only one band while this function expects three bands for RGB. This function applies the following equation for converting RGB to Grayscale: 0.2989 * RED + 0.5870 * GREEN + 0.1140 * BLUE

Availability: 2.5.0

□□

This self-contained example paints an apple with gradients and highlights into three raster bands. It compares the original RGB raster with the equivalent grayscale results produced by both function variants.

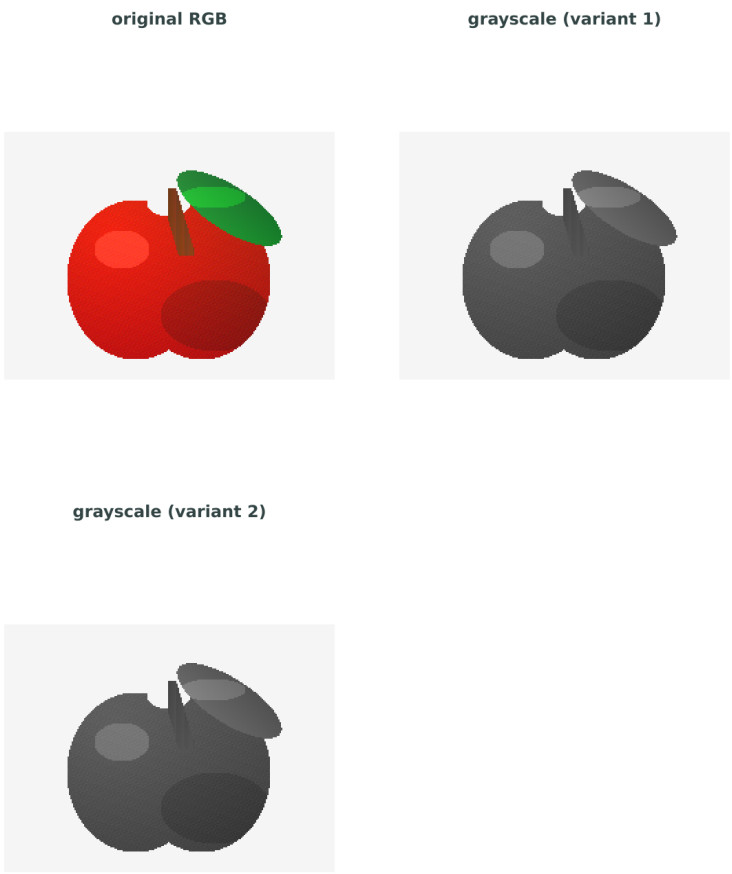
Code

```
WITH pixels AS (
    SELECT
        col,
        row,
        x,
        y,
        (((x - 48.0)^2 / (25.0^2) + (y - 54.0)^2 / (29.0^2) < 1)
         OR ((x - 72.0)^2 / (25.0^2) + (y - 54.0)^2 / (29.0^2) < 1))
         AND NOT ((x - 60.0)^2 / (8.0^2) + (y - 26.0)^2 / (5.0^2) < 1) AS fruit,
        ((x - 82.0) * 0.84 + (y - 28.0) * 0.54)^2 / (22.0^2)
          + (-(x - 82.0) * 0.54 + (y - 28.0) * 0.84)^2 / (8.0^2) < 1 AS leaf,
        abs((x - 62.0) * 0.96 - (y - 29.0) * 0.28) < 3
          AND x BETWEEN 60 AND 69 AND y BETWEEN 21 AND 45 AS stem
    FROM (
        SELECT
            col,
            row,
            col / 2.0 AS x,
            row / 2.0 AS y
        FROM generate_series(1, 180) AS row
        CROSS JOIN generate_series(1, 240) AS col
    ) AS source_pixels
), shaded AS (
    SELECT
        col,
        row,
        x,
        y,
        CASE
```

```
            WHEN stem THEN 95 + y * 0.7 + (x % 3) * 8
            WHEN leaf THEN 28 + (90 - x) * 0.5
            WHEN fruit THEN LEAST(255, GREATEST(0,
                150
                + (85 - sqrt((x - 39.0)^2 + (y - 37.0)^2)) * 1.1
                + CASE
                    WHEN (x - 43.0)^2 / (10.0^2) + (y - 43.0)^2 / (7.0^2) < 1
                    THEN 35
                    ELSE 0
                END
                - CASE
                    WHEN (x - 77.0)^2 / (20.0^2) + (y - 67.0)^2 / (13.0^2) < 1
                    THEN 35
                    ELSE 0
                END
                + ((x * 13 + y * 7) % 19 - 9)
            ))
            ELSE 245
        END AS red,
        CASE
            WHEN stem THEN 50 + y * 0.35
            WHEN leaf THEN LEAST(190, GREATEST(40,
                105 + (95 - x) * 1.2 + (y - 20) * 1.5
                + CASE
                    WHEN (x - 76.0)^2 / (12.0^2) + (y - 24.0)^2 / (4.0^2) < 1
                    THEN 45
                    ELSE 0
                END
            ))
            WHEN fruit THEN LEAST(95, GREATEST(15,
                24 + (68 - y) * 0.35
                + CASE
                    WHEN (x - 43.0)^2 / (10.0^2) + (y - 43.0)^2 / (7.0^2) < 1
                    THEN 30
                    ELSE 0
                END
                + ((x * 5 + y * 11) % 13 - 6)
            ))
            ELSE 245
        END AS green,
        CASE
            WHEN stem THEN 22 + (x % 4) * 3
            WHEN leaf THEN 42 + (100 - x) * 0.4
            WHEN fruit THEN LEAST(75, GREATEST(10,
                18
                + CASE
                    WHEN (x - 43.0)^2 / (10.0^2) + (y - 43.0)^2 / (7.0^2) < 1
                    THEN 25
                    ELSE 0
                END
                + ((x * 3 + y * 5) % 11 - 5)
            ))
            ELSE 245
        END AS blue
    FROM pixels
), rows AS (
    SELECT
        row,
        array_agg(red::double precision ORDER BY col) AS red_values,
        array_agg(green::double precision ORDER BY col) AS green_values,
        array_agg(blue::double precision ORDER BY col) AS blue_values
    FROM shaded
```

```
    GROUP BY row
), source AS (
    SELECT ST_SetValues(ST_SetValues(ST_SetValues(
        ST_AddBand(ST_AddBand(ST_AddBand(
            ST_MakeEmptyRaster(240, 180, 0, 180, 1, -1, 0, 0, 0),
            '8BUI'::text, 245::double precision, 0::double precision),
            '8BUI'::text, 245::double precision, 0::double precision),
            '8BUI'::text, 245::double precision, 0::double precision),
        1, 1, 1, array_agg(red_values ORDER BY row)::double precision[][]),
        2, 1, 1, array_agg(green_values ORDER BY row)::double precision[][]),
        3, 1, 1, array_agg(blue_values ORDER BY row)::double precision[][]) AS rast
    FROM rows
)
SELECT ST_AsPNG(rast) AS "original RGB",
    ST_AsPNG(ST_Grayscale(rast)) AS "grayscale (variant 1)",
    ST_AsPNG(ST_Grayscale(ARRAY[
        ROW(rast, 1)::rastbandarg,
        ROW(rast, 2)::rastbandarg,
        ROW(rast, 3)::rastbandarg
    ]::rastbandarg[])) AS "grayscale (variant 2)"
FROM source;
```

Figure



Original photographic illustration:



original apple



grayscale apple

⊠⊠

ST_AsPNG, ST_Reclass, ST_ColorMap

11.12.4 ST_Intersection

ST_Intersection — ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠-⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

setof geomval **ST_Intersection**(geometry geom, raster rast, integer band_num=1);
setof geomval **ST_Intersection**(raster rast, geometry geom);
setof geomval **ST_Intersection**(raster rast, integer band, geometry geomin);
raster **ST_Intersection**(raster rast1, raster rast2, double precision[] nodataval);
raster **ST_Intersection**(raster rast1, raster rast2, text returnband, double precision[] nodataval);
raster **ST_Intersection**(raster rast1, integer band1, raster rast2, integer band2, double precision[] nodataval);
raster **ST_Intersection**(raster rast1, integer band1, raster rast2, integer band2, text returnband, double precision[] nodataval);

Description

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠-⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠.

geomval ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠ (ST_DumpAsPolygon ⊠⊠⊠) ⊠ ⊠⊠⊠ geomval ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠ ST_Intersection(geometry, geometry) PostGIS ⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠ NODATA ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. WHERE ⊠⊠ ST_Intersect ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠⊠⊠ geomval ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠’.geom’ ⊠⊠’.val’ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠. ⊠: (ST_Intersection(rast, geom)).geom

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ST_MapAlgebraExpr ⊠⊠⊠⊠⊠⊠ 2 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠ `returnband` ⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠’BAND1’, ’BAND2’ ⊠⊠’BOTH’ ⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠ NODATA ⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ NODATA ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠, NODATA ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ NODATA ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

ST_Intersection ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ NODATA ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠ ⊠’BAND1’, ’BAND2’ ⊠⊠’BOTH’ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ 1 ⊠⊠⊠ 2 ⊠⊠ NODATA ⊠⊠⊠⊠ `nodataval[]` ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ NODATA ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠ NODATA ⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ NODATA ⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠ NODATA ⊠⊠⊠⊠⊠⊠⊠⊠ NODATA ⊠⊠⊠⊠⊠⊠⊠⊠⊠, ST_MinPossibleValue ⊠⊠⊠⊠⊠ NODATA ⊠⊠⊠⊠⊠⊠⊠⊠. NODATA ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠.

⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ 1 ⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ST_Clip ⊠⊠⊠⊠⊠⊠⊠⊠.

> ⊠⊠
> For more control over the result extent or NODATA handling, use the two-raster variant of ST_MapAlgebraExpr. To compute a raster-space intersection with a geometry, use ST_Clip. Use this function together with ST_Intersects and an index on the raster or geometry column when filtering table data.

⊠⊠⊠⊠: 2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. 2.0.0 ⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠: ⊠⊠, ⊠⊠⊠ -- ⊠⊠-⊠⊠⊠⊠⊠⊠

Code

```
SELECT
    foo.rid,
    foo.gid,
    GeometryType((foo.geomval).geom) AS geom_type,
    (foo.geomval).val AS value
FROM (
    SELECT
        A.rid,
        g.gid,
        ST_Intersection(A.rast, g.geom) As geomval
    FROM dummy_rast AS A
    CROSS JOIN (
        VALUES
            (1, ST_Point(3427928, 5793243.85) ),
            (2, 'LINESTRING(3427927.85 5793243.75,3427927.8 5793243.75,3427927.8 5793243.8) ↩
    '::geometry),
            (3, 'LINESTRING(1 2,3 4)'::geometry)
    ) As g(gid, geom)
    WHERE A.rid = 2
) AS foo;
```

Output

```
rid | gid |      geom_type       | value
-----+-----+---------------------+-------
   2 |   1 | POINT               |   249
   2 |   1 | POINT               |   253
   2 |   2 | POINT               |   254
   2 |   2 | POINT               |   251
   2 |   2 | POINT               |   253
```

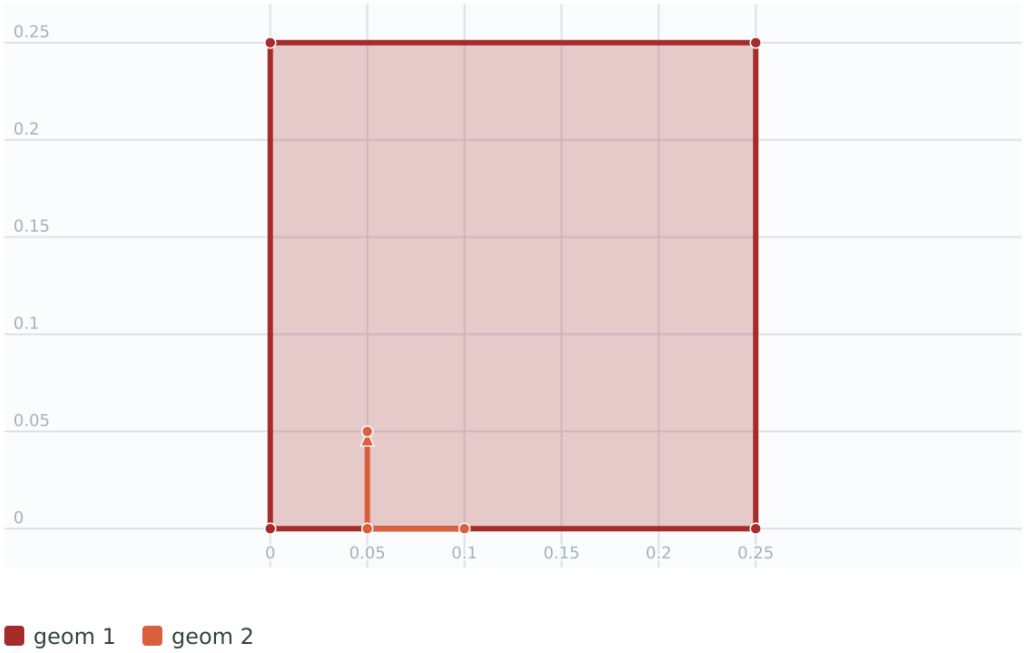
```
   2 |   2 | LINESTRING          |   252
   2 |   2 | MULTILINESTRING     |   250
   2 |   3 | GEOMETRYCOLLECTION  |
```

The same line example rendered in a common coordinate frame. The raster footprint and complete input line remain visible behind the returned intersection pieces.

Code

```
WITH source AS (
    SELECT
        rast,
        ST_GeomFromText(
            'LINESTRING(3427927.85 5793243.75,'
            '3427927.8 5793243.75,3427927.8 5793243.8)'
        ) AS input_geometry
    FROM dummy_rast
    WHERE rid = 2
), intersection AS (
SELECT
    rast,
    input_geometry,
    ST_Collect(gv.geom) AS result_geometry
FROM source
CROSS JOIN LATERAL ST_Intersection(rast, input_geometry) AS gv
GROUP BY rast, input_geometry
), layers AS (
    SELECT
        'raster' AS title,
        ST_Translate(ST_Envelope(rast), -3427927.75, -5793243.75) AS geom
    FROM intersection
    UNION ALL
    SELECT
        'input',
        ST_Translate(input_geometry, -3427927.75, -5793243.75)
    FROM intersection
    UNION ALL
    SELECT
        'intersection',
        ST_Translate(result_geometry, -3427927.75, -5793243.75)
    FROM intersection
)
SELECT title, geom AS geom
FROM layers
ORDER BY CASE title
    WHEN 'raster' THEN 1
    WHEN 'input' THEN 2
    ELSE 3
END;
```

Figure



□□

geomval, ST_Intersects, ST_MapAlgebraExpr, ST_Clip, ST_AsText

11.12.5 ST_MapAlgebra (callback function version)

ST_MapAlgebra (callback function version) — □□□□□□ - □□□ 1 □□□, □□□□□, □□□□□□□ □□□□□ 1 □□□□□□□□□ 1 □□□□□□□□□□□□□□.

□□

raster **ST_MapAlgebra**(rastbandarg[] rastbandargset, regprocedure callbackfunc, text pixeltype=NULL, text extenttype=INTERSECTION, raster customextent=NULL, integer distancex=0, integer distancey=0, text[] VARIADIC userargs=NULL);
raster **ST_MapAlgebra**(raster rast, integer[] nband, regprocedure callbackfunc, text pixeltype=NULL, text extenttype=FIRST, raster customextent=NULL, integer distancex=0, integer distancey=0, text[] VARIADIC userargs=NULL);
raster **ST_MapAlgebra**(raster rast, integer nband, regprocedure callbackfunc, text pixeltype=NULL, text extenttype=FIRST, raster customextent=NULL, integer distancex=0, integer distancey=0, text[] VARIADIC userargs=NULL);
raster **ST_MapAlgebra**(raster rast1, integer nband1, raster rast2, integer nband2, regprocedure callbackfunc, text pixeltype=NULL, text extenttype=INTERSECTION, raster customextent=NULL, integer distancex=0, integer distancey=0, text[] VARIADIC userargs=NULL);
raster **ST_MapAlgebra**(raster rast, integer nband, regprocedure callbackfunc, float8[] mask, boolean weighted, text pixeltype=NULL, text extenttype=INTERSECTION, raster customextent=NULL, text[] VARIADIC userargs=NULL);

Description

⊠⊠⊠ 1 ⊠⊠⊠, ⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ 1 ⊠⊠⊠⊠⊠⊠⊠⊠ 1 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

rast,rast1,rast2, rastbandargset ⊠⊠⊠ (代數) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠

rastbandargset ⊠⊠⊠⊠⊠⊠⊠⊠⊠/⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠ 1 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

nband, nband1, nband2 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. nband ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. nband1 ⊠ rast1 ⊠⊠⊠⊠⊠⊠⊠⊠ nband2 ⊠⊠⊠⊠ 2 ⊠/⊠⊠ 2 ⊠⊠⊠⊠ rast2 ⊠⊠⊠⊠⊠⊠⊠⊠.

callbackfunc The `callbackfunc` parameter must be the name and signature of an SQL or PL/pgSQL function, cast to a regprocedure. An example PL/pgSQL function example is:

Code

```
CREATE OR REPLACE FUNCTION sample_callbackfunc(value double precision[][][], pos ↵
    integer[][], VARIADIC userargs text[])
    RETURNS double precision
    AS $$
    BEGIN
        RETURN 0;
    END;
    $$ LANGUAGE 'plpgsql' IMMUTABLE;
```

The `callbackfunc` must have two or three arguments: a 3-dimension double precision array, a 2-dimension integer array and, optionally, a variadic 1-dimension text array. The first argument `value` is the set of values (as double precision) from all input rasters. The three dimensions (where indexes are 1-based) are: raster #, row y, column x. The second argument `position` is the set of pixel positions from the output raster and input rasters. The outer dimension (where indexes are 0-based) is the raster #. The position at outer dimension index 0 is the output raster's pixel position. For each outer dimension, there are two elements in the inner dimension for X and Y. The optional third argument `userargs` is for passing through any user-specified arguments.

The callback return value is used as the value of the output pixel. If the callback returns `NULL`, the output pixel is set to NODATA when the output band has a NODATA value. If the output band has no NODATA value, a `NULL` callback result raises an error.

Passing a regprocedure argument to a SQL function requires the full function signature to be passed, then cast to a regprocedure type. To pass the above example PL/pgSQL function as an argument, the SQL for the argument is:

Code

```
'sample_callbackfunc(double precision[], integer[], text[])'::regprocedure
```

Note that PostgreSQL array types do not encode dimensions in function signatures, so multidimensional callback arguments are written as double precision[] and integer[] in the regprocedure signature.

mask An n-dimensional array (matrix) of numbers used to filter what cells get passed to map algebra call-back function. 0 means a neighbor cell value should be treated as no-data and 1 means value should be treated as data. If weight is set to true, then the values, are used as multipliers to multiple the pixel value of that value in the neighborhood position.

weighted mask ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠) ⊠⊠ (mask ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠) ⊠⊠⊠⊠⊠⊠⊠ (⊠/⊠⊠) ⊠⊠⊠.

pixeltype pixeltype ⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. pixeltype ⊠ NULL ⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (⊠⊠⊠⊠⊠⊠⊠⊠: INTERSECTION, UNION, FIRST, CUSTOM), ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (⊠⊠⊠⊠⊠⊠⊠⊠: SECOND, LAST) ⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠ pixeltype ⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ST_BandPixelType ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠, NULL ⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠.

extenttype ⊠⊠⊠⊠⊠⊠⊠⊠ INTERSECTION(⊠⊠⊠), UNION, FIRST(⊠⊠⊠ 1 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠), SECOND, LAST, CUSTOM ⊠⊠⊠.

customextent extentype ⊠ CUSTOM ⊠⊠⊠, ⊠⊠⊠⊠ customextent ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠ 1 ⊠ 4 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

distancex The distance in pixels from the reference cell in x direction. So width of resulting matrix would be `2*distancex + 1`.If not specified only the reference cell is considered (neighborhood of 0).

distancey ⊠⊠⊠⊠⊠⊠⊠⊠ Y ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ `2*distancey + 1` ⊠⊠⊠⊠ ⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (⊠⊠⊠⊠⊠⊠⊠⊠ 0) ⊠⊠⊠⊠⊠⊠.

userargs If the `callbackfunc` declares a third argument, that argument is a variadic text array. All trailing text arguments are passed through to the specified `callbackfunc`, and are contained in the `userargs` argument.

> **Note**
> ⊠⊠
> (⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠) VARIADIC ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, PostgreSQL ⊠⊠⊠ ⊠⊠ Query Language (SQL) Functions ⊠"SQL Functions with Variable Numbers of Arguments" ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠ 1 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠/⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ rastbandarg ⊠⊠⊠⊠⊠⊠ ⊠⊠⊠. ⊠⊠ 1 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠ 2 ⊠ 3 ⊠⊠⊠⊠⊠⊠⊠ 1 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠ 2 ⊠ 3 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠ 4 ⊠⊠⊠⊠⊠⊠⊠⊠⊠ 1 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ 2 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠ 4 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

2.2.0 ⊠⊠⊠⊠⊠ mask ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

2.1.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

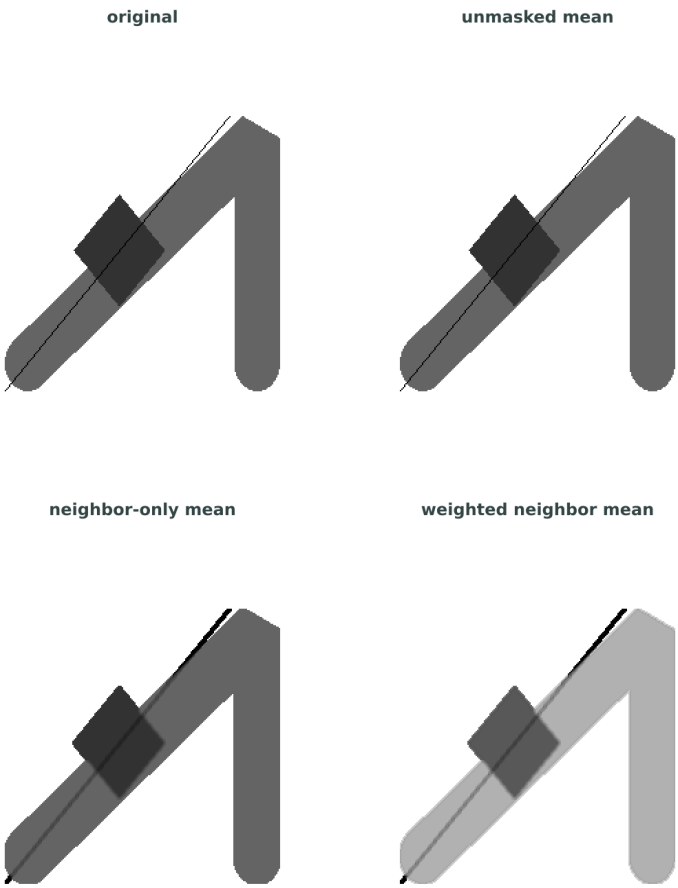
Compare an unmasked mean with neighbor-only and weighted-neighbor masks. The weighted mask multiplies neighboring values by 2 before computing the result.

Code

```
WITH foo AS (
    SELECT ST_SetBandNoDataValue(
        ST_SetValue(
            ST_SetValue(
                ST_AsRaster(
                    ST_Buffer('LINESTRING(50 50,100 90,100 50)'::geometry, 5, 'join=bevel') ↩
    ,
                    200, 200, ARRAY['8BUI'], ARRAY[100], ARRAY[0]
                ),
                ST_Buffer('POINT(70 70)'::geometry, 10, 'quad_segs=1'),
                50
            ),
```

```
            'LINESTRING(20 20,100 100,150 98)'::geometry,
            1
        ),
        0
    ) AS rast
), variants AS (
    SELECT 'original' AS title, rast
    FROM foo
    UNION ALL
    SELECT 'unmasked mean' AS title,
        ST_MapAlgebra(rast, 1, 'ST_Mean4ma(double precision[], int[], text[])':: ↩
    regprocedure) AS rast
    FROM foo
    UNION ALL
    SELECT 'neighbor-only mean' AS title,
        ST_MapAlgebra(
            rast,
            1,
            'ST_Mean4ma(double precision[], int[], text[])'::regprocedure,
            '{{1,1,1}, {1,0,1}, {1,1,1}}'::double precision[],
            false
        ) AS rast
    FROM foo
    UNION ALL
    SELECT 'weighted neighbor mean' AS title,
        ST_MapAlgebra(
            rast,
            1,
            'ST_Mean4ma(double precision[], int[], text[])'::regprocedure,
            '{{2,2,2}, {2,0,2}, {2,2,2}}'::double precision[],
            true
        ) AS rast
    FROM foo
)
SELECT title, ST_AsPNG(rast) AS image
FROM variants;
```

Figure



Variant 1.

□□□ 1 □, □□ 1 □

Code

```
WITH foo AS (
    SELECT 1 AS rid, ST_AddBand(ST_MakeEmptyRaster(2, 2, 0, 0, 1, -1, 0, 0, 0), 1, '16BUI', ↩
     1, 0) AS rast
)
SELECT
    ST_MapAlgebra(ARRAY[ROW(rast, 1)]::rastbandarg[],
        'sample_callbackfunc(double precision[], int[], text[])'::regprocedure
    ) AS rast
FROM foo
```

□□□ 1 □, □□□□

Code

```
WITH foo AS (
    SELECT 1 AS rid, ST_AddBand(ST_AddBand(ST_AddBand(ST_MakeEmptyRaster(2, 2, 0, 0, 1, -1, ↩
     0, 0, 0), 1, '16BUI', 1, 0), 2, '8BUI', 10, 0), 3, '32BUI', 100, 0) AS rast
)
```

```
SELECT
    ST_MapAlgebra(ARRAY[ROW(rast, 3), ROW(rast, 1), ROW(rast, 3), ROW(rast, 2)]:: ↩
    rastbandarg[],
        'sample_callbackfunc(double precision[], int[], text[])'::regprocedure
    ) AS rast
FROM foo
```

□□□□□, □□□□

Code

```
WITH foo AS (
    SELECT 1 AS rid, ST_AddBand(ST_AddBand(ST_AddBand(ST_MakeEmptyRaster(2, 2, 0, 0, 1, -1, ↩
     0, 0, 0), 1, '16BUI', 1, 0), 2, '8BUI', 10, 0), 3, '32BUI', 100, 0) AS rast UNION ALL
    SELECT 2 AS rid, ST_AddBand(ST_AddBand(ST_AddBand(ST_MakeEmptyRaster(2, 2, 0, 1, 1, -1, ↩
     0, 0, 0), 1, '16BUI', 2, 0), 2, '8BUI', 20, 0), 3, '32BUI', 300, 0) AS rast
)
SELECT
    ST_MapAlgebra(ARRAY[ROW(t1.rast, 3), ROW(t2.rast, 1), ROW(t2.rast, 3), ROW(t1.rast, 2) ↩
    ]::rastbandarg[],
        'sample_callbackfunc(double precision[], int[], text[])'::regprocedure
    ) AS rast
FROM foo t1
CROSS JOIN foo t2
WHERE t1.rid = 1
    AND t2.rid = 2
```

Complete example of tiles of a coverage with neighborhood.

Code

```
WITH foo AS (
    SELECT 0 AS rid, ST_AddBand(ST_MakeEmptyRaster(2, 2, 0, 0, 1, -1, 0, 0, 0), 1, '16BUI', ↩
     1, 0) AS rast UNION ALL
    SELECT 1, ST_AddBand(ST_MakeEmptyRaster(2, 2, 2, 0, 1, -1, 0, 0, 0), 1, '16BUI', 2, 0)  ↩
    AS rast UNION ALL
    SELECT 2, ST_AddBand(ST_MakeEmptyRaster(2, 2, 4, 0, 1, -1, 0, 0, 0), 1, '16BUI', 3, 0)  ↩
    AS rast UNION ALL

    SELECT 3, ST_AddBand(ST_MakeEmptyRaster(2, 2, 0, -2, 1, -1, 0, 0, 0), 1, '16BUI', 10,  ↩
    0) AS rast UNION ALL
    SELECT 4, ST_AddBand(ST_MakeEmptyRaster(2, 2, 2, -2, 1, -1, 0, 0, 0), 1, '16BUI', 20,  ↩
    0) AS rast UNION ALL
    SELECT 5, ST_AddBand(ST_MakeEmptyRaster(2, 2, 4, -2, 1, -1, 0, 0, 0), 1, '16BUI', 30,  ↩
    0) AS rast UNION ALL

    SELECT 6, ST_AddBand(ST_MakeEmptyRaster(2, 2, 0, -4, 1, -1, 0, 0, 0), 1, '16BUI', 100,  ↩
    0) AS rast UNION ALL
    SELECT 7, ST_AddBand(ST_MakeEmptyRaster(2, 2, 2, -4, 1, -1, 0, 0, 0), 1, '16BUI', 200,  ↩
    0) AS rast UNION ALL
    SELECT 8, ST_AddBand(ST_MakeEmptyRaster(2, 2, 4, -4, 1, -1, 0, 0, 0), 1, '16BUI', 300,  ↩
    0) AS rast
)
SELECT
    t1.rid,
    ST_MapAlgebra(ARRAY[ROW(ST_Union(t2.rast), 1)]::rastbandarg[],
        'sample_callbackfunc(double precision[], int[], text[])'::regprocedure,
        '32BUI',
        'CUSTOM', t1.rast,
        1, 1
```

```
    ) AS rast
FROM foo t1
CROSS JOIN foo t2
WHERE t1.rid = 4
    AND t2.rid BETWEEN 0 AND 8
    AND ST_Intersects(t1.rast, t2.rast)
GROUP BY t1.rid, t1.rast
```

Example like the prior one for tiles of a coverage with neighborhood but using only basic SQL features.

Code

```
WITH src AS (
    SELECT 0 AS rid, ST_AddBand(ST_MakeEmptyRaster(2, 2, 0, 0, 1, -1, 0, 0, 0), 1, '16BUI', ↩
     1, 0) AS rast UNION ALL
    SELECT 1, ST_AddBand(ST_MakeEmptyRaster(2, 2, 2, 0, 1, -1, 0, 0, 0), 1, '16BUI', 2, 0)  ↩
    AS rast UNION ALL
    SELECT 2, ST_AddBand(ST_MakeEmptyRaster(2, 2, 4, 0, 1, -1, 0, 0, 0), 1, '16BUI', 3, 0)  ↩
    AS rast UNION ALL

    SELECT 3, ST_AddBand(ST_MakeEmptyRaster(2, 2, 0, -2, 1, -1, 0, 0, 0), 1, '16BUI', 10,  ↩
    0) AS rast UNION ALL
    SELECT 4, ST_AddBand(ST_MakeEmptyRaster(2, 2, 2, -2, 1, -1, 0, 0, 0), 1, '16BUI', 20,  ↩
    0) AS rast UNION ALL
    SELECT 5, ST_AddBand(ST_MakeEmptyRaster(2, 2, 4, -2, 1, -1, 0, 0, 0), 1, '16BUI', 30,  ↩
    0) AS rast UNION ALL

    SELECT 6, ST_AddBand(ST_MakeEmptyRaster(2, 2, 0, -4, 1, -1, 0, 0, 0), 1, '16BUI', 100,  ↩
    0) AS rast UNION ALL
    SELECT 7, ST_AddBand(ST_MakeEmptyRaster(2, 2, 2, -4, 1, -1, 0, 0, 0), 1, '16BUI', 200,  ↩
    0) AS rast UNION ALL
    SELECT 8, ST_AddBand(ST_MakeEmptyRaster(2, 2, 4, -4, 1, -1, 0, 0, 0), 1, '16BUI', 300,  ↩
    0) AS rast
), foo AS (
    SELECT
        t1.rid,
        ST_Union(t2.rast) AS rast
    FROM src t1
    JOIN src t2
        ON ST_Intersects(t1.rast, t2.rast)
        AND t2.rid BETWEEN 0 AND 8
    WHERE t1.rid = 4
    GROUP BY t1.rid
), bar AS (
    SELECT
        t1.rid,
        ST_MapAlgebra(
            ARRAY[ROW(t2.rast, 1)]::rastbandarg[],
            'sample_callbackfunc(double precision[], int[], text[])'::regprocedure,
            '32BUI',
            'CUSTOM', t1.rast,
            1, 1
        ) AS rast
    FROM src t1
    JOIN foo t2
        ON t1.rid = t2.rid
)
SELECT
    rid,
    (ST_MetaData(rast)),
    (ST_BandMetaData(rast, 1)),
```

```
    ST_Value(rast, 1, 1, 1)
FROM bar;
```

Variants 2 and 3.

□□□ 1 □, □□□□

Code

```
WITH foo AS (
    SELECT 1 AS rid, ST_AddBand(ST_AddBand(ST_AddBand(ST_MakeEmptyRaster(2, 2, 0, 0, 1, -1, ↩
     0, 0, 0), 1, '16BUI', 1, 0), 2, '8BUI', 10, 0), 3, '32BUI', 100, 0) AS rast
)
SELECT
    ST_MapAlgebra(
        rast,
        ARRAY[3, 1, 3, 2]::integer[],
        'sample_callbackfunc(double precision[], int[], text[])'::regprocedure
    ) AS rast
FROM foo
```

□□□ 1 □, □□ 1 □

Code

```
WITH foo AS (
    SELECT 1 AS rid, ST_AddBand(ST_AddBand(ST_AddBand(ST_MakeEmptyRaster(2, 2, 0, 0, 1, -1, ↩
     0, 0, 0), 1, '16BUI', 1, 0), 2, '8BUI', 10, 0), 3, '32BUI', 100, 0) AS rast
)
SELECT
    ST_MapAlgebra(
        rast,
        2,
        'sample_callbackfunc(double precision[], int[], text[])'::regprocedure
    ) AS rast
FROM foo
```

Variant 4.

□□□ 2 □, □□ 2 □

Code

```
WITH foo AS (
    SELECT 1 AS rid, ST_AddBand(ST_AddBand(ST_AddBand(ST_MakeEmptyRaster(2, 2, 0, 0, 1, -1, ↩
     0, 0, 0), 1, '16BUI', 1, 0), 2, '8BUI', 10, 0), 3, '32BUI', 100, 0) AS rast UNION ALL
    SELECT 2 AS rid, ST_AddBand(ST_AddBand(ST_AddBand(ST_MakeEmptyRaster(2, 2, 0, 1, 1, -1, ↩
     0, 0, 0), 1, '16BUI', 2, 0), 2, '8BUI', 20, 0), 3, '32BUI', 300, 0) AS rast
)
SELECT
    ST_MapAlgebra(
        t1.rast,
        2,
        t2.rast,
        1,
        'sample_callbackfunc(double precision[], int[], text[])'::regprocedure
    ) AS rast
FROM foo t1
CROSS JOIN foo t2
WHERE t1.rid = 1
```

```
    AND t2.rid = 2
```

⊠⊠

rastbandarg, ST_Union, ST_MapAlgebra (expression version)

11.12.6 ST_MapAlgebra (expression version)

ST_MapAlgebra (expression version) — ⊠⊠⊠⊠⊠ - ⊠⊠⊠⊠⊠ 1 ⊠⊠⊠ 2 ⊠, ⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠ ⊠ SQL ⊠⊠⊠ 1 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ 1 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

raster **ST_MapAlgebra**(raster rast, integer nband, text pixeltype, text expression, double precision nodataval=NULL);
raster **ST_MapAlgebra**(raster rast, text pixeltype, text expression, double precision nodataval=NULL);
raster **ST_MapAlgebra**(raster rast1, integer nband1, raster rast2, integer nband2, text expression, text pixeltype=NULL, text extenttype=INTERSECTION, text nodata1expr=NULL, text nodata2expr=NULL, double precision nodatanodataval=NULL);
raster **ST_MapAlgebra**(raster rast1, raster rast2, text expression, text pixeltype=NULL, text extenttype=INTERSECTION, text nodata1expr=NULL, text nodata2expr=NULL, double precision nodatanodataval=NULL);

Description

⊠⊠⊠⊠⊠ - ⊠⊠⊠⊠⊠ 1 ⊠⊠⊠ 2 ⊠, ⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠ SQL ⊠⊠⊠ 1 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ 1 ⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

2.1.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠: ⊠⊠ 1, 2 (⊠⊠⊠ 1 ⊠)

⊠⊠⊠⊠⊠ (rast) ⊠⊠⊠ expression ⊠⊠⊠⊠⊠⊠⊠⊠ PostgreSQL ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠ 1 ⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. nband ⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠ 1 ⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠ 1 ⊠⊠⊠⊠⊠⊠⊠⊠.

pixeltype ⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. pixeltype ⊠ NULL ⊠⊠⊠, ⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠ rast ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

- expression ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
  1. [rast] - ⊠⊠⊠⊠⊠⊠⊠⊠
  2. [rast.val] - ⊠⊠⊠⊠⊠⊠⊠⊠
  3. [rast.x] - ⊠⊠⊠⊠⊠ 1-⊠⊠⊠⊠⊠
  4. [rast.y] - ⊠⊠⊠⊠⊠ 1-⊠⊠⊠⊠⊠

☒☒: ☒☒ 3, 4 (☒☒☒ 2 ☒)

☒☒☒☒☒☒☒☒ rast1, (rast2) ☒☒☒ expression ☒☒☒☒☒☒☒☒ 2 ☒☒☒☒, ☒☒☒ PostgreSQL ☒☒☒☒☒☒☒☒☒☒☒☒☒, ☒☒ 1 ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒. band1, band2 ☒☒☒☒☒☒☒☒☒☒, ☒☒ 1 ☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒ (☒☒, ☒☒☒☒☒☒☒☒☒☒☒) ☒☒☒☒☒☒☒☒. extenttype ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.

expression ☒☒☒ 2 ☒☒☒☒☒ PostgreSQL ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒ PostgreSQL ☒☒☒☒☒/☒☒☒☒☒☒☒. ☒: (([rast1] + [rast2])/2.0)::integer

pixeltype ☒☒☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒ ST_BandPixelType ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒, ☒☒☒☒☒☒, NULL ☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒ NULL ☒☒☒☒☒☒☒☒☒, ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.

extenttype ☒☒☒☒☒☒☒☒☒☒☒

1. INTERSECTION - ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒.
2. UNION - ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.
3. FIRST - ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.
4. SECOND - ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.

nodata1expr rast1 ☒☒☒☒☒☒ NODATA ☒☒☒☒☒☒☒☒☒☒☒☒☒ rast2 ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒, ☒☒ rast2 ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.

nodata2expr rast2 ☒☒☒☒☒☒ NODATA ☒☒☒☒☒☒☒☒☒☒☒☒☒ rast1 ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒, ☒☒ rast1 ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.

nodatanodataval ☒☒☒☒☒☒☒☒☒☒ rast1 ☒ rast2 ☒☒☒☒☒☒☒☒ NODATA ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.

- expression, nodata1expr ☒ nodata2expr ☒☒☒☒☒☒☒☒☒☒☒☒.

1. [rast1] - rast1 ☒☒☒☒☒☒☒☒☒☒☒
2. [rast1.val] - rast1 ☒☒☒☒☒☒☒☒☒☒☒
3. [rast1.x] - rast1 ☒☒☒☒☒☒☒☒ 1-☒☒☒☒☒
4. [rast1.y] - rast1 ☒☒☒☒☒☒☒☒ 1-☒☒☒☒☒
5. [rast2] - rast2 ☒☒☒☒☒☒☒☒☒☒☒
6. [rast2.val] - rast2 ☒☒☒☒☒☒☒☒☒☒☒
7. [rast2.x] - rast2 ☒☒☒☒☒☒☒☒ 1-☒☒☒☒☒
8. [rast2.y] - rast2 ☒☒☒☒☒☒☒☒ 1-☒☒☒☒☒

☒☒

Variants 1 and 2.

Code

```
WITH foo AS (
    SELECT ST_AddBand(ST_MakeEmptyRaster(10, 10, 0, 0, 1, 1, 0, 0, 0), '32BF'::text, 1, -1) ↩
     AS rast
)
SELECT
    ST_MapAlgebra(rast, 1, NULL, 'ceil([rast]*[rast.x]/[rast.y]+[rast.val])')
FROM foo;
```

Variant 3 and 4.

Code

```
WITH foo AS (
    SELECT 1 AS rid, ST_AddBand(ST_AddBand(ST_AddBand(ST_MakeEmptyRaster(2, 2, 0, 0, 1, -1, ↵
     0, 0, 0), 1, '16BUI', 1, 0), 2, '8BUI', 10, 0), 3, '32BUI'::text, 100, 0) AS rast UNION ↵
     ALL
    SELECT 2 AS rid, ST_AddBand(ST_AddBand(ST_AddBand(ST_MakeEmptyRaster(2, 2, 0, 1, 1, -1, ↵
     0, 0, 0), 1, '16BUI', 2, 0), 2, '8BUI', 20, 0), 3, '32BUI'::text, 300, 0) AS rast
)
SELECT
    ST_MapAlgebra(
        t1.rast,
        2,
        t2.rast,
        1,
        '([rast2] + [rast1.val]) / 2'
    ) AS rast
FROM foo t1
CROSS JOIN foo t2
WHERE t1.rid = 1
    AND t2.rid = 2;
```

⊠⊠

rastbandarg, ST_Union, ST_MapAlgebra (callback function version), 10.4.1⊠, 10.4.2⊠

11.12.7 ST_MapAlgebraExpr

ST_MapAlgebraExpr — ⊠⊠⊠⊠⊠ 1 ⊠⊠⊠: ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ PostgreSQL ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠ 1 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠ 1 ⊠⊠⊠⊠⊠⊠.

⊠⊠

raster **ST_MapAlgebraExpr**(raster rast, integer band, text pixeltype, text expression, double precision nodataval=NULL);
raster **ST_MapAlgebraExpr**(raster rast, text pixeltype, text expression, double precision nodataval=NULL);

Description

⊠⊠
ST_MapAlgebraExpr 2.1.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠ ST_MapAlgebra (expression version) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠⊠⊠⊠ (rast) ⊠⊠⊠ expression ⊠⊠⊠⊠⊠⊠⊠⊠⊠ PostgreSQL ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠ 1 ⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. nband ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠ 1 ⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠ 1 ⊠⊠⊠⊠⊠⊠⊠⊠⊠.

pixeltype ⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. pixeltype ⊠ NULL ⊠⊠⊠, ⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠ rast ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

□□□□□□□□□□□□□□□□□□□□ [rast], 1-□□□□□□□□□□□□□□□ [rast.x], 1-□□□□□□ □□□□□□□□□ [rast.y] □□□□□□□□□□□□.

2.0.0 □□□□□□□□□□□□.

□□

□□□□□ 2 □□□□□□□□□□ (modulo) □□□□□□□□□□□□ 1 □□□□□□□□□□□□□□□.

Code

```
ALTER TABLE dummy_rast ADD COLUMN map_rast raster;
UPDATE dummy_rast SET map_rast = ST_MapAlgebraExpr(rast, NULL, 'mod([rast]::numeric,2)')  ↩
    WHERE rid = 2;

SELECT
    ST_Value(rast, 1, i, j) As origval,
    ST_Value(map_rast, 1, i, j) As mapval
FROM dummy_rast
CROSS JOIN generate_series(1, 3) AS i
CROSS JOIN generate_series(1, 3) AS j
WHERE rid = 2;
```

Output

```
origval | mapval
---------+--------
     253 |      1
     254 |      0
     253 |      1
     253 |      1
     254 |      0
     254 |      0
     250 |      0
     254 |      0
     254 |      0
```

□□□□□□□ NODATA □□ 0 □□□□□□□□□□□□□□□□ 2BUI □, □□ 1 □□□□□□□□□□□ □□□.

Code

```
ALTER TABLE dummy_rast ADD COLUMN map_rast2 raster;
UPDATE dummy_rast SET
    map_rast2 = ST_MapAlgebraExpr(rast, '2BUI'::text, 'CASE WHEN [rast] BETWEEN 100 and 250 ↩
     THEN 1 WHEN [rast] = 252 THEN 2 WHEN [rast] BETWEEN 253 and 254 THEN 3 ELSE 0 END':: ↩
    text, '0')
WHERE rid = 2;

SELECT DISTINCT
    ST_Value(rast, 1, i, j) As origval,
    ST_Value(map_rast2, 1, i, j) As mapval
FROM dummy_rast
CROSS JOIN generate_series(1, 5) AS i
CROSS JOIN generate_series(1, 5) AS j
WHERE rid = 2;
```

Output

```
origval | mapval
---------+--------
     249 |      1
     250 |      1
     251 |
     252 |      2
     253 |      3
     254 |      3
```

Code

```
SELECT
    ST_BandPixelType(map_rast2) As b1pixtyp
FROM dummy_rast
WHERE rid = 2;
```

Output

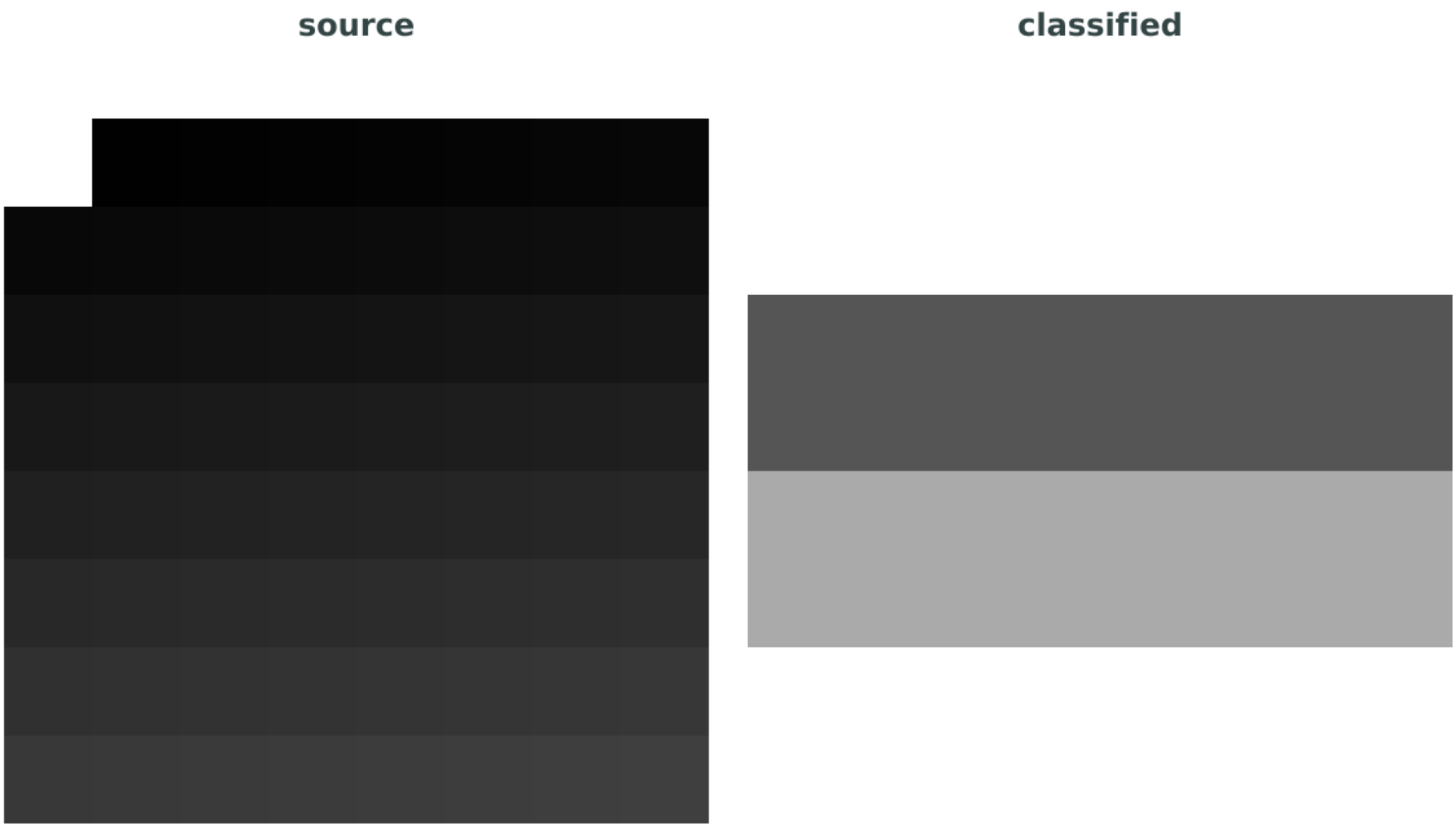
```
2BUI
```

A self-contained source raster and the raster produced by classifying its values into four display levels.

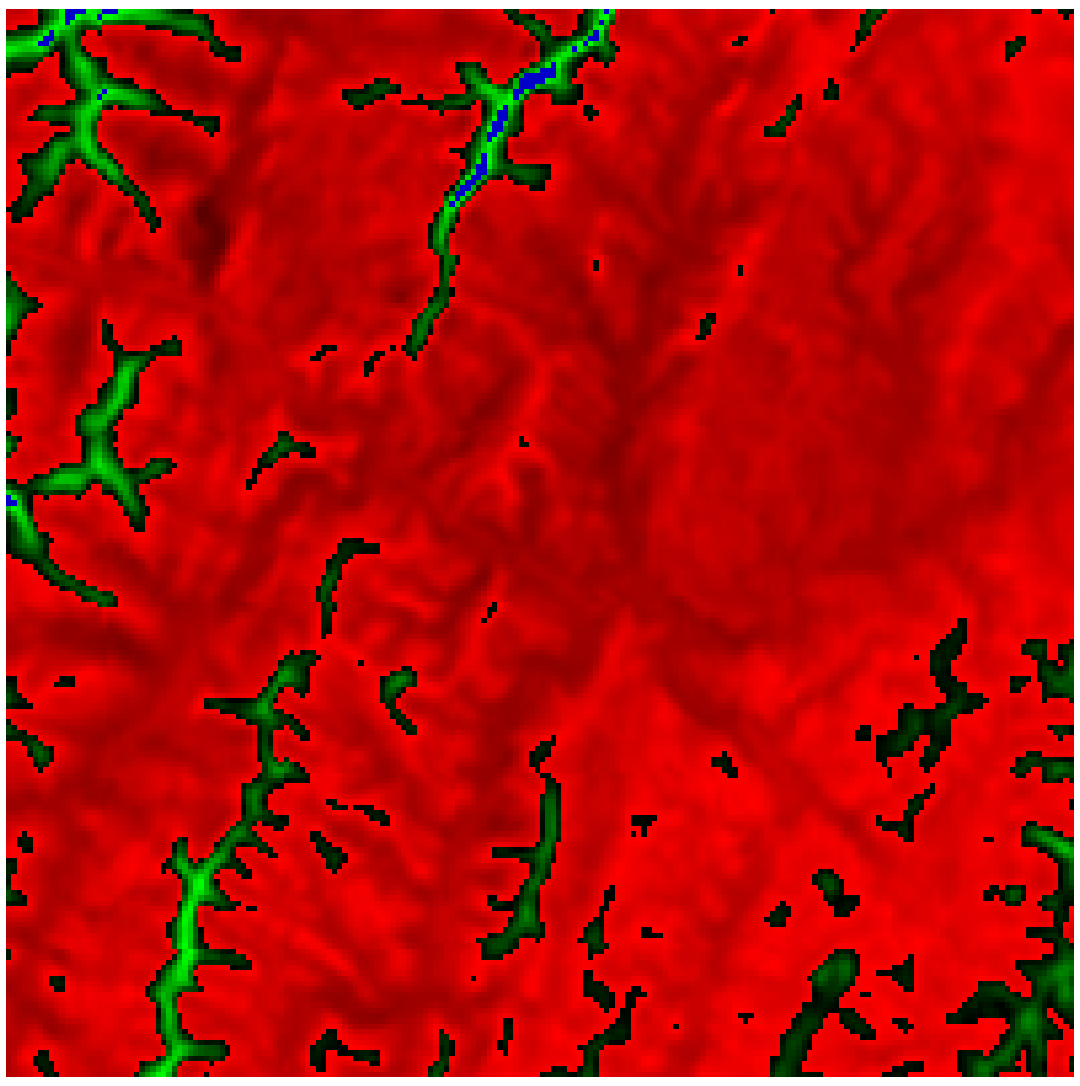
Code

```
WITH source AS (
    SELECT ST_SetValues(
        ST_AddBand(
            ST_MakeEmptyRaster(8, 8, 0, 8, 1, -1, 0, 0, 0),
            1, '8BUI', 0, 0
        ),
        1, 1, 1,
        ARRAY[
            [0, 1, 2, 3, 4, 5, 6, 7],
            [8, 9, 10, 11, 12, 13, 14, 15],
            [16, 17, 18, 19, 20, 21, 22, 23],
            [24, 25, 26, 27, 28, 29, 30, 31],
            [32, 33, 34, 35, 36, 37, 38, 39],
            [40, 41, 42, 43, 44, 45, 46, 47],
            [48, 49, 50, 51, 52, 53, 54, 55],
            [56, 57, 58, 59, 60, 61, 62, 63]
        ]::double precision[][]
    ) AS rast
), variants AS (
    SELECT 'source' AS title, rast AS rendered
    FROM source
    UNION ALL
    SELECT
        'classified',
        ST_MapAlgebraExpr(rast, '8BUI', 'floor([rast] / 16) * 85')
    FROM source
)
SELECT title, ST_AsPNG(rendered) AS image
FROM variants
ORDER BY title DESC;
```

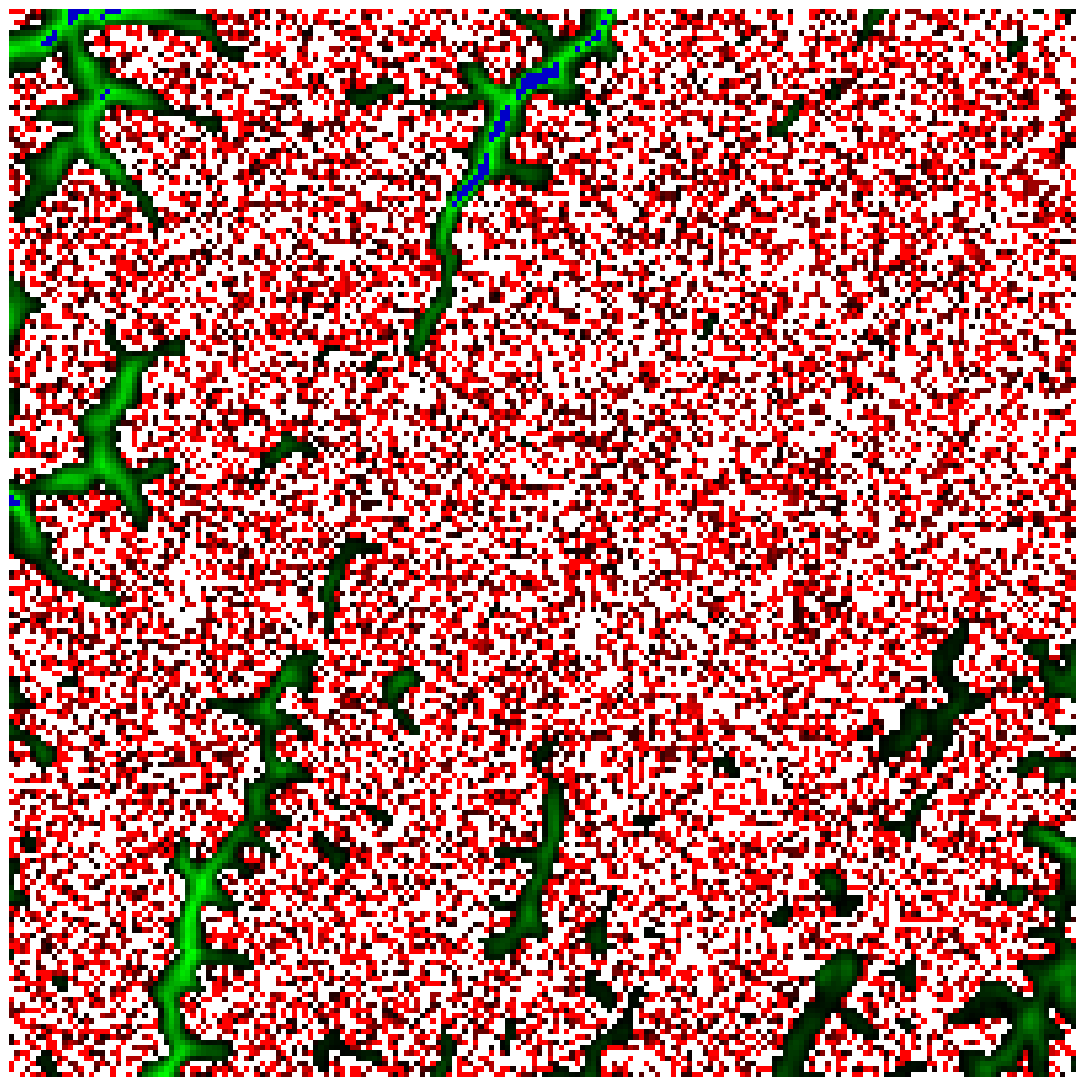
Figure



Map algebra altering the first band of a three-band raster.



original (column rast_view)



rast_view_ma

□□ 3 □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□, □□□□□□□□ □□ 3 □□□□□□□□□□□□□□.

Code

```
SELECT
    ST_AddBand(ST_AddBand(ST_AddBand(ST_MakeEmptyRaster(rast_view),
                ST_MapAlgebraExpr(rast_view, 1, NULL, 'tan([rast])*[rast]')
            ),
            ST_Band(rast_view, 2)
        ),
        ST_Band(rast_view, 3)
    )  As rast_view_ma
FROM wind
WHERE rid=167;
```

⊠⊠

ST_MapAlgebraExpr, ST_MapAlgebraFct, ST_BandPixelType, ST_GeoReference, ST_Value

11.12.8 ST_MapAlgebraExpr

ST_MapAlgebraExpr — ⊠⊠⊠⊠⊠ 2 ⊠⊠⊠: ⊠⊠⊠⊠⊠⊠ 2 ⊠⊠⊠⊠⊠⊠⊠ PostgreSQL ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠ 1 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠ ⊠⊠⊠ 1 ⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠) ⊠⊠⊠ ⊠⊠⊠⊠. extenttype ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. extenttype ⊠⊠⊠ INTERSECTION, UNION, FIRST, SECOND ⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

raster **ST_MapAlgebraExpr**(raster rast1, raster rast2, text expression, text pixeltype=same_as_rast1_band, text extenttype=INTERSECTION, text nodata1expr=NULL, text nodata2expr=NULL, double precision nodatanodataval=NULL);
raster **ST_MapAlgebraExpr**(raster rast1, integer band1, raster rast2, integer band2, text expression, text pixeltype=same_as_rast1_band, text extenttype=INTERSECTION, text nodata1expr=NULL, text nodata2expr=NULL, double precision nodatanodataval=NULL);

Description

⊠⊠
ST_MapAlgebraExpr 2.1.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠ ST_MapAlgebra (expression version) ⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠⊠⊠⊠⊠⊠ rast1, (rast2) ⊠⊠⊠ expression ⊠⊠⊠⊠⊠⊠⊠ 2 ⊠⊠⊠⊠, ⊠⊠⊠ PostgreSQL ⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠ 1 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. band1, band2 ⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠ 1 ⊠ ⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠) ⊠⊠⊠⊠⊠⊠⊠. extenttype ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

expression ⊠⊠⊠ 2 ⊠⊠⊠⊠⊠ PostgreSQL ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ PostgreSQL ⊠⊠⊠⊠/⊠⊠⊠⊠⊠⊠. ⊠: (([rast1] + [rast2])/2.0)::integer

pixeltype ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠ ST_BandPixelType ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠, NULL ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ NULL ⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

extenttype ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠

1. INTERSECTION - ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠.
2. UNION - ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
3. FIRST - ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
4. SECOND - ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

nodata1expr rast1 ⊠⊠⊠⊠⊠ NODATA ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ rast2 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠ rast2 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

nodata2expr rast2 ⊠⊠⊠⊠⊠ NODATA ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ rast1 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠ rast1 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

nodatanodataval ⊠⊠⊠⊠⊠⊠⊠⊠⊠ rast1 ⊠ rast2 ⊠⊠⊠⊠⊠⊠⊠ NODATA ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠.

pixeltype ⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. pixeltype ⊠ NULL ⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ rast1 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ [rast1.val], [rast2.val], ⊠⊠⊠⊠/⊠⊠⊠⊠⊠⊠⊠⊠⊠ [rast1.x], [rast1.y] ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

2 Band Intersection and Union.

⊠⊠⊠⊠⊠ 2 ⊠⊠⊠⊠⊠⊠⊠⊠⊠ (modulo) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ 1 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

Create a cool set of rasters.

Insert some cool shapes around Boston in Massachusetts state plane meters, then map their intersection and union.

Code

```
DROP TABLE IF EXISTS fun_shapes;
CREATE TABLE fun_shapes(rid serial PRIMARY KEY, fun_name text, rast raster);

INSERT INTO fun_shapes(fun_name, rast)
VALUES ('ref', ST_AsRaster(ST_MakeEnvelope(235229, 899970, 237229, 901930, 26986), 200, ↩
    200, '8BUI', 0, 0));

INSERT INTO fun_shapes(fun_name, rast)
WITH ref(rast) AS (SELECT rast FROM fun_shapes WHERE fun_name = 'ref' )
SELECT 'area' AS fun_name, ST_AsRaster(ST_Buffer(ST_SetSRID(ST_Point(236229, 900930), ↩
    26986), 1000),
            ref.rast, '8BUI', 10, 0) As rast
FROM ref
UNION ALL
SELECT 'rand bubbles',
       ST_AsRaster(
           (
               SELECT ST_Collect(geom)
               FROM (
                   SELECT ST_Buffer(
```

```
                        ST_SetSRID(ST_Point(236229 + i*random()*100, 900930 + j*random() ↩
    *100), 26986),
                        random()*20
                    ) AS geom
                    FROM generate_series(1, 10) AS i,
                         generate_series(1, 10) AS j
                ) AS foo
            ),
            ref.rast, '8BUI', 200, 0
        )
FROM ref;

SELECT  ST_MapAlgebraExpr(area.rast, bub.rast, '[rast2.val]', '8BUI', 'INTERSECTION', '[ ↩
    rast2.val]', '[rast1.val]') As interrast,
        ST_MapAlgebraExpr(area.rast, bub.rast, '[rast2.val]', '8BUI', 'UNION', '[rast2.val] ↩
    ', '[rast1.val]') As unionrast
FROM
  (SELECT rast FROM fun_shapes WHERE fun_name = 'area') As area
CROSS JOIN
  (SELECT rast FROM fun_shapes WHERE fun_name = 'rand bubbles') As bub
```

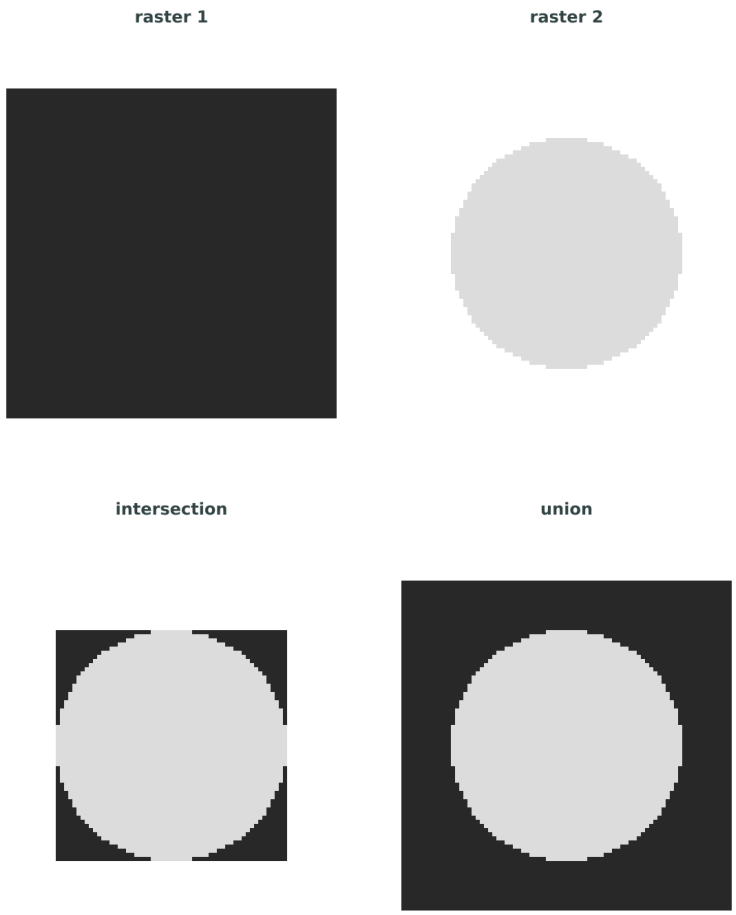
The same intersection and union operations using two deterministic, self-contained rasters.

Code

```
WITH canvas AS (
    SELECT ST_AddBand(
        ST_MakeEmptyRaster(80, 80, 0, 80, 1, -1, 0, 0, 0),
        1, '8BUI', 40, 0
    ) AS rast
), inputs AS (
    SELECT
        rast AS raster_1,
        ST_AsRaster(
            ST_Buffer(ST_Point(40, 40), 28),
            rast,
            '8BUI', 220, 0
        ) AS raster_2
    FROM canvas
), variants AS (
    SELECT 'raster 1' AS title, raster_1 AS rendered FROM inputs
    UNION ALL
    SELECT 'raster 2', raster_2 FROM inputs
    UNION ALL
    SELECT 'intersection', ST_MapAlgebraExpr(
        raster_1, raster_2,
        '[rast2.val]', '8BUI', 'INTERSECTION',
        '[rast2.val]', '[rast1.val]'
    ) FROM inputs
    UNION ALL
    SELECT 'union', ST_MapAlgebraExpr(
        raster_1, raster_2,
        '[rast2.val]', '8BUI', 'UNION',
        '[rast2.val]', '[rast1.val]'
    ) FROM inputs
)
SELECT title, ST_AsPNG(rendered) AS image
FROM variants
ORDER BY CASE title
    WHEN 'raster 1' THEN 1
    WHEN 'raster 2' THEN 2
```

```
    WHEN 'intersection' THEN 3
    ELSE 4
END;
```

Figure



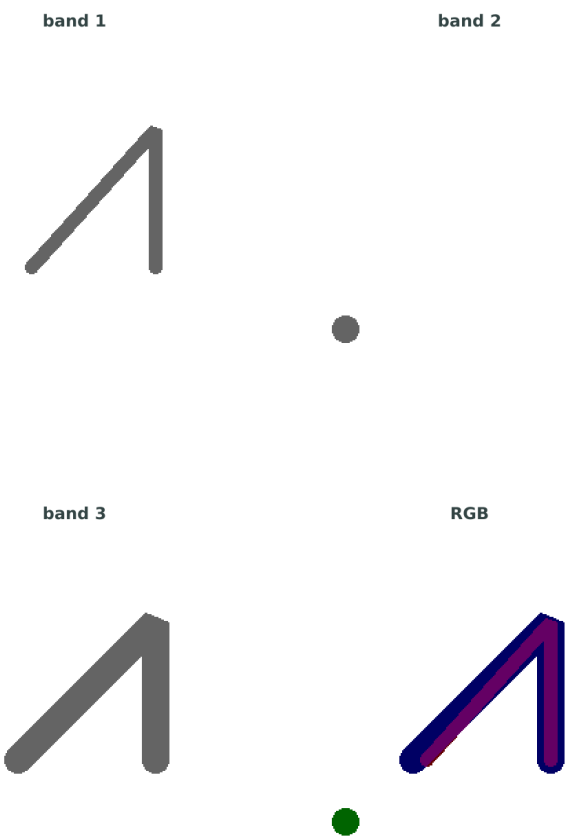
Overlaying rasters on a one-unit-per-pixel canvas as separate bands. We use ST_AsPNG to render the image so all single band ones look grey.

Code

```
WITH mygeoms AS (
    SELECT 2 AS bnum, ST_Buffer(ST_Point(1, 5), 10) AS geom
    UNION ALL
    SELECT 3,
        ST_Buffer(
            ST_GeomFromText('LINESTRING(50 50,150 150,150 50)'),
            10,
            'join=bevel'
        )
    UNION ALL
    SELECT 1,
        ST_Buffer(
            ST_GeomFromText('LINESTRING(60 50,150 150,150 50)'),
            5,
```

```
            'join=bevel'
        )
), canvas AS (
    SELECT ST_AddBand(
        ST_MakeEmptyRaster(
            200,
            200,
            ST_XMin(e)::integer,
            ST_YMax(e)::integer,
            1, -1, 0, 0),
        '8BUI'::text,
        0
    ) AS rast
    FROM (
      SELECT ST_Extent(geom) AS e
      FROM mygeoms
    ) AS bounds
), rbands AS (
    SELECT ARRAY(
        SELECT ST_MapAlgebraExpr(
            canvas.rast,
            ST_AsRaster(m.geom, canvas.rast, '8BUI', 100),
            '[rast2.val]', '8BUI', 'FIRST',
            '[rast2.val]', '[rast1.val]'
        )
        FROM mygeoms AS m
        CROSS JOIN canvas
        ORDER BY m.bnum
    ) AS rasts
)
SELECT title, ST_AsPNG(rendered) AS image
FROM rbands
CROSS JOIN LATERAL (VALUES
    ('band 1', rasts[1]),
    ('band 2', rasts[2]),
    ('band 3', rasts[3]),
    ('RGB', ST_AddBand(ST_AddBand(rasts[1], rasts[2]), rasts[3]))
) AS variants(title, rendered);
```

Figure



Overlay a two-meter boundary of selected parcels over aerial imagery. The query first unions the parcel geometries, clips the source raster shards to the region, unions those smaller shards, and finally adds the first two bands plus a third-band map algebra overlay.

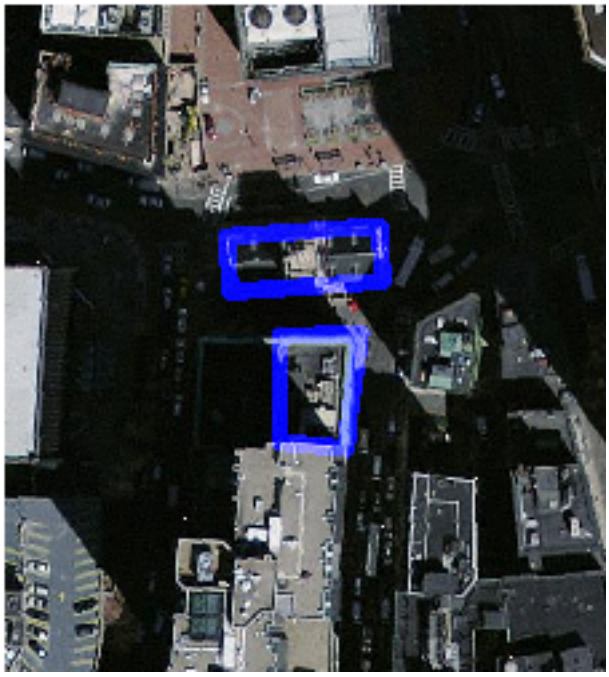
Create a new three-band raster composed of the first two clipped bands and the third band overlaid with the geometry. This query took 3.6 seconds on a 64-bit PostGIS Windows installation.

Code

```
WITH pr AS (
    SELECT ST_Clip(rast, ST_Expand(geom, 50)) AS rast, g.geom
    FROM aerials.o_2_boston AS r
    INNER JOIN (
        SELECT ST_Union(ST_Transform(geom, 26986)) AS geom
        FROM landparcels
        WHERE pid IN ('0303890000', '0303900000')
    ) AS g
        ON ST_Intersects(rast::geometry, ST_Expand(g.geom, 50))
), prunion AS (
    SELECT ST_AddBand(
        NULL,
        ARRAY[
            ST_Union(rast, 1),
            ST_Union(rast, 2),
```

```
            ST_Union(rast, 3)
        ]
    ) AS clipped,
    geom
    FROM pr
    GROUP BY geom
)
SELECT ST_AddBand(
    ST_Band(clipped, ARRAY[1, 2]),
    ST_MapAlgebraExpr(
     ST_Band(clipped, 3),
     ST_AsRaster(ST_Buffer(ST_Boundary(geom), 2), clipped, '8BUI', 250),
     '[rast2.val]',
     '8BUI',
     'FIRST',
     '[rast2.val]',
     '[rast1.val]') ) As rast
FROM prunion;
```

Parcel-boundary overlay.



The blue lines are the boundaries of selected parcels

The generated four-panel canvas example above shows the same band-overlay technique without requiring an external aerial-imagery or parcel table.

☒☒

ST_MapAlgebraExpr, ST_AddBand, ST_AsPNG, ST_AsRaster, ST_MapAlgebraFct, ST_BandPixelType, ST_GeoReference, ST_Value, ST_Union, ST_Union

11.12.9 ST_MapAlgebraFct

ST_MapAlgebraFct — □□□□□ 1 □□□: □□□□□□□□□□□□ PostgreSQL □□□□□□□□□□□□□□□, □□□□□□□□□□□, □□ 1 □□□□□□□□□□□□□□□□. □□□□□□□□□□□□□□□, □□ 1 □□□□□□□.

□□

raster **ST_MapAlgebraFct**(raster rast, regprocedure onerasteruserfunc);
raster **ST_MapAlgebraFct**(raster rast, regprocedure onerasteruserfunc, text[] VARIADIC args);
raster **ST_MapAlgebraFct**(raster rast, text pixeltype, regprocedure onerasteruserfunc);
raster **ST_MapAlgebraFct**(raster rast, text pixeltype, regprocedure onerasteruserfunc, text[] VARIADIC args);
raster **ST_MapAlgebraFct**(raster rast, integer band, regprocedure onerasteruserfunc);
raster **ST_MapAlgebraFct**(raster rast, integer band, regprocedure onerasteruserfunc, text[] VARIADIC args);
raster **ST_MapAlgebraFct**(raster rast, integer band, text pixeltype, regprocedure onerasteruserfunc);
raster **ST_MapAlgebraFct**(raster rast, integer band, text pixeltype, regprocedure onerasteruserfunc, text[] VARIADIC args);

Description



□□
ST_MapAlgebraFct 2.1.0 □□□□□□□□□□□□□□□□□□□□. □□ ST_MapAlgebra (callback function version) □□□□□□□□□□.

□□□□□ (rast) □□□ onerasteruserfunc □□□□□□□□ PostgreSQL □□□□□□□□□□, □□ 1 □□□□□□□□□□□□□□□□. band □□□□□□□□□□, □□ 1 □□□□□□□. □□□□□□□□□□□□□□□□ □□□□□, □□□□□□□□□, □□□□ 1 □□□□□□□□□.

pixeltype □□□□□□□, □□□□□□□□□□□□□□□□□□□□□□. pixeltype □ NULL □□□□, □□□□□ □□□□□□□ rast □□□□□□□□□□□□□□□□□□□.

The onerasteruserfunc parameter must be the name and signature of a SQL or PL/pgSQL function, cast to a regprocedure. A very simple and quite useless PL/pgSQL function example is:

Code

```
CREATE OR REPLACE FUNCTION simple_function(pixel FLOAT, pos INTEGER[], VARIADIC args TEXT ↩
    [])
RETURNS FLOAT
AS $$ BEGIN
    RETURN 0.0;
END; $$
LANGUAGE 'plpgsql' IMMUTABLE;
```

The userfunction may accept two or three arguments: a float value, an optional integer array, and a variadic text array. The first argument is the value of an individual raster cell (regardless of the raster datatype). The second argument is the position of the current processing cell in the form '{x,y}'. The third argument indicates that all remaining parameters to ST_MapAlgebraFct shall be passed through to the userfunction.

Passing a regprodedure argument to a SQL function requires the full function signature to be passed, then cast to a regprocedure type. To pass the above example PL/pgSQL function as an argument, the SQL for the argument is:

Code

```
'simple_function(float,integer[],text[])'::regprocedure
```

Note that the argument contains the name of the function, the types of the function arguments, quotes around the name and argument types, and a cast to a regprocedure.

userfunction ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ variadic text ⊠⊠⊠⊠⊠. ⊠⊠ ST_MapAlgebraFct ⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ userfunction ⊠⊠⊠⊠⊠, args ⊠⊠⊠⊠⊠⊠⊠⊠.

> **Note!**
> ⊠⊠
> (⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠) VARIADIC ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, PostgreSQL ⊠⊠⊠ ⊠⊠ Query Language (SQL) Functions ⊠"SQL Functions with Variable Numbers of Arguments" ⊠⊠⊠⊠⊠⊠⊠⊠⊠.

> **Note!**
> ⊠⊠
> ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, userfunction ⊠⊠⊠⊠⊠ text[] ⊠⊠⊠⊠⊠⊠⊠⊠.

2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

⊠⊠⊠⊠⊠ 2 ⊠⊠⊠⊠⊠⊠⊠⊠⊠ (modulo) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ 1 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

Code

```
ALTER TABLE dummy_rast ADD COLUMN map_rast raster;
CREATE FUNCTION mod_fct(pixel float, pos integer[], variadic args text[])
RETURNS float
AS $$
BEGIN
    RETURN pixel::integer % 2;
END;
$$
LANGUAGE 'plpgsql' IMMUTABLE;

UPDATE dummy_rast SET map_rast = ST_MapAlgebraFct(rast, NULL, 'mod_fct(float,integer[],text ↩
    [])'::regprocedure) WHERE rid = 2;

SELECT ST_Value(rast, 1, i, j) As origval, ST_Value(map_rast, 1, i, j) As mapval
FROM dummy_rast CROSS JOIN generate_series(1, 3) AS i CROSS JOIN generate_series(1, 3) AS j
WHERE rid = 2;
```

Output

```
origval | mapval
---------+--------
     253 |      1
     254 |      0
     253 |      1
     253 |      1
     254 |      0
     254 |      0
     250 |      0
     254 |      0
     254 |      0
```

⊠⊠⊠⊠⊠⊠⊠ NODATA ⊠⊠⊠⊠⊠⊠⊠ (0) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ 2BUI ⊠, ⊠ ⊠ 1 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

Code

```
ALTER TABLE dummy_rast ADD COLUMN map_rast2 raster;
CREATE FUNCTION classify_fct(pixel float, pos integer[], variadic args text[])
RETURNS float
AS
$$
DECLARE
    nodata float := 0;
BEGIN
    IF NOT args[1] IS NULL THEN
        nodata := args[1];
    END IF;
    IF pixel < 251 THEN
        RETURN 1;
    ELSIF pixel = 252 THEN
        RETURN 2;
    ELSIF pixel
> 252 THEN
        RETURN 3;
    ELSE
        RETURN nodata;
    END IF;
END;
$$
LANGUAGE 'plpgsql';
UPDATE dummy_rast SET map_rast2 = ST_MapAlgebraFct(rast,'2BUI','classify_fct(float,integer ←
    [],text[])'::regprocedure, '0') WHERE rid = 2;

SELECT DISTINCT ST_Value(rast,1,i,j) As origval, ST_Value(map_rast2, 1, i, j) As mapval
FROM dummy_rast CROSS JOIN generate_series(1, 5) AS i CROSS JOIN generate_series(1,5) AS j
WHERE rid = 2;
```

Output

```
 origval | mapval
---------+--------
     249 |      1
     250 |      1
     251 |
     252 |      2
     253 |      3
     254 |      3
```

Code

```
SELECT ST_BandPixelType(map_rast2) As b1pixtyp
FROM dummy_rast WHERE rid = 2;
```

Output

```
 b1pixtyp
----------
 2BUI
```

☒☒ 3 ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒, ☒☒☒☒☒☒☒☒ ☒☒ 3 ☒☒☒☒☒☒☒☒☒☒☒☒☒☒.

Code

```
CREATE FUNCTION rast_plus_tan(pixel float, pos integer[], variadic args text[])
RETURNS float
AS
$$
```

```
BEGIN
    RETURN tan(pixel) * pixel;
END;
$$
LANGUAGE 'plpgsql';

SELECT ST_AddBand(ST_AddBand(ST_AddBand(ST_MakeEmptyRaster(rast_view),
            ST_MapAlgebraFct(rast_view, 1, NULL, 'rast_plus_tan(float,integer[],text[])':: ↩
    regprocedure)
        ),
        ST_Band(rast_view, 2)
    ),
    ST_Band(rast_view, 3) As rast_view_ma
)
FROM wind
WHERE rid=167;
```

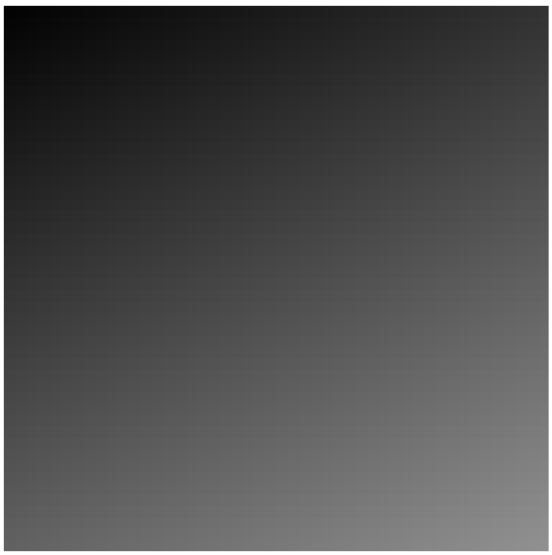
The deprecated callback form above is equivalent to a modern ST_MapAlgebra (callback function version) expression for simple per-pixel transforms. This build-time example renders the original raster and a modulo-2 classification.

Code

```
WITH pixels AS (
    SELECT x, y, (x + y * 2)::double precision AS value
    FROM generate_series(1, 48) AS x
    CROSS JOIN generate_series(1, 48) AS y
), rows AS (
    SELECT y, array_agg(value ORDER BY x) AS values
    FROM pixels
    GROUP BY y
), source AS (
    SELECT ST_SetValues(
        ST_AddBand(
            ST_MakeEmptyRaster(48, 48, 0, 48, 1, -1, 0, 0, 0),
            1, '8BUI', 0, 0
        ),
        1, 1, 1, array_agg(values ORDER BY y)::double precision[][]
    ) AS rast
    FROM rows
), variants AS (
    SELECT 'source' AS title, rast AS rendered FROM source
    UNION ALL
    SELECT 'modulo 2', ST_MapAlgebra(
        rast, 1,
        '8BUI',
        'mod([rast]::integer,2) * 255'
    )
    FROM source
)
SELECT title, ST_AsPNG(rendered) AS image
FROM variants
ORDER BY title DESC;
```

Figure

source modulo 2



□□

ST_MapAlgebraExpr, ST_BandPixelType, ST_GeoReference, ST_SetValue

11.12.10 ST_MapAlgebraFct

ST_MapAlgebraFct — □□□□□ 2 □□□: □□□□□ 2 □□□□□□□ PostgreSQL □□□□□□□□□□□□□, □□□□□□□□□□□, □□ 1 □□□□□□□□□□□□□□□. □□□□□□□□□□□□□□, □□ 1 □□□□□□□. □□□□□□□□□□□□□□□□□□□□□ INTERSECTION □□□.

□□

raster **ST_MapAlgebraFct**(raster rast1, raster rast2, regprocedure tworastuserfunc, text pixeltype=same_a
text extenttype=INTERSECTION, text[] VARIADIC userargs);
raster **ST_MapAlgebraFct**(raster rast1, integer band1, raster rast2, integer band2, regprocedure tworastuserfunc, text pixeltype=same_as_rast1, text extenttype=INTERSECTION, text[] VARIADIC userargs);

Description

⊠⊠
ST_MapAlgebraFct 2.1.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠ ST_MapAlgebra (callback function version) ⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠⊠⊠⊠ rast1, rast2 ⊠⊠⊠ tworastuserfunc ⊠⊠⊠⊠⊠ PostgreSQL ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠ 1 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. band1 ⊠⊠ band2 ⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠ 1 ⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠ 1 ⊠⊠⊠⊠⊠⊠⊠⊠.

pixeltype ⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. pixeltype ⊠ NULL ⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ rast1 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

The tworastuserfunc parameter must be the name and signature of an SQL or PL/pgSQL function, cast to a regprocedure. An example PL/pgSQL function example is:

Code

```
CREATE OR REPLACE FUNCTION simple_function_for_two_rasters(pixel1 FLOAT, pixel2 FLOAT, pos ↩
    INTEGER[], VARIADIC args TEXT[])
RETURNS FLOAT
AS $$ BEGIN
    RETURN 0.0;
END; $$
LANGUAGE 'plpgsql' IMMUTABLE;
```

The tworastuserfunc may accept three or four arguments: a double precision value, a double precision value, an optional integer array, and a variadic text array. The first argument is the value of an individual raster cell in rast1 (regardless of the raster datatype). The second argument is an individual raster cell value in rast2. The third argument is the position of the current processing cell in the form '{x,y}'. The fourth argument indicates that all remaining parameters to ST_MapAlgebraFct shall be passed through to the tworastuserfunc.

Passing a regprodedure argument to a SQL function requires the full function signature to be passed, then cast to a regprocedure type. To pass the above example PL/pgSQL function as an argument, the SQL for the argument is:

Code

```
'simple_function(double precision, double precision, integer[], text[])'::regprocedure
```

Note that the argument contains the name of the function, the types of the function arguments, quotes around the name and argument types, and a cast to a regprocedure.

The fourth argument to the tworastuserfunc is a variadic text array. All trailing text arguments to any ST_MapAlgebraFct call are passed through to the specified tworastuserfunc, and are contained in the userargs argument.

⊠⊠
(⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠) VARIADIC ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, PostgreSQL ⊠⊠⊠ ⊠⊠ Query Language (SQL) Functions ⊠"SQL Functions with Variable Numbers of Arguments" ⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠
⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, tworastuserfunc ⊠⊠⊠⊠⊠ text[] ⊠⊠⊠⊠⊠⊠⊠⊠.

2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

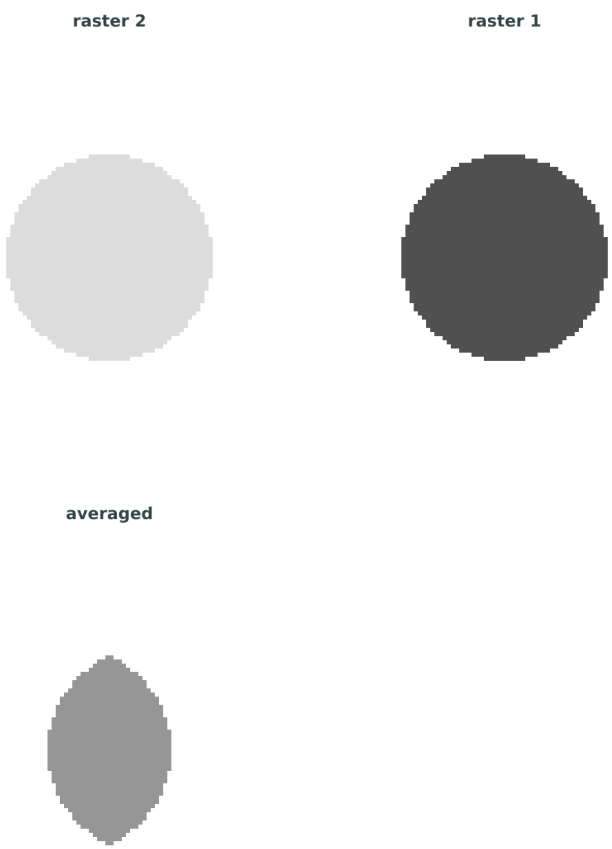
□□: □□□□□□□□□□□□□□□□□□□□□□

The deprecated two-raster callback form can often be expressed with the modern two-raster ST_MapAlgebra (callback function version) expression variant. This build-time example renders two input rasters and their averaged overlap.

Code

```
WITH canvas AS (
    SELECT ST_AddBand(
        ST_MakeEmptyRaster(80, 80, 0, 80, 1, -1, 0, 0, 0),
        1, '8BUI', 0, 0
    ) AS rast
), inputs AS (
    SELECT
        ST_AsRaster(ST_Buffer(ST_Point(30, 40), 25), rast, '8BUI', 80, 0)
            AS raster_1,
        ST_AsRaster(ST_Buffer(ST_Point(50, 40), 25), rast, '8BUI', 220, 0)
            AS raster_2
    FROM canvas
), variants AS (
    SELECT 'raster 1' AS title, raster_1 AS rendered FROM inputs
    UNION ALL
    SELECT 'raster 2', raster_2 FROM inputs
    UNION ALL
    SELECT 'averaged', ST_MapAlgebra(
        raster_1, 1,
        raster_2, 1,
        '([rast1] + [rast2]) / 2',
        '8BUI',
        'UNION'
    ) FROM inputs
)
SELECT title, ST_AsPNG(rendered) AS image
FROM variants
ORDER BY title DESC;
```

Figure



Define our user defined function.

Code

```
CREATE OR REPLACE FUNCTION raster_mapalgebra_union(
    rast1 double precision,
    rast2 double precision,
    pos integer[],
    VARIADIC userargs text[]
)
    RETURNS double precision
    AS $$
    DECLARE
    BEGIN
        CASE
            WHEN rast1 IS NOT NULL AND rast2 IS NOT NULL THEN
                RETURN ((rast1 + rast2)/2.);
            WHEN rast1 IS NULL AND rast2 IS NULL THEN
                RETURN NULL;
            WHEN rast1 IS NULL THEN
                RETURN rast2;
            ELSE
                RETURN rast1;
```

```
        END CASE;

        RETURN NULL;
    END;
    $$ LANGUAGE 'plpgsql' IMMUTABLE COST 1000;
```

Prepare a test table of rasters.

Code

```
DROP TABLE IF EXISTS map_shapes;
CREATE TABLE map_shapes(rid serial PRIMARY KEY, rast raster, bnum integer, descrip text);
INSERT INTO map_shapes(rast,bnum, descrip)
WITH mygeoms
    AS ( SELECT 2 As bnum, ST_Buffer(ST_Point(90,90),30) As geom, 'circle' As descrip
            UNION ALL
            SELECT 3 AS bnum,
                ST_Buffer(ST_GeomFromText('LINESTRING(50 50,150 150,150 50)'), 15) As geom, ↩
     'big road' As descrip
            UNION ALL
            SELECT 1 As bnum,
                ST_Translate(ST_Buffer(ST_GeomFromText('LINESTRING(60 50,150 150,150 50)'), ↩
     8,'join=bevel'), 10,-6) As geom, 'small road' As descrip
            ),
   -- define our canvas to be 1 to 1 pixel to geometry
   canvas
    AS ( SELECT ST_AddBand(
        ST_MakeEmptyRaster(
            250,
            250,
            ST_XMin(e)::integer,
            ST_YMax(e)::integer,
            1, -1, 0, 0),
        '8BUI'::text,
        0) As rast
        FROM (SELECT ST_Extent(geom) As e,
                    Max(ST_SRID(geom)) As srid
                    from mygeoms
                    ) As foo
            )
-- return our rasters aligned with our canvas
SELECT ST_AsRaster(m.geom, canvas.rast, '8BUI', 240) As rast, bnum, descrip
                FROM mygeoms AS m CROSS JOIN canvas
UNION ALL
SELECT canvas.rast, 4, 'canvas'
FROM canvas;
```

Map algebra on single-band rasters and then collect the result with ST_AddBand.

Code

```
INSERT INTO map_shapes(rast,bnum,descrip)
SELECT ST_AddBand(ST_AddBand(rasts[1], rasts[2]),rasts[3]), 4, 'map bands overlay fct union ↩
     (canvas)'
    FROM (SELECT ARRAY(SELECT ST_MapAlgebraFct(
            m1.rast,
            m2.rast,
            'raster_mapalgebra_union(double precision, double precision, integer[], text[]) ↩
    '::regprocedure,
            '8BUI',
            'FIRST')
                FROM map_shapes As m1 CROSS JOIN map_shapes As m2
```

```
    WHERE m1.descrip = 'canvas' AND m2.descrip <
> 'canvas' ORDER BY m2.bnum) As rasts) As foo;
```

A compact, executable callback example. The source rasters and their averaged overlap are rendered at build time.

Code

```
CREATE OR REPLACE FUNCTION pg_temp.raster_average(
    value_1 double precision,
    value_2 double precision,
    position integer[],
    VARIADIC userargs text[]
) RETURNS double precision
LANGUAGE SQL IMMUTABLE
AS 'SELECT COALESCE(($1 + $2) / 2.0, $1, $2)';

WITH canvas AS (
    SELECT ST_AddBand(
        ST_MakeEmptyRaster(80, 80, 0, 80, 1, -1, 0, 0, 0),
        1, '8BUI', 0, 0
    ) AS rast
), inputs AS (
    SELECT
        ST_AsRaster(ST_Buffer(ST_Point(30, 40), 25), rast, '8BUI', 80, 0)
            AS raster_1,
        ST_AsRaster(ST_Buffer(ST_Point(50, 40), 25), rast, '8BUI', 220, 0)
            AS raster_2
    FROM canvas
), variants AS (
    SELECT 'raster 1' AS title, raster_1 AS rendered FROM inputs
    UNION ALL
    SELECT 'raster 2', raster_2 FROM inputs
    UNION ALL
    SELECT 'averaged', ST_MapAlgebraFct(
        raster_1,
        raster_2,
        'pg_temp.raster_average(double precision,double precision,integer[],text[])':: ↩
    regprocedure,
        '8BUI',
        'UNION'
    ) FROM inputs
)
SELECT title, ST_AsPNG(rendered) AS image
FROM variants
ORDER BY title DESC;
```

Output

```
title    | image
----------+--------------------------
 raster 2 | PNG image, 50 x 50 pixels
 raster 1 | PNG image, 50 x 50 pixels
 averaged | PNG image, 70 x 50 pixels
```

□□□□□□□□□□□□□□□□

Code

```
CREATE OR REPLACE FUNCTION raster_mapalgebra_userargs(
    rast1 double precision,
    rast2 double precision,
    pos integer[],
    VARIADIC userargs text[]
)
    RETURNS double precision
    AS $$
    DECLARE
    BEGIN
        CASE
            WHEN rast1 IS NOT NULL AND rast2 IS NOT NULL THEN
                RETURN least(userargs[1]::integer,(rast1 + rast2)/2.);
            WHEN rast1 IS NULL AND rast2 IS NULL THEN
                RETURN userargs[2]::integer;
            WHEN rast1 IS NULL THEN
                RETURN greatest(rast2, random()*userargs[3]::integer)::integer;
            ELSE
                RETURN greatest(rast1, random()*userargs[4]::integer)::integer;
        END CASE;

        RETURN NULL;
    END;
    $$ LANGUAGE 'plpgsql' VOLATILE COST 1000;

SELECT ST_MapAlgebraFct(
    m1.rast,
    1,
    m1.rast,
    3,
    'raster_mapalgebra_userargs(double precision, double precision, integer[], text[])':: ↩
    regprocedure,
    '8BUI',
    'INTERSECT',
    '100', '200', '200', '0')
                FROM map_shapes As m1
    WHERE m1.descrip = 'map bands overlay fct union (canvas)';
```

The generated callback figure above demonstrates the raster alignment and overlap used by this variant; the exact callback result remains available as text.

□□

ST_MapAlgebraExpr, ST_BandPixelType, ST_GeoReference, ST_SetValue

11.12.11 ST_MapAlgebraFctNgb

ST_MapAlgebraFctNgb — □□□□□ 1 □□□: □□□□□ PostgreSQL □□□□□□□□□□□□□□□□ (Map Algebra Nearest Neighbor) □□□. □□□□□□□□□□□□ (neighborhood) □□□□ PostgreSQL □□□□□□□□□□□□□□□□□□□□□□□□□□□.

□□

raster **ST_MapAlgebraFctNgb**(raster rast, integer band, text pixeltype, integer ngbwidth, integer ngbheight, regprocedure onerastngbuserfunc, text nodatamode, text[] VARIADIC args);

Description

> ⊠⊠
> ST_MapAlgebraFctNgb 2.1.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠ ST_MapAlgebra (callback function version) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠⊠ 1 ⊠⊠⊠: ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (neighborhood) ⊠⊠⊠⊠ PostgreSQL ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

rast ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠

band ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (⊠⊠⊠⊠ 1)

pixeltype ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠ ST_BandPixelType ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠, NULL ⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠ NULL ⊠⊠⊠⊠⊠⊠⊠⊠⊠, rast ⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

ngbwidth ⊠⊠ (neighborhood) ⊠⊠⊠⊠⊠⊠

ngbheight ⊠⊠ (neighborhood) ⊠⊠⊠⊠⊠⊠

onerastngbuserfunc ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ PL/pgSQL ⊠⊠ psql ⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ 2 ⊠⊠⊠⊠⊠⊠⊠⊠⊠.

nodatamode NODATA ⊠⊠ NULL ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

’ignore’: ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ NODATA ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠ NODATA ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

’NULL’: ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ NODATA ⊠⊠⊠⊠⊠⊠⊠ NULL ⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠.

’value’: ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ NODATA ⊠⊠⊠⊠⊠⊠ (⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠) ⊠⊠⊠⊠⊠⊠. ⊠⊠⊠ NODATA ⊠⊠⊠, (⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠) ’NULL’ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

args ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠

2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

⊠⊠⊠⊠⊠ https://gdal.org/user/drivers/raster/postgisraster.html ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ST_Rescale ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

A simple ’callback’ user function that averages up all the values in a neighborhood.

Code

```
CREATE OR REPLACE FUNCTION rast_avg(matrix float[][], nodatamode text, variadic args text ↩
    [])
    RETURNS float AS
    $$
    DECLARE
        _matrix float[][];
        x1 integer;
        x2 integer;
        y1 integer;
        y2 integer;
```

```
        sum float;
    BEGIN
        _matrix := matrix;
        sum := 0;
        FOR x in array_lower(matrix, 1)..array_upper(matrix, 1) LOOP
            FOR y in array_lower(matrix, 2)..array_upper(matrix, 2) LOOP
                sum := sum + _matrix[x][y];
            END LOOP;
        END LOOP;
        RETURN (sum*1.0/(array_upper(matrix,1)*array_upper(matrix,2) ))::integer ;
    END;
    $$
LANGUAGE 'plpgsql' IMMUTABLE COST 1000;
```

Apply the raster averaging callback within two pixels of each input pixel in the X and Y directions.

Code

```
SELECT ST_MapAlgebraFctNgb(
        rast,
        1,
        '8BUI',
        4,
        4,
        'rast_avg(float[][], text, text[])'::regprocedure,
        'NULL',
        NULL) As nn_with_border
    FROM katrinas_rescaled
    limit 1;
```

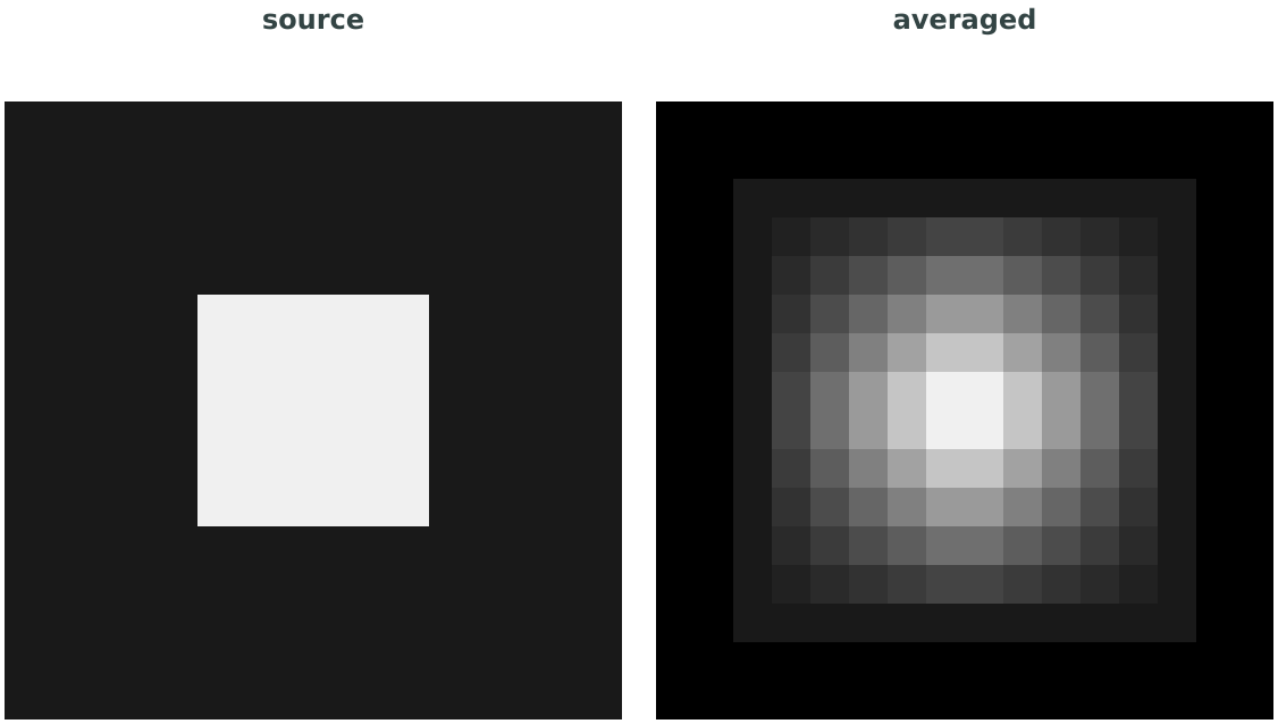
A self-contained neighborhood average. The source raster has a sharp central square, while the result spreads those values across neighboring pixels. The source uses a non-NODATA background so `nodatamode ignore` averages background values with the square values, instead of treating the background as absent.

Code

```
WITH source AS (
    SELECT ST_SetValues(
        ST_AddBand(
            ST_MakeEmptyRaster(16, 16, 0, 16, 1, -1, 0, 0, 0),
            1, '8BUI', 25, NULL
        ),
        1, 6, 6,
        ARRAY[
            [240, 240, 240, 240, 240, 240],
            [240, 240, 240, 240, 240, 240],
            [240, 240, 240, 240, 240, 240],
            [240, 240, 240, 240, 240, 240],
            [240, 240, 240, 240, 240, 240],
            [240, 240, 240, 240, 240, 240]
        ]::double precision[][]
    ) AS rast
), variants AS (
    SELECT 'source' AS title, rast AS rendered FROM source
    UNION ALL
    SELECT 'averaged', ST_SetBandNoDataValue(ST_MapAlgebraFctNgb(
        rast,
        1,
        '8BUI',
```

```
        2,
        2,
        'ST_Mean4ma(float[][],text,text[])'::regprocedure,
        'ignore',
        NULL
    ), NULL) FROM source
)
SELECT title, ST_AsPNG(rendered) AS image
FROM variants
ORDER BY title DESC;
```

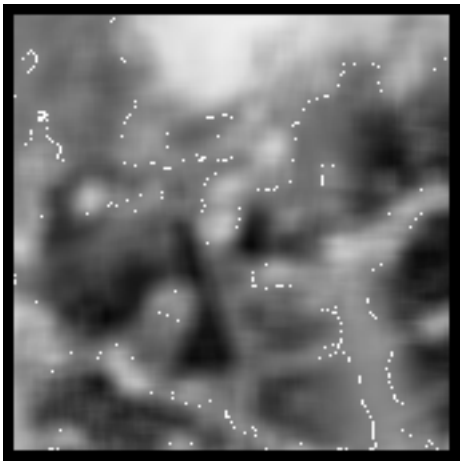
Figure



Neighborhood averaging.



□□□□□□□□□□



Averaged result

□□

ST_MapAlgebraFct, ST_MapAlgebraExpr, ST_Rescale

11.12.12 ST_Reclass

ST_Reclass — □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□. nband □□□□□□□□□□ □□□□. nband □□□□□□□□□□□□□□□□ 1 □□□□□□□. □□□□□□□□□□□□□□□□□□□□. □□□□□: □□□□□□□□□□□□□□□□□□□□□□□ 16BUI □□□ 8BUI □□□□□□□□□□□.

□□

raster **ST_Reclass**(raster rast, integer nband, text reclassexpr, text pixeltype, double precision nodataval=NULL);
raster **ST_Reclass**(raster rast, reclassarg[] VARIADIC reclassargset);
raster **ST_Reclass**(raster rast, text reclassexpr, text pixeltype);

Description

Creates a new raster formed by applying a reclassification operation defined by the `reclassexpr` on the input raster (`rast`). Refer to reclassarg for the description of reclassification expressions. If no `band` is specified band 1 is assumed.

The new raster will have the same georeference, width, and height as the original raster. The bands of the new raster have pixel type of `pixeltype`. If `reclassargset` is specified then each `reclassarg` defines the type of the target band. Bands not designated are returned unchanged.

2.0.0 □□□□□□□□□□□□□□.

□□

Basic.

□□□□□ 2 □ 8BUI □□ 4BUI □□□□□ 101 □□ 254 □□□□□□□ NODATA □□□□□□□□□□□ □□□□□.

Code

```
ALTER TABLE dummy_rast ADD COLUMN reclass_rast raster;
UPDATE dummy_rast SET reclass_rast = ST_Reclass(rast, 2, '0-87:1-10, 88-100:11-15,  ↩
    101-254:0-0', '4BUI', 0) WHERE rid = 2;

SELECT i as col, j as row, ST_Value(rast, 2, i, j) As origval,
    ST_Value(reclass_rast, 2, i, j) As reclassval,
    ST_Value(reclass_rast, 2, i, j, false) As reclassval_include_nodata
FROM dummy_rast CROSS JOIN generate_series(1, 3) AS i CROSS JOIN generate_series(1, 3) AS j
WHERE rid = 2;
```

Output

```
col | row | origval | reclassval | reclassval_include_nodata
----+-----+---------+------------+--------------------------
  1 |   1 |      78 |          9 |                         9
  2 |   1 |      98 |         14 |                        14
  3 |   1 |     122 |            |                         0
  1 |   2 |      96 |         14 |                        14
  2 |   2 |     118 |            |                         0
  3 |   2 |     180 |            |                         0
  1 |   3 |      99 |         15 |                        15
  2 |   3 |     112 |            |                         0
  3 |   3 |     169 |            |                         0
```

Advanced using multiple reclassargs.

□□□□□□□□ 1, 2, 3 □□□ 1BB, 4BUI, 4BUI □□□□□□□□□□□□□□□□□□□□. □□□□ (□□□□□□□□□□□□□□□□□□□□) □□□□□□□□□□□□□□□□□□□□, □□□□□□□□□□□□□ reclassarg □□□□□□□□□□□□□□□□□.

Code

```
UPDATE dummy_rast SET reclass_rast =
    ST_Reclass(
        rast,
        ROW(2, '0-87]:1-10, (87-100]:11-15, (101-254]:0-0', '4BUI', NULL)::reclassarg,
        ROW(1, '0-253]:1, 254:0', '1BB', NULL)::reclassarg,
        ROW(3, '0-70]:1, (70-86:2, [86-150):3, [150-255:4', '4BUI', NULL)::reclassarg
        ) WHERE rid = 2;

SELECT i as col, j as row, ST_Value(rast, 1, i, j) As ov1, ST_Value(reclass_rast, 1, i, j)  ↩
    As rv1,
    ST_Value(rast, 2, i, j) As ov2, ST_Value(reclass_rast, 2, i, j) As rv2,
    ST_Value(rast, 3, i, j) As ov3, ST_Value(reclass_rast, 3, i, j) As rv3
FROM dummy_rast CROSS JOIN generate_series(1, 3) AS i CROSS JOIN generate_series(1, 3) AS j
WHERE rid = 2;
```

Output

```
col | row | ov1 | rv1 | ov2 | rv2 | ov3 | rv3
----+-----+-----+-----+-----+-----+-----+-----
  1 |   1 | 253 |   1 |  78 |   9 |  70 |   1
  2 |   1 | 254 |   0 |  98 |  14 |  86 |   3
  3 |   1 | 253 |   1 | 122 |   0 | 100 |   3
  1 |   2 | 253 |   1 |  96 |  14 |  80 |   2
  2 |   2 | 254 |   0 | 118 |   0 | 108 |   3
  3 |   2 | 254 |   0 | 180 |   0 | 162 |   4
  1 |   3 | 250 |   1 |  99 |  15 |  90 |   3
  2 |   3 | 254 |   0 | 112 |   0 | 108 |   3
  3 |   3 | 254 |   0 | 169 |   0 | 175 |   4
```

Advanced Map a single band 32BF raster to multiple viewable bands.

32BF □□□□□□□□□□□□□□□□□□□□ 3 □ ((8BUI,8BUI,8BUI) □□□□□□□□□□□□□□□□□□□□.

Code

```
ALTER TABLE wind ADD COLUMN rast_view raster;
UPDATE wind
    set rast_view = ST_AddBand(
        NULL,
        ARRAY[
    ST_Reclass(rast, 1, '0.1-10]:1-10,9-10]:11,(11-33:0'::text, '8BUI'::text, 0),
    ST_Reclass(rast, 1, '11-33):0-255,[0-32:0,(34-1000:0'::text, '8BUI'::text, 0),
    ST_Reclass(rast, 1, '0-32]:0,(32-100:100-255'::text, '8BUI'::text, 0)
    ]
    );
```

□□

ST_AddBand, ST_Band, ST_BandPixelType, ST_MakeEmptyRaster, reclassarg, 10.4.1□, ST_Value

11.12.13 ST_ReclassExact

ST_ReclassExact — Creates a new raster composed of bands reclassified from original, using a 1:1 mapping from values in the original band to new values in the destination band.

□□

raster **ST_ReclassExact**(raster rast, double precision[] inputvalues, double precision[] outputvalues, integer bandnumber=1, text pixeltype=32BF, double precision nodatavalue=NULL);

Description

Creates a new raster formed by applying a reclassification operation defined by the `inputvalues` and `outputvalues` arrays. Pixel values found in the input array are mapped to the corresponding value in the output array. All other pixel values are mapped to the `nodatavalue`.

The output pixel type defaults to float, but can be specified using the `pixeltype` parameter. If no `bandnumber` is specified band 1 is assumed.

The new raster will have the same georeference, width, and height as the original raster. Bands not designated are returned unchanged.

Availability: 3.6.0

Example: Basic

Create a small raster and map its pixels to new values.

Create a raster with four pixels: first row 1 2, second row 3 4.

Code

```
CREATE TABLE reclassexact (
    id integer,
    rast raster
);
INSERT INTO reclassexact (id, rast)
```

```
SELECT 1, ST_SetValues(ST_AddBand(ST_MakeEmptyRaster(2, -- width in pixels
        2, -- height in pixels
        0, -- upper-left x-coordinate
        0, -- upper-left y-coordinate
        1, -- pixel size in x-direction
        -1, -- pixel size in y-direction (negative for north-up)
        0, -- skew in x-direction
        0, -- skew in y-direction
        4326  -- SRID (e.g., WGS 84)
      ),
      '32BUI'::text, -- pixel type (e.g., '32BF' for float, '8BUI' for unsigned 8-bit int)
      0.0, -- initial value for the band (e.g., 0.0 or a no-data value)
      -99            -- nodatavalue
    ),
    1, -- band number (usually 1 for single-band rasters)
    1, -- x origin for setting values (usually 1)
    1, -- y origin for setting values (usually 1)
    ARRAY[
      ARRAY[1, 2],
      ARRAY[3, 4]
    ]::double precision[][] -- 2D array of values
  );
```

Reclass the values to new values and dump the values of the new raster for display.

Code

```
WITH rc AS (
  SELECT ST_ReclassExact(rast, -- input raster
    ARRAY[4, 3, 2, 1], -- input map
    ARRAY[14, 13, 12, 11], -- output map
    1, -- band number to remap
    '32BUI'              -- output raster pixel type
    ) AS rast
  FROM reclassexact
  WHERE id = 1
  )
SELECT 'rce-1', (ST_DumpValues(rc.rast)).*
FROM rc;
```

□□

ST_Reclass, ST_AddBand, ST_Band, ST_MakeEmptyRaster

11.12.14 ST_Union

ST_Union — □□□□□□□□□ 1 □□□□□□□□□□□□□□□□□□□□□□□□□.

□□

raster **ST_Union**(setof raster rast);
raster **ST_Union**(setof raster rast, unionarg[] unionargset);
raster **ST_Union**(setof raster rast, integer nband);
raster **ST_Union**(setof raster rast, text uniontype);
raster **ST_Union**(setof raster rast, integer nband, text uniontype);

Description

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ 1 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠. ⊠⊠⊠⊠⊠⊠⊠, `uniontype` ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. `uniontype` ⊠⊠⊠ LAST(⊠⊠⊠), FIRST, MIN, MAX, COUNT, SUM, MEAN, RANGE ⊠⊠⊠⊠⊠⊠⊠⊠.

> ⊠⊠
> In order for rasters to be unioned, they must all have the same alignment. Use ST_SameAlignment and ST_NotSameAlignmentReason for more details and help. One way to fix alignment issues is to use ST_Resample and use the same reference raster for alignment.

2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠⊠⊠: 2.1.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (⊠⊠⊠ C ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠).

2.1.0 ⊠⊠⊠⊠ ST_Union(rast, unionarg) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠⊠⊠: 2.1.0 ⊠⊠⊠⊠ ST_Union(rast) ⊠⊠ 1 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. PostGIS ⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠⊠⊠: 2.1.0 ⊠⊠⊠⊠ ST_Union(rast, uniontype) ⊠⊠ 4 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

Reconstitute a single band chunked raster tile.

This creates a single band from first band of raster tiles. That form the original file system tile.

Code

```
SELECT filename, ST_Union(rast, 1) As file_rast
FROM sometable WHERE filename IN('dem01', 'dem02') GROUP BY filename;
```

Return a multi-band raster that is the union of tiles intersecting geometry.

This creates a multi band raster collecting all the tiles that intersect a line. Note that In 2.0, this would have just returned a single band raster. , new union works on all bands by default. This is equivalent to unionarg: ARRAY[ROW(1, 'LAST'), ROW(2, 'LAST'), ROW(3, 'LAST')]::unionarg[].

Code

```
SELECT ST_Union(rast)
FROM aerials.boston
WHERE ST_Intersects(rast, ST_GeomFromText('LINESTRING(230486 887771,230500 88772)', 26986) ↩
    );
```

Select and reorder bands while unioning tiles that intersect a geometry.

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

This creates a multi band raster collecting all the tiles that intersect a line.

Code

```
SELECT ST_Union(rast, ARRAY[ROW(2, 'LAST'), ROW(1, 'LAST'), ROW(3, 'LAST')]::unionarg[])
FROM aerials.boston
WHERE ST_Intersects(rast, ST_GeomFromText('LINESTRING(230486 887771,230500 88772)', 26986) ↩
    );
```

□□

unionarg, ST_Envelope, ST_ConvexHull, ST_Clip, ST_Union

11.13 □□□□□□□□□

11.13.1 ST_Distinct4ma

ST_Distinct4ma — □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□.

□□

float8 **ST_Distinct4ma**(float8[][] matrix, text nodatamode, text[] VARIADIC args);
double precision **ST_Distinct4ma**(double precision[][][] value, integer[][] pos, text[] VARIADIC user-args);

Description

□□□□□□□□□□□□□□□□□□□□□□□□.

> **Note**
> □□
> □□ 1 □ ST_MapAlgebraFctNgb □□□□□□□□□□□□□□□□□□□□□.

> **Note**
> □□
> □□ 2 □ ST_MapAlgebra (callback function version) □□□□□□□□□□□□□□□□□□□□□
> □.

> **Warning**
> □□
> 2.1.0 □□□□ ST_MapAlgebraFctNgb □□□□□□□□□□□□ 1 □□□□□□□□□□□□□.

2.0.0 □□□□□□□□□□□□.

□□□□: 2.1.0 □□□□□□ 2 □□□□□□□.

□□

Code

```
SELECT
    rid,
    ST_Value(ST_MapAlgebraFctNgb(rast, 1, NULL, 1, 1, 'ST_Distinct4ma(float[][],text,text ↩
    [])'::regprocedure, 'ignore', NULL), 2, 2
    )
FROM dummy_rast
WHERE rid = 2;
```

Output

```
rid | st_value
-----+----------
   2 |        3
(1 row)
```

□□

ST_MapAlgebraFctNgb, ST_MapAlgebra (callback function version), ST_Min4ma, ST_Max4ma, ST_Sum4ma, ST_Mean4ma, ST_Distinct4ma, ST_StdDev4ma

11.13.2 ST_InvDistWeight4ma

ST_InvDistWeight4ma — □□□□□□□□□□□□□□□□□□□□□□□□□□□□□.

□□

double precision **ST_InvDistWeight4ma**(double precision[][][] value, integer[][] pos, text[] VARIADIC userargs);

Description

□□□□□□□ (Inverse Distance Weighted method) □□□□□□□□□□□□□□□□□□□.

userargs □□□□□□□□□□□□□□□□□ 2 □□□□□. □□□□□□□□□□□□□□□□□□□□□□ 0 □ 1 □□□□□ (力率) □□□ (□□□□□□ k □□). □□□□□□□□□□, □□□□ 1 □□□. □□□□□□ □□. □□□□□□□□□□□ □□□□□□□□□□□, □□□□□□□□□.

The basic inverse-distance weighted estimate is: $z = (\Sigma_i z_i / d_i^k) / (\Sigma_i 1 / d_i^k)$ Here z_i is a neighboring pixel value, d_i is its distance from the pixel being estimated, and k is a real-valued power factor between 0 and 1.

> **Note**
> □□
> □□□□ ST_MapAlgebra (callback function version) □□□□□□□□□□□□□□□□□□□□□ □.

2.1.0 □□□□□□□□□□□□.

□□

Code

```
WITH foo AS (
    SELECT ST_SetValues(ST_AddBand(ST_MakeEmptyRaster(3, 3, 0, 0, 1, -1, 0, 0, 0), 1, '32BF ↩
    ', NULL::double precision, -9999),
        1, 1, 1, ARRAY[
            [10, 10, 10],
            [10, NULL, 20],
            [20, 20, 20]
        ]::double precision[][]
```

```
    ) AS rast
)
SELECT ST_Value(
    ST_MapAlgebra(
        rast,
        1,
        'ST_InvDistWeight4ma(double precision[][][],integer[][],text[])'::regprocedure,
        '32BF',
        'FIRST',
        NULL,
        1,
        1
    ),
    2, 2
) AS interpolated_value
FROM foo;
```

Output

```
15
```

⊠⊠

ST_MapAlgebra (callback function version), ST_MinDist4ma

11.13.3 ST_Max4ma

ST_Max4ma — ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

float8 **ST_Max4ma**(float8[][] matrix, text nodatamode, text[] VARIADIC args);
double precision **ST_Max4ma**(double precision[][][] value, integer[][] pos, text[] VARIADIC userargs);

Description

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠ 2 ⊠⊠⊠, ⊠⊠⊠⊠ userargs ⊠⊠⊠⊠ NODATA ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

> **Note:** ⊠⊠
> ⊠⊠ 1 ⊠ ST_MapAlgebraFctNgb ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

> **Note:** ⊠⊠
> ⊠⊠ 2 ⊠ ST_MapAlgebra (callback function version) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠
2.1.0 ⊠⊠⊠⊠ ST_MapAlgebraFctNgb ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ 1 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠⊠⊠: 2.1.0 ⊠⊠⊠⊠⊠⊠ 2 ⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

Code

```
SELECT
    rid,
    ST_Value(ST_MapAlgebraFctNgb(rast, 1, NULL, 1, 1, 'ST_Max4ma(float[][],text,text[])':: ↩
    regprocedure, 'ignore', NULL), 2, 2
    )
FROM dummy_rast
WHERE rid = 2;
```

Output

```
rid | st_value
-----+----------
   2 |      254
(1 row)
```

⊠⊠

ST_MapAlgebraFctNgb, ST_MapAlgebra (callback function version), ST_Min4ma, ST_Sum4ma, ST_Mean4m
ST_Range4ma, ST_Distinct4ma, ST_StdDev4ma

11.13.4 ST_Mean4ma

ST_Mean4ma — ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

float8 **ST_Mean4ma**(float8[][] matrix, text nodatamode, text[] VARIADIC args);
double precision **ST_Mean4ma**(double precision[][][] value, integer[][] pos, text[] VARIADIC user-args);

Description

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠ 2 ⊠⊠⊠, ⊠⊠⊠⊠ userargs ⊠⊠⊠⊠ NODATA ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠
⊠⊠ 1 ⊠ ST_MapAlgebraFctNgb ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠
⊠⊠ 2 ⊠ ST_MapAlgebra (callback function version) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠
2.1.0 ⊠⊠⊠⊠ ST_MapAlgebraFctNgb ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ 1 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠⊠⊠: 2.1.0 ⊠⊠⊠⊠⊠⊠ 2 ⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

Variant 1.

Code

```
SELECT
    rid,
    ST_Value(ST_MapAlgebraFctNgb(rast, 1, '32BF', 1, 1, 'ST_Mean4ma(float[][],text,text[])' ←
    ::regprocedure, 'ignore', NULL), 2, 2
    )
FROM dummy_rast
WHERE rid = 2;
```

Output

```
rid |     st_value
-----+------------------
   2 | 253.222229003906
(1 row)
```

Variant 2.

Code

```
SELECT
  rid,
  ST_Value(
      ST_MapAlgebra(rast, 1, 'ST_Mean4ma(double precision[][][], integer[][], text[])':: ←
    regprocedure, '32BF', 'FIRST', NULL, 1, 1),
      2,
      2)
FROM dummy_rast
 WHERE rid = 2;
```

Output

```
rid |     st_value
-----+------------------
   2 | 253.222229003906
(1 row)
```

⊠⊠

ST_MapAlgebraFctNgb, ST_MapAlgebra (callback function version), ST_Min4ma, ST_Max4ma, ST_Sum4ma, ST_Range4ma, ST_StdDev4ma

11.13.5 ST_Min4ma

ST_Min4ma — □□□□□□□□□□□□□□□□□□□□□□□□□□.

□□

float8 **ST_Min4ma**(float8[][] matrix, text nodatamode, text[] VARIADIC args);
double precision **ST_Min4ma**(double precision[][][] value, integer[][] pos, text[] VARIADIC userargs);

Description

□□□□□□□□□□□□□□□□□□□.

□□ 2 □□□, □□□□ userargs □□□□ NODATA □□□□□□□□□□□□□□□.

□□
□□ 1 □ ST_MapAlgebraFctNgb □□□□□□□□□□□□□□□□□□□□□.

□□
□□ 2 □ ST_MapAlgebra (callback function version) □□□□□□□□□□□□□□□□□□□□
□.

□□
2.1.0 □□□□ ST_MapAlgebraFctNgb □□□□□□□□□□□□ 1 □□□□□□□□□□□□□.

2.0.0 □□□□□□□□□□□□.

□□□□: 2.1.0 □□□□□□ 2 □□□□□□□.

□□

Code

```
SELECT
    rid,
    ST_Value(ST_MapAlgebraFctNgb(rast, 1, NULL, 1, 1, 'ST_Min4ma(float[][],text,text[])':: ↩
    regprocedure, 'ignore', NULL), 2, 2
    )
FROM dummy_rast
WHERE rid = 2;
```

Output

```
rid | st_value
-----+----------
   2 |      250
(1 row)
```

□□

ST_MapAlgebraFctNgb, ST_MapAlgebra (callback function version), ST_Max4ma, ST_Sum4ma, ST_Mean4m
ST_Range4ma, ST_Distinct4ma, ST_StdDev4ma

11.13.6 ST_MinDist4ma

ST_MinDist4ma — □□□□□□□□□□□□□□□□□□□□□□□□ (□□□□□□) □□□□□□□□□□□□□□□□.

□□

double precision **ST_MinDist4ma**(double precision[][][] value, integer[][] pos, text[] VARIADIC user-args);

Description

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ (□□□□□□) □□□□□.

> **Note!** □□
> □□□□□□□□ ST_InvDistWeight4ma □□. □□□□□□□□□□□□□□□□□□□□□□.

> **Note!** □□
> □□□□ ST_MapAlgebra (callback function version) □□□□□□□□□□□□□□□□□□□□□□.

2.1.0 □□□□□□□□□□□□.

□□

Code

```
WITH foo AS (
    SELECT ST_SetValues(ST_AddBand(ST_MakeEmptyRaster(3, 3, 0, 0, 1, -1, 0, 0, 0), 1, '32BF ↩
    ', NULL::double precision, -9999),
        1, 1, 1, ARRAY[
            [10, 10, 10],
            [10, NULL, 20],
            [20, 20, 20]
        ]::double precision[][]
    ) AS rast
)
SELECT ST_Value(
    ST_MapAlgebra(
        rast,
        1,
        'ST_MinDist4ma(double precision[][][],integer[][],text[])'::regprocedure,
        '32BF',
```

```
        'FIRST',
        NULL,
        1,
        1
    ),
    2, 2
) AS min_distance
FROM foo;
```

Output

```
1
```

⊠⊠

ST_MapAlgebra (callback function version), ST_InvDistWeight4ma

11.13.7 ST_Range4ma

ST_Range4ma — ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

float8 **ST_Range4ma**(float8[][] matrix, text nodatamode, text[] VARIADIC args);
double precision **ST_Range4ma**(double precision[][][] value, integer[][] pos, text[] VARIADIC userargs);

Description

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠ 2 ⊠⊠⊠, ⊠⊠⊠⊠ userargs ⊠⊠⊠⊠ NODATA ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

> ⊠⊠
> ⊠⊠ 1 ⊠ ST_MapAlgebraFctNgb ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

> ⊠⊠
> ⊠⊠ 2 ⊠ ST_MapAlgebra (callback function version) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

> ⊠⊠
> 2.1.0 ⊠⊠⊠⊠ ST_MapAlgebraFctNgb ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ 1 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠⊠⊠: 2.1.0 ⊠⊠⊠⊠⊠⊠ 2 ⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

Code

```
SELECT
    rid,
    ST_Value(ST_MapAlgebraFctNgb(rast, 1, NULL, 1, 1, 'ST_Range4ma(float[][],text,text[])' ↩
    ::regprocedure, 'ignore', NULL), 2, 2
    )
FROM dummy_rast
WHERE rid = 2;
```

Output

```
rid | st_value
-----+----------
   2 |        4
(1 row)
```

□□

ST_MapAlgebraFctNgb, ST_MapAlgebra (callback function version), ST_Min4ma, ST_Max4ma, ST_Sum4ma, ST_Mean4ma, ST_Distinct4ma, ST_StdDev4ma

11.13.8 ST_StdDev4ma

ST_StdDev4ma — □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□.

□□

float8 **ST_StdDev4ma**(float8[][] matrix, text nodatamode, text[] VARIADIC args);
double precision **ST_StdDev4ma**(double precision[][][] value, integer[][] pos, text[] VARIADIC user-args);

Description

□□□□□□□□□□□□□□□□□□□□□□□.

□□
□□ 1 □ ST_MapAlgebraFctNgb □□□□□□□□□□□□□□□□□□□□□.

□□
□□ 2 □ ST_MapAlgebra (callback function version) □□□□□□□□□□□□□□□□□□□□
□.

□□
2.1.0 □□□□ ST_MapAlgebraFctNgb □□□□□□□□□□□□ 1 □□□□□□□□□□□□□.

2.0.0 □□□□□□□□□□□□□.

□□□□: 2.1.0 □□□□□□ 2 □□□□□□□.

⊠⊠

Code

```
SELECT
    rid,
    ST_Value(ST_MapAlgebraFctNgb(rast, 1, '32BF', 1, 1, 'ST_StdDev4ma(float[][],text,text ←
    [])'::regprocedure, 'ignore', NULL), 2, 2
    )
FROM dummy_rast
WHERE rid = 2;
```

Output

```
rid |     st_value
-----+------------------
   2 | 1.30170822143555
(1 row)
```

⊠⊠

ST_MapAlgebraFctNgb, ST_MapAlgebra (callback function version), ST_Min4ma, ST_Max4ma, ST_Sum4ma, ST_Mean4ma, ST_Distinct4ma, ST_StdDev4ma

11.13.9 ST_Sum4ma

ST_Sum4ma — ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

float8 **ST_Sum4ma**(float8[][] matrix, text nodatamode, text[] VARIADIC args);
double precision **ST_Sum4ma**(double precision[][][] value, integer[][] pos, text[] VARIADIC userargs);

Description

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠ 2 ⊠⊠⊠, ⊠⊠⊠⊠ userargs ⊠⊠⊠⊠ NODATA ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

> **Note!** ⊠⊠
> ⊠⊠ 1 ⊠ ST_MapAlgebraFctNgb ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

> **Note!** ⊠⊠
> ⊠⊠ 2 ⊠ ST_MapAlgebra (callback function version) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.



□□
2.1.0 □□□□ ST_MapAlgebraFctNgb □□□□□□□□□□□□ 1 □□□□□□□□□□□□□.

2.0.0 □□□□□□□□□□□□.

□□□□: 2.1.0 □□□□□□□ 2 □□□□□□□□.

□□

Code

```
SELECT
    rid,
    ST_Value(ST_MapAlgebraFctNgb(rast, 1, '32BF', 1, 1, 'ST_Sum4ma(float[][],text,text[])' ↩
    ::regprocedure, 'ignore', NULL), 2, 2
    )
FROM dummy_rast
WHERE rid = 2;
```

Output

```
rid | st_value
-----+----------
   2 |     2279
(1 row)
```

□□

ST_MapAlgebraFctNgb, ST_MapAlgebra (callback function version), ST_Min4ma, ST_Max4ma, ST_Mean4m
ST_Range4ma, ST_Distinct4ma, ST_StdDev4ma

11.14 □□□□□□

11.14.1 ST_Aspect

ST_Aspect — □□□□□□□□□ (□□□□□□) □□□□□□. □□□□□□□□□□□□□□.

□□

raster **ST_Aspect**(raster rast, integer band=1, text pixeltype=32BF, text units=DEGREES, boolean interpolate_nodata=FALSE);
raster **ST_Aspect**(raster rast, integer band, raster customextent, text pixeltype=32BF, text units=DEGREES
boolean interpolate_nodata=FALSE);

Description

⊠⊠⊠⊠⊠⊠⊠⊠⊠ (⊠⊠⊠⊠⊠⊠) ⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

`units` ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. RADIANS, DEGREES(⊠⊠⊠) ⊠⊠⊠⊠⊠⊠⊠⊠.

`units` = RADIANS ⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ 0 ⊠ 2π ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

`units` = DEGREES ⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ 0 ⊠⊠ 360 ⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠⊠⊠⊠⊠ 0 ⊠⊠⊠, ⊠⊠⊠⊠⊠ -1 ⊠⊠⊠.

> **Note**
> ⊠⊠
> ⊠⊠ (slope), ⊠ (aspect), ⊠⊠⊠⊠ (hillshade) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ESRI - How hillshade works ⊠ ERDAS Field Guide - Aspect Images ⊠⊠⊠⊠⊠⊠⊠⊠.

2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠⊠⊠: 2.1.0 ⊠⊠⊠⊠ ST_MapAlgebra() ⊠⊠⊠⊠⊠, ⊠⊠⊠⊠ `interpolate_nodata` ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠.

⊠⊠⊠⊠: 2.1.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. 2.1.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠.

⊠⊠

Variant 1.

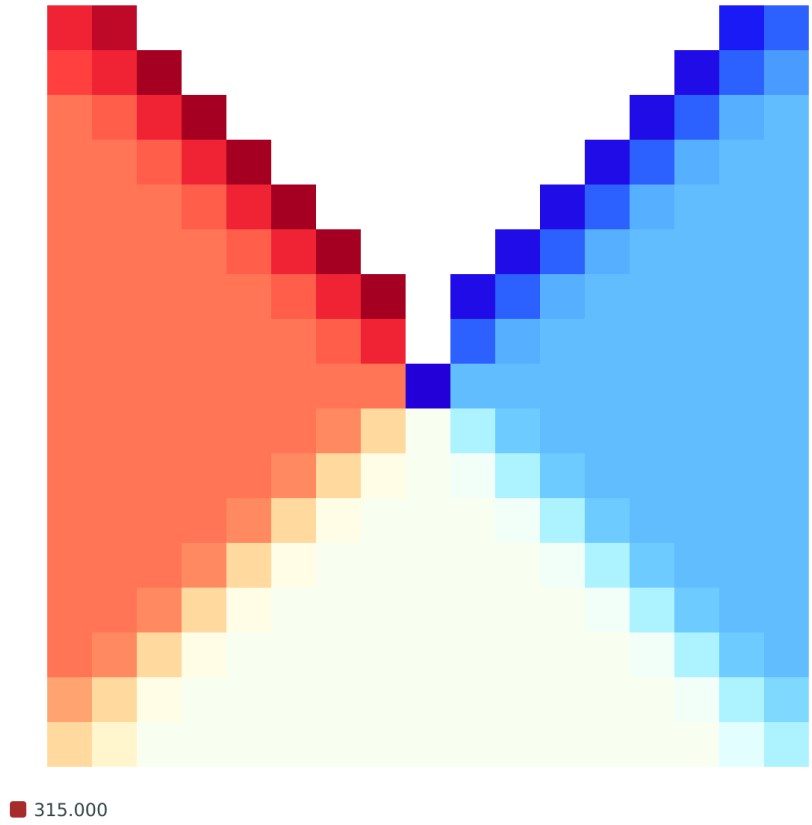
This symmetric peak highlights the expected compass directions at the corners and the `-1` aspect at the summit. The generated PNG shows the full aspect surface without relying on a manual illustration. The generated source raster is large enough for the PNG to show the directional field around the summit.

Code

```
WITH pixels AS (
    SELECT
        x,
        y,
        (20 - greatest(abs(x - 9), abs(y - 9)))::double precision AS value
    FROM generate_series(1, 17) AS y
    CROSS JOIN generate_series(1, 17) AS x
), rows AS (
    SELECT
        y,
        array_agg(value ORDER BY x) AS row_values
    FROM pixels
    GROUP BY y
), source AS (
    SELECT ST_SetValues(
        ST_AddBand(
            ST_MakeEmptyRaster(17, 17, 0, 0, 1, -1, 0, 0, 0),
            1, '32BF', 0, -9999
        ),
        1, 1, 1,
        array_agg(row_values ORDER BY y)::double precision[][]
    ) AS rast
    FROM rows
), aspect AS (
    SELECT ST_Aspect(rast, 1, '32BF') AS rast
```

```
    FROM source
)
SELECT
    round(ST_Value(rast, 1, 1, 1)::numeric, 3) AS northwest_degrees,
    round(ST_Value(rast, 1, 9, 9)::numeric, 3) AS summit_degrees,
    round(ST_Value(rast, 1, 17, 17)::numeric, 3) AS southeast_degrees,
    ST_AsPNG(ST_ColorMap(rast, 1, 'bluered')) AS image
FROM aspect;
```

Figure



Variant 2.

Complete example of tiles of a coverage.

Code

```
WITH foo AS (
    SELECT ST_Tile(ST_SetValues(ST_AddBand(ST_MakeEmptyRaster(6, 6, 0, 0, 1, -1, 0, 0, 0),
                1, '32BF', 0, -9999
            ),
            1, 1, 1, ARRAY[
                [1, 1, 1, 1, 1, 1],
                [1, 1, 1, 1, 2, 1],
                [1, 2, 2, 3, 3, 1],
                [1, 1, 3, 2, 1, 1],
                [1, 2, 2, 1, 2, 1],
                [1, 1, 1, 1, 1, 1]
            ]::double precision[]
        ),
        2, 2
    ) AS rast
```

```
)
SELECT
    t1.rast,
    ST_Aspect(ST_Union(t2.rast), 1, t1.rast)
FROM foo t1
CROSS JOIN foo t2
WHERE ST_Intersects(t1.rast, t2.rast)
GROUP BY t1.rast;
```

⊠⊠

ST_MapAlgebra (callback function version), ST_TRI, ST_TPI, ST_Roughness, ST_HillShade, ST_Slope

11.14.2 ST_HillShade

ST_HillShade — ⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

raster **ST_HillShade**(raster rast, integer band=1, text pixeltype=32BF, double precision azimuth=315, double precision altitude=45, double precision max_bright=255, double precision scale=1.0, boolean interpolate_nodata=FALSE);
raster **ST_HillShade**(raster rast, integer band, raster customextent, text pixeltype=32BF, double precision azimuth=315, double precision altitude=45, double precision max_bright=255, double precision scale=1.0, boolean interpolate_nodata=FALSE);

Description

⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠ 0 ⊠ 255 ⊠⊠⊠⊠⊠⊠.

azimuth ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ 0 ⊠⊠ 360 ⊠⊠⊠⊠⊠⊠⊠⊠⊠.

altitude ⊠ 0 ⊠⊠⊠⊠⊠, 90 ⊠⊠⊠⊠ (天頂) ⊠ 0 ⊠⊠ 90 ⊠⊠⊠⊠⊠⊠⊠⊠⊠.

max_bright ⊠ 0 ⊠⊠⊠⊠⊠⊠, 255 ⊠⊠⊠⊠⊠⊠ 0 ⊠ 255 ⊠⊠⊠⊠⊠⊠⊠.

scale ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠: ⊠⊠⊠⊠⊠⊠ scale=370400, ⊠⊠: ⊠⊠⊠⊠⊠⊠ scale=111120 ⊠⊠⊠⊠⊠⊠⊠.

interpolate_nodata ⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ST_InvDistWeight4ma ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ NODATA ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

> **⊠⊠**
> ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, How hillshade works ⊠⊠⊠⊠⊠⊠⊠.

2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠⊠⊠: 2.1.0 ⊠⊠⊠⊠ ST_MapAlgebra() ⊠⊠⊠⊠⊠, ⊠⊠⊠⊠ interpolate_nodata ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠⊠⊠: 2.1.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. 2.1.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

□□

Variant 1.

Code

```
WITH foo AS (
    SELECT ST_SetValues(ST_AddBand(ST_MakeEmptyRaster(5, 5, 0, 0, 1, -1, 0, 0, 0), 1, '32BF ↩
    ', 0, -9999),
        1, 1, 1, ARRAY[
            [1, 1, 1, 1, 1],
            [1, 2, 2, 2, 1],
            [1, 2, 3, 2, 1],
            [1, 2, 2, 2, 1],
            [1, 1, 1, 1, 1]
        ]::double precision[][]
    ) AS rast
)
SELECT
    ST_DumpValues(ST_HillShade(rast, 1, '32BF'))
FROM foo

                                                
```

□□

ST_MapAlgebra (callback function version), ST_TRI, ST_TPI, ST_Roughness, ST_Aspect, ST_Slope

11.14.3 ST_Roughness

ST_Roughness — DEM □□□□” □□□ (roughness)” □□□□□□□□□□□□.

□□

raster **ST_Roughness**(raster rast, integer nband, raster customextent, text pixeltype=”32BF” , boolean interpolate_nodata=FALSE);

Description

□□□□□□□□□□□□□□□□□ DEM □” □□□” □□□□□□.
2.1.0 □□□□□□□□□□□.

□□

Code

```
WITH foo AS (
    SELECT ST_SetValues(ST_AddBand(ST_MakeEmptyRaster(5, 5, 0, 0, 1, -1, 0, 0, 0), 1, '32BF ↩
    ', 0, -9999),
        1, 1, 1, ARRAY[
            [1, 1, 2, 3, 3],
            [1, 2, 4, 15, 3],
            [2, 4, 10, 7, 5],
            [3, 5, 7, 6, 4],
            [3, 3, 5, 4, 4]
        ]::double precision[][]
    ) AS rast
)
SELECT ST_Value(ST_Roughness(rast, 1, '32BF'), 3, 3) AS roughness
FROM foo;
```

Output

```
13
```

□□

ST_MapAlgebra (callback function version), ST_TRI, ST_TPI, ST_Slope, ST_HillShade, ST_Aspect

11.14.4 ST_Slope

ST_Slope — □□□□□□□□□□ (□□□□□□) □□□□□□. □□□□□□□□□□□□□□.

☒☒

raster **ST_Slope**(raster rast, integer nband=1, text pixeltype=32BF, text units=DEGREES, double precision scale=1.0, boolean interpolate_nodata=FALSE);
raster **ST_Slope**(raster rast, integer nband, raster customextent, text pixeltype=32BF, text units=DEGREES, double precision scale=1.0, boolean interpolate_nodata=FALSE);

Description

☒☒☒☒☒☒☒☒☒☒☒ (☒☒☒☒☒☒☒) ☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.

`units` ☒☒☒☒☒☒☒☒☒☒☒☒☒☒. RADIANS, DEGREES(☒☒☒), PERCENT ☒☒☒☒☒☒☒☒.

`scale` ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒: ☒☒☒☒☒☒☒ scale=370400, ☒☒: ☒☒☒☒☒☒☒ scale=111120 ☒☒☒☒☒☒☒☒.

`interpolate_nodata` ☒☒☒☒☒☒, ☒☒☒☒☒☒☒☒☒☒☒☒☒ ST_InvDistWeight4ma ☒☒☒☒☒☒☒☒☒☒☒ ☒☒☒☒ NODATA ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.

> ☒☒
> ☒☒ (slope), ☒ (aspect), ☒☒☒☒ (hillshade) ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒, ESRI - How hillshade works ☒ ERDAS Field Guide - Slope Images ☒☒☒☒☒☒☒☒.

2.0.0 ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.

☒☒☒☒: 2.1.0 ☒☒☒☒☒ ST_MapAlgebra() ☒☒☒☒☒☒, ☒☒☒☒☒ `units`, `scale`, `interpolate_nodata` ☒ ☒☒☒☒☒☒☒☒☒☒☒☒☒☒.

☒☒☒☒: 2.1.0 ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒. 2.1.0 ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒ ☒☒☒☒☒☒.

☒☒

Variant 1.

Code

```
WITH foo AS (
    SELECT ST_SetValues(ST_AddBand(ST_MakeEmptyRaster(5, 5, 0, 0, 1, -1, 0, 0, 0), 1, '32BF ↩
    ', 0, -9999),
        1, 1, 1, ARRAY[
            [1, 1, 1, 1, 1],
            [1, 2, 2, 2, 1],
            [1, 2, 3, 2, 1],
            [1, 2, 2, 2, 1],
            [1, 1, 1, 1, 1]
        ]::double precision[][]
    ) AS rast
)
SELECT
    ST_DumpValues(ST_Slope(rast, 1, '32BF'))
FROM foo

                            st_dumpvalues
```

Output

```
----------------------------------------------------------------------------------------------------------------------------------------------------------------

----------------------------------------------------------------------------------------------------------------------------------------------------------------

-------------------------------------------------------------------
 ↩
    (1,"{{10.0249881744385,21.5681285858154,26.5650520324707,21.5681285858154,10.0249881744385},{21.5681285858154,35.2643890380
{26.5650520324707,36.8698959350586,0,36.8698959350586,26.5650520324707},{21.5681285858154,35.2643890380859,36.8698959350586,35.2
5681285858154,26.5650520324707,21.5681285858154,10.0249881744385}}")
(1 row)
```

Variant 2.

Complete example of tiles of a coverage.

Code

```
WITH foo AS (
    SELECT ST_Tile(ST_SetValues(ST_AddBand(ST_MakeEmptyRaster(6, 6, 0, 0, 1, -1, 0, 0, 0),
                1, '32BF', 0, -9999
            ),
            1, 1, 1, ARRAY[
                [1, 1, 1, 1, 1, 1],
                [1, 1, 1, 1, 2, 1],
                [1, 2, 2, 3, 3, 1],
                [1, 1, 3, 2, 1, 1],
                [1, 2, 2, 1, 2, 1],
                [1, 1, 1, 1, 1, 1]
            ]::double precision[]
        ),
        2, 2
    ) AS rast
)
SELECT
    t1.rast,
    ST_Slope(ST_Union(t2.rast), 1, t1.rast)
FROM foo t1
CROSS JOIN foo t2
WHERE ST_Intersects(t1.rast, t2.rast)
GROUP BY t1.rast;
```

☒☒

ST_MapAlgebra (callback function version), ST_TRI, ST_TPI, ST_Roughness, ST_HillShade, ST_Aspect

11.14.5 ST_TPI

ST_TPI — ☒☒☒☒☒☒☒☒☒ (Topographic Position Index) ☒☒☒☒☒☒☒☒☒☒☒☒.

☒☒

raster **ST_TPI**(raster rast, integer nband, raster customextent, text pixeltype=”32BF” , boolean interpolate_nodata=FALSE);

Description

Calculates the Topographic Position Index, which is defined as the focal mean with radius of one minus the center cell.



□□
□□□□ 1 □□□□□□□ (focalmean radius of one) □□□□□□□.

2.1.0 □□□□□□□□□□□□.

□□

Code

```
WITH foo AS (
    SELECT ST_SetValues(ST_AddBand(ST_MakeEmptyRaster(5, 5, 0, 0, 1, -1, 0, 0, 0), 1, '32BF ↩
    ', 0, -9999),
        1, 1, 1, ARRAY[
            [1, 1, 2, 3, 3],
            [1, 2, 4, 15, 3],
            [2, 4, 10, 7, 5],
            [3, 5, 7, 6, 4],
            [3, 3, 5, 4, 4]
        ]::double precision[][]
    ) AS rast
)
SELECT ST_Value(ST_TPI(rast, 1, '32BF'), 3, 3) AS tpi
FROM foo;
```

Output

```
3.75
```

□□

ST_MapAlgebra (callback function version), ST_TRI, ST_Roughness, ST_Slope, ST_HillShade, ST_Aspect

11.14.6 ST_TRI

ST_TRI — □□□□□□□□□□ (Terrain Ruggedness Index) □□□□□□□□□□□□.

□□

raster **ST_TRI**(raster rast, integer nband, raster customextent, text pixeltype=”32BF” , boolean interpolate_nodata=FALSE);

Description

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (Terrain Ruggedness Index) ⊠⊠⊠⊠⊠⊠.

> **Note!**
> ⊠⊠
> ⊠⊠⊠⊠ 1 ⊠⊠⊠⊠⊠⊠ (focalmean radius of one) ⊠⊠⊠⊠⊠⊠⊠.

2.1.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

Code

```
WITH foo AS (
    SELECT ST_SetValues(ST_AddBand(ST_MakeEmptyRaster(5, 5, 0, 0, 1, -1, 0, 0, 0), 1, '32BF ↩
    ', 0, -9999),
        1, 1, 1, ARRAY[
            [1, 1, 2, 3, 3],
            [1, 2, 4, 15, 3],
            [2, 4, 10, 7, 5],
            [3, 5, 7, 6, 4],
            [3, 3, 5, 4, 4]
        ]::double precision[][]
    ) AS rast
)
SELECT ST_Value(ST_TRI(rast, 1, '32BF'), 3, 3) AS tri
FROM foo;
```

Output

```
5
```

⊠⊠

ST_MapAlgebra (callback function version), ST_Roughness, ST_TPI, ST_Slope, ST_HillShade, ST_Aspect

11.14.7 ST_InterpolateRaster

ST_InterpolateRaster — Interpolates a gridded surface based on an input set of 3-d points, using the X- and Y-values to position the points on the grid and the Z-value of the points as the surface elevation.

⊠⊠

raster **ST_InterpolateRaster**(geometry input_points, text algorithm_options, raster template, integer template_band_num=1);

Description

Interpolates a gridded surface based on an input set of 3-d points, using the X- and Y-values to position the points on the grid and the Z-value of the points as the surface elevation. There are five interpolation algorithms available: inverse distance, inverse distance nearest-neighbor, moving average, nearest neighbor, and linear interpolation. See the gdal_grid documentation for more details on the algorithms and their parameters. For more information on how interpolations are calculated, see the GDAL grid tutorial.

Input parameters are:

input_points The points to drive the interpolation. Any geometry with Z-values is acceptable, all points in the input will be used.

algorithm_options A string defining the algorithm and algorithm options, in the format used by gdal_grid. For example, for an inverse-distance interpolation with a smoothing of 2, you would use "invdist:smoothing=2.0"

template A raster template to drive the geometry of the output raster. The width, height, pixel size, spatial extent and pixel type will be read from this template.

template_band_num By default the first band in the template raster is used to drive the output raster, but that can be adjusted with this parameter.

Availability: 3.2.0

☒☒

Code

```
SELECT ST_InterpolateRaster('MULTIPOINT(10.5 9.5 1000,11.5 8.5 1000,10.5 8.5 500,11.5 9.5 ↩
    500)'::geometry,
    'invdist:smoothing:2.0',
    ST_AddBand(ST_MakeEmptyRaster(200, 400, 10, 10, 0.01, -0.005, 0, 0), '16BSI')
)
```

☒☒

ST_Contour

11.14.8 ST_Contour

ST_Contour — Generates a set of vector contours from the provided raster band, using the GDAL contouring algorithm.

☒☒

table(geom geometry, id integer, value float8) **ST_Contour**(raster rast, integer bandnumber=1, double precision level_interval=100.0, double precision level_base=0.0, double precision[] fixed_levels=ARRAY boolean polygonize=false);

Description

Generates a set of vector contours from the provided raster band, using the GDAL contouring algorithm.

When the `fixed_levels` parameter is a non-empty array, the `level_interval` and `level_base` parameters are ignored.

Input parameters are:

`rast` The raster to generate the contour of

`bandnumber` The band to generate the contour of

`level_interval` The elevation interval between contours generated

`level_base` The "base" relative to which contour intervals are applied, this is normally zero, but could be different. To generate 10m contours at 5, 15, 25, ... the LEVEL_BASE would be 5.

`fixed_levels` The elevation interval between contours generated

`polygonize` If `true`, contour polygons will be created, rather than polygon lines.

Return values are a set of records with the following attributes:

`geom` The geometry of the contour line.

`id` A unique identifier given to the contour line by GDAL.

`value` The raster value the line represents. For an elevation DEM input, this would be the elevation of the output contour.

Availability: 3.2.0

□□

Code

```
WITH c AS (
SELECT (ST_Contour(rast, 1, fixed_levels => ARRAY[100.0, 200.0, 300.0])).*
FROM dem_grid WHERE rid = 1
)
SELECT geom, id, value
FROM c;
```

□□

ST_InterpolateRaster

11.15 □□□□□□□□

11.15.1 Box3D

Box3D — □□□□□□□□□□ BOX3D □□□□□□□□□.

⊠⊠

box3d **Box3D**(raster rast);

Description

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ((MINX, MINY), (MAXX, MAXY)) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠⊠⊠: 2.0.0 ⊠⊠⊠⊠⊠⊠⊠ BOX3D ⊠⊠ BOX2D ⊠⊠⊠⊠⊠⊠⊠. BOX2D ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, 2.0.0 ⊠⊠⊠⊠ BOX3D ⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

Code

```
SELECT
    rid,
    Box3D(rast) AS rastbox
FROM dummy_rast;
```

Output

```
rid |        rastbox
----+-------------------------------------------------
1   | BOX3D(0.5 0.5 0,20.5 60.5 0)
2   | BOX3D(3427927.75 5793243.5 0,3427928 5793244 0)
```

⊠⊠

ST_Envelope

11.15.2 ST_ConvexHull

ST_ConvexHull — BandNoDataValue ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ST_Envelope ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

geometry **ST_ConvexHull**(raster rast);

Description

NoDataBandValue ⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠, ST_Envelope ⊠⊠.

> **⊠⊠**
> ST_Envelope ⊠⊠⊠⊠⊠⊠⊠⊠ (floor) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠ ST_ConvexHull ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

See `doc/development/internals/raster-affine.md` for the scale, skew, and rotation terms that make the convex hull and envelope differ for skewed rasters.

For regular non-skewed rasters, the envelope and convex hull are more or less the same.

Code

```
SELECT ST_ConvexHull(rast) As convhull, ST_Envelope(rast) As env
FROM dummy_rast WHERE rid=1;
```

Output

```
convhull                        |                env
--------------------------------------------------------+-----------------------------------
 POLYGON((0.5 0.5,20.5 0.5,20.5 60.5,0.5 60.5,0.5 0.5)) | POLYGON((0 0,20 0,20 60,0 60,0 0))
```

Now we skew the raster. Note how the convex hull and envelope are now different.

Code

```
SELECT ST_ConvexHull(rast) As convhull, ST_Envelope(rast) As env
FROM (SELECT ST_SetRotation(rast, 0.1, 0.1) As rast
    FROM dummy_rast WHERE rid=1) As foo;
```

Output

```
convhull                        |                env
--------------------------------------------------------+-----------------------------------
 POLYGON((0.5 0.5,20.5 1.5,22.5 61.5,2.5 60.5,0.5 0.5)) | POLYGON((0 0,22 0,22 61,0 61,0 0))
```

⊠⊠

ST_Envelope, ST_MinConvexHull, ST_ConvexHull, ST_AsText

11.15.3 ST_DumpAsPolygons

ST_DumpAsPolygons — ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ geomval(geom, val) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ 1 ⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

setof geomval **ST_DumpAsPolygons**(raster rast, integer band_num=1, boolean exclude_nodata_value=TRU

Description

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (SRF; Set-Returning Function) ⊠⊠⊠. ⊠⊠ (geom) ⊠⊠⊠⊠⊠⊠⊠ (val) ⊠⊠⊠⊠⊠⊠⊠ geomval ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ val ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠.

ST_DumpAsPolygon ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ GROUP BY ⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠/⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

Changed 3.3.0, validation and fixing is disabled to improve performance. May result invalid geometries.

Changed 3.7.0, the polygonization honours PostgreSQL interrupts so cancellations and statement timeouts halt processing promptly.

GDAL 1.7 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠
If there is a no data value set for a band, pixels with that value will not be returned except in the case of exclude_nodata_value=false.

⊠⊠
⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ST_ValueCount ⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠
⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ST_PixelAsPolygons ⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

This syntax uses a LATERAL join.

Code

```
SELECT val,geom As geomwkt
FROM (
SELECT dp.*
FROM dummy_rast, LATERAL ST_DumpAsPolygons(rast) AS dp
WHERE rid = 2
) As foo
WHERE val BETWEEN 249 and 251
ORDER BY val;
```

Output

```
val |                                                       geomwkt
-----+--------------------------------------------------------------------------
 249 | POLYGON((3427927.95 5793243.95,3427927.95 5793243.85,3427928 5793243.85,
        3427928 5793243.95,3427927.95 5793243.95))
 250 | POLYGON((3427927.75 5793243.9,3427927.75 5793243.85,3427927.8 5793243.85,
        3427927.8 5793243.9,3427927.75 5793243.9))
 250 | POLYGON((3427927.8 5793243.8,3427927.8 5793243.75,3427927.85 5793243.75,
        3427927.85 5793243.8, 3427927.8 5793243.8))
 251 | POLYGON((3427927.75 5793243.85,3427927.75 5793243.8,3427927.8 5793243.8,
        3427927.8 5793243.85,3427927.75 5793243.85))
```

⊠⊠

geomval, ST_AsRasterAgg, ST_Value, ST_Polygon, ST_ValueCount

11.15.4 ST_Envelope

ST_Envelope — ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

geometry **ST_Envelope**(raster rast);

Description

□□□□□□□□□□□□□ SRID □□□□□□□□□□□□□□□□□□□. □□□□□□□□□□□□□□□□□□□ float8 □□□□□□□□□□□□.

□□□□□□□□□□□□□□□□□□□□□□□□□□□ ((MINX, MINY), (MINX, MAXY), (MAXX, MAXY), (MAXX, MINY), (MINX, MINY)).

□□

Code

```
SELECT rid, ST_Envelope(rast) As envgeomwkt
FROM dummy_rast;
```

Output

```
rid |                                        envgeomwkt
-----+-------------------------------------------------------------------------
   1 | POLYGON((0 0,20 0,20 60,0 60,0 0))
   2 | POLYGON((3427927 5793243,3427928 5793243,
        3427928 5793244,3427927 5793244, 3427927 5793243))
```

□□

ST_Envelope, ST_AsText, ST_SRID

11.15.5 ST_MinConvexHull

ST_MinConvexHull — □□□□ NODATA □□□□□□□□□□□□□□□□□□□□□.

□□

geometry **ST_MinConvexHull**(raster rast, integer nband=NULL);

Description

□□□□ NODATA □□□□□□□□□□□□□□□□□□□□. nband □ NULL □□□, □□□□□□□□□□□□□□□□□.

2.1.0 □□□□□□□□□□□□□.

□□

Code

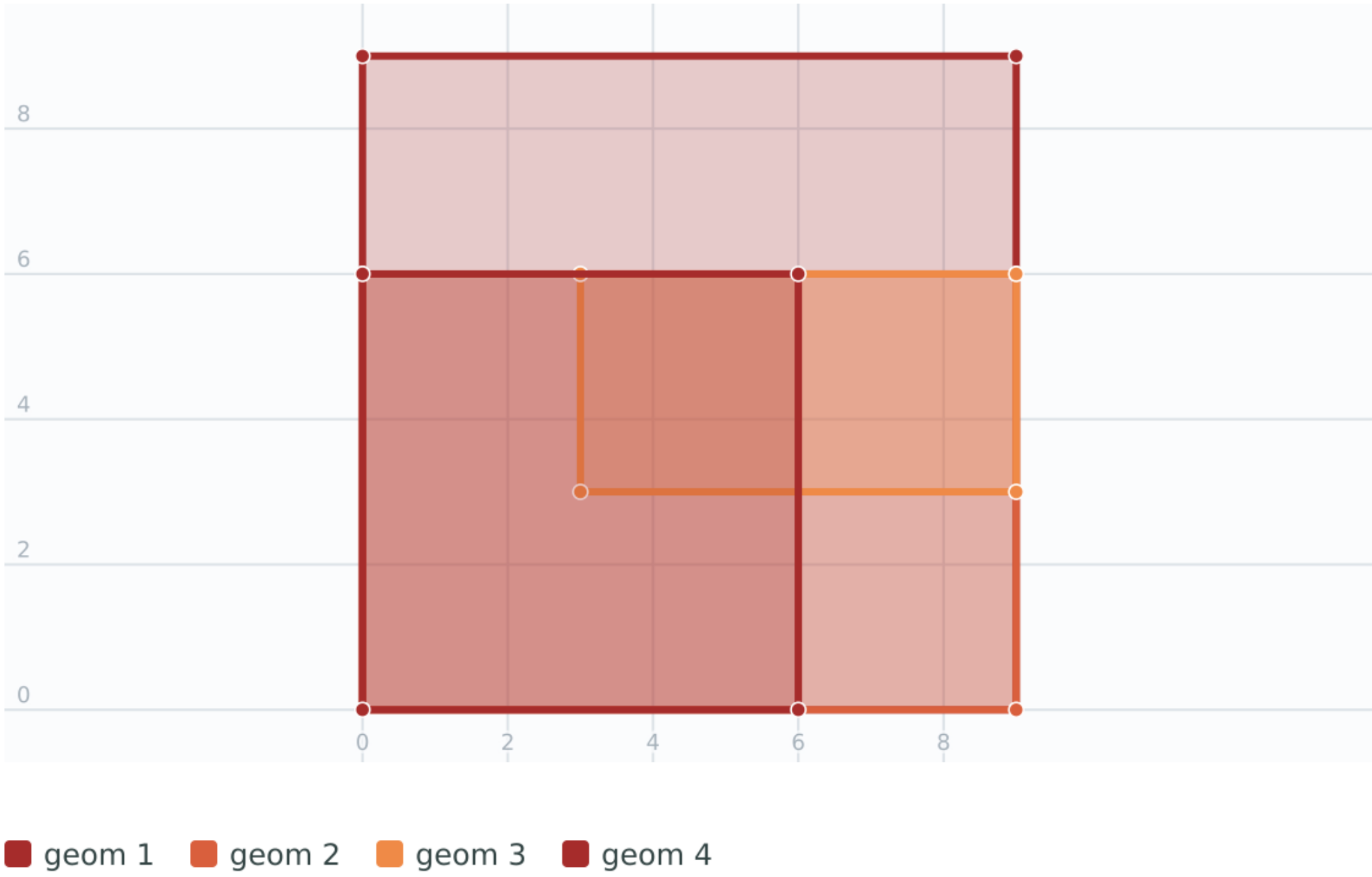
```
WITH foo AS (
    SELECT
        ST_SetValues(ST_SetValues(ST_AddBand(ST_AddBand(ST_MakeEmptyRaster(9, 9, 0, 0, 1,  ↩
    -1, 0, 0, 0), 1, '8BUI', 0, 0), 2, '8BUI', 1, 0),
                1, 1, 1,
                ARRAY[
                    [0, 0, 0, 0, 0, 0, 0, 0, 0],
```

```
                    [0, 0, 0, 0, 0, 0, 0, 0, 0],
                    [0, 0, 0, 0, 0, 0, 0, 0, 0],
                    [0, 0, 0, 1, 0, 0, 0, 0, 1],
                    [0, 0, 0, 1, 1, 0, 0, 0, 0],
                    [0, 0, 0, 1, 0, 0, 0, 0, 0],
                    [0, 0, 0, 0, 0, 0, 0, 0, 0],
                    [0, 0, 0, 0, 0, 0, 0, 0, 0],
                    [0, 0, 0, 0, 0, 0, 0, 0, 0]
                ]::double precision[][]
            ),
            2, 1, 1,
            ARRAY[
                [0, 0, 0, 0, 0, 0, 0, 0, 0],
                [0, 0, 0, 0, 0, 0, 0, 0, 0],
                [0, 0, 0, 0, 0, 0, 0, 0, 0],
                [1, 0, 0, 0, 0, 1, 0, 0, 0],
                [0, 0, 0, 0, 1, 1, 0, 0, 0],
                [0, 0, 0, 0, 0, 1, 0, 0, 0],
                [0, 0, 0, 0, 0, 0, 0, 0, 0],
                [0, 0, 0, 0, 0, 0, 0, 0, 0],
                [0, 0, 1, 0, 0, 0, 0, 0, 0]
            ]::double precision[][]
        ) AS rast
)
SELECT title, geom AS geom
FROM foo
CROSS JOIN LATERAL (VALUES
    ('raster hull', ST_ConvexHull(rast)),
    ('all bands', ST_MinConvexHull(rast)),
    ('band 1', ST_MinConvexHull(rast, 1)),
    ('band 2', ST_MinConvexHull(rast, 2))
) AS variants(title, geom);
```

Output

```
title    |                geom
------------+-------------------------------------
 raster hull | POLYGON((0 0,9 0,9 -9,0 -9,0 0))
 all bands   | POLYGON((0 -3,9 -3,9 -9,0 -9,0 -3))
 band 1      | POLYGON((3 -3,9 -3,9 -6,3 -6,3 -3))
 band 2      | POLYGON((0 -3,6 -3,6 -9,0 -9,0 -3))
```

Figure



⊠⊠

ST_Envelope, ST_ConvexHull, ST_ConvexHull, ST_AsText

11.15.6 ST_Polygon

ST_Polygon — NODATA ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

geometry **ST_Polygon**(raster rast, integer band_num=1);

Description

Changed 3.3.0, validation and fixing is disabled to improve performance. May result invalid geometries.

0.1.6 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. GDAL 1.7 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠⊠⊠: 2.1.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (⊠⊠⊠ C ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠). ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠⊠⊠: 2.1.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

Compare polygonization of every pixel with polygonization after zero is assigned as the NODATA value. The latter preserves the gap formed by zero-valued cells.

Code

```
WITH source AS (
    SELECT ST_SetValues(
        ST_AddBand(
            ST_MakeEmptyRaster(4, 4, 0, 4, 1, -1, 0, 0, 0),
            1, '8BUI', 1, NULL
        ),
        1, 1, 1,
        ARRAY[
            [1, 1, 1, 1],
            [1, 0, 0, 1],
            [1, 1, 0, 1],
            [1, 1, 1, 1]
        ]::double precision[][]
    ) AS rast
), variants AS (
    SELECT 'all pixels' AS title, ST_Polygon(rast) AS geom
    FROM source
    UNION ALL
    SELECT
        'zero excluded',
        ST_Polygon(ST_SetBandNoDataValue(rast, 1, 0))
    FROM source
)
SELECT title, geom AS geom
FROM variants
ORDER BY CASE title
    WHEN 'all pixels' THEN 1
    ELSE 2
END;
```

Figure



□□

ST_Value, ST_DumpAsPolygons

11.15.7 ST_IntersectionFractions

ST_IntersectionFractions — Calculates the fraction of each raster cell that is covered by a given geometry.

□□

raster **ST_IntersectionFractions**(raster rast, geometry geom);

Description

Calculates the fraction of each raster cell that is covered by a given geometry. The first argument is a raster, which defines the grid geometry to use for the calculation. The extent and cell size are read from the raster parameter. The second argument is a geometry, which is overlaid with the grid, and each grid populated based on overlaying the geometry on the grid. For polygons, the value returned for each cell is the proportion of its area that is covered by the geometry. For linestrings, the value returned for each cell is the length contained in the cell.

Availability: 3.6.0 Requires GEOS 3.14 or higher.

□□

This rotated square polygon covers half of each cell in the raster.

Code

```
CREATE TABLE raster_proportions_rast (
    name text,
    rast raster
);

INSERT INTO raster_proportions_rast (name, rast) VALUES (
  '2x2 raster covering 0,0 to 10,10',
  ST_MakeEmptyRaster(2, 2, -- raster width/height in pixels
    0, 10, -- upper-left corner x/y coordinates
    5, -5, -- pixel width/height in ground units
    0, 0, -- skew x/y
    0        -- SRID
  ));

SELECT name, ST_DumpValues(
    ST_IntersectionFractions(
        rast,
        'POLYGON((5 0,0 5,5 10,10 5,5 0))'::geometry),
    1)
FROM raster_proportions_rast;
```

Output

```
2x2 raster covering 0,0 to 10,10
---------------------------------
 {{0.5,0.5},{0.5,0.5}}
```

⊠⊠

ST_MakeEmptyRaster

11.16 ⊠⊠⊠⊠⊠⊠

11.16.1 &&

&& — A ⊠⊠⊠⊠⊠⊠ B ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ TRUE ⊠⊠⊠⊠⊠⊠.

⊠⊠

boolean **&&**(raster A , raster B);
boolean **&&**(raster A , geometry B);
boolean **&&**(geometry B , raster A);

Description

&& ⊠⊠⊠⊠⊠⊠⊠/⊠⊠ A ⊠⊠⊠⊠⊠⊠⊠⊠⊠/⊠⊠ B ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ TRUE ⊠⊠⊠⊠⊠⊠.

> **Note!** ⊠⊠
> ⊠⊠⊠⊠⊠ (operand) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

Code

```
SELECT A.rid As a_rid, B.rid As b_rid, A.rast && B.rast As intersect
FROM dummy_rast AS A CROSS JOIN dummy_rast AS B LIMIT 3;
```

Output

```
a_rid | b_rid | intersect
-------+-------+---------
     2 |     2 | t
     2 |     3 | f
     2 |     1 | f
```

11.16.2 &<

&< — A ⊠⊠⊠⊠⊠⊠ B ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ TRUE ⊠⊠⊠⊠⊠⊠.

⊠⊠

boolean **&<**(raster A , raster B);

Description

&< ⊠⊠⊠⊠⊠⊠⊠⊠ A ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ B ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠ B ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, TRUE ⊠⊠⊠⊠⊠⊠⊠.

> **Note!**
> ⊠⊠
> ⊠⊠⊠⊠⊠ (operand) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

Code

```
SELECT A.rid As a_rid, B.rid As b_rid, A.rast &< B.rast As overleft
 FROM dummy_rast AS A CROSS JOIN dummy_rast AS B;
```

Output

```
a_rid | b_rid | overleft
------+-------+----------
    2 |     2 | t
    2 |     3 | f
    2 |     1 | f
    3 |     2 | t
    3 |     3 | t
    3 |     1 | f
    1 |     2 | t
    1 |     3 | t
    1 |     1 | t
```

11.16.3 &>

&> — A ⊠⊠⊠⊠⊠⊠⊠ B ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ TRUE ⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

boolean **&>**(raster A , raster B);

Description

&> ⊠⊠⊠⊠⊠⊠⊠⊠ A ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ B ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠ B ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, TRUE ⊠⊠⊠⊠⊠⊠⊠.

> **Note!**
> ⊠⊠
> ⊠⊠⊠⊠⊠ (operand) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

Code

```
SELECT A.rid As a_rid, B.rid As b_rid, A.rast &
> B.rast As overright
 FROM dummy_rast AS A CROSS JOIN dummy_rast AS B;
```

Output

```
 a_rid | b_rid | overright
-------+-------+----------
     2 |     2 | t
     2 |     3 | t
     2 |     1 | t
     3 |     2 | f
     3 |     3 | t
     3 |     1 | f
     1 |     2 | f
     1 |     3 | t
     1 |     1 | t
```

11.16.4 =

= — A ⊠⊠⊠⊠⊠⊠⊠ B ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ TRUE ⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

boolean **=**(raster A , raster B);

Description

= ⊠⊠⊠⊠⊠⊠⊠⊠⊠ A ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ B ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ TRUE ⊠⊠⊠⊠⊠⊠⊠. PostgreSQL ⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ =, <, ⊠ > ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (⊠: GROUP BY ⊠⊠ ORDER BY ⊠⊠⊠).

> **⊠⊠**
> ⊠⊠⊠⊠⊠⊠ (operand) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠ ~= ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (group by) ⊠⊠⊠⊠⊠⊠⊠.

2.1.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

~=

11.16.5 @

@ — B ⊠⊠⊠⊠⊠⊠⊠ A ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ TRUE ⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

boolean **@**(raster A , raster B);
boolean **@**(geometry A , raster B);
boolean **@**(raster B , geometry A);

Description

@ ⊠⊠⊠⊠⊠⊠⊠/⊠⊠ B ⊠⊠⊠⊠⊠⊠⊠⊠⊠/⊠⊠ A ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ TRUE ⊠⊠⊠⊠⊠⊠.

> **Note!**
> ⊠⊠
> ⊠⊠⊠⊠⊠ (operand) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

2.0.0 ⊠⊠⊠⊠ raster @ raster, raster @ geometry ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

2.0.5 ⊠⊠⊠⊠ geometry @ raster ⊠⊠⊠⊠⊠⊠.

⊠⊠

~

11.16.6 ~=

~= — A ⊠⊠⊠⊠⊠⊠ B ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ TRUE ⊠⊠⊠⊠⊠⊠.

⊠⊠

boolean **~=**(raster A , raster B);

Description

~= ⊠⊠⊠⊠⊠⊠⊠ A ⊠⊠⊠⊠⊠⊠⊠⊠⊠ B ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ TRUE ⊠⊠⊠⊠⊠⊠.

> **Note!**
> ⊠⊠
> ⊠⊠⊠⊠⊠ (operand) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ 2 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠.

Code

```
SELECT ST_AddBand(prec.rast, alt.rast) As new_rast
FROM prec INNER JOIN alt ON (prec.rast ~= alt.rast);
```

⊠⊠

ST_AddBand, =

11.16.7 ~

~ — A ⊠⊠⊠⊠⊠⊠ B ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ TRUE ⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

boolean **~**(raster A , raster B);
boolean **~**(geometry A , raster B);
boolean **~**(raster B , geometry A);

Description

~ ⊠⊠⊠⊠⊠⊠⊠/⊠⊠ A ⊠⊠⊠⊠⊠⊠⊠⊠⊠/⊠⊠ B ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ TRUE ⊠⊠⊠⊠⊠⊠.

> **Note!**
> ⊠⊠
> ⊠⊠⊠⊠⊠ (operand) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

@

11.17 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠

11.17.1 ST_Contains

ST_Contains — ⊠⊠⊠ rastA ⊠⊠⊠⊠⊠⊠⊠⊠ rastB ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠ rastB ⊠⊠⊠⊠⊠⊠⊠⊠ ⊠ rastA ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

boolean **ST_Contains**(raster rastA , integer nbandA , raster rastB , integer nbandB);
boolean **ST_Contains**(raster rastA , raster rastB);

Description

⊠⊠⊠ rastA ⊠⊠⊠⊠⊠⊠⊠⊠ rastB ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠ rastB ⊠⊠⊠⊠⊠⊠⊠⊠ rastA ⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠ rastA ⊠ rastB ⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ NULL ⊠⊠⊠⊠⊠⊠, ⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠ (NODATA ⊠⊠⊠) ⊠⊠⊠⊠⊠ ⊠⊠⊠⊠.

> **Note!**
> ⊠⊠
> ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

☒☒
☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒, ST_Contains(ST_Polygon(raster), geometry) ☒☒ ST_Contains(geometry, ST_Polygon(raster)) ☒☒☒☒☒☒ ST_Polygon ☒☒☒☒☒☒☒☒☒.

☒☒
ST_Contains() ☒ ST_Within() ☒☒☒☒☒☒☒. ☒☒☒, ST_Contains(rastA, rastB) ☒☒☒☒ ST_Within(rastB, rastA) ☒☒☒☒☒☒☒☒☒☒☒☒.

2.1.0 ☒☒☒☒☒☒☒☒☒☒☒☒.

☒☒

Specified band numbers.

Code

```
SELECT r1.rid, r2.rid, ST_Contains(r1.rast, 1, r2.rast, 1) FROM dummy_rast r1 CROSS JOIN ↩
    dummy_rast r2 WHERE r1.rid = 1;
```

Output

```
NOTICE:  The first raster provided has no bands
 rid | rid | st_contains
-----+-----+-------------
   1 |   1 |
   1 |   2 | f
```

No band numbers specified.

Code

```
SELECT r1.rid, r2.rid, ST_Contains(r1.rast, r2.rast) FROM dummy_rast r1 CROSS JOIN ↩
    dummy_rast r2 WHERE r1.rid = 1;
```

Output

```
rid | rid | st_contains
-----+-----+-------------
   1 |   1 | t
   1 |   2 | f
```

☒☒

ST_Intersects, ST_Within

11.17.2 ST_ContainsProperly

ST_ContainsProperly — rastB ☒ rastA ☒☒☒☒☒☒☒☒☒☒ rastA ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒ ☒☒☒☒☒☒.

☒☒

boolean **ST_ContainsProperly**(raster rastA , integer nbandA , raster rastB , integer nbandB);
boolean **ST_ContainsProperly**(raster rastA , raster rastB);

Description

⊠⊠⊠ rastB ⊠⊠⊠⊠ rastA ⊠⊠⊠⊠⊠⊠⊠⊠⊠ rastA ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ rastA ⊠ rastB ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ NULL ⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠ (NODATA ⊠⊠⊠) ⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠⊠ rastA ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

> **Note!**
> ⊠⊠
> ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

> **Note!**
> ⊠⊠
> ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ST_ContainsProperly(ST_Polygon(raster), geometry) ⊠⊠ ST_ContainsProperly(geometry, ST_Polygon(raster)) ⊠⊠⊠⊠⊠⊠ ST_Polygon ⊠⊠⊠⊠⊠⊠⊠⊠⊠.

2.1.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

Code

```
SELECT r1.rid, r2.rid, ST_ContainsProperly(r1.rast, 1, r2.rast, 1) FROM dummy_rast r1 CROSS ↩
    JOIN dummy_rast r2 WHERE r1.rid = 2;
```

Output

```
rid | rid | st_containsproperly
-----+-----+---------------------
   2 |   1 | f
   2 |   2 | f
```

⊠⊠

ST_Intersects, ST_Contains

11.17.3 ST_Covers

ST_Covers — ⊠⊠⊠ rastB ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ rastA ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

boolean **ST_Covers**(raster rastA , integer nbandA , raster rastB , integer nbandB);
boolean **ST_Covers**(raster rastA , raster rastB);

Description

⊠⊠⊠ rastB ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ rastA ⊠⊠⊠⊠⊠⊠⊠ rastA ⊠ rastB ⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠ NULL ⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠ ⊠⊠⊠⊠⊠⊠ (NODATA ⊠⊠⊠) ⊠⊠⊠⊠⊠⊠⊠⊠⊠.

> **Note!**
> ⊠⊠
> ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

> **Note!**
> ⊠⊠
> ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ST_Covers(ST_Polygon(raster), geometry) ⊠⊠ ST_Covers(geometry, ST_Polygon(raster)) ⊠⊠⊠⊠⊠⊠ ST_Polygon ⊠⊠⊠⊠⊠⊠⊠⊠⊠.

2.1.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

Code

```
SELECT r1.rid, r2.rid, ST_Covers(r1.rast, 1, r2.rast, 1) FROM dummy_rast r1 CROSS JOIN ↩
    dummy_rast r2 WHERE r1.rid = 2;
```

Output

```
rid | rid | st_covers
-----+-----+-----------
   2 |   1 | f
   2 |   2 | t
```

⊠⊠

ST_Intersects, ST_CoveredBy

11.17.4 ST_CoveredBy

ST_CoveredBy — ⊠⊠⊠ rastA ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ rastB ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

boolean **ST_CoveredBy**(raster rastA , integer nbandA , raster rastB , integer nbandB);
boolean **ST_CoveredBy**(raster rastA , raster rastB);

Description

⊠⊠⊠ rastA ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ rastB ⊠⊠⊠⊠⊠⊠⊠ rastA ⊠ rastB ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ NULL ⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (NODATA ⊠⊠⊠) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

> **Note**
> ⊠⊠
> ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

> **Note**
> ⊠⊠
> ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ST_CoveredBy(ST_Polygon(raster), geometry) ⊠⊠ ST_CoveredBy(geometry, ST_Polygon(raster)) ⊠⊠⊠⊠⊠⊠ ST_Polygon ⊠⊠⊠⊠⊠⊠⊠⊠⊠.

2.1.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

Code

```
SELECT r1.rid, r2.rid, ST_CoveredBy(r1.rast, 1, r2.rast, 1) FROM dummy_rast r1 CROSS JOIN ↩
    dummy_rast r2 WHERE r1.rid = 2;
```

Output

```
rid | rid | st_coveredby
-----+-----+--------------
   2 |   1 | f
   2 |   2 | t
```

⊠⊠

ST_Intersects, ST_Covers

11.17.5 ST_Disjoint

ST_Disjoint — ⊠⊠⊠ rastA ⊠⊠⊠⊠ rastB ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

boolean **ST_Disjoint**(raster rastA , integer nbandA , raster rastB , integer nbandB);
boolean **ST_Disjoint**(raster rastA , raster rastB);

Description

⊠⊠⊠ rastA ⊠⊠⊠⊠ rastB ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ rastA ⊠ rastB ⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ NULL ⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠ (NODATA ⊠⊠⊠) ⊠⊠⊠⊠⊠⊠⊠⊠⊠.

> **Note!** ⊠⊠
> ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

> **Note!** ⊠⊠
> ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ST_Disjoint(ST_Polygon(raster), geometry) ⊠⊠⊠⊠⊠⊠ ST_Polygon ⊠⊠⊠⊠⊠⊠⊠⊠⊠.

2.1.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

Rid = 1 has no bands, hence the NOTICE and the NULL value for st_disjoint.

Code

```
SELECT r1.rid, r2.rid, ST_Disjoint(r1.rast, 1, r2.rast, 1) FROM dummy_rast r1 CROSS JOIN ↩
    dummy_rast r2 WHERE r1.rid = 2;
```

Output

```
NOTICE:  The second raster provided has no bands
 rid | rid | st_disjoint
-----+-----+-------------
   2 |   1 |
   2 |   2 | f
```

This time, without specifying band numbers.

Code

```
SELECT r1.rid, r2.rid, ST_Disjoint(r1.rast, r2.rast) FROM dummy_rast r1 CROSS JOIN ↩
    dummy_rast r2 WHERE r1.rid = 2;
```

Output

```
rid | rid | st_disjoint
-----+-----+-------------
   2 |   1 | t
   2 |   2 | f
```

⊠⊠

ST_Intersects

11.17.6 ST_Intersects

ST_Intersects — ⊠⊠⊠ rastA ⊠⊠⊠⊠ rastB ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

boolean **ST_Intersects**(raster rastA , integer nbandA , raster rastB , integer nbandB);
boolean **ST_Intersects**(raster rastA , raster rastB);
boolean **ST_Intersects**(raster rast , integer nband , geometry geommin);
boolean **ST_Intersects**(raster rast , geometry geommin , integer nband=NULL);
boolean **ST_Intersects**(geometry geommin , raster rast , integer nband=NULL);

Description

⊠⊠⊠ rastA ⊠⊠⊠⊠ rastB ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ NULL ⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (NODATA ⊠⊠⊠) ⊠⊠⊠⊠⊠⊠⊠⊠⊠.

> **⊠⊠**
> ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠⊠⊠: 2.0.0 ⊠⊠⊠⊠⊠⊠⊠/⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

> **⊠⊠**
> ⊠⊠⊠⊠: 2.1.0 ⊠⊠⊠⊠ ST_Intersects(geometry, raster) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ST_Intersects(raster, geometry) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

Different bands of same raster.

Code

```
SELECT ST_Intersects(rast, 2, rast, 3) FROM dummy_rast WHERE rid = 2;
```

Output

```
t
```

⊠⊠

ST_Intersection, ST_Disjoint

11.17.7 ST_Overlaps

ST_Overlaps — ⊠⊠⊠ rastA ⊠⊠⊠⊠ rastB ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

boolean **ST_Overlaps**(raster rastA , integer nbandA , raster rastB , integer nbandB);
boolean **ST_Overlaps**(raster rastA , raster rastB);

Description

⊠⊠⊠ rastA ⊠⊠⊠⊠ rastB ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠ rastA ⊠ rastB ⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ NULL ⊠⊠⊠⊠⊠⊠, ⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (NODATA ⊠⊠⊠) ⊠⊠⊠⊠ ⊠⊠⊠⊠⊠.

> ⊠⊠
> ⊠⊠.

> ⊠⊠
> ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ST_Overlaps(ST_Polygon(raster), geometry) ⊠⊠⊠⊠⊠⊠ ST_Polygon ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

2.1.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

Comparing different bands of same raster.

Code

```
SELECT ST_Overlaps(rast, 1, rast, 2) FROM dummy_rast WHERE rid = 2;
```

Output

```
f
```

⊠⊠

ST_Intersects

11.17.8 ST_Touches

ST_Touches — ⊠⊠⊠ rastA ⊠⊠⊠⊠ rastB ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ TRUE ⊠⊠⊠⊠⊠⊠.

⊠⊠

boolean **ST_Touches**(raster rastA , integer nbandA , raster rastB , integer nbandB);
boolean **ST_Touches**(raster rastA , raster rastB);

Description

☒☒☒ rastA ☒☒☒☒ rastB ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒ rastA ☒☒☒☒ rastB ☒☒☒☒ ☒☒☒☒☒☒☒☒☒☒☒☒☒, ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒ NULL ☒☒☒☒☒☒, ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒, ☒☒☒☒☒☒☒☒☒☒ (NODATA ☒☒☒) ☒ ☒☒☒☒☒☒☒☒☒.

☒☒
☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.

☒☒
☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒, ST_Touches(ST_Polygon(raster), geometry) ☒☒☒☒☒☒ ST_Polygon ☒☒☒☒☒☒☒☒☒.

2.1.0 ☒☒☒☒☒☒☒☒☒☒☒☒☒.

☒☒

Code

```
SELECT r1.rid, r2.rid, ST_Touches(r1.rast, 1, r2.rast, 1) FROM dummy_rast r1 CROSS JOIN ↩
    dummy_rast r2 WHERE r1.rid = 2;
```

Output

```
rid | rid | st_touches
-----+-----+------------
   2 |   1 | f
   2 |   2 | f
```

☒☒

ST_Intersects

11.17.9 ST_SameAlignment

ST_SameAlignment — ☒☒☒☒☒☒☒☒☒☒☒, ☒☒, ☒☒☒☒☒☒☒, ☒☒☒☒☒☒☒☒☒☒☒☒☒☒ (☒☒☒ ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒) ☒☒☒☒☒☒, ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒ ☒☒☒☒☒☒☒☒☒☒☒.

☒☒

boolean **ST_SameAlignment**(raster rastA , raster rastB);
boolean **ST_SameAlignment**(double precision ulx1 , double precision uly1 , double precision scalex1 , double precision scaley1 , double precision skewx1 , double precision skewy1 , double precision ulx2 , double precision uly2 , double precision scalex2 , double precision scaley2 , double precision skewx2 , double precision skewy2);
boolean **ST_SameAlignment**(raster set rastfield);

Description

⊠⊠⊠⊠⊠⊠⊠⊠⊠ (⊠⊠ 1, 2): (⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠, ⊠⊠⊠, SRID ⊠⊠⊠⊠⊠⊠) ⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠, ⊠⊠⊠, SRID ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ 4 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (NOTICE) ⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠.

⊠⊠⊠⊠⊠⊠ (⊠⊠ 3): ⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ST_SameAlignment(⊠⊠⊠ PostgreSQL ⊠⊠⊠⊠⊠” ⊠⊠ (aggregate)” ⊠⊠⊠⊠⊠. ⊠ SUM() ⊠ AVG() ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠⊠⊠: 2.1.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠: ⊠⊠⊠

Code

```
SELECT ST_SameAlignment(ST_MakeEmptyRaster(1, 1, 0, 0, 1, 1, 0, 0),
    ST_MakeEmptyRaster(1, 1, 0, 0, 1, 1, 0, 0)
) as sm;
```

Output

```
t
```

Code

```
SELECT ST_SameAlignment(A.rast, b.rast)
FROM dummy_rast AS A CROSS JOIN dummy_rast AS B;
```

Output

```
NOTICE:  The two rasters provided have different SRIDs
NOTICE:  The two rasters provided have different SRIDs
 t
 f
 f
 f
```

⊠⊠

10.1⊠, ST_NotSameAlignmentReason, ST_MakeEmptyRaster

11.17.10 ST_NotSameAlignmentReason

ST_NotSameAlignmentReason — ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

text **ST_NotSameAlignmentReason**(raster rastA, raster rastB);

Description

□□□□□□□□□□□□□□, □□□□□□□□□□□□□□□□□□□□□□□□□□□□.

> **Note!**
> □□
> □□□□□□□□□□□□□□□□□□□□□□, □□□□□ (□□□□□□□□□□) □□□□□□□□
> □.

2.1.0 □□□□□□□□□□□□.

□□

Code

```
SELECT
    ST_SameAlignment(ST_MakeEmptyRaster(1, 1, 0, 0, 1, 1, 0, 0),
        ST_MakeEmptyRaster(1, 1, 0, 0, 1.1, 1.1, 0, 0)
    ),
    ST_NotSameAlignmentReason(ST_MakeEmptyRaster(1, 1, 0, 0, 1, 1, 0, 0),
        ST_MakeEmptyRaster(1, 1, 0, 0, 1.1, 1.1, 0, 0)
    )
;
```

Output

```
st_samealignment |            st_notsamealignmentreason
------------------+-------------------------------------------------
 f                | The rasters have different scales on the X axis
(1 row)
```

□□

10.1□, ST_SameAlignment

11.17.11 ST_Within

ST_Within — □□□ rastB □□□□□□□□ rastA □□□□□□□□□□, □□□ rastA □□□□□□□□□ rastB □□□□□□□□□□□□□□□□.

□□

boolean **ST_Within**(raster rastA , integer nbandA , raster rastB , integer nbandB);
boolean **ST_Within**(raster rastA , raster rastB);

Description

□□□ rastB □□□□□□□□ rastA □□□□□□□□□□, □□□ rastA □□□□□□□□□ rastB □□□□□□□□□□ rastA □ rastB □□□□□□□□□□. □□□□□□□□□□□□□□□ NULL □□□□□□, □□□□□□□□□□□□□□□□□□□□. □□□□□□□□□□, □□□□□□□□□ (NODATA □□□) □□□□□□□□□.

⊠⊠
⊠⊠⊠⊠⊠ (operand) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠
⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ST_Within(ST_Polygon(raster), geometry) ⊠⊠ ST_Within(geometry, ST_Polygon(raster)) ⊠⊠⊠⊠⊠⊠ ST_Polygon ⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠
ST_Within() ⊠ ST_Contains() ⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠, ST_Within(rastA, rastB) ⊠⊠⊠⊠ ST_Contains(rastB, rastA) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

2.1.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

Code

```
SELECT r1.rid, r2.rid, ST_Within(r1.rast, 1, r2.rast, 1) FROM dummy_rast r1 CROSS JOIN ↩
    dummy_rast r2 WHERE r1.rid = 2;
```

Output

```
rid | rid | st_within
-----+-----+-----------
   2 |   1 | f
   2 |   2 | t
```

⊠⊠

ST_Intersects, ST_Contains, ST_DWithin, ST_DFullyWithin

11.17.12 ST_DWithin

ST_DWithin — ⊠⊠⊠ rastA ⊠⊠⊠⊠ rastB ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

boolean **ST_DWithin**(raster rastA , integer nbandA , raster rastB , integer nbandB , double precision distance_of_srid);
boolean **ST_DWithin**(raster rastA , raster rastB , double precision distance_of_srid);

Description

⊠⊠⊠ rastA ⊠⊠⊠⊠ rastB ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ NULL ⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠ (NODATA ⊠⊠⊠) ⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠ SRID ⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

> **Note**
> ⊠⊠
> ⊠⊠⊠⊠⊠ (operand) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

> **Note**
> ⊠⊠
> ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ST_DWithin(ST_Polygon(raster), geometry) ⊠⊠⊠⊠⊠⊠ ST_Polygon ⊠⊠⊠⊠⊠⊠⊠⊠⊠.

2.1.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

Code

```
SELECT r1.rid, r2.rid, ST_DWithin(r1.rast, 1, r2.rast, 1, 3.14) FROM dummy_rast r1 CROSS ↩
    JOIN dummy_rast r2 WHERE r1.rid = 2;
```

Output

```
rid | rid | st_dwithin
-----+-----+------------
   2 |   1 | f
   2 |   2 | t
```

⊠⊠

ST_Within, ST_DFullyWithin

11.17.13 ST_DFullyWithin

ST_DFullyWithin — ⊠⊠⊠ rastA ⊠⊠⊠⊠ rastB ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

⊠⊠

boolean **ST_DFullyWithin**(raster rastA , integer nbandA , raster rastB , integer nbandB , double precision distance_of_srid);
boolean **ST_DFullyWithin**(raster rastA , raster rastB , double precision distance_of_srid);

Description

□□□ rastA □□□□ rastB □□□□□□□□□□□□□□□□□□□□□□□□□□□. □□□□□□□□□□□□ □□ NULL □□□□□□□, □□□□□□□□□□□□□□□□□□□□. □□□□□□□□□□, □□□□□□□□□ (NODATA □□□) □□□□□□□□□□.

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□. □□□□□□□□□□□□, □□□□□□□□□□□□ □□ SRID □□□□, □□□□□□□□□□□□□□□□.

Note! □□
□□□□□ (operand) □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□.

Note! □□
□□□□□□□□□□□□□□□□□□□□□, ST_DFullyWithin(ST_Polygon(raster), geometry) □□□□ □□ ST_Polygon □□□□□□□□□□.

2.1.0 □□□□□□□□□□□□□.

□□

Code

```
SELECT r1.rid, r2.rid, ST_DFullyWithin(r1.rast, 1, r2.rast, 1, 3.14) FROM dummy_rast r1 ↩
    CROSS JOIN dummy_rast r2 WHERE r1.rid = 2;
```

Output

```
rid | rid | st_dfullywithin
-----+-----+-----------------
   2 |   1 | f
   2 |   2 | t
```

□□

ST_Within, ST_DWithin

11.18 Raster Tips

11.18.1 Out-DB Rasters

11.18.1.1 Directory containing many files

When GDAL opens a file, GDAL eagerly scans the directory of that file to build a catalog of other files. If this directory contains many files (e.g. thousands, millions), opening that file becomes extremely slow (especially if that file happens to be on a network drive such as NFS).

To control this behavior, GDAL provides the following environment variable: GDAL_DISABLE_READDIR_ON
Set `GDAL_DISABLE_READDIR_ON_OPEN` to `TRUE` to disable directory scanning.

In Ubuntu (and assuming you are using PostgreSQL's packages for Ubuntu), `GDAL_DISABLE_READDIR_ON_OPE` can be set in */etc/postgresql/POSTGRESQL_VERSION/CLUSTER_NAME/environment* (where POSTGRESQL_VERSION is the version of PostgreSQL, e.g. 9.6 and CLUSTER_NAME is the name of the cluster, e.g. maindb). You can also set PostGIS environment variables here as well.

Code

```
# environment variables for postmaster process
# This file has the same syntax as postgresql.conf:
#  VARIABLE = simple_value
#  VARIABLE2 = 'any value!'
# I. e. you need to enclose any value which does not only consist of letters,
# numbers, and '-', '_', '.' in single quotes. Shell commands are not
# evaluated.
POSTGIS_GDAL_ENABLED_DRIVERS = 'ENABLE_ALL'

POSTGIS_ENABLE_OUTDB_RASTERS = 1

GDAL_DISABLE_READDIR_ON_OPEN = 'TRUE'
```

11.18.1.2 Maximum Number of Open Files

The maximum number of open files permitted by Linux and PostgreSQL are typically conservative (typically 1024 open files per process) given the assumption that the system is consumed by human users. For Out-DB Rasters, a single valid query can easily exceed this limit (e.g. a dataset of 10 year's worth of rasters with one raster for each day containing minimum and maximum temperatures and we want to know the absolute min and max value for a pixel in that dataset).

The easiest change to make is the following PostgreSQL setting: max_files_per_process. The default is set to 1000, which is far too low for Out-DB Rasters. A safe starting value could be 65536 but this really depends on your datasets and the queries run against those datasets. This setting can only be made on server start and probably only in the PostgreSQL configuration file (e.g. */etc/postgresql/-POSTGRESQL_VERSION/CLUSTER_NAME/postgresql.conf* in Ubuntu environments).

Code

```
...
# - Kernel Resource Usage -

max_files_per_process = 65536           # min 25
                                        # (change requires restart)
...
```

The major change to make is the Linux kernel's open files limits. There are two parts to this:

- Maximum number of open files for the entire system
- Maximum number of open files per process

11.18.1.2.1 Maximum number of open files for the entire system

You can inspect the current maximum number of open files for the entire system with the following example:

Code

```
$ sysctl -a | grep fs.file-max
fs.file-max = 131072
```

If the value returned is not large enough, add a file to */etc/sysctl.d/* as per the following example:

Code

```
$ echo "fs.file-max = 6145324" >> /etc/sysctl.d/fs.conf

$ cat /etc/sysctl.d/fs.conf
fs.file-max = 6145324

$ sysctl -p --system
* Applying /etc/sysctl.d/fs.conf ...
fs.file-max = 2097152
* Applying /etc/sysctl.conf ...

$ sysctl -a | grep fs.file-max
fs.file-max = 6145324
```

11.18.1.2.2 Maximum number of open files per process

We need to increase the maximum number of open files per process for the PostgreSQL server processes.

To see what the current PostgreSQL service processes are using for maximum number of open files, do as per the following example (make sure to have PostgreSQL running):

Code

```
$ ps aux | grep postgres
```

Output

```
postgres 31713  0.0  0.4 179012 17564 pts/0    S    Dec26   0:03 /home/dustymugs/devel/postgresql/sandbox/10/usr/local ↩
     /bin/postgres -D /home/dustymugs/devel/postgresql/sandbox/10/pgdata
postgres 31716  0.0  0.8 179776 33632 ?        Ss   Dec26   0:01 postgres: checkpointer process
postgres 31717  0.0  0.2 179144  9416 ?        Ss   Dec26   0:05 postgres: writer process
postgres 31718  0.0  0.2 179012  8708 ?        Ss   Dec26   0:06 postgres: wal writer process
postgres 31719  0.0  0.1 179568  7252 ?        Ss   Dec26   0:03 postgres: autovacuum launcher process
postgres 31720  0.0  0.1  34228  4124 ?        Ss   Dec26   0:09 postgres: stats collector process
postgres 31721  0.0  0.1 179308  6052 ?        Ss   Dec26   0:00 postgres: bgworker: logical replication launcher
```

Code

```
$ cat /proc/31718/limits
```

Output

```
Limit                     Soft Limit           Hard Limit           Units
Max cpu time              unlimited            unlimited            seconds
Max file size             unlimited            unlimited            bytes
Max data size             unlimited            unlimited            bytes
Max stack size            8388608              unlimited            bytes
Max core file size        0                    unlimited            bytes
Max resident set          unlimited            unlimited            bytes
Max processes             15738                15738                processes
Max open files            1024                 4096                 files
Max locked memory         65536                65536                bytes
Max address space         unlimited            unlimited            bytes
Max file locks            unlimited            unlimited            locks
Max pending signals       15738                15738                signals
Max msgqueue size         819200               819200               bytes
Max nice priority         0                    0
Max realtime priority     0                    0
Max realtime timeout      unlimited            unlimited            us
```

In the example above, we inspected the open files limit for Process 31718. It doesn't matter which PostgreSQL process, any of them will do. The response we are interested in is *Max open files*.

We want to increase *Soft Limit* and *Hard Limit* of *Max open files* to be greater than the value we specified for the PostgreSQL setting `max_files_per_process`. In our example, we set `max_files_per_process` to 65536.

In Ubuntu (and assuming you are using PostgreSQL's packages for Ubuntu), the easiest way to change the *Soft Limit* and *Hard Limit* is to edit **/etc/init.d/postgresql** (SysV) or **/lib/systemd/system/postgresql*.service** (systemd).

Let's first address the SysV Ubuntu case where we add **ulimit -H -n 262144** and **ulimit -n 131072** to **/etc/init.d/postgresql**.

Code

```
...
case "$1" in
    start|stop|restart|reload)
        if [ "$1" = "start" ]; then
            create_socket_directory
        fi
    if [ -z "`pg_lsclusters -h`" ]; then
        log_warning_msg 'No PostgreSQL clusters exist; see "man pg_createcluster"'
        exit 0
    fi

    ulimit -H -n 262144
    ulimit -n 131072

    for v in $versions; do
        $1 $v || EXIT=$?
    done
    exit ${EXIT:-0}
        ;;
    status)
...
```

Now to address the systemd Ubuntu case. We will add **LimitNOFILE=131072** to every **/lib/systemd/system/postgresql*.service** file in the **[Service]** section.

Code

```
...
[Service]

LimitNOFILE=131072

...

[Install]
WantedBy=multi-user.target
...
```

After making the necessary systemd changes, make sure to reload the daemon

Code

```
systemctl daemon-reload
```

Chapter 12

PostGIS Extras

This chapter documents features found in the extras folder of the PostGIS source tarballs and source repository. These are not always packaged with PostGIS binary releases, but are usually PL/pgSQL based or standard shell scripts that can be run as is.

Chapter 13

PostGIS Special Functions Index

13.1 PostGIS Aggregate Functions

The functions below are spatial aggregate functions that are used in the same way as SQL aggregate function such as `sum` and `average`.

- ST_3DUnion - Perform 3D union using postgis_sfcgal.
- ST_3DExtent - Aggregate function that returns the 3D bounding box of geometries.
- ST_AsFlatGeobuf - Return a FlatGeobuf representation of a set of rows.
- ST_AsGeobuf - Return a Geobuf representation of a set of rows.
- ST_AsMVT - Aggregate function returning a MVT representation of a set of rows.
- ST_ClusterDBSCAN - Window function that returns a cluster id for each input geometry using the DBSCAN algorithm.
- ST_ClusterIntersecting - Aggregate function that clusters input geometries into connected sets.
- ST_ClusterIntersectingWin - Window function that returns a cluster id for each input geometry, clustering input geometries into connected sets.
- ST_ClusterKMeans - Window function that returns a cluster id for each input geometry using the K-means algorithm.
- ST_ClusterRelateWin - Window function that returns a cluster id for each input geometry, clustering input geometries into connected sets using the relate pattern to determine whether the geometries are connected.
- ST_ClusterWithin - Aggregate function that clusters geometries by separation distance.
- ST_ClusterWithinWin - Window function that returns a cluster id for each input geometry, clustering using separation distance.
- ST_Collect - Creates a GeometryCollection or Multi* geometry from a set of geometries.
- ST_CoverageClean - Computes a clean (edge matched, non-overlapping, gap-cleared) polygonal coverage, given a non-clean input.
- ST_CoverageInvalidEdges - Window function that finds locations where polygons fail to form a valid coverage.
- ST_CoverageSimplify - Window function that simplifies the edges of a polygonal coverage.

- ST_CoverageUnion - Computes the union of a set of polygons forming a coverage by removing shared edges.
- ST_Extent - Aggregate function that returns the bounding box of geometries.
- ST_MakeLine - ⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_MemUnion - Aggregate function which unions geometries in a memory-efficent but slower way
- ST_MinimumSpanningTree - Window function that returns a tree id for each input geometry, clustering input geometries into connected trees using the Minimum Spanning Tree algorithm.
- ST_Polygonize - Computes a collection of polygons formed from the linework of a set of geometries.
- ST_SameAlignment - ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠) ⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_Union - Computes a geometry representing the point-set union of the input geometries.
- ST_Union - ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ 1 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- TopoElementArray_Agg - Returns a topoelementarray for a set of element_id, type arrays (topoelements).

13.2 PostGIS Window Functions

The functions below are spatial window functions that are used in the same way as SQL window functions such as `row_number()`, `lead()`, and `lag()`. They must be followed by an `OVER()` clause.

- ST_ClusterDBSCAN - Window function that returns a cluster id for each input geometry using the DBSCAN algorithm.
- ST_ClusterIntersectingWin - Window function that returns a cluster id for each input geometry, clustering input geometries into connected sets.
- ST_ClusterKMeans - Window function that returns a cluster id for each input geometry using the K-means algorithm.
- ST_ClusterRelateWin - Window function that returns a cluster id for each input geometry, clustering input geometries into connected sets using the relate pattern to determine whether the geometries are connected.
- ST_ClusterWithinWin - Window function that returns a cluster id for each input geometry, clustering using separation distance.
- ST_CoverageClean - Computes a clean (edge matched, non-overlapping, gap-cleared) polygonal coverage, given a non-clean input.
- ST_CoverageInvalidEdges - Window function that finds locations where polygons fail to form a valid coverage.
- ST_CoverageSimplify - Window function that simplifies the edges of a polygonal coverage.
- ST_MinimumSpanningTree - Window function that returns a tree id for each input geometry, clustering input geometries into connected trees using the Minimum Spanning Tree algorithm.

13.3 PostGIS SQL-MM Compliant Functions

The functions given below are PostGIS functions that conform to the SQL/MM 3 standard

- CG_3DArea - 3 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠ 0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- CG_3DDifference - 3 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- CG_3DIntersection - 3 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- CG_3DUnion - Perform 3D union using postgis_sfcgal.
- CG_Volume - Computes the volume of a 3D geometry. Closed surfaces can have non-zero volume.
- ST_3DDWithin - Tests if two 3D geometries are within a given 3D distance
- ST_3DDistance - ⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠ (SRS ⊠⊠⊠⊠) 3 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠.
- ST_3DIntersects - Tests if two geometries spatially intersect in 3D - only for points, linestrings, polygons, polyhedral surface (area)
- ST_3DLength - ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_3DPerimeter - ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_AddEdgeModFace - ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_AddEdgeNewFaces - ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠ 2 ⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_AddIsoEdge - ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ anode ⊠ anothernode ⊠⊠⊠⊠⊠ alinestring ⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ID ⊠⊠⊠⊠⊠⊠⊠.
- ST_AddIsoNode - ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (isolated) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ID ⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠ NULL ⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_Area - ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_AsBinary - Return the OGC/ISO Well-Known Binary (WKB) representation of the geometry/geography without SRID meta data.
- ST_AsGML - ⊠⊠⊠ GML 2 ⊠⊠ GML 3 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_AsText - ⊠⊠/⊠⊠⊠⊠⊠ WKT(Well-Known Text) ⊠⊠⊠⊠ SRID ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_Boundary - ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_Buffer - Computes a geometry covering all points within a given distance from a geometry.
- ST_Centroid - ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_ChangeEdgeGeom - ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_Contains - Tests if every point of B lies in A, and their interiors have a point in common
- ST_ConvexHull - Computes the convex hull of a geometry.
- ST_CoordDim - ST_Geometry ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_CreateTopoGeo - ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠.
- ST_Crosses - Tests if two geometries have some, but not all, interior points in common

- ST_CurveN - Returns the Nth component curve geometry of a CompoundCurve.
- ST_CurveToLine - Converts a geometry containing curves to a linear geometry.
- ST_Difference - Computes a geometry representing the part of geometry A that does not intersect geometry B.
- ST_Dimension - ST_Geometry ⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧.
- ST_Disjoint - Tests if two geometries have no points in common
- ST_Distance - ⌧⌧⌧⌧⌧⌧ 3 ⌧⌧⌧⌧ (longest) ⌧⌧⌧⌧⌧⌧⌧⌧.
- ST_EndPoint - Returns the last point of a LineString, CircularLineString, or NURBSCurve.
- ST_Envelope - ⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧ (double precision; float8) ⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧.
- ST_Equals - Tests if two geometries include the same set of points
- ST_ExteriorRing - ⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧.
- ST_GMLToSQL - Returns a geometry from GML. Alias for ST_GeomFromGML.
- ST_GeomCollFromText - Makes a collection Geometry from collection WKT with the given SRID. If SRID is not given, it defaults to 0.
- ST_GeomFromText - WKT ⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧ ST_Geometry ⌧⌧⌧⌧⌧⌧⌧.
- ST_GeomFromWKB - WKB(Well-Known Binary) ⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧ SRID ⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧.
- ST_GeometryFromText - Returns a geometry from Well-Known Text (WKT). Alias for ST_GeomFromText.
- ST_GeometryN - ST_Geometry ⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧.
- ST_GeometryType - ST_Geometry ⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧.
- ST_GetFaceEdges - aface ⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧.
- ST_GetFaceGeometry - ⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧ ID ⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧.
- ST_InitTopoGeo - Creates a new topology schema and registers it in the topology.topology table.
- ST_InteriorRingN - ⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧.
- ST_Intersection - Computes a geometry representing the shared portion of geometries A and B.
- ST_Intersects - Tests if two geometries intersect (they have at least one point in common)
- ST_IsClosed - LINESTRING ⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧ TRUE ⌧⌧⌧⌧⌧⌧. ⌧⌧⌧⌧⌧⌧⌧⌧ (⌧⌧⌧⌧⌧⌧) ⌧⌧⌧⌧ TRUE ⌧⌧⌧⌧⌧⌧.
- ST_IsEmpty - Tests if a geometry is empty.
- ST_IsRing - Tests if a LineString is closed and simple.
- ST_IsSimple - ⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧ TRUE ⌧⌧⌧⌧⌧⌧.
- ST_IsValid - Tests if a geometry is well-formed in 2D.
- ST_Length - ⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧.
- ST_LineFromText - ⌧⌧⌧ SRID ⌧⌧⌧ WKT ⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧. SRID ⌧⌧⌧⌧⌧⌧⌧⌧⌧, ⌧⌧⌧⌧ 0 ⌧⌧⌧⌧.
- ST_LineFromWKB - Creates a LINESTRING from a WKB representation, optionally using a given SRID.

- ST_LinestringFromWKB - Creates a LINESTRING from a WKB representation, optionally using a given SRID.
- ST_LocateAlong - Returns the point(s) on a geometry that match a measure value.
- ST_LocateBetween - Returns the portions of a geometry that match a measure range.
- ST_M - Returns the M coordinate of a Point.
- ST_MLineFromText - WKT ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ST_MultiLineString ⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_MPointFromText - Makes a Geometry from WKT with the given SRID. If SRID is not given, it defaults to 0.
- ST_MPolyFromText - Makes a MultiPolygon Geometry from WKT with the given SRID. If SRID is not given, it defaults to 0.
- ST_ModEdgeHeal - Heals two edges by deleting the node connecting them, modifying the first edge and deleting the second edge. Returns the id of the deleted node.
- ST_ModEdgeSplit - ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_MoveIsoNode - Moves an isolated node in a topology from one point to another. If new apoint geometry exists as a node an error is thrown. Returns description of move.
- ST_NewEdgeHeal - Heals two edges by deleting the node connecting them, deleting both edges, and replacing them with an edge whose direction is the same as the first edge provided.
- ST_NewEdgesSplit - ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ 2 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ID ⊠⊠⊠⊠⊠⊠⊠.
- ST_NumCurves - Return the number of component curves in a CompoundCurve.
- ST_NumGeometries - ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_NumInteriorRings - ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_NumPatches - Returns the number of faces in a PolyhedralSurface or TIN.
- ST_NumPoints - ST_LineString ⊠⊠ ST_CircularString ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_OrderingEquals - Tests if two geometries represent the same geometry and have points in the same directional order
- ST_Overlaps - Tests if two geometries have the same dimension and intersect, but each has at least one point not in the other
- ST_PatchN - Returns the Nth face of a PolyhedralSurface or TIN.
- ST_Perimeter - Returns the length of the boundary of a polygonal geometry or geography.
- ST_Point - Creates a Point with X, Y and SRID values.
- ST_PointFromText - ⊠⊠⊠ SRID ⊠⊠⊠ WKT ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. SRID ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠ 0 ⊠⊠⊠⊠.
- ST_PointFromWKB - ⊠⊠⊠ SRID ⊠⊠⊠ WKB ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_PointN - ST_LineString ⊠⊠ ST_CircularString ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_PointOnSurface - Computes a point guaranteed to lie in a polygon, or on a geometry.
- ST_Polygon - Creates a Polygon from a LineString with a specified SRID.
- ST_PolygonFromText - Makes a Geometry from WKT with the given SRID. If SRID is not given, it defaults to 0.

- ST_Relate - Tests if two geometries have a topological relationship matching an Intersection Matrix pattern, or computes their Intersection Matrix
- ST_RemEdgeModFace - Removes an edge, and if the edge separates two faces deletes one face and modifies the other face to cover the space of both.
- ST_RemEdgeNewFace - □□□□□□□□, □□□□□□□□□□□□□□□□□□□□, □□□□□□□□□ □□□□□□□□□□□□□□.
- ST_RemoveIsoEdge - Removes an isolated edge and returns description of action. If the edge is not isolated, then an exception is thrown.
- ST_RemoveIsoNode - □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□. □□□□□□□□□ (□□□□□□□□□□□□□□□) □□, □□□□□□□□□.
- ST_SRID - Returns the spatial reference identifier for a geometry.
- ST_StartPoint - Returns the first point of a LineString, CircularLineString, or NURBSCurve.
- ST_SymDifference - Computes a geometry representing the portions of geometries A and B that do not intersect.
- ST_Touches - Tests if two geometries have at least one point in common, but their interiors do not intersect
- ST_Transform - Return a new geometry with coordinates transformed to a different spatial reference system.
- ST_Union - Computes a geometry representing the point-set union of the input geometries.
- ST_WKBToSQL - Returns a geometry from Well-Known Binary (WKB). SQL/MM alias for ST_GeomFromWK without an SRID argument.
- ST_WKTToSQL - Returns a geometry from Well-Known Text (WKT). SQL/MM alias for ST_GeomFromText.
- ST_Within - Tests if every point of A lies in B, and their interiors have a point in common
- ST_X - Returns the X coordinate of a Point.
- ST_Y - Returns the Y coordinate of a Point.
- ST_Z - Returns the Z coordinate of a Point.

13.4 PostGIS Geography Support Functions

The functions and operators given below are PostGIS functions/operators that take as input or return as output a geography data type object.



□□
Functions with a (T) are not native geodetic functions, and use a ST_Transform call to and from geometry to do the operation. As a result, they may not behave as expected when going over dateline, poles, and for large geometries or geometry pairs that cover more than one UTM zone. Basic transform - (favoring UTM, Lambert Azimuthal (North/South), and falling back on mercator in worst case scenario)

- ST_Area - □□□□□□□□□□□□□□□□□□□□.
- ST_AsBinary - Return the OGC/ISO Well-Known Binary (WKB) representation of the geometry/geography without SRID meta data.

- ST_AsEWKT - □□□ WKT(Well-Known Text) □□□□ SRID □□□□□□□□□□□□□□.
- ST_AsGML - □□□ GML 2 □□ GML 3 □□□□□□□□□□.
- ST_AsGeoJSON - Return a geometry or feature in GeoJSON format.
- ST_AsKML - □□□ GML 2 □□ GML 3 □□□□□□□□□□.
- ST_AsSVG - Returns SVG path data for a geometry.
- ST_AsText - □□/□□□□ WKT(Well-Known Text) □□□□ SRID □□□□□□□□□□□□.
- ST_Azimuth - □□□□□□ 2 □□□□□□□□□□□□.
- ST_Buffer - Computes a geometry covering all points within a given distance from a geometry.
- ST_Centroid - □□□□□□□□□□□□□□□□.
- ST_ClosestPoint - Returns the 2D point on g1 that is closest to g2. This is the first point of the shortest line from one geometry to the other.
- ST_CoveredBy - Tests if every point of A lies in B
- ST_Covers - Tests if every point of B lies in A
- ST_DWithin - Tests if two geometries are within a given distance
- ST_Distance - □□□□□□□ 3 □□□□ (longest) □□□□□□□□.
- ST_GeogFromText - WKT (□□) □□□□□□□□□□□□□□□□□□□□□□□.
- ST_GeogFromWKB - WKB □□□□□□□□ EWKB(□□ WKB) □□□□□□□□□□□□□□□□□□.
- ST_GeographyFromText - WKT (□□) □□□□□□□□□□□□□□□□□□□□□□□.
- = - Returns TRUE if the coordinates and coordinate order geometry/geography A are the same as the coordinates and coordinate order of geometry/geography B.
- ST_Intersection - Computes a geometry representing the shared portion of geometries A and B.
- ST_Intersects - Tests if two geometries intersect (they have at least one point in common)
- ST_Length - □□□□□□□□□□□□□□□□□.
- ST_LineInterpolatePoint - Returns a point interpolated along a line at a fractional location.
- ST_LineInterpolatePoints - Returns points interpolated along a line at a fractional interval.
- ST_LineLocatePoint - Returns the fractional location of the closest point on a line to a point.
- ST_LineSubstring - Returns the part of a line between two fractional locations.
- ST_Perimeter - Returns the length of the boundary of a polygonal geometry or geography.
- ST_Project - Returns a point projected from a start point by a distance and bearing (azimuth).
- ST_Segmentize - Returns a modified geometry/geography having no segment longer than a given distance.
- ST_ShortestLine - □□□□□□□ 2 □□□□□□□□□□□□□.
- ST_Summary - □□□□□□□□□□□□□□□□□□□□.
- <-> - A □ B □□□ 2 □□□□□□□□□□□.
- && - A □ 2D □□□□□□ B □ 2D □□□□□□□□□□□□ TRUE □□□□□□□.

13.5 PostGIS Raster Support Functions

The functions and operators given below are PostGIS functions/operators that take as input or return as output a raster data type object. Listed in alphabetical order.

- Box3D - ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ BOX3D ⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- @ - B ⊠⊠⊠⊠⊠⊠ A ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ TRUE ⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ~ - A ⊠⊠⊠⊠⊠⊠ B ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ TRUE ⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- = - A ⊠⊠⊠⊠⊠⊠ B ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ TRUE ⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- && - A ⊠⊠⊠⊠⊠⊠ B ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ TRUE ⊠⊠⊠⊠⊠⊠.
- &< - A ⊠⊠⊠⊠⊠⊠ B ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ TRUE ⊠⊠⊠⊠⊠⊠.
- &> - A ⊠⊠⊠⊠⊠⊠ B ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ TRUE ⊠⊠⊠⊠⊠⊠.
- ~= - A ⊠⊠⊠⊠⊠⊠ B ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ TRUE ⊠⊠⊠⊠⊠⊠.
- ST_Retile - ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_AddBand - ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (⊠) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_AsBinary/ST_AsWKB - Return the Well-Known Binary (WKB) representation of the raster.
- ST_AsGDALRaster - Return the raster tile in the designated GDAL Raster format. Raster formats are one of those supported by your compiled library. Use ST_GDALDrivers() to get a list of formats supported by your library.
- ST_AsHexWKB - Return the Well-Known Binary (WKB) in Hex representation of the raster.
- ST_AsJPEG - ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ JPEG(Joint Photographic Exports Group) ⊠⊠⊠ (⊠⊠⊠⊠⊠) ⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠ 1 ⊠⊠⊠⊠⊠ 3 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠ 3 ⊠⊠⊠⊠⊠⊠⊠ 3 ⊠⊠⊠⊠⊠⊠ RGB ⊠⊠⊠⊠⊠⊠⊠.
- ST_AsPNG - ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ PNG(Portable Network Graphics) ⊠⊠⊠ (⊠⊠⊠⊠⊠) ⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠ 1 ⊠, 3 ⊠, ⊠⊠ 4 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠ 2 ⊠⊠⊠ 4 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠ 1 ⊠⊠⊠⊠. ⊠⊠⊠ RGB ⊠⊠ RGBA ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_AsRaster - PostGIS ⊠⊠⊠ PostGIS ⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_AsRasterAgg - Aggregate. Renders PostGIS geometries into a new raster.
- ST_AsTIFF - Return the raster selected bands as a single TIFF image (byte array). If no band is specified or any of specified bands does not exist in the raster, then will try to use all bands.
- ST_Aspect - ⊠⊠⊠⊠⊠⊠⊠⊠⊠ (⊠⊠⊠⊠⊠⊠) ⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_Band - ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_BandFileSize - Returns the file size of a band stored in file system. If no bandnum specified, 1 is assumed.
- ST_BandFileTimestamp - Returns the file timestamp of a band stored in file system. If no bandnum specified, 1 is assumed.
- ST_BandIsNoData - ⊠⊠⊠ NODATA ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_BandMetaData - ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ 1 ⊠⊠⊠⊠⊠⊠⊠⊠⊠.

- ST_BandNoDataValue - ⊠⊠⊠⊠⊠⊠⊠ NODATA ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ 1 ⊠⊠⊠⊠⊠⊠.
- ST_BandPath - ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. bandnum ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ 1 ⊠⊠⊠⊠⊠⊠.
- ST_BandPixelType - ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. bandnum ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ 1 ⊠⊠⊠⊠⊠⊠.
- ST_Clip - Returns the raster clipped by the input geometry. If band number is not specified, all bands are processed. If crop is not specified or TRUE, the output raster is cropped. If touched is set to TRUE, then touched pixels are included, otherwise only if the center of the pixel is in the geometry it is included.
- ST_ColorMap - ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ 8BUI ⊠⊠ (grayscale, RGB, RGBA) ⊠ 4 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ 1 ⊠⊠⊠⊠⊠⊠.
- ST_Contains - ⊠⊠⊠ rastA ⊠⊠⊠⊠⊠⊠⊠⊠ rastB ⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠ rastB ⊠⊠⊠⊠⊠⊠⊠⊠⊠ rastA ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_ContainsProperly - rastB ⊠ rastA ⊠⊠⊠⊠⊠⊠⊠⊠⊠ rastA ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_Contour - Generates a set of vector contours from the provided raster band, using the GDAL contouring algorithm.
- ST_ConvexHull - BandNoDataValue ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ST_Envelope ⊠⊠.
- ST_Count - ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ 1 ⊠⊠⊠. exclude_nodata_value ⊠⊠⊠⊠⊠⊠⊠⊠⊠, NODATA ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_CountAgg - ⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ 1 ⊠⊠⊠. exclude_nodata_value ⊠⊠⊠⊠⊠⊠⊠⊠⊠, NODATA ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_CoveredBy - ⊠⊠⊠ rastA ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ rastB ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_Covers - ⊠⊠⊠ rastB ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ rastA ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_DFullyWithin - ⊠⊠⊠ rastA ⊠⊠⊠⊠ rastB ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_DWithin - ⊠⊠⊠ rastA ⊠⊠⊠⊠ rastB ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_Disjoint - ⊠⊠⊠ rastA ⊠⊠⊠⊠ rastB ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_DumpAsPolygons - ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ geomval(geom, val) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ 1 ⊠⊠⊠⊠⊠⊠.
- ST_DumpValues - ⊠⊠⊠⊠⊠⊠⊠⊠⊠ 2 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_Envelope - ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_FromGDALRaster - ⊠⊠ GDAL ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_GeoReference - ⊠⊠ (world) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ GDAL ⊠⊠ ESRI ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠ GDAL ⊠⊠⊠.
- ST_Grayscale - Creates a new one-8BUI band raster from the source raster and specified bands representing Red, Green and Blue
- ST_HasNoBand - ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠ 1 ⊠⊠⊠⊠⊠⊠.

- ST_Height - ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_HillShade - ⊠⊠⊠⊠⊠⊠, ⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠.
- ST_Histogram - ⊠ (bin; ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_InterpolateRaster - Interpolates a gridded surface based on an input set of 3-d points, using the X- and Y-values to position the points on the grid and the Z-value of the points as the surface elevation.
- ST_Intersection - ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠-⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_IntersectionFractions - Calculates the fraction of each raster cell that is covered by a given geometry.
- ST_Intersects - ⊠⊠⊠ rastA ⊠⊠⊠⊠ rastB ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_IsEmpty - ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (width = 0, height = 0) ⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠.
- ST_MakeEmptyCoverage - Cover georeferenced area with a grid of empty raster tiles.
- ST_MakeEmptyRaster - ⊠⊠⊠⊠⊠ (⊠⊠ & ⊠⊠), ⊠⊠⊠ X ⊠ Y, ⊠⊠⊠⊠, ⊠⊠ (scalex, scaley, skewx & skewy) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (SRID) ⊠⊠⊠⊠⊠ (⊠⊠⊠⊠⊠) ⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠ ⊠⊠, ⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠ SRID ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. SRID ⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠ 0(unknown) ⊠⊠⊠⊠⊠⊠⊠.
- ST_MapAlgebra (callback function version) - ⊠⊠⊠⊠⊠⊠ - ⊠⊠⊠ 1 ⊠⊠⊠, ⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠ 1 ⊠⊠⊠⊠⊠⊠⊠⊠ 1 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_MapAlgebraExpr - ⊠⊠⊠⊠⊠ 1 ⊠⊠⊠: ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ PostgreSQL ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠ 1 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠ 1 ⊠⊠ ⊠⊠⊠⊠.
- ST_MapAlgebraExpr - ⊠⊠⊠⊠⊠ 2 ⊠⊠⊠: ⊠⊠⊠⊠⊠ 2 ⊠⊠⊠⊠⊠⊠⊠ PostgreSQL ⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠ 1 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠, ⊠⊠⊠⊠⊠⊠⊠ 1 ⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (⊠⊠, ⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠) ⊠⊠⊠⊠⊠⊠⊠⊠. extenttype ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. extenttype ⊠⊠⊠ INTERSECTION, UNION, FIRST, SECOND ⊠⊠⊠⊠⊠⊠⊠.
- ST_MapAlgebraFct - ⊠⊠⊠⊠⊠ 1 ⊠⊠⊠: ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ PostgreSQL ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠ 1 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠ 1 ⊠⊠ ⊠⊠⊠⊠.
- ST_MapAlgebraFct - ⊠⊠⊠⊠⊠ 2 ⊠⊠⊠: ⊠⊠⊠⊠⊠ 2 ⊠⊠⊠⊠⊠⊠⊠ PostgreSQL ⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠ 1 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠ 1 ⊠⊠ ⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ INTERSECTION ⊠⊠⊠.
- ST_MapAlgebraFctNgb - ⊠⊠⊠⊠⊠ 1 ⊠⊠⊠: ⊠⊠⊠⊠⊠ PostgreSQL ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠ (Map Algebra Nearest Neighbor) ⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (neighborhood) ⊠⊠⊠⊠ PostgreSQL ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_MapAlgebra (expression version) - ⊠⊠⊠⊠⊠ - ⊠⊠⊠⊠⊠ 1 ⊠⊠⊠ 2 ⊠, ⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠ ⊠⊠ SQL ⊠⊠⊠ 1 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ 1 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_MemSize - ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (⊠⊠⊠⊠⊠⊠) ⊠⊠⊠⊠⊠.
- ST_MetaData - ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠ (skew), ⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_MinConvexHull - ⊠⊠⊠⊠ NODATA ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

- ST_NearestValue - columnx ⊠ rowy, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ NODATA ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_Neighborhood - columnx ⊠ rowy, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ NODATA ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ 2 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_NotSameAlignmentReason - ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_NumBands - ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_Overlaps - ⊠⊠⊠ rastA ⊠⊠⊠⊠ rastB ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠.
- ST_PixelAsCentroid - ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (⊠⊠⊠⊠⊠) ⊠⊠⊠⊠⊠⊠.
- ST_PixelAsCentroids - ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (⊠⊠⊠⊠⊠) ⊠⊠⊠⊠⊠⊠⊠ X, Y ⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_PixelAsPoint - ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_PixelAsPoints - ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ X, Y ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_PixelAsPolygon - ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_PixelAsPolygons - ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ X, Y ⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠.
- ST_PixelHeight - ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_PixelOfValue - ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ columnx, rowy ⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_PixelWidth - ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_Polygon - NODATA ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_Quantile - ⊠⊠⊠⊠⊠⊠⊠⊠⊠ (population) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (quantile) ⊠⊠⊠⊠⊠⊠. ⊠⊠⊠, ⊠⊠⊠⊠ 25%, 50%, 75% ⊠⊠⊠ (percentile) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠.
- ST_RastFromHexWKB - Return a raster value from a Hex representation of Well-Known Binary (WKB) raster.
- ST_RastFromWKB - Return a raster value from a Well-Known Binary (WKB) raster.
- ST_RasterToWorldCoord - ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ X, Y(⊠⊠, ⊠⊠) ⊠⊠⊠⊠⊠⊠⊠ ⊠. ⊠⊠⊠⊠⊠ 1 ⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_RasterToWorldCoordX - ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ X ⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠ 1 ⊠⊠⊠⊠⊠⊠⊠.
- ST_RasterToWorldCoordY - ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ Y ⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠ 1 ⊠⊠⊠⊠⊠⊠⊠.
- ST_Reclass - ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. nband ⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠. nband ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ 1 ⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠ ⊠: ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ 16BUI ⊠⊠⊠ 8BUI ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_ReclassExact - Creates a new raster composed of bands reclassified from original, using a 1:1 mapping from values in the original band to new values in the destination band.
- ST_Resample - ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

- ST_Rescale - Resample a raster by adjusting only its scale (or pixel size). New pixel values are computed using the NearestNeighbor (english or american spelling), Bilinear, Cubic, CubicSpline, Lanczos, Max or Min resampling algorithm. Default is NearestNeighbor.
- ST_Resize - ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠/⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_Reskew - ⊠⊠⊠ (⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. NearestNeighbor(⊠⊠⊠⊠⊠⊠⊠⊠), Bilinear, Cubic, CubicSpline ⊠⊠ Lanczos ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠ NearestNeighbor ⊠⊠⊠.
- ST_Rotation - ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_Roughness - DEM ⊠⊠⊠⊠" ⊠⊠⊠ (roughness)" ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_SRID - spatial_ref_sys ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_SameAlignment - ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠) ⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_ScaleX - ⊠⊠⊠⊠⊠ X ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_ScaleY - ⊠⊠⊠⊠⊠ Y ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_SetBandIndex - Update the external band number of an out-db band
- ST_SetBandIsNoData - ⊠⊠⊠ isnodata ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_SetBandNoDataValue - NODATA ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ 1 ⊠⊠⊠⊠⊠⊠. ⊠⊠⊠ NODATA ⊠⊠⊠⊠⊠⊠⊠⊠⊠, nodata value = NULL ⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_SetBandPath - Update the external path and band number of an out-db band
- ST_SetGeoReference - ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ 6 ⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. GDAL ⊠⊠ ESRI ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠ GDAL ⊠⊠⊠.
- ST_SetM - Returns a geometry with the same X/Y coordinates as the input geometry, and values from the raster copied into the M dimension using the requested resample algorithm.
- ST_SetRotation - ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_SetSRID - ⊠⊠⊠⊠ SRID ⊠ spatial_ref_sys ⊠⊠⊠⊠⊠⊠⊠⊠⊠ SRID ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_SetScale - X ⊠ Y ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠/⊠⊠⊠⊠/⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_SetSkew - ⊠⊠⊠⊠ X ⊠ Y ⊠⊠⊠ (skew)(⊠⊠⊠⊠⊠⊠⊠⊠⊠) ⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠, X ⊠ Y ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_SetUpperLeft - Sets the value of the upper left corner of the pixel of the raster to projected X and Y coordinates.
- ST_SetValue - ⊠⊠⊠ columnx, rowy ⊠⊠⊠. ⊠⊠⊠⊠⊠ 1 ⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ 1 ⊠⊠⊠⊠⊠⊠.
- ST_SetValues - ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_SetZ - Returns a geometry with the same X/Y coordinates as the input geometry, and values from the raster copied into the Z dimension using the requested resample algorithm.
- ST_SkewX - ⊠⊠⊠⊠ X ⊠⊠⊠ (skew)(⊠⊠⊠⊠⊠⊠⊠⊠⊠) ⊠⊠⊠⊠⊠⊠.
- ST_SkewY - ⊠⊠⊠⊠ Y ⊠⊠⊠ (⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠) ⊠⊠⊠⊠⊠⊠.
- ST_Slope - ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (⊠⊠⊠⊠⊠⊠) ⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

- ST_SnapToGrid - ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. NearestNeighbor(⊠⊠⊠⊠⊠⊠⊠⊠), Bilinear, Cubic, CubicSpline ⊠⊠ Lanczos ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠ NearestNeighbor ⊠⊠⊠.
- ST_Summary - ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_SummaryStats - Returns summarystats consisting of count, sum, mean, stddev, min, max for a given raster band of a raster or raster coverage. Band 1 is assumed if no band is specified.
- ST_SummaryStatsAgg - Aggregate. Returns summarystats consisting of count, sum, mean, stddev, min, max for a given raster band of a set of raster. Band 1 is assumed if no band is specified.
- ST_TPI - ⊠⊠⊠⊠⊠⊠⊠⊠⊠ (Topographic Position Index) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_TRI - ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (Terrain Ruggedness Index) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_Tile - ⊠⊠.
- ST_Touches - ⊠⊠⊠ rastA ⊠⊠⊠⊠ rastB ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ TRUE ⊠⊠⊠⊠⊠⊠.
- ST_Transform - ⊠⊠. NearestNeighbor, Bilinear, Cubic, CubicSpline, Lanczos ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠ NearestNeighbor ⊠⊠⊠.
- ST_Union - ⊠⊠⊠⊠⊠⊠⊠⊠ 1 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_UpperLeftX - ⊠⊠⊠⊠⊠⊠⊠ X ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_UpperLeftY - ⊠⊠⊠⊠⊠⊠⊠ Y ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_Value - ⊠⊠ columnx, rowy ⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠ 1 ⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ 1 ⊠⊠⊠⊠⊠⊠. exclude_nodata_value ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, nodata ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. exclude_nodata_value ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_ValueCount - ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ 1 ⊠⊠⊠. ⊠⊠⊠⊠⊠ NODATA ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_Width - ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_Within - ⊠⊠⊠ rastB ⊠⊠⊠⊠⊠⊠⊠⊠ rastA ⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠ rastA ⊠⊠⊠⊠⊠⊠⊠⊠⊠ rastB ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_WorldToRasterCoord - ⊠⊠⊠⊠⊠⊠⊠ X, Y(⊠⊠, ⊠⊠) ⊠⊠.
- ST_WorldToRasterCoordX - ⊠⊠⊠⊠⊠ (pt) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ X, Y ⊠⊠⊠⊠ (xw, yw) ⊠⊠⊠⊠⊠⊠.
- ST_WorldToRasterCoordY - ⊠⊠⊠⊠⊠ (pt) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ X, Y ⊠⊠⊠⊠ (xw, yw) ⊠⊠⊠⊠⊠⊠.
- UpdateRasterSRID - ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ SRID ⊠⊠⊠⊠⊠⊠.

13.6 PostGIS Geometry / Geography / Raster Dump Functions

The functions given below are PostGIS functions that take as input or return as output a set of or single geometry_dump or geomval data type object.

- ST_DumpAsPolygons - □□□□□□□□□□ geomval(geom, val) □□□□□□□□□□. □□□□□□ □□□□□□□□□□□□□□□ 1 □□□□□□□.
- ST_Intersection - □□□□□□□□□□□□□□, □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□-□□□□□□□□□□□□□□□.
- ST_Dump - Returns a set of geometry_dump rows for the components of a geometry.
- ST_DumpPoints - □□□□□□□□□□□□□□□□□□□□.
- ST_DumpRings - Returns a set of geometry_dump rows for the exterior and interior rings of a Polygon.
- ST_DumpSegments - □□□□□□□□□□□□□□□□□□□□□□.

13.7 PostGIS Box Functions

The functions given below are PostGIS functions that take as input or return as output the box* family of PostGIS spatial types. The box family of types consists of box2d, and box3d

- Box2D - Returns a BOX2D representing the 2D extent of a geometry.
- Box3D - Returns a BOX3D representing the 3D extent of a geometry.
- MakeTopologyPrecise - Snap topology vertices to precision grid.
- Box3D - □□□□□□□□□□ BOX3D □□□□□□□□□□.
- ST_3DExtent - Aggregate function that returns the 3D bounding box of geometries.
- ST_3DMakeBox - Creates a BOX3D defined by two 3D point geometries.
- ST_AsMVTGeom - Transforms a geometry into the coordinate space of a MVT tile.
- ST_AsTWKB - □□□ TWKB(Tiny Well-Known Binary) □□□□□□.
- ST_Box2dFromGeoHash - GeoHash □□□□□□□ BOX2D □□□□□□.
- ST_ClipByBox2D - Computes the portion of a geometry falling within a rectangle.
- ST_EstimatedExtent - Returns the estimated extent of a spatial table.
- ST_Expand - Returns a bounding box expanded from another bounding box or a geometry.
- ST_Extent - Aggregate function that returns the bounding box of geometries.
- ST_MakeBox2D - Creates a BOX2D defined by two 2D point geometries.
- ST_RemoveIrrelevantPointsForView - Removes points that are irrelevant for rendering a specific rectangular view of a geometry.
- ST_XMax - Returns the X maxima of a 2D or 3D bounding box or a geometry.
- ST_XMin - Returns the X minima of a 2D or 3D bounding box or a geometry.
- ST_XSize - Returns the X size of a 2D or 3D bounding box or a geometry.

- ST_YMax - Returns the Y maxima of a 2D or 3D bounding box or a geometry.
- ST_YMin - Returns the Y minima of a 2D or 3D bounding box or a geometry.
- ST_YSize - Returns the Y size of a 2D or 3D bounding box or a geometry.
- ST_ZMax - Returns the Z maxima of a 2D or 3D bounding box or a geometry.
- ST_ZMin - Returns the Z minima of a 2D or 3D bounding box or a geometry.
- ST_ZSize - Returns the Z size of a 2D or 3D bounding box or a geometry.
- RemoveUnusedPrimitives - Removes topology primitives which not needed to define existing TopoGeometry objects.
- ValidateTopology - Returns a set of validatetopology_returntype objects detailing issues with topology.
- ValidateTopologyPrecision - Returns non-precise vertices in the topology.
- ~(box2df,box2df) - Returns TRUE if a 2D float precision bounding box (BOX2DF) contains another 2D float precision bounding box (BOX2DF).
- ~(box2df,geometry) - Returns TRUE if a 2D float precision bounding box (BOX2DF) contains a geometry's 2D bonding box.
- ~(geometry,box2df) - Returns TRUE if a geometry's 2D bonding box contains a 2D float precision bounding box (GIDX).
- @(box2df,box2df) - Returns TRUE if a 2D float precision bounding box (BOX2DF) is contained into another 2D float precision bounding box.
- @(box2df,geometry) - Returns TRUE if a 2D float precision bounding box (BOX2DF) is contained into a geometry's 2D bounding box.
- @(geometry,box2df) - Returns TRUE if a geometry's 2D bounding box is contained into a 2D float precision bounding box (BOX2DF).
- &&(box2df,box2df) - Returns TRUE if two 2D float precision bounding boxes (BOX2DF) intersect each other.
- &&(box2df,geometry) - Returns TRUE if a 2D float precision bounding box (BOX2DF) intersects a geometry's (cached) 2D bounding box.
- &&(geometry,box2df) - Returns TRUE if a geometry's (cached) 2D bounding box intersects a 2D float precision bounding box (BOX2DF).

13.8 PostGIS Functions that support 3D

The functions given below are PostGIS functions that do not throw away the Z-Index.

- AddGeometryColumn - □□□□□□□□□□□□□□□□□□□□□□□□□.
- Box3D - Returns a BOX3D representing the 3D extent of a geometry.
- CG_3DAlphaWrapping - Computes a 3D Alpha-wrapping strictly enclosing a geometry.
- CG_3DArea - 3 □□□□□□□□□□□□□□□□□. □□□□□ 0 □□□□□□□□□.
- CG_3DConvexHull - □□□□□□□□□□□□□□□□□.
- CG_3DDifference - 3 □□□□□□□□□□.

- CG_3DIntersection - 3 □□□□□□□□□□.
- CG_3DRotate - Rotates a geometry in 3D space around an axis vector.
- CG_3DScale - Scales a geometry by separate factors along X, Y, and Z axes.
- CG_3DScaleAroundCenter - Scales a geometry in 3D space around a specified center point.
- CG_3DTranslate - Translates (moves) a geometry by given offsets in 3D space.
- CG_3DUnion - Perform 3D union using postgis_sfcgal.
- CG_ConstrainedDelaunayTriangles - Return a constrained Delaunay triangulation around the given input geometry.
- CG_Extrude - □□□□□□□□□□□□□□□.
- CG_ForceLHR - LHR(Left Hand Reverse; □□□□) □□□□□□□□□.
- CG_IsPlanar - □□□□□□□□□□□□□□□□□.
- CG_IsSimple - Tests whether a geometry is simple according to SFCGAL
- CG_IsSimpleDetail - Returns a reason when a geometry is not simple according to SFCGAL
- CG_IsSolid - □□□□□□□□□□□□□□. □□□□□□□□□□□□□□□□□□.
- CG_IsValid - Tests whether a geometry is valid according to SFCGAL
- CG_IsValidDetail - Returns a valid_detail row stating whether a geometry is valid according to SFCGAL, and if not a reason and a location.
- CG_MakeSolid - □□□□□□□□□□□□. □□□□□□□□□□□□□□□□□. □□□□□□□□□□□□, □□□□□□□□□□□□□□□□□□ TIN □□□□□□□□.
- CG_NurbsCurveApproximate - Creates an approximating NURBS curve fitting data points within a tolerance
- CG_NurbsCurveDerivative - Computes the derivative (tangent, curvature) of a NURBS curve at a given parameter
- CG_NurbsCurveInterpolate - Creates an interpolating NURBS curve passing through all given data points
- CG_Orientation - □□□□□□ (orientation) □□□□□□□.
- CG_RotateX - Rotates a geometry around the X-axis by a given angle.
- CG_RotateY - Rotates a geometry around the Y-axis by a given angle.
- CG_RotateZ - Rotates a geometry around the Z-axis by a given angle.
- CG_Simplify - Reduces the complexity of a geometry while preserving essential features and Z/M values.
- CG_Tesselate - □□□□□□□□□□□□□□□□□□ (tessellation) □□□□□□□□□ TIN □□ TIN □□□□□□□□□□□□.
- CG_Visibility - Compute a visibility polygon from a point or a segment in a polygon geometry
- CG_Volume - Computes the volume of a 3D geometry. Closed surfaces can have non-zero volume.
- DropGeometryColumn - □□□□□□□□□□□□□□□□□□□□□□□□□.
- GeometryType - ST_Geometry □□□□□□□□□□□□.

- ST_3DClosestPoint - g2 □□□□□□ g1 □□□□ 3 □□□□□□□□□□□□. □□□□□□ 3D □□□□□ □□□□□□□□□□.
- ST_3DDFullyWithin - Tests if two 3D geometries are entirely within a given 3D distance
- ST_3DDWithin - Tests if two 3D geometries are within a given 3D distance
- ST_3DDistance - □□□□□□□, □□□□□□ (SRS □□□□) 3 □□□□□□□□□□□□□□□□□□□□□□□□.
- ST_3DExtent - Aggregate function that returns the 3D bounding box of geometries.
- ST_3DInterpolatePoint - Returns the interpolated measure of a geometry closest to a point in 3D.
- ST_3DIntersects - Tests if two geometries spatially intersect in 3D - only for points, linestrings, polygons, polyhedral surface (area)
- ST_3DLength - □□□□□□□□□□□□□□□□.
- ST_3DLineInterpolatePoint - Returns a point interpolated along a 3D line at a fractional location.
- ST_3DLongestLine - □□□□□□ 3 □□□□ (longest) □□□□□□□□.
- ST_3DMaxDistance - □□□□□□□□, □□□□□□□ (SRS □□□□) 3 □□□□□□□□□□□□□□□□□□□□□□□.
- ST_3DPerimeter - □□□□□□□□□□□□□□□□.
- ST_3DShortestLine - □□□□□□□ 3 □□□□ (shortest) □□□□□□□□.
- ST_AddMeasure - Interpolates measures along a linear geometry.
- ST_AddPoint - □□□□□□□□□□□□□□□□.
- ST_Affine - Apply a 3D affine transformation to a geometry.
- ST_AsBinary - Return the OGC/ISO Well-Known Binary (WKB) representation of the geometry/geography without SRID meta data.
- ST_AsEWKB - Return the Extended Well-Known Binary (EWKB) representation of the geometry with SRID meta data.
- ST_AsEWKT - □□□ WKT(Well-Known Text) □□□□ SRID □□□□□□□□□□□□□□.
- ST_AsGML - □□□ GML 2 □□ GML 3 □□□□□□□□□□.
- ST_AsGeoJSON - Return a geometry or feature in GeoJSON format.
- ST_AsHEXEWKB - □□□□□□□ (NDR) □□□□□□□ (XDR) □□□□□□□ HEXEWKB (□□□) □□□ □□□□□□□.
- ST_AsKML - □□□ GML 2 □□ GML 3 □□□□□□□□□□.
- ST_AsX3D - □□□ X3D XML □□□□□□□: ISO-IEC-19776-1.2-X3DEncodings-XML □□□□□□.
- ST_Boundary - □□□□□□□□□□□□□□□□□□□□□□□□.
- ST_BoundingDiagonal - □□□□□□□□□□□□□□□□□□□□□□.
- ST_CPAWithin - Tests if the closest point of approach of two trajectories is within the specified distance.
- ST_CatmullRomSmoothing - Returns a smoothed version of a geometry, using the Catmull-Rom spline algorithm
- ST_ChaikinSmoothing - Returns a smoothed version of a geometry, using the Chaikin algorithm

- ST_ClosestPointOfApproach - Returns a measure at the closest point of approach of two trajectories.
- ST_Collect - Creates a GeometryCollection or Multi* geometry from a set of geometries.
- ST_ConvexHull - Computes the convex hull of a geometry.
- ST_CoordDim - ST_Geometry □□□□□□□□□□□□□.
- ST_CurveN - Returns the Nth component curve geometry of a CompoundCurve.
- ST_CurveToLine - Converts a geometry containing curves to a linear geometry.
- ST_DelaunayTriangles - Returns the Delaunay triangulation of the vertices of a geometry.
- ST_Difference - Computes a geometry representing the part of geometry A that does not intersect geometry B.
- ST_DistanceCPA - Returns the distance between the closest point of approach of two trajectories.
- ST_Dump - Returns a set of geometry_dump rows for the components of a geometry.
- ST_DumpPoints - □□□□□□□□□□□□□□□□□□□□.
- ST_DumpRings - Returns a set of geometry_dump rows for the exterior and interior rings of a Polygon.
- ST_DumpSegments - □□□□□□□□□□□□□□□□□□□□.
- ST_EndPoint - Returns the last point of a LineString, CircularLineString, or NURBSCurve.
- ST_ExteriorRing - □□□□□□□□□□□□□□□□□□□□.
- ST_FlipCoordinates - Returns a version of a geometry with X and Y axis flipped.
- ST_Force2D - □□□”2 □□□□” □□□□□□.
- ST_ForceCurve - □□□□□□□□, □□□□□□□□□□□□□□□□□□□□ (upcast) □□□.
- ST_ForcePolygonCCW - Orients all exterior rings counter-clockwise and all interior rings clockwise.
- ST_ForcePolygonCW - Orients all exterior rings clockwise and all interior rings counter-clockwise.
- ST_ForceRHR - □□□□□□□□□□□□□□□ (orientation) □□□□□□ (Right-Hand Rule) □□□□□ □□□□□.
- ST_ForceSFS - Force geometries to use SFS 1.1 or 1.2 geometry types.
- ST_Force3D - □□□ XYZ □□□□□□□□. ST_Force3DZ □□□□□□.
- ST_Force3DZ - □□□ XYZ □□□□□□□□.
- ST_Force4D - □□□ XYZM □□□□□□□□.
- ST_ForceCollection - □□□□□□□□□□□□□□□.
- ST_GeomFromEWKB - EWKB(Extended Well-Known Binary) □□□□□□ ST_Geometry □□□□□□□.
- ST_GeomFromEWKT - EWKT(Extended Well-Known Text) □□□□□□ ST_Geometry □□□□□□□.
- ST_GeomFromGML - □□□ GML □□□□□□□□ PostGIS □□□□□□□□□□.
- ST_GeomFromGeoJSON - GeoJSON □□□□□□□□ PostGIS □□□□□□□□□□.
- ST_GeomFromKML - □□□ KML □□□□□□□□ PostGIS □□□□□□□□□□.
- ST_GeometricMedian - □□□□□□□□□□□□□□ (median) □□□□□□.

- ST_GeometryN - ST_Geometry □□□□□□□□□□□□.
- ST_GeometryType - ST_Geometry □□□□□□□□□□□□.
- ST_HasArc - Tests if a geometry contains a circular arc
- ST_HasM - Checks if a geometry has an M (measure) dimension.
- ST_HasZ - Checks if a geometry has a Z dimension.
- ST_InteriorRingN - □□□□□□□□□□□□□□□□□□□□.
- ST_InterpolatePoint - □□□□□□□□□□□□□□□□□□□□□□ (M □□) □□□□□□□□.
- ST_Intersection - Computes a geometry representing the shared portion of geometries A and B.
- ST_IsClosed - LINESTRING □□□□□□□□□□□□□□□ TRUE □□□□□□. □□□□□□□□ (□□□□□□) □□□□ TRUE □□□□□□.
- ST_IsCollection - □□□□□□□□□□□, □□□, □□□□□□□ TRUE □□□□□□.
- ST_IsPolygonCCW - Tests if Polygons have exterior rings oriented counter-clockwise and interior rings oriented clockwise.
- ST_IsPolygonCW - Tests if Polygons have exterior rings oriented clockwise and interior rings oriented counter-clockwise.
- ST_IsSimple - □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ TRUE □□□□□□.
- ST_IsValidTrajectory - Tests if the geometry is a valid trajectory.
- ST_LengthSpheroid - □□□□□□□□□□□□□□□.
- ST_LineFromMultiPoint - □□□□□□□□□□□□□□□□□□□□□□□□.
- ST_LineInterpolatePoint - Returns a point interpolated along a line at a fractional location.
- ST_LineInterpolatePoints - Returns points interpolated along a line at a fractional interval.
- ST_LineSubstring - Returns the part of a line between two fractional locations.
- ST_LineToCurve - Converts a linear geometry to a curved geometry.
- ST_LocateBetweenElevations - Returns the portions of a geometry that lie in an elevation (Z) range.
- ST_M - Returns the M coordinate of a Point.
- ST_MMax - Returns the M maxima of a geometry.
- ST_MMin - Returns the M minima of a geometry.
- ST_MSize - Returns the M size of a geometry.
- ST_MakeLine - □□□□, □□□□□□□□□□□□□□□□□□□□□□□□□□□□.
- ST_MakePoint - Creates a 2D, 3DZ or 4D Point.
- ST_MakePolygon - Creates a Polygon or CurvePolygon from a shell and optional list of holes.
- ST_MakeValid - Attempts to make an invalid geometry valid without losing vertices.
- ST_MemSize - ST_Geometry □□□□□□□□□□□□.
- ST_MemUnion - Aggregate function which unions geometries in a memory-efficent but slower way
- ST_NDims - ST_Geometry □□□□□□□□□□□□.

- ST_NPoints - □□□□□□□□□□□□ (□□□) □□□□□□□□□□.
- ST_NRings - □□□□□□□□□□□□□□□□□□□□□□.
- ST_Node - Nodes a collection of lines.
- ST_NumCurves - Return the number of component curves in a CompoundCurve.
- ST_NumGeometries - □□□□□□□□□□□□□□□□□□□□□□. □□□□□□□□□□□□□□□□.
- ST_NumPatches - Returns the number of faces in a PolyhedralSurface or TIN.
- ST_PatchN - Returns the Nth face of a PolyhedralSurface or TIN.
- ST_PointFromWKB - □□□ SRID □□□ WKB □□□□□□□□□□□.
- ST_PointN - ST_LineString □□ ST_CircularString □□□□□□□□□□□□□□□□□□.
- ST_PointOnSurface - Computes a point guaranteed to lie in a polygon, or on a geometry.
- ST_Points - □□□□□□□□□□□□□□□□□□□□□□□□□□□□.
- ST_Polygon - Creates a Polygon from a LineString with a specified SRID.
- ST_RemovePoint - Remove a point from a linestring.
- ST_RemoveRepeatedPoints - Returns a version of a geometry with duplicate points removed.
- ST_Reverse - □□□□□□□□□□□□□□□□□□□□□□.
- ST_Rotate - Rotates a geometry about an origin point.
- ST_RotateX - Rotates a geometry about the X axis.
- ST_RotateY - Rotates a geometry about the Y axis.
- ST_RotateZ - Rotates a geometry about the Z axis.
- ST_Scale - Scales a geometry by given factors.
- ST_Scroll - Change start point of a closed LineString.
- ST_SetPoint - □□□□□□□□□□□□□□□□□□□□□□□□□□□□.
- ST_ShiftLongitude - Shifts the longitude coordinates of a geometry between -180..180 and 0..360.
- ST_SnapToGrid - □□□□□□□□□□□□□□□□□□□□□□ (snap) □□□□□.
- ST_StartPoint - Returns the first point of a LineString, CircularLineString, or NURBSCurve.
- ST_SwapOrdinates - □□□□□□□□□□□□□□□□□□□□□□□□.
- ST_SymDifference - Computes a geometry representing the portions of geometries A and B that do not intersect.
- ST_TransScale - Translates and scales a geometry by given offsets and factors.
- ST_Translate - Translates a geometry by given offsets.
- ST_UnaryUnion - Computes the union of the components of a single geometry.
- ST_Union - Computes a geometry representing the point-set union of the input geometries.
- ST_WrapX - X □□□□□□□□□□□□□□□.
- ST_X - Returns the X coordinate of a Point.
- ST_XMax - Returns the X maxima of a 2D or 3D bounding box or a geometry.

- ST_XMin - Returns the X minima of a 2D or 3D bounding box or a geometry.
- ST_XSize - Returns the X size of a 2D or 3D bounding box or a geometry.
- ST_Y - Returns the Y coordinate of a Point.
- ST_YMax - Returns the Y maxima of a 2D or 3D bounding box or a geometry.
- ST_YMin - Returns the Y minima of a 2D or 3D bounding box or a geometry.
- ST_YSize - Returns the Y size of a 2D or 3D bounding box or a geometry.
- ST_Z - Returns the Z coordinate of a Point.
- ST_ZMFlag - ST_Geometry □□□□□□□□□□□□.
- ST_ZMax - Returns the Z maxima of a 2D or 3D bounding box or a geometry.
- ST_ZMin - Returns the Z minima of a 2D or 3D bounding box or a geometry.
- ST_ZSize - Returns the Z size of a 2D or 3D bounding box or a geometry.
- Equals - □ TopoGeometry □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□.
- Intersects - □ TopoGeometry □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□.
- UpdateGeometrySRID - Updates the SRID of all features in a geometry column, and the table metadata.
- &&& - A □ n □□□□□□□□ B □ n □□□□□□□□□□□□□□□ TRUE □□□□□□□.
- &&&(geometry,gidx) - Returns TRUE if a geometry's (cached) n-D bounding box intersects a n-D float precision bounding box (GIDX).
- &&&(gidx,geometry) - Returns TRUE if a n-D float precision bounding box (GIDX) intersects a geometry's (cached) n-D bounding box.
- &&&(gidx,gidx) - Returns TRUE if two n-D float precision bounding boxes (GIDX) intersect each other.

13.9 PostGIS Curved Geometry Support Functions

The functions given below are PostGIS functions that can use CIRCULARSTRING, CURVEPOLYGON, and other curved geometry types

- AddGeometryColumn - □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□.
- Box2D - Returns a BOX2D representing the 2D extent of a geometry.
- Box3D - Returns a BOX3D representing the 3D extent of a geometry.
- DropGeometryColumn - □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□.
- GeometryType - ST_Geometry □□□□□□□□□□□□□□.
- PostGIS_AddBBox - □□□□□□□□□□□□□□□□.
- PostGIS_DropBBox - □□□□□□□□□□□□□□□□□□□□□□□□.
- PostGIS_HasBBox - □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□, □□□□□□□□□□□□□□□□□□□□.
- ST_3DExtent - Aggregate function that returns the 3D bounding box of geometries.
- ST_Affine - Apply a 3D affine transformation to a geometry.

- ST_AsBinary - Return the OGC/ISO Well-Known Binary (WKB) representation of the geometry/geography without SRID meta data.
- ST_AsEWKB - Return the Extended Well-Known Binary (EWKB) representation of the geometry with SRID meta data.
- ST_AsEWKT - □□□ WKT(Well-Known Text) □□□□ SRID □□□□□□□□□□□□□□.
- ST_AsHEXEWKB - □□□□□□□ (NDR) □□□□□□ (XDR) □□□□□□ HEXEWKB (□□□) □□□ □□□□□□.
- ST_AsSVG - Returns SVG path data for a geometry.
- ST_AsText - □□/□□□□ WKT(Well-Known Text) □□□□ SRID □□□□□□□□□□□□.
- ST_ClusterDBSCAN - Window function that returns a cluster id for each input geometry using the DBSCAN algorithm.
- ST_ClusterWithin - Aggregate function that clusters geometries by separation distance.
- ST_ClusterWithinWin - Window function that returns a cluster id for each input geometry, clustering using separation distance.
- ST_Collect - Creates a GeometryCollection or Multi* geometry from a set of geometries.
- ST_CoordDim - ST_Geometry □□□□□□□□□□□□.
- ST_CurveToLine - Converts a geometry containing curves to a linear geometry.
- ST_Distance - □□□□□□ 3 □□□□ (longest) □□□□□□□□.
- ST_Dump - Returns a set of geometry_dump rows for the components of a geometry.
- ST_DumpPoints - □□□□□□□□□□□□□□□□□□.
- ST_EndM - Returns the M coordinate of the last point of a LineString, CircularLineString, or NURBSCurve.
- ST_EndPoint - Returns the last point of a LineString, CircularLineString, or NURBSCurve.
- ST_EstimatedExtent - Returns the estimated extent of a spatial table.
- ST_FlipCoordinates - Returns a version of a geometry with X and Y axis flipped.
- ST_Force2D - □□□”2 □□□□” □□□□□□.
- ST_ForceCurve - □□□□□□□□, □□□□□□□□□□□□□□□□□□□□ (upcast) □□□.
- ST_ForceSFS - Force geometries to use SFS 1.1 or 1.2 geometry types.
- ST_Force3D - □□□ XYZ □□□□□□□□. ST_Force3DZ □□□□□□.
- ST_Force3DM - □□□ XYM □□□□□□□□.
- ST_Force3DZ - □□□ XYZ □□□□□□□□.
- ST_Force4D - □□□ XYZM □□□□□□□□.
- ST_ForceCollection - □□□□□□□□□□□□□□□.
- ST_GeoHash - □□□ GeoHash □□□□□□□□□.
- ST_GeogFromWKB - WKB □□□□□□□□ EWKB(□□ WKB) □□□□□□□□□□□□□□□□□.
- ST_GeomFromEWKB - EWKB(Extended Well-Known Binary) □□□□□□ ST_Geometry □□□□□□□.

- ST_GeomFromEWKT - EWKT(Extended Well-Known Text) □□□□□□ ST_Geometry □□□□□□□.
- ST_GeomFromText - WKT □□□□□□□□□□ ST_Geometry □□□□□□□.
- ST_GeomFromWKB - WKB(Well-Known Binary) □□□□□□□□□□ SRID □□□□□□□□□□□□□ □□□.
- ST_GeometryN - ST_Geometry □□□□□□□□□□□□.
- = - Returns TRUE if the coordinates and coordinate order geometry/geography A are the same as the coordinates and coordinate order of geometry/geography B.
- &<| - A □□□□□□ B □□□□□□□□□□□□□□□□□□ TRUE □□□□□□.
- ST_HasArc - Tests if a geometry contains a circular arc
- ST_Intersects - Tests if two geometries intersect (they have at least one point in common)
- ST_IsClosed - LINESTRING □□□□□□□□□□□□□□□ TRUE □□□□□□. □□□□□□□□ (□□□□□□) □□□□ TRUE □□□□□□.
- ST_IsCollection - □□□□□□□□□□□, □□□, □□□□□□□ TRUE □□□□□□.
- ST_IsEmpty - Tests if a geometry is empty.
- ST_LineToCurve - Converts a linear geometry to a curved geometry.
- ST_MMax - Returns the M maxima of a geometry.
- ST_MMin - Returns the M minima of a geometry.
- ST_MSize - Returns the M size of a geometry.
- ST_MemSize - ST_Geometry □□□□□□□□□□□□.
- ST_NPoints - □□□□□□□□□□□ (□□□) □□□□□□□□□.
- ST_NRings - □□□□□□□□□□□□□□□□□□□□.
- ST_PointFromWKB - □□□ SRID □□□ WKB □□□□□□□□□□.
- ST_PointN - ST_LineString □□ ST_CircularString □□□□□□□□□□□□□□□□.
- ST_Points - □□□□□□□□□□□□□□□□□□□□□□□□□□.
- ST_Rotate - Rotates a geometry about an origin point.
- ST_RotateZ - Rotates a geometry about the Z axis.
- ST_SRID - Returns the spatial reference identifier for a geometry.
- ST_Scale - Scales a geometry by given factors.
- ST_SetEndM - Sets the M coordinate of the last point of a LineString, CircularLineString, or NURBSCurve.
- ST_SetSRID - Set the SRID on a geometry.
- ST_SetStartM - Sets the M coordinate of the first point of a LineString, CircularLineString, or NURBSCurve.
- ST_StartM - Returns the M coordinate of the first point of a LineString, CircularLineString, or NURBSCurve.
- ST_StartPoint - Returns the first point of a LineString, CircularLineString, or NURBSCurve.
- ST_Summary - □□□□□□□□□□□□□□□□□□.

- ST_SwapOrdinates - □□□□□□□□□□□□□□□□□□□□□□□□.
- ST_TransScale - Translates and scales a geometry by given offsets and factors.
- ST_Transform - Return a new geometry with coordinates transformed to a different spatial reference system.
- ST_Translate - Translates a geometry by given offsets.
- ST_XMax - Returns the X maxima of a 2D or 3D bounding box or a geometry.
- ST_XMin - Returns the X minima of a 2D or 3D bounding box or a geometry.
- ST_XSize - Returns the X size of a 2D or 3D bounding box or a geometry.
- ST_YMax - Returns the Y maxima of a 2D or 3D bounding box or a geometry.
- ST_YMin - Returns the Y minima of a 2D or 3D bounding box or a geometry.
- ST_YSize - Returns the Y size of a 2D or 3D bounding box or a geometry.
- ST_ZMFlag - ST_Geometry □□□□□□□□□□□□□.
- ST_ZMax - Returns the Z maxima of a 2D or 3D bounding box or a geometry.
- ST_ZMin - Returns the Z minima of a 2D or 3D bounding box or a geometry.
- ST_ZSize - Returns the Z size of a 2D or 3D bounding box or a geometry.
- UpdateGeometrySRID - Updates the SRID of all features in a geometry column, and the table metadata.
- ~(box2df,box2df) - Returns TRUE if a 2D float precision bounding box (BOX2DF) contains another 2D float precision bounding box (BOX2DF).
- ~(box2df,geometry) - Returns TRUE if a 2D float precision bounding box (BOX2DF) contains a geometry's 2D bonding box.
- ~(geometry,box2df) - Returns TRUE if a geometry's 2D bonding box contains a 2D float precision bounding box (GIDX).
- && - A □ 2D □□□□□ B □ 2D □□□□□□□□□□□ TRUE □□□□□□.
- &&& - A □ n □□□□□□□ B □ n □□□□□□□□□□□□□ TRUE □□□□□□.
- @(box2df,box2df) - Returns TRUE if a 2D float precision bounding box (BOX2DF) is contained into another 2D float precision bounding box.
- @(box2df,geometry) - Returns TRUE if a 2D float precision bounding box (BOX2DF) is contained into a geometry's 2D bounding box.
- @(geometry,box2df) - Returns TRUE if a geometry's 2D bounding box is contained into a 2D float precision bounding box (BOX2DF).
- &&(box2df,box2df) - Returns TRUE if two 2D float precision bounding boxes (BOX2DF) intersect each other.
- &&(box2df,geometry) - Returns TRUE if a 2D float precision bounding box (BOX2DF) intersects a geometry's (cached) 2D bounding box.
- &&(geometry,box2df) - Returns TRUE if a geometry's (cached) 2D bounding box intersects a 2D float precision bounding box (BOX2DF).
- &&&(geometry,gidx) - Returns TRUE if a geometry's (cached) n-D bounding box intersects a n-D float precision bounding box (GIDX).
- &&&(gidx,geometry) - Returns TRUE if a n-D float precision bounding box (GIDX) intersects a geometry's (cached) n-D bounding box.
- &&&(gidx,gidx) - Returns TRUE if two n-D float precision bounding boxes (GIDX) intersect each other.

13.10 PostGIS Polyhedral Surface Support Functions

The functions given below are PostGIS functions that can use POLYHEDRALSURFACE, POLYHEDRALSURFACEM geometries

- Box2D - Returns a BOX2D representing the 2D extent of a geometry.
- Box3D - Returns a BOX3D representing the 3D extent of a geometry.
- CG_3DArea - 3 □□□□□□□□□□□□□□□□. □□□□□ 0 □□□□□□□□.
- CG_3DConvexHull - □□□□□□□□□□□□□□□□.
- CG_3DDifference - 3 □□□□□□□□□□.
- CG_3DIntersection - 3 □□□□□□□□□□.
- CG_3DUnion - Perform 3D union using postgis_sfcgal.
- CG_ApproximateMedialAxis - □□□□□□□□□□□□□□□□.
- CG_Extrude - □□□□□□□□□□□□□□.
- CG_ForceLHR - LHR(Left Hand Reverse; □□□□) □□□□□□□□.
- CG_GenerateFlatRoof - Generate a flat (box) roof from a footprint polygon.
- CG_GenerateGableRoof - Generate a gable roof from a footprint polygon.
- CG_GenerateHippedRoof - Generate a hipped roof from a footprint polygon.
- CG_GenerateRoof - Generate a roof of the requested type from a footprint polygon.
- CG_GenerateSkillionRoof - Generate a skillion (single-slope) roof from a footprint polygon.
- CG_IsPlanar - □□□□□□□□□□□□□□□□.
- CG_IsSimple - Tests whether a geometry is simple according to SFCGAL
- CG_IsSimpleDetail - Returns a reason when a geometry is not simple according to SFCGAL
- CG_IsSolid - □□□□□□□□□□□□□□. □□□□□□□□□□□□□□□□.
- CG_IsValid - Tests whether a geometry is valid according to SFCGAL
- CG_IsValidDetail - Returns a valid_detail row stating whether a geometry is valid according to SFCGAL, and if not a reason and a location.
- CG_MakeSolid - □□□□□□□□□□□□. □□□□□□□□□□□□□□□□. □□□□□□□□□□□□, □□□ □□□□□□□□□□□□□□□□ TIN □□□□□□□□.
- CG_PolygonRepair - Repair an invalid polygon or multipolygon.
- CG_StraightSkeleton - □□□□□□□□□□□□ (straight skeleton) □□□□□□□.
- CG_Tesselate - □□□□□□□□□□□□□□□□□□ (tessellation) □□□□□□□□□ TIN □□ TIN □□□□□ □□□□□□.
- CG_Visibility - Compute a visibility polygon from a point or a segment in a polygon geometry
- CG_Volume - Computes the volume of a 3D geometry. Closed surfaces can have non-zero volume.
- GeometryType - ST_Geometry □□□□□□□□□□□□□.
- ST_3DClosestPoint - g2 □□□□□□□ g1 □□□□□ 3 □□□□□□□□□□□□□. □□□□□□□ 3D □□□□□□ □□□□□□□□□□□.

- ST_3DDFullyWithin - Tests if two 3D geometries are entirely within a given 3D distance
- ST_3DDWithin - Tests if two 3D geometries are within a given 3D distance
- ST_3DDistance - □□□□□□□, □□□□□□ (SRS □□□) 3 □□□□□□□□□□□□□□□□□□□□□□□□.
- ST_3DExtent - Aggregate function that returns the 3D bounding box of geometries.
- ST_3DIntersects - Tests if two geometries spatially intersect in 3D - only for points, linestrings, polygons, polyhedral surface (area)
- ST_3DLongestLine - □□□□□□ 3 □□□□ (longest) □□□□□□□□.
- ST_3DMaxDistance - □□□□□□□, □□□□□□ (SRS □□□□) 3 □□□□□□□□□□□□□□□□□□□□□□□□.
- ST_3DShortestLine - □□□□□□ 3 □□□□ (shortest) □□□□□□□□.
- ST_Affine - Apply a 3D affine transformation to a geometry.
- ST_Area - □□□□□□□□□□□□□□□□.
- ST_AsBinary - Return the OGC/ISO Well-Known Binary (WKB) representation of the geometry/geography without SRID meta data.
- ST_AsEWKB - Return the Extended Well-Known Binary (EWKB) representation of the geometry with SRID meta data.
- ST_AsEWKT - □□□ WKT(Well-Known Text) □□□□ SRID □□□□□□□□□□□□□□.
- ST_AsGML - □□□ GML 2 □□ GML 3 □□□□□□□□□□.
- ST_AsX3D - □□□ X3D XML □□□□□□: ISO-IEC-19776-1.2-X3DEncodings-XML □□□□□□.
- ST_CoordDim - ST_Geometry □□□□□□□□□□□□.
- ST_Dimension - ST_Geometry □□□□□□□□□□□□.
- ST_Dump - Returns a set of geometry_dump rows for the components of a geometry.
- ST_DumpPoints - □□□□□□□□□□□□□□□□□□□□.
- ST_Expand - Returns a bounding box expanded from another bounding box or a geometry.
- ST_Extent - Aggregate function that returns the bounding box of geometries.
- ST_FlipCoordinates - Returns a version of a geometry with X and Y axis flipped.
- ST_Force2D - □□□”2 □□□□” □□□□□□.
- ST_ForceRHR - □□□□□□□□□□□□□□□ (orientation) □□□□□□ (Right-Hand Rule) □□□□□□□□□□.
- ST_ForceSFS - Force geometries to use SFS 1.1 or 1.2 geometry types.
- ST_Force3D - □□□ XYZ □□□□□□□□. ST_Force3DZ □□□□□□.
- ST_Force3DZ - □□□ XYZ □□□□□□□□.
- ST_ForceCollection - □□□□□□□□□□□□□□□.
- ST_GeomFromEWKB - EWKB(Extended Well-Known Binary) □□□□□□ ST_Geometry □□□□□□□.
- ST_GeomFromEWKT - EWKT(Extended Well-Known Text) □□□□□□ ST_Geometry □□□□□□□.

- ST_GeomFromGML - □□□ GML □□□□□□□□ PostGIS □□□□□□□□□□.
- ST_GeometryN - ST_Geometry □□□□□□□□□□□□.
- ST_GeometryType - ST_Geometry □□□□□□□□□□□□.
- = - Returns TRUE if the coordinates and coordinate order geometry/geography A are the same as the coordinates and coordinate order of geometry/geography B.
- &<| - A □□□□□□□ B □□□□□□□□□□□□□□□□□□□ TRUE □□□□□□□.
- ~= - A □□□□□□□ B □□□□□□□□□□□□ TRUE □□□□□□□.
- ST_IsClosed - LINESTRING □□□□□□□□□□□□□□□□ TRUE □□□□□□□. □□□□□□□□□ (□□□□□□□) □□□□□ TRUE □□□□□□□.
- ST_MemSize - ST_Geometry □□□□□□□□□□□□□.
- ST_NPoints - □□□□□□□□□□□□ (□□□) □□□□□□□□□□.
- ST_NumGeometries - □□□□□□□□□□□□□□□□□□□□□. □□□□□□□□□□□□□□□□.
- ST_NumPatches - Returns the number of faces in a PolyhedralSurface or TIN.
- ST_PatchN - Returns the Nth face of a PolyhedralSurface or TIN.
- ST_RemoveRepeatedPoints - Returns a version of a geometry with duplicate points removed.
- ST_Reverse - □□□□□□□□□□□□□□□□□□□□□.
- ST_Rotate - Rotates a geometry about an origin point.
- ST_RotateX - Rotates a geometry about the X axis.
- ST_RotateY - Rotates a geometry about the Y axis.
- ST_RotateZ - Rotates a geometry about the Z axis.
- ST_Scale - Scales a geometry by given factors.
- ST_ShiftLongitude - Shifts the longitude coordinates of a geometry between -180..180 and 0..360.
- ST_Summary - □□□□□□□□□□□□□□□□□□□□.
- ST_SwapOrdinates - □□□□□□□□□□□□□□□□□□□□□.
- ST_Transform - Return a new geometry with coordinates transformed to a different spatial reference system.
- ~(box2df,box2df) - Returns TRUE if a 2D float precision bounding box (BOX2DF) contains another 2D float precision bounding box (BOX2DF).
- ~(box2df,geometry) - Returns TRUE if a 2D float precision bounding box (BOX2DF) contains a geometry's 2D bonding box.
- ~(geometry,box2df) - Returns TRUE if a geometry's 2D bonding box contains a 2D float precision bounding box (GIDX).
- && - A □ 2D □□□□□□ B □ 2D □□□□□□□□□□□□ TRUE □□□□□□□.
- &&& - A □ n □□□□□□□□ B □ n □□□□□□□□□□□□□□□ TRUE □□□□□□□.
- @(box2df,box2df) - Returns TRUE if a 2D float precision bounding box (BOX2DF) is contained into another 2D float precision bounding box.
- @(box2df,geometry) - Returns TRUE if a 2D float precision bounding box (BOX2DF) is contained into a geometry's 2D bounding box.

- @(geometry,box2df) - Returns TRUE if a geometry's 2D bounding box is contained into a 2D float precision bounding box (BOX2DF).
- &&(box2df,box2df) - Returns TRUE if two 2D float precision bounding boxes (BOX2DF) intersect each other.
- &&(box2df,geometry) - Returns TRUE if a 2D float precision bounding box (BOX2DF) intersects a geometry's (cached) 2D bounding box.
- &&(geometry,box2df) - Returns TRUE if a geometry's (cached) 2D bounding box intersects a 2D float precision bounding box (BOX2DF).
- &&&(geometry,gidx) - Returns TRUE if a geometry's (cached) n-D bounding box intersects a n-D float precision bounding box (GIDX).
- &&&(gidx,geometry) - Returns TRUE if a n-D float precision bounding box (GIDX) intersects a geometry's (cached) n-D bounding box.
- &&&(gidx,gidx) - Returns TRUE if two n-D float precision bounding boxes (GIDX) intersect each other.

13.11 PostGIS Function Support Matrix

Below is an alphabetical listing of spatial specific functions in PostGIS and the kinds of spatial types they work with or OGC/SQL compliance they try to conform to.

- A ✓ means the function works with the type or subtype natively.
- A means it works but with a transform cast built-in using cast to geometry, transform to a "best srid" spatial ref and then cast back. Results may not be as expected for large areas or areas at poles and may accumulate floating point junk.
- A means the function works with the type because of a auto-cast to another such as to box3d rather than direct type support.
- A means the function only available if PostGIS compiled with SFCGAL support.
- geom - Basic 2D geometry support (x,y).
- geog - Basic 2D geography support (x,y).
- 2.5D - basic 2D geometries in 3 D/4D space (has Z or M coord).
- PS - Polyhedral surfaces
- T - Triangles and Triangulated Irregular Network surfaces (TIN)

Function	geom	geog	2.5D	Curves	SQL MM	PS	T
ST_Collect	✓		✓	✓			
ST_LineFromMu	✓ Point		✓				
ST_MakeEnvelo	✓						
ST_MakeLine	✓		✓				

Function	geom	geog	2.5D	Curves	SQL MM	PS	T
ST_MakePoint	✓		✓				
ST_MakePointM	✓						
ST_MakePolygo	✓		✓				
ST_Point	✓				✓		
ST_PointZ	✓						
ST_PointM	✓						
ST_PointZM	✓						
ST_Polygon	✓		✓		✓		
ST_TileEnvelop	✓						
ST_HexagonGri	✓						
ST_Hexagon	✓						
ST_SquareGrid	✓						
ST_Square	✓						
ST_Letters	✓						
ST_MakeNurbs	✓ ve						
GeometryType	✓		✓	✓		✓	✓
ST_Boundary	✓		✓		✓		
ST_BoundingDi	✓ nal		✓				
ST_CoordDim	✓		✓	✓	✓	✓	✓
ST_Dimension	✓				✓	✓	✓
ST_Dump	✓		✓	✓		✓	✓
ST_DumpPoints	✓		✓	✓		✓	✓
ST_DumpSegme	✓ s		✓				✓
ST_DumpRings	✓		✓				
ST_EndPoint	✓		✓	✓	✓		
ST_Envelope	✓				✓		
ST_ExteriorRing	✓		✓		✓		
ST_GeometryN	✓		✓	✓	✓	✓	✓
ST_GeometryTy	✓		✓		✓	✓	
ST_HasArc	✓		✓	✓			
ST_InteriorRing	✓		✓		✓		
ST_NumCurves	✓		✓		✓		
ST_CurveN	✓		✓		✓		
ST_IsClosed	✓		✓	✓	✓	✓	

Function	geom	geog	2.5D	Curves	SQL MM	PS	T
ST_IsCollection	✓		✓	✓			
ST_IsEmpty	✓			✓	✓		
ST_IsPolygonCC	✓		✓				
ST_IsPolygonCW	✓		✓				
ST_IsRing	✓				✓		
ST_IsSimple	✓		✓		✓		
ST_IsSimpleDet	✓						
ST_M	✓		✓		✓		
ST_MemSize	✓		✓	✓		✓	✓
ST_NDims	✓		✓				
ST_NPoints	✓		✓	✓		✓	
ST_NRings	✓		✓	✓			
ST_NumGeome	✓	s	✓		✓	✓	✓
ST_NumInterior	✓	ıgs			✓		
ST_NumInterior	✓	ıg					
ST_NumPatches	✓		✓		✓	✓	
ST_NumPoints	✓				✓		
ST_PatchN	✓		✓		✓	✓	✓
ST_PointN	✓		✓	✓	✓		
ST_Points	✓		✓	✓			
ST_StartPoint	✓		✓	✓	✓		
ST_StartM	✓			✓			
ST_EndM	✓			✓			
ST_SetStartM	✓			✓			
ST_SetEndM	✓			✓			
ST_Summary	✓	✓		✓		✓	✓
ST_X	✓		✓		✓		
ST_Y	✓		✓		✓		
ST_Z	✓		✓		✓		
ST_ZMFlag	✓		✓	✓			
ST_HasZ	✓		✓				
ST_HasM	✓		✓				
ST_ControlPoin	✓						
ST_Degree	✓						

Function	geom	geog	2.5D	Curves	SQL MM	PS	T
ST_Weights	✓						
ST_Knots	✓						
ST_NumControl	✓	nts					
ST_NurbsCurve	✓	ational					
ST_IsNurbsCur	✓						
ST_Evaluate	✓						
ST_NurbsToLin	✓	ring					
ST_AddPoint	✓		✓				
ST_CollectionEx	✓	ct					
ST_CollectionH	✓	ogenize					
ST_CurveToLine	✓		✓	✓	✓		
ST_Scroll	✓		✓				
ST_FlipCoordina	✓	s	✓	✓		✓	✓
ST_Force2D	✓		✓	✓		✓	
ST_Force3D	✓		✓	✓		✓	
ST_Force3DZ	✓		✓	✓		✓	
ST_Force3DM	✓			✓			
ST_Force4D	✓		✓	✓			
ST_ForceCollec	✓	ı	✓	✓		✓	
ST_ForceCurve	✓		✓	✓			
ST_ForcePolygo	✓	CW	✓				
ST_ForcePolygo	✓	W	✓				
ST_ForceSFS	✓		✓	✓		✓	✓
ST_ForceRHR	✓		✓			✓	
ST_LineExtend	✓						
ST_LineToCurve	✓		✓	✓			
ST_Multi	✓						
ST_Normalize	✓						
ST_Project	✓	✓					
ST_QuantizeCo	✓	inates					
ST_RemovePoin	✓		✓				
ST_RemoveRep	✓	edPoints	✓			✓	
ST_RemoveIrrel	✓	ıntPointsForView					
ST_RemoveSma	✓	arts					

Function	geom	geog	2.5D	Curves	SQL MM	PS	T
ST_Reverse	✓		✓			✓	
ST_Segmentize	✓	✓					
ST_SetPoint	✓		✓				
ST_ShiftLongitu	✓		✓			✓	✓
ST_WrapX	✓		✓				
ST_SnapToGrid	✓		✓				
ST_Snap	✓						
ST_SwapOrdina	✓		✓	✓		✓	✓
ST_IsValid	✓				✓		
ST_IsValidDetai	✓						
ST_IsValidReas	✓						
ST_MakeValid	✓		✓				
ST_InverseTran	✓	rmPipeline					
ST_SetSRID	✓			✓			
ST_SRID	✓			✓	✓		
ST_Transform	✓			✓	✓	✓	
ST_TransformPi	✓	ine					
postgis_srs_codes							
postgis_srs	✓						
postgis_srs_all	✓						
postgis_srs_sea	✓						
ST_BdPolyFrom	✓	t					
ST_BdMPolyFro	✓	ext					
ST_GeogFromText		✓					
ST_GeographyFromText		✓					
ST_GeomCollFr	✓	Text			✓		
ST_GeomFromE	✓	T		✓	✓	✓	✓
ST_GeomFromM	✓	RC21					
ST_GeometryFr	✓	Text			✓		
ST_GeomFromT	✓			✓	✓		
ST_LineFromTe	✓				✓		
ST_MLineFrom	✓	t			✓		
ST_MPointFrom	✓	t			✓		
ST_MPolyFromT	✓				✓		

Function	geom	geog	2.5D	Curves	SQL MM	PS	T
ST_PointFromTe	✓				✓		
ST_PolygonFror xt	✓				✓		
ST_WKTToSQL	✓				✓		
ST_GeogFromWKB		✓		✓			
ST_GeomFromE CB	✓		✓	✓		✓	✓
ST_GeomFromW B	✓			✓	✓		
ST_LineFromWI	✓				✓		
ST_LinestringFr WKB	✓				✓		
ST_PointFromW	✓		✓	✓	✓		
ST_WKBToSQL	✓				✓		
ST_Box2dFromG Hash	✗						
ST_GeomFromG Hash	✓						
ST_GeomFromG	✓		✓			✓	✓
ST_GeomFromG JSON	✓		✓				
ST_GeomFromK	✓		✓				
ST_GeomFromT CB	✓						
ST_GMLToSQL	✓				✓		
ST_LineFromEn edPolyline	✓						
ST_PointFromGeoHash							
ST_FromFlatGeobufToTable							
ST_FromFlatGeobuf							
ST_AsEWKT	✓	✓	✓	✓		✓	✓
ST_AsText	✓	✓		✓	✓		
ST_AsBinary	✓	✓	✓	✓	✓	✓	✓
ST_AsEWKB	✓		✓	✓		✓	✓
ST_AsHEXEWK	✓		✓	✓			
ST_AsEncodedP line	✓						
ST_AsFlatGeobu	✗						
ST_AsGeobuf	✗						
ST_AsGeoJSON	✓	✓	✓				
ST_AsGML	✓	✓	✓		✓	✓	✓
ST_AsKML	✓	✓	✓				
ST_AsLatLonTe	✓						
ST_AsMARC21	✓						
ST_AsMVTGeon	✓						

Function	geom	geog	2.5D	Curves	SQL MM	PS	T
ST_AsMVT	☑						
ST_AsSVG	✓	✓		✓			
ST_AsTWKB	✓						
ST_AsX3D	✓		✓			✓	✓
ST_GeoHash	✓			✓			
&&	✓	✓		✓		✓	
&&(geometry,box2df)	✓			✓		✓	
&&(box2df,geometry)	✓			✓		✓	
&&(box2df,box2df)	☑			✓		✓	
&&&	✓		✓	✓		✓	✓
&&&(geometry,gidx)	✓		✓	✓		✓	✓
&&&(gidx,geometry)	✓		✓	✓		✓	✓
&&&(gidx,gidx)			✓	✓		✓	✓
&<	✓						
&<|	✓			✓		✓	
&>	✓						
<<	✓						
<<|	✓						
=	✓	✓		✓		✓	
>>	✓						
@	✓						
@(geometry,box2df)	✓			✓		✓	
@(box2df,geometry)	✓			✓		✓	
@(box2df,box2df)	☑			✓		✓	
|&>	✓						
|>>	✓						
~	✓						
~(geometry,box2df)	✓			✓		✓	
~(box2df,geometry)	✓			✓		✓	
~(box2df,box2df)	☑			✓		✓	
~=	✓					✓	
<->	✓	✓					
|=|	✓						
<#>	✓						

Function	geom	geog	2.5D	Curves	SQL MM	PS	T
<<->>	✓						
ST_3DIntersects	✓		✓		✓	✓	✓
ST_Contains	✓				✓		
ST_ContainsProperly	✓						
ST_CoveredBy	✓	✓					
ST_Covers	✓	✓					
ST_Crosses	✓				✓		
ST_Disjoint	✓				✓		
ST_Equals	✓				✓		
ST_Intersects	✓	✓		✓	✓		✓
ST_LineCrossingDirection	✓						
ST_OrderingEquals	✓				✓		
ST_Overlaps	✓				✓		
ST_Relate	✓				✓		
ST_RelateMatch							
ST_Touches	✓				✓		
ST_Within	✓				✓		
ST_3DDWithin	✓		✓		✓	✓	
ST_3DDFullyWithin	✓		✓			✓	
ST_DFullyWithin	✓						
ST_DWithin	✓	✓					
ST_PointInsideCircle	✓						
ST_Area	✓	✓			✓	✓	
ST_Azimuth	✓	✓					
ST_Angle	✓						
ST_ClosestPoint	✓	✓					
ST_3DClosestPoint	✓		✓			✓	
ST_Distance	✓	✓		✓	✓		
ST_3DDistance	✓		✓		✓	✓	
ST_DistanceSphere	✓						
ST_DistanceSpheroid	✓						
ST_FrechetDistance	✓						
ST_HausdorffDistance	✓						
ST_Length	✓	✓			✓		

Function	geom	geog	2.5D	Curves	SQL MM	PS	T
ST_Length2D	✓						
ST_3DLength	✓		✓		✓		
ST_LengthSphe✓d	✓		✓				
ST_LongestLine	✓						
ST_3DLongestL	✓		✓			✓	
ST_MaxDistanc	✓						
ST_3DMaxDista✓e	✓		✓			✓	
ST_MinimumCl✓ance	✓						
ST_MinimumCl✓anceLine	✓						
ST_Perimeter	✓	✓			✓		
ST_Perimeter2D	✓						
ST_3DPerimeter	✓		✓		✓		
ST_ShortestLin	✓	✓					
ST_3DShortestL✓e	✓		✓			✓	
ST_ClipByBox2D	✓						
ST_Difference	✓		✓		✓		
ST_Intersection	✓	😁	✓		✓		
ST_MemUnion	✓		✓				
ST_Node	✓		✓				
ST_Split	✓						
ST_Subdivide	✓						
ST_SymDifferen	✓		✓		✓		
ST_UnaryUnion	✓		✓				
ST_Union	✓		✓		✓		
ST_Buffer	✓	😁			✓		
ST_BuildArea	✓						
ST_Centroid	✓	✓			✓		
ST_ChaikinSmo✓ing	✓		✓				
ST_CatmullRom✓oothing	✓		✓				
ST_ConcaveHul	✓						
ST_ConvexHull	✓		✓		✓		
ST_DelaunayTri✓gles	✓		✓				✓
ST_FilterByM	✓						
ST_GeneratePoi	✓						

Function	geom	geog	2.5D	Curves	SQL MM	PS	T
ST_GeometricM an	✓		✓				
ST_LineMerge	✓						
ST_MaximumIn bedCircle	✓						
ST_LargestEmp ircle	✓						
ST_MinimumBo lingCircle	✓						
ST_MinimumBo lingRadius	✓						
ST_OrientedEnv pe	✓						
ST_OffsetCurve	✓						
ST_PointOnSurf	✓		✓		✓		
ST_Polygonize	✓						
ST_ReducePrec n	✓						
ST_SharedPaths	✓						
ST_Simplify	✓						
ST_SimplifyPres veTopology	✓						
ST_SimplifyPoly nHull	✓						
ST_SimplifyVW	✓						
ST_SetEffective a	✓						
ST_TriangulateI gon	✓						
ST_VoronoiLine	✓						
ST_VoronoiPoly s	✓						
ST_CoverageInv dEdges	✓						
ST_CoverageCl	✓						
ST_CoverageSi ify	✓						
ST_CoverageEd	✓						
ST_CoverageUn	✓						
ST_Affine	✓		✓	✓		✓	✓
ST_Rotate	✓		✓	✓		✓	✓
ST_RotateX	✓		✓			✓	✓
ST_RotateY	✓		✓			✓	✓
ST_RotateZ	✓		✓	✓		✓	✓
ST_Scale	✓		✓	✓		✓	✓
ST_Translate	✓		✓	✓			
ST_TransScale	✓		✓	✓			
ST_ClusterDBS N	✓			✓			

Function	geom	geog	2.5D	Curves	SQL MM	PS	T
ST_ClusterInter ting	✓						
ST_ClusterInter tingWin	✓						
ST_ClusterRela Win	✓						
ST_MinimumSp ingTree	✓						
ST_ClusterKMe	✓						
ST_ClusterWith	✓			✓			
ST_ClusterWith Win	✓			✓			
Box2D	✓			✓		✓	✓
Box3D	✓		✓	✓		✓	✓
ST_EstimatedEx t	☑			✓			
ST_Expand	✓					✓	✓
ST_Extent	✓					✓	✓
ST_3DExtent	✓		✓	✓		✓	✓
ST_MakeBox2D	✓						
ST_3DMakeBox	✓						
ST_XMax	☑		✓	✓			
ST_XMin	☑		✓	✓			
ST_YMax	☑		✓	✓			
ST_YMin	☑		✓	✓			
ST_ZMax	☑		✓	✓			
ST_ZMin	☑		✓	✓			
ST_XSize	☑		✓	✓			
ST_YSize	☑		✓	✓			
ST_ZSize	☑		✓	✓			
ST_MMin	✓		✓	✓			
ST_MMax	✓		✓	✓			
ST_MSize	✓		✓	✓			
ST_LineInterpol Point	✓	✓	✓				
ST_3DLineInter atePoint	✓		✓				
ST_LineInterpol Points	✓	✓	✓				
ST_LineLocateP t	✓	✓					
ST_LineSubstri	✓	✓	✓				
ST_LocateAlong	✓				✓		
ST_LocateBetw	✓				✓		

Function	geom	geog	2.5D	Curves	SQL MM	PS	T
ST_LocateBetw	✓		✓				
Elevations							
ST_InterpolateP	✓		✓				
t							
ST_3DInterpola	✓		✓				
oint							
ST_AddMeasure	✓		✓				
ST_IsValidTraje	✓		✓				
ry							
ST_ClosestPoint	✓		✓				
Approach							
ST_DistanceCPA	✓		✓				
ST_CPAWithin	✓		✓				
postgis.gdal_datapath							
postgis.gdal_enabled_drivers							
postgis.enable_outdb_rasters							
postgis.gdal_vsi_options							
postgis.gdal_cpl_debug							
PostGIS_AddBB	✓			✓			
PostGIS_DropBB	✓			✓			
PostGIS_HasBB	✓			✓			
postgis_sfcgal_version							
postgis_sfcgal_full_version							
CG_ForceLHR	🧊		🧊			🧊	🧊
CG_IsPlanar	🧊		🧊			🧊	🧊
CG_IsValid	🧊		🧊			🧊	🧊
CG_IsValidDeta	🧊		🧊			🧊	🧊
CG_IsSimple	🧊		🧊			🧊	🧊
CG_IsSimpleDe	🧊		🧊			🧊	🧊
CG_IsSolid	🧊		🧊			🧊	🧊
CG_MakeSolid	🧊		🧊			🧊	🧊
CG_Orientation	🧊		🧊				
CG_Area	🧊						
CG_3DArea	🧊		🧊		🧊	🧊	🧊
CG_Volume	🧊		🧊		🧊	🧊	🧊
CG_Intersectio	🧊						

Function	geom	geog	2.5D	Curves	SQL MM	PS	T
CG_Intersects							
CG_3DIntersec							
CG_Difference							
CG_3DDifferen							
CG_Distance							
CG_3DDistanc							
CG_3DConvexH							
CG_3DIntersec							
CG_Union							
CG_3DUnion							
CG_AlphaShap							
CG_ApproxCon		artition					
CG_Approxima		dialAxis					
CG_GenerateF		of					
CG_GenerateH		dRoof					
CG_GenerateG		Roof					
CG_GenerateSl		nRoof					
CG_GenerateR							
CG_PolygonRej							
CG_Constraine		aunayTriangles					
CG_Extrude							
CG_ExtrudeStr		tSkeleton					
CG_GreeneApp		onvexPartition					
CG_Minkowski							

Function	geom	geog	2.5D	Curves	SQL MM	PS	T
CG_OptimalAlp		ıape					
CG_OptimalCo		Partition					
CG_StraightSk		n					
CG_Tesselate							
CG_Triangulate							
CG_Visibility							
CG_YMonotone		ition					
CG_StraightSk		nPartition					
CG_3DBuffer							
CG_Rotate							
CG_2DRotate							
CG_3DRotate							
CG_RotateX							
CG_RotateY							
CG_RotateZ							
CG_Scale							
CG_3DScale							
CG_3DScaleAr		Center					
CG_Translate							
CG_3DTranslat							
CG_Simplify							
CG_3DAlphaW		ng					
CG_NurbsCurv		ɛrpolate					
CG_NurbsCurv		ɔroximate					

Function	geom	geog	2.5D	Curves	SQL MM	PS	T
CG_NurbsCurv ivative							
getfaceedges_returntype							
TopoGeometry							
validatetopology_returntype							
TopoElement							
TopoElementArray							
AddTopoGeometryColumn							
RenameTopoGeometryColumn							
DropTopology							
RenameTopology							
DropTopoGeometryColumn							
FixCorruptTopoGeometryColumn							
Populate_Topology_Layer							
TopologySummary							
ValidateTopolog	✓						
ValidateTopologyRelation							
ValidateTopolog ecision	✓						
MakeTopologyP ise	✓						
FindTopology	✓						
FindLayer	✓						
TotalTopologySize							
UpgradeTopology							
CreateTopology							
CopyTopology							
ST_InitTopoGeo					✓		
ST_CreateTopo	✓				✓		
TopoGeo_AddPo	✓						
TopoGeo_AddLi tring	✓						
TopoGeo_AddPo on	✓						
TopoGeo_LoadG metry	✓						
ST_AddIsoNode	✓				✓		
ST_AddIsoEdge	✓				✓		
ST_AddEdgeNe aces	✓				✓		
ST_AddEdgeMo ace	✓				✓		
ST_RemEdgeNewFace					✓		
ST_RemEdgeModFace					✓		
ST_ChangeEdg om	✓				✓		
ST_ModEdgeSp	✓				✓		
ST_ModEdgeHeal					✓		
ST_NewEdgeHeal					✓		

Function	geom	geog	2.5D	Curves	SQL MM	PS	T
ST_MoveIsoNod	✓				✓		
ST_NewEdgesS	✓				✓		
ST_RemoveIsoNode					✓		
ST_RemoveIsoEdge					✓		
GetEdgeByPoin	✓						
GetFaceByPoint	✓						
GetFaceContain	✓ Point						
GetNodeByPoin	✓						
GetTopologyID							
GetTopologySRID							
GetTopologyName							
ST_GetFaceEdges					✓		
ST_GetFaceGeo	✓ try				✓		
GetRingEdges							
GetNodeEdges							
FindVertexSegn	✓ tPairsBelowDistance						
Polygonize							
AddNode	✓						
AddEdge	✓						
AddFace	✓						
ST_Simplify	✓						
RemoveUnused	✓ nitives						
CreateTopoGeo	✓						
toTopoGeom	✓						
TopoElementArray_Agg							
TopoElement	✓						
clearTopoGeom	✓						
TopoGeom_addl	✓ nent						
TopoGeom_rem	✓ ment						
TopoGeom_add	✓ oGeom						
toTopoGeom							
GetTopoGeomElementArray							
GetTopoGeomElements							
ST_SRID	✓						
AsGML	✓						
AsTopoJSON	✓						
Equals	✓		✓				
Intersects	✓		✓				

Function	geom	geog	2.5D	Curves	SQL MM	PS	T
geomval							
addbandarg							
rastbandarg							
raster							
reclassarg							
summarystats							
unionarg							
AddRasterConstraints							
DropRasterConstraints							
AddOverviewConstraints							
DropOverviewConstraints							
PostGIS_GDAL_Version							
PostGIS_Raster_Lib_Build_Date							
PostGIS_Raster_Lib_Version							
ST_GDALDrivers							
UpdateRasterSRID							
ST_CreateOverview							
ST_AddBand							
ST_AsRaster	✓						
ST_AsRasterAg	✓						
ST_Band							
ST_MakeEmptyCoverage							
ST_MakeEmptyRaster							
ST_Tile							
ST_Retile	✓						
ST_FromGDALRaster							
ST_GeoReference							
ST_Height							
ST_IsEmpty							
ST_MemSize							
ST_MetaData							
ST_NumBands							
ST_PixelHeight							
ST_PixelWidth							
ST_ScaleX							
ST_ScaleY							
ST_RasterToWorldCoord							
ST_RasterToWorldCoordX							
ST_RasterToWorldCoordY							
ST_Rotation							
ST_SkewX							
ST_SkewY							
ST_SRID							
ST_Summary							
ST_UpperLeftX							
ST_UpperLeftY							
ST_Width							
ST_WorldToRas	✓	Coord					
ST_WorldToRas	✓	CoordX					
ST_WorldToRas	✓	CoordY					
ST_BandMetaData							
ST_BandNoDataValue							

Function	geom	geog	2.5D	Curves	SQL MM	PS	T
ST_BandIsNoData							
ST_BandPath							
ST_BandFileSize							
ST_BandFileTimestamp							
ST_BandPixelType							
ST_MinPossibleValue							
ST_HasNoBand							
ST_PixelAsPolyg	✓						
ST_PixelAsPolyg	✓ s						
ST_PixelAsPoint	✓						
ST_PixelAsPoint	✓						
ST_PixelAsCent	✓ l						
ST_PixelAsCent	✓ ls						
ST_Value	✓						
ST_NearestValu	✓						
ST_SetZ	✓						
ST_SetM	✓						
ST_Neighborho	✓						
ST_SetValue	✓						
ST_SetValues							
ST_DumpValues							
ST_PixelOfValue							
ST_SetGeoReference							
ST_SetRotation							
ST_SetScale							
ST_SetSkew							
ST_SetSRID							
ST_SetUpperLeft							
ST_Resample							
ST_Rescale							
ST_Reskew							
ST_SnapToGrid							
ST_Resize							
ST_Transform							
ST_SetBandNoDataValue							
ST_SetBandIsNoData							
ST_SetBandPath							
ST_SetBandIndex							
ST_Count							
ST_CountAgg							
ST_Histogram							
ST_Quantile							
ST_SummaryStats							
ST_SummaryStatsAgg							
ST_ValueCount							
ST_RastFromWKB							
ST_RastFromHexWKB							
ST_AsBinary/ST_AsWKB							

Function	geom	geog	2.5D	Curves	SQL MM	PS	T
ST_AsHexWKB							
ST_AsGDALRaster							
ST_AsJPEG							
ST_AsPNG							
ST_AsTIFF							
ST_Clip	✓						
ST_ColorMap							
ST_Grayscale							
ST_Intersection	✓						
ST_MapAlgebra (callback function version)							
ST_MapAlgebra (expres-sion version)							
ST_MapAlgebraExpr							
ST_MapAlgebraExpr							
ST_MapAlgebraFct							
ST_MapAlgebraFct							
ST_MapAlgebraFctNgb							
ST_Reclass							
ST_ReclassExact							
ST_Union							
ST_Distinct4ma							
ST_InvDistWeight4ma							
ST_Max4ma							
ST_Mean4ma							
ST_Min4ma							
ST_MinDist4ma							
ST_Range4ma							
ST_StdDev4ma							
ST_Sum4ma							
ST_Aspect							
ST_HillShade							
ST_Roughness							
ST_Slope							
ST_TPI							
ST_TRI							
ST_InterpolateR ✓ er							
ST_Contour	✓						
Box3D	☑						
ST_ConvexHull	✓						
ST_DumpAsPolygons							
ST_Envelope	✓						
ST_MinConvexH ✓							
ST_Polygon	✓						
ST_Intersection ✓ ctions							

Function	geom	geog	2.5D	Curves	SQL MM	PS	T
&&	✓						
&<							
&>							
=							
@	✓						
~=							
~	✓						
ST_Contains							
ST_ContainsProperly							
ST_Covers							
ST_CoveredBy							
ST_Disjoint							
ST_Intersects	✓						
ST_Overlaps							
ST_Touches							
ST_SameAlignment							
ST_NotSameAlignmentReason							
ST_Within							
ST_DWithin							
ST_DFullyWithin							

13.12 New, Enhanced or changed PostGIS Functions

13.12.1 PostGIS Functions new or enhanced in 3.7

The functions given below are PostGIS functions that were added or enhanced.

Functions new in PostGIS 3.7

- CG_ApproximateMedialAxis - Availability: 3.7.0 - projected parameter. Requires SFCGAL >= 2.3.0 for projected result; falls back to non-projected with a notice on older versions. □□□□□□□□□ □□□□□□.
- CG_GenerateFlatRoof - Availability: 3.7.0 - requires SFCGAL >= 2.3.0. Generate a flat (box) roof from a footprint polygon.
- CG_GenerateGableRoof - Availability: 3.7.0 - requires SFCGAL >= 2.3.0. Generate a gable roof from a footprint polygon.
- CG_GenerateHippedRoof - Availability: 3.7.0 - requires SFCGAL >= 2.3.0. Generate a hipped roof from a footprint polygon.
- CG_GenerateRoof - Availability: 3.7.0 - requires SFCGAL >= 2.3.0. Generate a roof of the requested type from a footprint polygon.
- CG_GenerateSkillionRoof - Availability: 3.7.0 - requires SFCGAL >= 2.3.0. Generate a skillion (single-slope) roof from a footprint polygon.
- CG_NurbsCurveApproximate - Availability: 3.7.0 - requires SFCGAL >= 2.3.0. Creates an approximating NURBS curve fitting data points within a tolerance
- CG_NurbsCurveDerivative - Availability: 3.7.0 - requires SFCGAL >= 2.3.0. Computes the derivative (tangent, curvature) of a NURBS curve at a given parameter

- CG_NurbsCurveInterpolate - Availability: 3.7.0 - requires SFCGAL >= 2.3.0. Creates an interpolating NURBS curve passing through all given data points
- CG_PolygonRepair - Availability: 3.7.0 - requires SFCGAL >= 2.3.0 and CGAL >= 6.0. Repair an invalid polygon or multipolygon.
- FindVertexSegmentPairsBelowDistance - Availability: 3.7 Find pairs of topology vertex/segment that are closer than tolerated distance
- ST_3DInterpolatePoint - Availability: 3.7.0 Returns the interpolated measure of a geometry closest to a point in 3D.
- ST_CatmullRomSmoothing - Availability: 3.7.0 Returns a smoothed version of a geometry, using the Catmull-Rom spline algorithm
- ST_ClusterRelateWin - Availability: 3.7.0 Window function that returns a cluster id for each input geometry, clustering input geometries into connected sets using the relate pattern to determine whether the geometries are connected.
- ST_ControlPoints - Availability: 3.7.0 Returns the control points of a NURBS curve as a MULTIPOINT geometry.
- ST_CoverageEdges - Availability: 3.7.0 Computes the unique edges of a polygonal coverage.
- ST_Degree - Availability: 3.7.0 Returns the polynomial degree of a NURBS curve.
- ST_Evaluate - Availability: 3.7.0 Evaluates a NURBS curve at a specific parameter value and returns the resulting point.
- ST_IsNurbsCurve - Availability: 3.7.0 Checks if a geometry is a NURBS curve.
- ST_IsSimpleDetail - Availability: 3.7.0 Report locations of self-intersection or self-tangency, if any.
- ST_Knots - Availability: 3.7.0 Returns the knot vector of a NURBS curve.
- ST_MSize - Availability: 3.7.0 Returns the M size of a geometry.
- ST_MakeLine - Availability: 3.7.0 - Support for MultiLineString input elements was introduced ⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_MakeNurbsCurve - Availability: 3.7.0 Creates a NURBS (Non-Uniform Rational B-Spline) curve from control points, optional weights, and optional knot vector.
- ST_MinimumSpanningTree - Availability: 3.7.0 Window function that returns a tree id for each input geometry, clustering input geometries into connected trees using the Minimum Spanning Tree algorithm.
- ST_NumControlPoints - Availability: 3.7.0 Returns the number of control points in a NURBS curve.
- ST_NurbsCurveIsRational - Availability: 3.7.0 Checks if a NURBS curve is rational (has weights).
- ST_NurbsToLineString - Availability: 3.7.0 Converts a NURBS curve to a LINESTRING by uniform sampling.
- ST_Weights - Availability: 3.7.0 Returns the weight array of a rational NURBS curve.
- ST_XSize - Availability: 3.7.0 Returns the X size of a 2D or 3D bounding box or a geometry.
- ST_YSize - Availability: 3.7.0 Returns the Y size of a 2D or 3D bounding box or a geometry.
- ST_ZSize - Availability: 3.7.0 Returns the Z size of a 2D or 3D bounding box or a geometry.

Functions enhanced in PostGIS 3.7

- ST_AsX3D - Enhanced: 3.7.0: Support for curve geometries. NURBSCURVE is emitted as X3D NurbsCurve; other curve types are linearized before output. □□□ X3D XML □□□□□□: ISO-IEC-19776-1.2-X3DEncodings-XML □□□□□□.
- ST_GMLToSQL - Enhanced: 3.7.0 support for GML ArcString, curved Ring, and CurvePolygon geometries was introduced. Returns a geometry from GML. Alias for ST_GeomFromGML.
- ST_GeomFromGML - Enhanced: 3.7.0 support for GML ArcString, curved Ring, and CurvePolygon geometries was introduced. □□□ GML □□□□□□□□ PostGIS □□□□□□□□□□.
- ST_LongestLine - Enhanced: 3.7.0 - support for a single geometry input. □□□□□□ 3 □□□□ (longest) □□□□□□□□.
- ST_MakePolygon - Enhanced: 3.7.0 - Support for curved input rings was introduced. Creates a Polygon or CurvePolygon from a shell and optional list of holes.
- ST_MaxDistance - Enhanced: 3.7.0 - support for a single geometry input. □□□□□□ 2 □□□□□ □□□□□□□□□□□□□.
- ST_Value - Enhanced: 3.7.0 resample accepts nearest-neighbor boundary options "nearest-UL", "nearest-UR", "nearest-LL", and "nearest-LR". □□ columnx, rowy □□□□□□□□□□, □□□□□ □□□□□□□□□□□□□□□□□□□□. □□□□□ 1 □□□□□□□, □□□□□□□□□□□ 1 □□□□□□□. exclude_nodata_value □□□□□□□□□□, nodata □□□□□□□□□□□□□□□□□□□□□□□□□□ □□□. exclude_nodata_value □□□□□□□□□□, □□□□□□□□□□□□□□□□□□□□.
- TopoGeo_AddLineString - Enhanced: 3.7.0 added support for limiting the number of new edges created in the topology. Adds a linestring to an existing topology using a tolerance and possibly splitting existing edges/faces.

Functions changed in PostGIS 3.7

- ST_AsGeoJSON - Changed: 3.7.0 added warning about duplicate keys Return a geometry or feature in GeoJSON format.
- toTopoGeom - Changed: 3.7.0 changes meaning of tolerance 0 to really mean 0 and -1 (new default) to request using the topology precision. Converts a simple Geometry into a topo geometry.

13.12.2 PostGIS Functions new or enhanced in 3.6

The functions given below are PostGIS functions that were added or enhanced.

Functions new in PostGIS 3.6

- CG_2DRotate - Availability: 3.6.0 - requires SFCGAL >= 2.0.0 Rotates a geometry by a given angle around a specified point in 2D.
- CG_3DAlphaWrapping - Availability: 3.6.0 - requires SFCGAL >= 2.1.0 Computes a 3D Alpha-wrapping strictly enclosing a geometry.
- CG_3DBuffer - Availability: 3.6.0 - requires SFCGAL >= 2.0.0 Computes a 3D buffer around a geometry.
- CG_3DRotate - Availability: 3.6.0 - requires SFCGAL >= 2.0.0 Rotates a geometry in 3D space around an axis vector.
- CG_3DScale - Availability: 3.6.0 - requires SFCGAL >= 2.0.0 Scales a geometry by separate factors along X, Y, and Z axes.
- CG_3DScaleAroundCenter - Availability: 3.6.0 - requires SFCGAL >= 2.0.0 Scales a geometry in 3D space around a specified center point.

- CG_3DTranslate - Availability: 3.6.0 - requires SFCGAL >= 2.0.0 Translates (moves) a geometry by given offsets in 3D space.
- CG_Rotate - Availability: 3.6.0 - requires SFCGAL >= 2.0.0 Rotates a geometry by a given angle around the origin (0,0).
- CG_RotateX - Availability: 3.6.0 - requires SFCGAL >= 2.0.0 Rotates a geometry around the X-axis by a given angle.
- CG_RotateY - Availability: 3.6.0 - requires SFCGAL >= 2.0.0 Rotates a geometry around the Y-axis by a given angle.
- CG_RotateZ - Availability: 3.6.0 - requires SFCGAL >= 2.0.0 Rotates a geometry around the Z-axis by a given angle.
- CG_Scale - Availability: 3.6.0 - requires SFCGAL >= 2.0.0 Scales a geometry uniformly in all dimensions by a given factor.
- CG_Simplify - Availability: 3.6.0 - requires SFCGAL >= 2.1.0 Reduces the complexity of a geometry while preserving essential features and Z/M values.
- CG_StraightSkeletonPartition - Availability: 3.6.0 - requires SFCGAL >= 2.0.0. Computes the straight skeleton partition of a polygon.
- CG_Translate - Availability: 3.6.0 - requires SFCGAL >= 2.0.0 Translates (moves) a geometry by given offsets in 2D space.
- FixCorruptTopoGeometryColumn - Availability: 3.6.1 Fixes topogeometry corruption caused by upgrade to postgis_topology 3.6.0 and higher
- MakeTopologyPrecise - Availability: 3.6.0 Snap topology vertices to precision grid.
- ST_AsRasterAgg - Availability: 3.6.0 Aggregate. Renders PostGIS geometries into a new raster.
- ST_CoverageClean - Availability: 3.6.0 - requires GEOS >= 3.14.0 Computes a clean (edge matched, non-overlapping, gap-cleared) polygonal coverage, given a non-clean input.
- ST_IntersectionFractions - Availability: 3.6.0 Requires GEOS 3.14 or higher. Calculates the fraction of each raster cell that is covered by a given geometry.
- ST_ReclassExact - Availability: 3.6.0 Creates a new raster composed of bands reclassified from original, using a 1:1 mapping from values in the original band to new values in the destination band.
- TotalTopologySize - Availability: 3.6.0 Total disk space used by the specified topology, including all indexes and TOAST data.
- UpgradeTopology - Availability: 3.6.0 Upgrades the specified topology to support large ids (int8) for topology and primitive ids.
- ValidateTopologyPrecision - Availability: 3.6.0 Returns non-precise vertices in the topology.
- postgis.gdal_cpl_debug - Availability: 3.6.0 A boolean configuration to turn logging of GDAL debug messages on and off.

Functions enhanced in PostGIS 3.6

- CreateTopology - Enhanced: 3.6.0 added topoid and usesLargeIDs parameters. Creates a new topology schema and registers it in the topology.topology table.

Functions changed in PostGIS 3.6

- ST_GeometryN - Changed: 3.6.0 TIN and PolyhedralSurface are treated as unitary geometries and are not decomposed into patches; use ST_PatchN to access faces. ST_Geometry □□□□□□□□□ □□□.
- ST_NumGeometries - Changed: 3.6.0 TIN and PolyhedralSurface are treated as unitary geometries and return 1; use ST_NumPatches to count faces. □□□□□□□□□□□□□□□□□□□□. □□□□ □□□□□□□□□□.

13.12.3 PostGIS Functions new or enhanced in 3.5

The functions given below are PostGIS functions that were added or enhanced.

Functions new in PostGIS 3.5

- CG_3DArea - Availability: 3.5.0 3 □□□□□□□□□□□□□□□□. □□□□□ 0 □□□□□□□□.
- CG_3DConvexHull - Availability: 3.5.0 □□□□□□□□□□□□□□□□.
- CG_3DDifference - Availability: 3.5.0 3 □□□□□□□□□□.
- CG_3DDistance - Availability: 3.5.0 Computes the minimum 3D distance between two geometries
- CG_3DIntersection - Availability: 3.5.0 3 □□□□□□□□□□.
- CG_3DIntersects - Availability: 3.5.0 Tests if two 3D geometries intersect
- CG_3DUnion - Availability: 3.5.0 Perform 3D union using postgis_sfcgal.
- CG_AlphaShape - Availability: 3.5.0 - requires SFCGAL >= 1.4.1. Computes an Alpha-shape enclosing a geometry
- CG_ApproxConvexPartition - Availability: 3.5.0 - requires SFCGAL >= 1.5.0. Computes approximal convex partition of the polygon geometry
- CG_ApproximateMedialAxis - Availability: 3.5.0 □□□□□□□□□□□□□□□□.
- CG_Area - Availability: 3.5.0 Calculates the area of a geometry
- CG_Difference - Availability: 3.5.0 Computes the geometric difference between two geometries
- CG_Distance - Availability: 3.5.0 Computes the minimum distance between two geometries
- CG_Extrude - Availability: 3.5.0 □□□□□□□□□□□□□□.
- CG_ExtrudeStraightSkeleton - Availability: 3.5.0 - requires SFCGAL >= 1.5.0. Straight Skeleton Extrusion
- CG_ForceLHR - Availability: 3.5.0 LHR(Left Hand Reverse; □□□□) □□□□□□□□.
- CG_GreeneApproxConvexPartition - Availability: 3.5.0 - requires SFCGAL >= 1.5.0. Computes approximal convex partition of the polygon geometry
- CG_Intersection - Availability: 3.5.0 Computes the intersection of two geometries
- CG_Intersects - Availability: 3.5.0 Tests if two geometries intersect (they have at least one point in common)
- CG_IsPlanar - Availability: 3.5.0 □□□□□□□□□□□□□□□□.
- CG_IsSolid - Availability: 3.5.0 □□□□□□□□□□□□□□□. □□□□□□□□□□□□□□□□□.
- CG_MakeSolid - Availability: 3.5.0 □□□□□□□□□□□. □□□□□□□□□□□□□□□□. □□□□□□ □□□□□□, □□□□□□□□□□□□□□□□□□ TIN □□□□□□□□.

- CG_MinkowskiSum - Availability: 3.5.0 □□□□□□□□□□□□□□.
- CG_OptimalAlphaShape - Availability: 3.5.0 - requires SFCGAL >= 1.4.1. Computes an Alpha-shape enclosing a geometry using an "optimal" alpha value.
- CG_OptimalConvexPartition - Availability: 3.5.0 - requires SFCGAL >= 1.5.0. Computes an optimal convex partition of the polygon geometry
- CG_Orientation - Availability: 3.5.0 □□□□□ (orientation) □□□□□□.
- CG_StraightSkeleton - Availability: 3.5.0 □□□□□□□□□□□ (straight skeleton) □□□□□□.
- CG_Tesselate - Availability: 3.5.0 □□□□□□□□□□□□□□□□□ (tessellation) □□□□□□□□ TIN □□ TIN □□□□□□□□□□.
- CG_Triangulate - Availability: 3.5.0 Triangulates a polygonal geometry
- CG_Union - Availability: 3.5.0 Computes the union of two geometries
- CG_Visibility - Availability: 3.5.0 - requires SFCGAL >= 1.5.0. Compute a visibility polygon from a point or a segment in a polygon geometry
- CG_Volume - Availability: 3.5.0 Computes the volume of a 3D geometry. Closed surfaces can have non-zero volume.
- CG_YMonotonePartition - Availability: 3.5.0 - requires SFCGAL >= 1.5.0. Computes y-monotone partition of the polygon geometry
- ST_HasM - Availability: 3.5.0 Checks if a geometry has an M (measure) dimension.
- ST_HasZ - Availability: 3.5.0 Checks if a geometry has a Z dimension.
- ST_RemoveIrrelevantPointsForView - Availability: 3.5.0 Removes points that are irrelevant for rendering a specific rectangular view of a geometry.
- ST_RemoveSmallParts - Availability: 3.5.0 Removes small parts (polygon rings or linestrings) of a geometry.
- TopoGeo_LoadGeometry - Availability: 3.5.0 Load a geometry into an existing topology, snapping and splitting as needed.

Functions enhanced in PostGIS 3.5

- ST_Clip - Enhanced: 3.5.0 - touched argument added. Returns the raster clipped by the input geometry. If band number is not specified, all bands are processed. If crop is not specified or TRUE, the output raster is cropped. If touched is set to TRUE, then touched pixels are included, otherwise only if the center of the pixel is in the geometry it is included.

Functions changed in PostGIS 3.5

- ST_AsGeoJSON - Changed: 3.5.0 allow specifying the column containing the feature id Return a geometry or feature in GeoJSON format.
- ST_DFullyWithin - Changed: 3.5.0 : the logic behind the function now uses a test of containment within a buffer, rather than the ST_MaxDistance algorithm. Results will differ from prior versions, but should be closer to user expectations. Tests if a geometry is entirely inside a distance of another

13.12.4 PostGIS Functions new or enhanced in 3.4

The functions given below are PostGIS functions that were added or enhanced.

Functions new in PostGIS 3.4

- PostGIS_GEOS_Compiled_Version - Availability: 3.4.0 Returns the version number of the GEOS library against which PostGIS was built.
- PostGIS_PROJ_Compiled_Version - Availability: 3.5.0 Returns the version number of the PROJ library against which PostGIS was built.
- RenameTopoGeometryColumn - Availability: 3.4.0 Renames a topogeometry column
- RenameTopology - Availability: 3.4.0 Renames a topology
- ST_ClusterIntersectingWin - Availability: 3.4.0 Window function that returns a cluster id for each input geometry, clustering input geometries into connected sets.
- ST_ClusterWithinWin - Availability: 3.4.0 Window function that returns a cluster id for each input geometry, clustering using separation distance.
- ST_CoverageInvalidEdges - Availability: 3.4.0 Window function that finds locations where polygons fail to form a valid coverage.
- ST_CoverageSimplify - Availability: 3.4.0 Window function that simplifies the edges of a polygonal coverage.
- ST_CoverageUnion - Availability: 3.4.0 - requires GEOS >= 3.8.0 Computes the union of a set of polygons forming a coverage by removing shared edges.
- ST_InverseTransformPipeline - Availability: 3.4.0 Return a new geometry with coordinates transformed to a different spatial reference system using the inverse of a defined coordinate transformation pipeline.
- ST_LargestEmptyCircle - Availability: 3.4.0. Computes the largest circle not overlapping a geometry.
- ST_LineExtend - Availability: 3.4.0 Returns a line extended forwards and backwards by specified distances.
- ST_TransformPipeline - Availability: 3.4.0 Return a new geometry with coordinates transformed to a different spatial reference system using a defined coordinate transformation pipeline.
- TopoElement - Availability: 3.4.0 Converts a topogeometry to a topoelement.
- postgis_srs - Availability: 3.4.0 Return a metadata record for the requested authority and srid.
- postgis_srs_all - Availability: 3.4.0 Return metadata records for every spatial reference system in the underlying Proj database.
- postgis_srs_codes - Availability: 3.4.0 Return the list of SRS codes associated with the given authority.
- postgis_srs_search - Availability: 3.4.0 Return metadata records for projected coordinate systems that have areas of usage that fully contain the bounds parameter.

Functions enhanced in PostGIS 3.4

- PostGIS_Full_Version - Enhanced: 3.4.0 now includes extra PROJ configurations NETWORK_ENABLED, URL_ENDPOINT and DATABASE_PATH of proj.db location Reports full PostGIS version and build configuration infos.

- PostGIS_PROJ_Version - Enhanced: 3.4.0 now includes NETWORK_ENABLED, URL_ENDPOINT and DATABASE_PATH of proj.db location Returns the version number of the PROJ4 library.
- ST_AsSVG - Enhanced: 3.4.0 to support all curve types Returns SVG path data for a geometry.
- ST_ClosestPoint - Enhanced: 3.4.0 - Support for geography. Returns the 2D point on g1 that is closest to g2. This is the first point of the shortest line from one geometry to the other.
- ST_LineSubstring - Enhanced: 3.4.0 - Support for geography was introduced. Returns the part of a line between two fractional locations.
- ST_Project - Enhanced: 3.4.0 Allow geometry arguments and two-point form omitting azimuth. Returns a point projected from a start point by a distance and bearing (azimuth).
- ST_Resample - Enhanced: 3.4.0 max and min resampling options added ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠.
- ST_Rescale - Enhanced: 3.4.0 max and min resampling options added Resample a raster by adjusting only its scale (or pixel size). New pixel values are computed using the NearestNeighbor (english or american spelling), Bilinear, Cubic, CubicSpline, Lanczos, Max or Min resampling algorithm. Default is NearestNeighbor.
- ST_ShortestLine - Enhanced: 3.4.0 - support for geography. ⊠⊠⊠⊠⊠⊠ 2 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

Functions changed in PostGIS 3.4

- PostGIS_Extensions_Upgrade - Changed: 3.4.0 to add target_version argument. Packages and upgrades PostGIS extensions (e.g. postgis_raster, postgis_topology, postgis_sfcgal) to given or latest version.

13.12.5 PostGIS Functions new or enhanced in 3.3

The functions given below are PostGIS functions that were added or enhanced.

Functions new in PostGIS 3.3

- RemoveUnusedPrimitives - Availability: 3.3.0 Removes topology primitives which not needed to define existing TopoGeometry objects.
- ST_AsMARC21 - Availability: 3.3.0 Returns geometry as a MARC21/XML record with a geographic datafield (034).
- ST_GeomFromMARC21 - Availability: 3.3.0, requires libxml2 2.6+ Takes MARC21/XML geographic data as input and returns a PostGIS geometry object.
- ST_Letters - Availability: 3.3.0 Returns the input letters rendered as geometry with a default start position at the origin and default text height of 100.
- ST_SimplifyPolygonHull - Availability: 3.3.0. Computes a simplified topology-preserving outer or inner hull of a polygonal geometry.
- ST_TriangulatePolygon - Availability: 3.3.0. Computes the constrained Delaunay triangulation of polygons
- postgis_sfcgal_full_version - Availability: 3.3.0 Returns the full version of SFCGAL in use including CGAL and Boost versions

Functions enhanced in PostGIS 3.3

- ST_ConcaveHull - Enhanced: 3.3.0, GEOS native implementation enabled for GEOS 3.11+ Computes a possibly concave geometry that contains all input geometry vertices
- ST_LineMerge - Enhanced: 3.3.0 accept a directed parameter. Return the lines formed by sewing together a MultiLineString.

Functions changed in PostGIS 3.3

- PostGIS_Extensions_Upgrade - Changed: 3.3.0 support for upgrades from any PostGIS version. Does not work on all systems. Packages and upgrades PostGIS extensions (e.g. postgis_raster, postgis_topology, postgis_sfcgal) to given or latest version.

13.12.6 PostGIS Functions new or enhanced in 3.2

The functions given below are PostGIS functions that were added or enhanced.

Functions new in PostGIS 3.2

- FindLayer - Availability: 3.2.0 Returns a topology.layer record by different means.
- FindTopology - Availability: 3.2.0 Returns a topology record by different means.
- GetFaceContainingPoint - Availability: 3.2.0 Finds the face containing a point.
- ST_AsFlatGeobuf - Availability: 3.2.0 Return a FlatGeobuf representation of a set of rows.
- ST_Contour - Availability: 3.2.0 Generates a set of vector contours from the provided raster band, using the GDAL contouring algorithm.
- ST_DumpSegments - Availability: 3.2.0 ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.
- ST_FromFlatGeobuf - Availability: 3.2.0 Reads FlatGeobuf data.
- ST_FromFlatGeobufToTable - Availability: 3.2.0 Creates a table based on the structure of FlatGeobuf data.
- ST_InterpolateRaster - Availability: 3.2.0 Interpolates a gridded surface based on an input set of 3-d points, using the X- and Y-values to position the points on the grid and the Z-value of the points as the surface elevation.
- ST_SRID - Availability: 3.2.0 Returns the spatial reference identifier for a topogeometry.
- ST_Scroll - Availability: 3.2.0 Change start point of a closed LineString.
- ST_SetM - Availability: 3.2.0 Returns a geometry with the same X/Y coordinates as the input geometry, and values from the raster copied into the M dimension using the requested resample algorithm.
- ST_SetZ - Availability: 3.2.0 Returns a geometry with the same X/Y coordinates as the input geometry, and values from the raster copied into the Z dimension using the requested resample algorithm.
- TopoGeom_addTopoGeom - Availability: 3.2 Adds element of a TopoGeometry to the definition of another TopoGeometry.
- ValidateTopologyRelation - Availability: 3.2.0 Returns info about invalid topology relation records
- postgis.gdal_vsi_options - Availability: 3.2.0 DB ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.

Functions enhanced in PostGIS 3.2

- GetFaceByPoint - Enhanced: 3.2.0 more efficient implementation and clearer contract, stops working with invalid topologies. Finds face intersecting a given point.

- ST_ClusterKMeans - Enhanced: 3.2.0 Support for max_radius Window function that returns a cluster id for each input geometry using the K-means algorithm.
- ST_MakeValid - Enhanced: 3.2.0, added algorithm options, 'linework' and 'structure' which requires GEOS >= 3.10.0. Attempts to make an invalid geometry valid without losing vertices.
- ST_PixelAsCentroid - □□□□: 2.1.0 □□□□ C □□□□□□□□□□. □□□□□□□□□□□□□□□ (□□□□□) □□□□□□□.
- ST_PixelAsCentroids - □□□□: 2.1.0 □□□□ C □□□□□□□□□□. □□□□□□□□□□□□□□□ (□□□□□) □□□□□□□□ X, Y □□□□□□□□□□□□□□. □□□□□□□□□□□□□□□□□□□□□□□□□□□□.
- ST_Point - Enhanced: 3.2.0 srid as an extra optional argument was added. Older installs require combining with ST_SetSRID to mark the srid on the geometry. Creates a Point with X, Y and SRID values.
- ST_PointM - Enhanced: 3.2.0 srid as an extra optional argument was added. Older installs require combining with ST_SetSRID to mark the srid on the geometry. Creates a Point with X, Y, M and SRID values.
- ST_PointZ - Enhanced: 3.2.0 srid as an extra optional argument was added. Older installs require combining with ST_SetSRID to mark the srid on the geometry. Creates a Point with X, Y, Z and SRID values.
- ST_PointZM - Enhanced: 3.2.0 srid as an extra optional argument was added. Older installs require combining with ST_SetSRID to mark the srid on the geometry. Creates a Point with X, Y, Z, M and SRID values.
- ST_RemovePoint - Enhanced: 3.2.0 Remove a point from a linestring.
- ST_RemoveRepeatedPoints - Enhanced: 3.2.0 Returns a version of a geometry with duplicate points removed.
- ST_StartPoint - Enhanced: 3.2.0 returns a point for all geometries. Prior behavior returns NULLs if input was not a LineString. Returns the first point of a LineString, CircularLineString, or NURBSCurve.
- ST_Value - □□□□: 2.1.0 □□□□□□□□ exclude_nodata_value □□□□□□□□□. □□ columnx, rowy □□□□□□□□□, □□□□□□□□□□□□□□□□□□□□□□□□. □□□□□ 1 □□□□□□□, □□ □□□□□□□□ 1 □□□□□□□. exclude_nodata_value □□□□□□□□□, nodata □□□□□□□□ □□□□□□□□□□□□□□□□□□□□. exclude_nodata_value □□□□□□□□□, □□□□□□□□□ □□□□□□□□□□.
- TopoGeo_AddLineString - Enhanced: 3.2.0 added support for returning signed identifier. Adds a linestring to an existing topology using a tolerance and possibly splitting existing edges/faces.

Functions changed in PostGIS 3.2

- ST_Boundary - Changed: 3.2.0 support for TIN, does not use geos, does not linearize curves □□□ □□□□□□□□□□□□□□□□□□□□.
- ValidateTopology - Changed: 3.2.0 added optional bbox parameter, perform face labeling and edge linking checks. Returns a set of validatetopology_returntype objects detailing issues with topology.

13.12.7 PostGIS Functions new or enhanced in 3.1

The functions given below are PostGIS functions that were added or enhanced.

Functions new in PostGIS 3.1

- ST_Hexagon - 2.1.0 □□□□□□□□□□□□□. Returns a single hexagon, using the provided edge size and cell coordinate within the hexagon grid space.
- ST_HexagonGrid - 2.1.0 □□□□□□□□□□□□□. Returns a set of hexagons and cell indices that completely cover the bounds of the geometry argument.
- ST_MaximumInscribedCircle - Availability: 3.1.0. □□□□□□□□□□□□□□□.
- ST_ReducePrecision - Availability: 3.1.0. Returns a valid geometry with points rounded to a grid tolerance.
- ST_Square - 2.1.0 □□□□□□□□□□□□□. Returns a single square, using the provided edge size and cell coordinate within the square grid space.
- ST_SquareGrid - 2.1.0 □□□□□□□□□□□□□. Returns a set of grid squares and cell indices that completely cover the bounds of the geometry argument.

Functions enhanced in PostGIS 3.1

- ST_AsEWKT - Enhanced: 3.1.0 support for optional precision parameter. □□□ WKT(Well-Known Text) □□□□ SRID □□□□□□□□□□□□□□.
- ST_ClusterKMeans - Enhanced: 3.1.0 Support for 3D geometries and weights Window function that returns a cluster id for each input geometry using the K-means algorithm.
- ST_Difference - Enhanced: 3.1.0 accept a gridSize parameter. Computes a geometry representing the part of geometry A that does not intersect geometry B.
- ST_Intersection - Enhanced: 3.1.0 accept a gridSize parameter Computes a geometry representing the shared portion of geometries A and B.
- ST_MakeValid - Enhanced: 3.1.0, added removal of Coordinates with NaN values. Attempts to make an invalid geometry valid without losing vertices.
- ST_Subdivide - Enhanced: 3.1.0 accept a gridSize parameter. Computes a rectilinear subdivision of a geometry.
- ST_SymDifference - Enhanced: 3.1.0 accept a gridSize parameter. Computes a geometry representing the portions of geometries A and B that do not intersect.
- ST_TileEnvelope - □□□□: 2.0.0 □□□□□□□□□□ SRID □□□□□□□□□□□. Creates a rectangular Polygon in Web Mercator (SRID:3857) using the XYZ tile system.
- ST_UnaryUnion - Enhanced: 3.1.0 accept a gridSize parameter. Computes the union of the components of a single geometry.
- ST_Union - Enhanced: 3.1.0 accept a gridSize parameter. Computes a geometry representing the point-set union of the input geometries.

Functions changed in PostGIS 3.1

- ST_Count - 2.2.0 □□□□□□□ ST_Count(rastertable, rastercolumn, ...) □□□□□□□□□□□□□. □□□□□□□□□□□□. □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□. □□□□□□□ □□□□□□□□□□□□ 1 □□□. exclude_nodata_value □□□□□□□□□, NODATA □□□□□□□ □□□□□□□□□□□.
- ST_Force3D - Changed: 3.1.0. Added support for supplying a non-zero Z value. □□□ XYZ □□□□ □□□□. ST_Force3DZ □□□□□□.
- ST_Force3DM - Changed: 3.1.0. Added support for supplying a non-zero M value. □□□ XYM □□ □□□□□□.

- ST_Force3DZ - Changed: 3.1.0. Added support for supplying a non-zero Z value. ⊠⊠⊠ XYZ ⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_Force4D - Changed: 3.1.0. Added support for supplying non-zero Z and M values. ⊠⊠⊠ XYZM ⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_Histogram - Changed: 3.1.0 Removed ST_Histogram(table_name, column_name) variant. ⊠ (bin; ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠) ⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_Quantile - Changed: 3.1.0 Removed ST_Quantile(table_name, column_name) variant. ⊠⊠⊠⊠⊠⊠⊠⊠ (population) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (quantile) ⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠, ⊠⊠⊠⊠ 25%, 50%, 75% ⊠⊠⊠ (percentile) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_SummaryStats - 2.2.0 ⊠⊠⊠⊠⊠⊠⊠ ST_SummaryStats(rastertable, rastercolumn, ...) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. Returns summarystats consisting of count, sum, mean, stddev, min, max for a given raster band of a raster or raster coverage. Band 1 is assumed if no band is specified.

13.12.8 PostGIS Functions new or enhanced in 3.0

The functions given below are PostGIS functions that were added or enhanced.

Functions new in PostGIS 3.0

- CG_ConstrainedDelaunayTriangles - 2.1.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. Return a constrained Delaunay triangulation around the given input geometry.
- ST_3DLineInterpolatePoint - 2.1.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. Returns a point interpolated along a 3D line at a fractional location.
- ST_TileEnvelope - 2.1.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. Creates a rectangular Polygon in Web Mercator (SRID:3857) using the XYZ tile system.

Functions enhanced in PostGIS 3.0

- ST_AsMVT - Enhanced: 3.0 - added support for Feature ID. Aggregate function returning a MVT representation of a set of rows.
- ST_Contains - Enhanced: 3.0.0 enabled support for GEOMETRYCOLLECTION Tests if every point of B lies in A, and their interiors have a point in common
- ST_ContainsProperly - Enhanced: 3.0.0 enabled support for GEOMETRYCOLLECTION Tests if every point of B lies in the interior of A
- ST_CoveredBy - Enhanced: 3.0.0 enabled support for GEOMETRYCOLLECTION Tests if every point of A lies in B
- ST_Covers - Enhanced: 3.0.0 enabled support for GEOMETRYCOLLECTION Tests if every point of B lies in A
- ST_Crosses - Enhanced: 3.0.0 enabled support for GEOMETRYCOLLECTION Tests if two geometries have some, but not all, interior points in common
- ST_CurveToLine - Enhanced: 3.0.0 implemented a minimum number of segments per linearized arc to prevent topological collapse. Converts a geometry containing curves to a linear geometry.
- ST_Disjoint - Enhanced: 3.0.0 enabled support for GEOMETRYCOLLECTION Tests if two geometries have no points in common

- ST_Equals - Enhanced: 3.0.0 enabled support for GEOMETRYCOLLECTION Tests if two geometries include the same set of points
- ST_GeneratePoints - Enhanced: 3.0.0, added seed parameter Generates a multipoint of random points contained in a Polygon or MultiPolygon.
- ST_GeomFromGeoJSON - Enhanced: 3.0.0 parsed geometry defaults to SRID=4326 if not specified otherwise. GeoJSON ☒☒☒☒☒☒☒☒ PostGIS ☒☒☒☒☒☒☒☒☒☒.
- ST_LocateBetween - Enhanced: 3.0.0 - added support for POLYGON, TIN, TRIANGLE. Returns the portions of a geometry that match a measure range.
- ST_LocateBetweenElevations - Enhanced: 3.0.0 - added support for POLYGON, TIN, TRIANGLE. Returns the portions of a geometry that lie in an elevation (Z) range.
- ST_Overlaps - Enhanced: 3.0.0 enabled support for GEOMETRYCOLLECTION Tests if two geometries have the same dimension and intersect, but each has at least one point not in the other
- ST_Relate - Enhanced: 3.0.0 enabled support for GEOMETRYCOLLECTION Tests if two geometries have a topological relationship matching an Intersection Matrix pattern, or computes their Intersection Matrix
- ST_Segmentize - Enhanced: 3.0.0 Segmentize geometry now produces equal-length subsegments Returns a modified geometry/geography having no segment longer than a given distance.
- ST_Touches - Enhanced: 3.0.0 enabled support for GEOMETRYCOLLECTION Tests if two geometries have at least one point in common, but their interiors do not intersect
- ST_Within - Enhanced: 3.0.0 enabled support for GEOMETRYCOLLECTION Tests if every point of A lies in B, and their interiors have a point in common

Functions changed in PostGIS 3.0

- PostGIS_Extensions_Upgrade - Changed: 3.0.0 to repackage loose extensions and support postgis_raster. Packages and upgrades PostGIS extensions (e.g. postgis_raster, postgis_topology, postgis_sfcgal) to given or latest version.
- ST_3DDistance - Changed: 3.0.0 - SFCGAL version removed ☒☒☒☒☒☒☒, ☒☒☒☒☒☒ (SRS ☒☒☒☒) 3 ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.
- ST_3DIntersects - Changed: 3.0.0 SFCGAL backend removed, GEOS backend supports TINs. Tests if two geometries spatially intersect in 3D - only for points, linestrings, polygons, polyhedral surface (area)
- ST_Area - Changed: 3.0.0 - does not depend on SFCGAL anymore. ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.
- ST_AsGeoJSON - Changed: 3.0.0 support records as input Return a geometry or feature in GeoJSON format.
- ST_AsGeoJSON - Changed: 3.0.0 output SRID if not EPSG:4326. Return a geometry or feature in GeoJSON format.
- ST_AsKML - Changed: 3.0.0 - Removed the ”versioned” variant signature ☒☒☒ GML 2 ☒☒ GML 3 ☒☒☒☒☒☒☒☒☒☒☒.
- ST_Distance - Changed: 3.0.0 - does not depend on SFCGAL anymore. ☒☒☒☒☒☒ 3 ☒☒☒☒ (longest) ☒☒☒☒☒☒☒☒.
- ST_Intersection - Changed: 3.0.0 does not depend on SFCGAL. Computes a geometry representing the shared portion of geometries A and B.
- ST_Intersects - Changed: 3.0.0 SFCGAL version removed and native support for 2D TINS added. Tests if two geometries intersect (they have at least one point in common)
- ST_Union - Changed: 3.0.0 does not depend on SFCGAL. Computes a geometry representing the point-set union of the input geometries.

13.12.9 PostGIS Functions new or enhanced in 2.5

The functions given below are PostGIS functions that were added or enhanced.

Functions new in PostGIS 2.5

- PostGIS_Extensions_Upgrade - Availability: 2.5.0 Packages and upgrades PostGIS extensions (e.g. postgis_raster, postgis_topology, postgis_sfcgal) to given or latest version.
- ST_Angle - Availability: 2.5.0 □□□□□□ 3 □□□□ (longest) □□□□□□□□.
- ST_AsHexWKB - Availability: 2.5.0 Return the Well-Known Binary (WKB) in Hex representation of the raster.
- ST_BandFileSize - Availability: 2.5.0 Returns the file size of a band stored in file system. If no bandnum specified, 1 is assumed.
- ST_BandFileTimestamp - Availability: 2.5.0 Returns the file timestamp of a band stored in file system. If no bandnum specified, 1 is assumed.
- ST_ChaikinSmoothing - Availability: 2.5.0 Returns a smoothed version of a geometry, using the Chaikin algorithm
- ST_FilterByM - Availability: 2.5.0 Removes vertices based on their M value
- ST_Grayscale - Availability: 2.5.0 Creates a new one-8BUI band raster from the source raster and specified bands representing Red, Green and Blue
- ST_LineInterpolatePoints - Availability: 2.5.0 Returns points interpolated along a line at a fractional interval.
- ST_OrientedEnvelope - Availability: 2.5.0. Returns a minimum-area rectangle containing a geometry.
- ST_QuantizeCoordinates - Availability: 2.5.0 Sets least significant bits of coordinates to zero
- ST_RastFromHexWKB - Availability: 2.5.0 Return a raster value from a Hex representation of Well-Known Binary (WKB) raster.
- ST_RastFromWKB - Availability: 2.5.0 Return a raster value from a Well-Known Binary (WKB) raster.
- ST_SetBandIndex - Availability: 2.5.0 Update the external band number of an out-db band
- ST_SetBandPath - Availability: 2.5.0 Update the external path and band number of an out-db band

Functions enhanced in PostGIS 2.5

- ST_AsBinary/ST_AsWKB - Enhanced: 2.5.0 Addition of ST_AsWKB Return the Well-Known Binary (WKB) representation of the raster.
- ST_AsMVT - Enhanced: 2.5.0 - added support parallel query. Aggregate function returning a MVT representation of a set of rows.
- ST_AsText - Enhanced: 2.5 - optional parameter precision introduced. □□/□□□□ WKT(Well-Known Text) □□□□ SRID □□□□□□□□□□□□.
- ST_BandMetaData - Enhanced: 2.5.0 to include outdbbandnum, filesize and filetimestamp for outdb rasters. □□□□□□□□□□□□□□□□□□□□□□□□□. □□□□□□□□□□□□□ 1 □□□□□□□□□□.
- ST_Buffer - Enhanced: 2.5.0 - ST_Buffer geometry support was enhanced to allow for side buffering specification side=both|left|right. Computes a geometry covering all points within a given distance from a geometry.

- ST_GeomFromGeoJSON - Enhanced: 2.5.0 can now accept json and jsonb as inputs. GeoJSON □□ □□□□□□ PostGIS □□□□□□□□□□.
- ST_GeometricMedian - Enhanced: 2.5.0 Added support for M as weight of points. □□□□□□□□ □□□□□ (median) □□□□□□.
- ST_Intersects - Enhanced: 2.5.0 Supports GEOMETRYCOLLECTION. Tests if two geometries intersect (they have at least one point in common)
- ST_OffsetCurve - Enhanced: 2.5 - added support for GEOMETRYCOLLECTION and MULTILINESTRING Returns an offset line at a given distance and side from an input line.
- ST_Scale - Enhanced: 2.5.0 support for scaling relative to a local origin (origin parameter) was introduced. Scales a geometry by given factors.
- ST_Split - Enhanced: 2.5.0 support for splitting a polygon by a multiline was introduced. Returns a collection of geometries created by splitting a geometry by another geometry.
- ST_Subdivide - Enhanced: 2.5.0 reuses existing points on polygon split, vertex count is lowered from 8 to 5. Computes a rectilinear subdivision of a geometry.

Functions changed in PostGIS 2.5

- ST_GDALDrivers - Changed: 2.5.0 - add can_read and can_write columns. Returns a list of raster formats supported by PostGIS through GDAL. Only those formats with can_write=True can be used by ST_AsGDALRaster

13.12.10 PostGIS Functions new or enhanced in 2.4

The functions given below are PostGIS functions that were added or enhanced.

Functions new in PostGIS 2.4

- ST_AsGeobuf - 2.2.0 □□□□□□□□□□□□. Return a Geobuf representation of a set of rows.
- ST_AsMVT - 2.2.0 □□□□□□□□□□□□. Aggregate function returning a MVT representation of a set of rows.
- ST_AsMVTGeom - 2.2.0 □□□□□□□□□□□□. Transforms a geometry into the coordinate space of a MVT tile.
- ST_Centroid - Availability: 2.4.0 support for geography was introduced. □□□□□□□□□□□□ □□.
- ST_ForcePolygonCCW - 2.2.0 □□□□□□□□□□□□. Orients all exterior rings counter-clockwise and all interior rings clockwise.
- ST_ForcePolygonCW - 2.2.0 □□□□□□□□□□□□. Orients all exterior rings clockwise and all interior rings counter-clockwise.
- ST_FrechetDistance - Availability: 2.4.0 - requires GEOS >= 3.7.0 □□□□□□ 3 □□□□ (shortest) □□□□□□□□.
- ST_IsPolygonCCW - 2.2.0 □□□□□□□□□□□□. Tests if Polygons have exterior rings oriented counter-clockwise and interior rings oriented clockwise.
- ST_IsPolygonCW - 2.2.0 □□□□□□□□□□□□. Tests if Polygons have exterior rings oriented clockwise and interior rings oriented counter-clockwise.
- ST_MakeEmptyCoverage - 2.2.0 □□□□□□□□□□□□. Cover georeferenced area with a grid of empty raster tiles.

Functions enhanced in PostGIS 2.4

- ST_AsTWKB - Enhanced: 2.4.0 memory and speed improvements. ☒☒☒ TWKB(Tiny Well-Known Binary) ☒☒☒☒☒☒.
- ST_Covers - Enhanced: 2.4.0 Support for polygon in polygon and line in polygon added for geography type Tests if every point of B lies in A
- ST_CurveToLine - Enhanced: 2.4.0 added support for max-deviation and max-angle tolerance, and for symmetric output. Converts a geometry containing curves to a linear geometry.
- ST_Project - Enhanced: 2.4.0 Allow negative distance and non-normalized azimuth. Returns a point projected from a start point by a distance and bearing (azimuth).
- ST_Reverse - Enhanced: 2.4.0 support for curves was introduced. ☒☒☒☒☒☒

- @(box2df,geometry) - Availability: 2.3.0 support for Block Range INdexes (BRIN) was introduced. Returns TRUE if a 2D float precision bounding box (BOX2DF) is contained into a geometry's 2D bounding box.
- @(geometry,box2df) - Availability: 2.3.0 support for Block Range INdexes (BRIN) was introduced. Returns TRUE if a geometry's 2D bounding box is contained into a 2D float precision bounding box (BOX2DF).
- Populate_Topology_Layer - 2.3.0 □□□□□□□□□□□□. Adds missing entries to topology.layer table by reading metadata from topo tables.
- ST_ClusterDBSCAN - 2.3.0 □□□□□□□□□□□□. Window function that returns a cluster id for each input geometry using the DBSCAN algorithm.
- ST_ClusterKMeans - 2.3.0 □□□□□□□□□□□□. Window function that returns a cluster id for each input geometry using the K-means algorithm.
- ST_GeneratePoints - 2.3.0 □□□□□□□□□□□□. Generates a multipoint of random points contained in a Polygon or MultiPolygon.
- ST_GeometricMedian - 2.3.0 □□□□□□□□□□□□. □□□□□□□□□□□□□□ (median) □□□□□□.
- ST_MakeLine - 2.0.0 □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□. □□□, □□□□□□□□□□ □□□□□□□□□□□□□□□□□□□□□□.
- ST_MinimumBoundingRadius - 2.3.0 □□□□□□□□□□□□. Returns the center point and radius of the smallest circle that contains a geometry.
- ST_MinimumClearance - 2.3.0 □□□□□□□□□□□□. □□□□□□□ (robustness) □□□□□□□□ □□□□ (clearance) □□□□□□□.
- ST_MinimumClearanceLine - 2.3.0 □□□□□□□□□□□□. GEOS 3.6.0 □□□□□□□□□□. □□ □ 2 □□□□□□□, □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□.
- ST_Normalize - 2.3.0 □□□□□□□□□□□□□. □□□□□□□□□□□□□□□□□□□□□□.
- ST_Points - 2.3.0 □□□□□□□□□□□□□. □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□.
- ST_VoronoiLines - 2.3.0 □□□□□□□□□□□□□. Returns the boundaries of the Voronoi diagram of the vertices of a geometry.
- ST_VoronoiPolygons - 2.3.0 □□□□□□□□□□□□□. Returns the cells of the Voronoi diagram of the vertices of a geometry.
- ST_WrapX - Availability: 2.3.0 requires GEOS X □□□□□□□□□□□□□□.
- TopoGeom_addElement - 2.3 □□□□□□□□□□□□□. Adds an element to the definition of a TopoGeometry.
- TopoGeom_remElement - 2.3 □□□□□□□□□□□□□. Removes an element from the definition of a TopoGeometry.
- ~(box2df,box2df) - Availability: 2.3.0 support for Block Range INdexes (BRIN) was introduced. Returns TRUE if a 2D float precision bounding box (BOX2DF) contains another 2D float precision bounding box (BOX2DF).
- ~(box2df,geometry) - Availability: 2.3.0 support for Block Range INdexes (BRIN) was introduced. Returns TRUE if a 2D float precision bounding box (BOX2DF) contains a geometry's 2D bonding box.
- ~(geometry,box2df) - Availability: 2.3.0 support for Block Range INdexes (BRIN) was introduced. Returns TRUE if a geometry's 2D bonding box contains a 2D float precision bounding box (GIDX).

Functions enhanced in PostGIS 2.3

- ST_Contains - Enhanced: 2.3.0 Enhancement to PIP short-circuit extended to support MultiPoints with few points. Prior versions only supported point in polygon. Tests if every point of B lies in A, and their interiors have a point in common
- ST_Covers - Enhanced: 2.3.0 Enhancement to PIP short-circuit for geometry extended to support MultiPoints with few points. Prior versions only supported point in polygon. Tests if every point of B lies in A
- ST_Expand - Enhanced: 2.3.0 support was added to expand a box by different amounts in different dimensions. Returns a bounding box expanded from another bounding box or a geometry.
- ST_Intersects - Enhanced: 2.3.0 Enhancement to PIP short-circuit extended to support MultiPoints with few points. Prior versions only supported point in polygon. Tests if two geometries intersect (they have at least one point in common)
- ST_Segmentize - Enhanced: 2.3.0 Segmentize geography now produces equal-length subsegments Returns a modified geometry/geography having no segment longer than a given distance.
- ST_Transform - Enhanced: 2.3.0 support for direct PROJ.4 text was introduced. Return a new geometry with coordinates transformed to a different spatial reference system.
- ST_Within - Enhanced: 2.3.0 Enhancement to PIP short-circuit for geometry extended to support MultiPoints with few points. Prior versions only supported point in polygon. Tests if every point of A lies in B, and their interiors have a point in common

Functions changed in PostGIS 2.3

- ST_PointN - ⊠⊠⊠⊠: 2.3.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (-1 ⊠⊠⊠⊠⊠⊠⊠⊠) ⊠⊠⊠⊠⊠⊠⊠⊠⊠. ST_LineString ⊠⊠ ST_CircularString ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

13.12.12 PostGIS Functions new or enhanced in 2.2

The functions given below are PostGIS functions that were added or enhanced.

Functions new in PostGIS 2.2

- <<->> - Availability: 2.2.0. Returns the n-D distance between the A and B geometries or bounding boxes
- ST_AsEncodedPolyline - 2.2.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠.
- ST_AsTWKB - 2.2.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠ TWKB(Tiny Well-Known Binary) ⊠⊠⊠⊠⊠⊠⊠.
- ST_BoundingDiagonal - 2.2.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_CPAWithin - 2.2.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. Tests if the closest point of approach of two trajectories is within the specified distance.
- ST_ClipByBox2D - 2.2.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. Computes the portion of a geometry falling within a rectangle.
- ST_ClosestPointOfApproach - 2.2.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. Returns a measure at the closest point of approach of two trajectories.
- ST_ClusterIntersecting - 2.2.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. Aggregate function that clusters input geometries into connected sets.

- ST_ClusterWithin - 2.2.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. Aggregate function that clusters geometries by separation distance.
- ST_CountAgg - 2.2.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ 1 ⊠⊠⊠. exclude_nodata_value ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, NODATA ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_CreateOverview - 2.2.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_DistanceCPA - 2.2.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. Returns the distance between the closest point of approach of two trajectories.
- ST_ForceCurve - 2.2.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (upcast) ⊠⊠⊠.
- ST_IsValidTrajectory - 2.2.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. Tests if the geometry is a valid trajectory.
- ST_LineFromEncodedPolyline - 2.2.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠ (polyline) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_MapAlgebra (callback function version) - 2.2.0 ⊠⊠⊠⊠ mask ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠ - ⊠⊠⊠ 1 ⊠⊠⊠, ⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ 1 ⊠⊠⊠⊠⊠⊠⊠⊠⊠ 1 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_MemSize - 2.2.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (⊠⊠⊠⊠⊠⊠⊠) ⊠⊠⊠⊠⊠⊠.
- ST_RemoveRepeatedPoints - 2.2.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. Returns a version of a geometry with duplicate points removed.
- ST_Retile - 2.2.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_SetEffectiveArea - 2.2.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. Sets the effective area for each vertex, using the Visvalingam-Whyatt algorithm.
- ST_SimplifyVW - 2.2.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. Returns a simplified representation of a geometry, using the Visvalingam-Whyatt algorithm
- ST_Subdivide - 2.2.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. Computes a rectilinear subdivision of a geometry.
- ST_SummaryStatsAgg - 2.2.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. Aggregate. Returns summarystats consisting of count, sum, mean, stddev, min, max for a given raster band of a set of raster. Band 1 is assumed if no band is specified.
- ST_SwapOrdinates - 2.2.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- postgis.enable_outdb_rasters - 2.2.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. DB ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- postgis.gdal_datapath - 2.2.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. GDAL ⊠ GDAL_DATA ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠ GDAL_DATA ⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- postgis.gdal_enabled_drivers - 2.2.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. PostGIS ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ GDAL ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. GDAL ⊠⊠⊠⊠ GDAL_SKIP ⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- |=| - Availability: 2.2.0. A ⊠ B ⊠⊠⊠⊠⊠⊠⊠⊠ (closest point of approach) ⊠⊠⊠⊠⊠ (trajectory) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

Functions enhanced in PostGIS 2.2

- <-> - Enhanced: 2.2.0 -- True KNN (”K nearest neighbor”) behavior for geometry and geography. Note for geography KNN is based on sphere rather than spheroid. A ⊠ B ⊠⊠⊠ 2 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

- AsTopoJSON - ⊠⊠⊠⊠: 2.2.1 ⊠⊠⊠⊠⊠⊠ (puntal) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. TopoGeometry ⊠ TopoJSON ⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_Area - ⊠⊠⊠⊠: 2.2.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ GeographicLib ⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ Proj 4.9.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_AsX3D - ⊠⊠⊠⊠: 2.2.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (x/y, ⊠⊠/⊠⊠) ⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠ X3D XML ⊠⊠⊠⊠⊠⊠: ISO-IEC-19776-1.2-X3DEncodings-XML ⊠⊠⊠⊠⊠⊠.
- ST_Azimuth - ⊠⊠⊠⊠: 2.2.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ GeographicLib ⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ Proj 4.9.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠ 2 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_Distance - ⊠⊠⊠⊠: 2.2.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ GeographicLib ⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ Proj 4.9.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠ 3 ⊠⊠⊠⊠ (longest) ⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_Scale - Enhanced: 2.2.0 support for scaling all dimension (factor parameter) was introduced. Scales a geometry by given factors.
- ST_Split - Enhanced: 2.2.0 support for splitting a line by a multiline, a multipoint or (multi)polygon boundary was introduced. Returns a collection of geometries created by splitting a geometry by another geometry.
- ST_Summary - ⊠⊠⊠⊠: 2.2.0 ⊠⊠⊠⊠ TIN ⊠⊠⊠⊠⊠ (curve) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

Functions changed in PostGIS 2.2

- <-> - Changed: 2.2.0 -- Old hybrid-syntax workarounds may be slower once true KNN is available. See examples below. A ⊠ B ⊠⊠⊠ 2 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_3DClosestPoint - ⊠⊠⊠⊠: 2.2.0 ⊠⊠⊠⊠ 2D ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, (⊠⊠⊠⊠⊠⊠ Z ⊠ 0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠) 2D ⊠⊠⊠⊠⊠⊠⊠⊠⊠. 2D ⊠ 3D ⊠⊠⊠, ⊠⊠⊠ Z ⊠⊠⊠⊠ Z ⊠ 0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. g2 ⊠⊠⊠⊠⊠⊠ g1 ⊠⊠⊠⊠ 3 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠ 3D ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_3DDistance - ⊠⊠⊠⊠: 2.2.0 ⊠⊠⊠⊠, 2D ⊠ 3D ⊠⊠⊠⊠⊠⊠ Z ⊠⊠⊠⊠ Z ⊠ 0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠ (SRS ⊠⊠⊠⊠) 3 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_3DLongestLine - ⊠⊠⊠⊠: 2.2.0 ⊠⊠⊠⊠ 2D ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, (⊠⊠⊠⊠⊠⊠ Z ⊠ 0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠) 2D ⊠⊠⊠⊠⊠⊠⊠⊠⊠. 2D ⊠ 3D ⊠⊠⊠, ⊠⊠⊠ Z ⊠⊠⊠⊠ Z ⊠ 0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠ 3 ⊠⊠⊠⊠ (longest) ⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_3DMaxDistance - ⊠⊠⊠⊠: 2.2.0 ⊠⊠⊠⊠, 2D ⊠ 3D ⊠⊠⊠⊠⊠⊠ Z ⊠⊠⊠⊠ Z ⊠ 0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠ (SRS ⊠⊠⊠⊠) 3 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_3DShortestLine - ⊠⊠⊠⊠: 2.2.0 ⊠⊠⊠⊠ 2D ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, (⊠⊠⊠⊠⊠⊠ Z ⊠ 0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠) 2D ⊠⊠⊠⊠⊠⊠⊠⊠⊠. 2D ⊠ 3D ⊠⊠⊠, ⊠⊠⊠ Z ⊠⊠⊠⊠ Z ⊠ 0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠ 3 ⊠⊠⊠⊠ (shortest) ⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_DistanceSphere - ⊠⊠⊠⊠: 2.2.0 ⊠⊠⊠⊠⊠⊠⊠⊠ ST_Distance_Sphere ⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. PostGIS 1.5 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_DistanceSpheroid - ⊠⊠⊠⊠: 2.2.0 ⊠⊠⊠⊠⊠⊠⊠⊠ ST_Distance_Spheroid ⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. PostGIS 1.5 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_Equals - Changed: 2.2.0 Returns true even for invalid geometries if they are binary equal Tests if two geometries include the same set of points
- ST_LengthSpheroid - ⊠⊠⊠⊠: 2.2.0 ⊠⊠⊠⊠⊠⊠⊠⊠ ST_Length_Spheroid ⊠⊠⊠⊠⊠⊠⊠⊠, ST_3DLength_Sp ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

- ST_MemSize - Changed: 2.2.0 name changed to ST_MemSize to follow naming convention. ST_Geometry ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_PointInsideCircle - Changed: 2.2.0 In prior versions this was called ST_Point_Inside_Circle Tests if a point geometry is inside a circle defined by a center and radius
- ValidateTopology - ⊠⊠⊠⊠: 2.2.0 ⊠⊠⊠⊠’edge crosses node’ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ id1 ⊠ id2 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. Returns a set of validatetopology_returntype objects detailing issues with topology.

13.12.13 PostGIS Functions new or enhanced in 2.1

The functions given below are PostGIS functions that were added or enhanced.

Functions new in PostGIS 2.1

- = - 2.1.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. A ⊠⊠⊠⊠⊠⊠ B ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ TRUE ⊠⊠⊠⊠⊠⊠. ⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- AsTopoJSON - 2.1.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. TopoGeometry ⊠ TopoJSON ⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_Box2dFromGeoHash - 2.1.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. GeoHash ⊠⊠⊠⊠⊠⊠⊠ BOX2D ⊠⊠⊠⊠⊠ ⊠.
- ST_ColorMap - 2.1.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ 8BUI ⊠⊠ (grayscale, RGB, RGBA) ⊠ 4 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ 1 ⊠⊠⊠⊠⊠⊠.
- ST_Contains - 2.1.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠ rastA ⊠⊠⊠⊠⊠⊠⊠⊠ rastB ⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠, ⊠⊠⊠ rastB ⊠⊠⊠⊠⊠⊠⊠⊠⊠ rastA ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_ContainsProperly - 2.1.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. rastB ⊠ rastA ⊠⊠⊠⊠⊠⊠⊠⊠⊠ rastA ⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_CoveredBy - 2.1.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠ rastA ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ rastB ⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_Covers - 2.1.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠ rastB ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ rastA ⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠.
- ST_DFullyWithin - 2.1.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠ rastA ⊠⊠⊠⊠ rastB ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_DWithin - 2.1.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠ rastA ⊠⊠⊠⊠ rastB ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠.
- ST_DelaunayTriangles - 2.1.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. Returns the Delaunay triangulation of the vertices of a geometry.
- ST_Disjoint - 2.1.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠ rastA ⊠⊠⊠⊠ rastB ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠.
- ST_DumpValues - 2.1.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠ 2 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_ForceSFS - 2.1.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. Force geometries to use SFS 1.1 or 1.2 geometry types.
- ST_FromGDALRaster - 2.1.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠ GDAL ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠.
- ST_GeomFromGeoHash - 2.1.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. GeoHash ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_InvDistWeight4ma - 2.1.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠.

- ST_MapAlgebra (callback function version) - 2.1.0 ☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒ - ☒☒☒ 1 ☒☒☒, ☒☒☒☒☒, ☒☒☒☒☒☒☒☒☒☒☒☒ 1 ☒☒☒☒☒☒☒☒ 1 ☒☒☒☒☒☒☒☒☒☒☒☒☒.
- ST_MapAlgebra (expression version) - 2.1.0 ☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒ - ☒☒☒☒☒ 1 ☒☒ ☒ 2 ☒, ☒☒☒☒☒, ☒☒☒☒☒☒☒☒ SQL ☒☒☒ 1 ☒☒☒☒☒☒☒☒☒☒ 1 ☒☒☒☒☒☒☒☒☒☒☒☒☒.
- ST_MinConvexHull - 2.1.0 ☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒ NODATA ☒☒☒☒☒☒☒☒☒☒☒☒☒☒ ☒☒☒☒.
- ST_MinDist4ma - 2.1.0 ☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒ (☒☒☒☒☒) ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.
- ST_NearestValue - 2.1.0 ☒☒☒☒☒☒☒☒☒☒☒☒. columnx ☒ rowy, ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒ ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒ NODATA ☒☒☒☒☒☒☒☒☒☒.
- ST_Neighborhood - 2.1.0 ☒☒☒☒☒☒☒☒☒☒☒☒. columnx ☒ rowy, ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒ ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒ NODATA ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒ 2 ☒☒☒☒☒☒☒☒☒☒.
- ST_NotSameAlignmentReason - 2.1.0 ☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒☒☒, ☒☒☒ ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.
- ST_Overlaps - 2.1.0 ☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒ rastA ☒☒☒☒ rastB ☒☒☒☒☒☒☒☒☒☒☒☒☒ ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.
- ST_PixelAsCentroid - 2.1.0 ☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒☒☒☒ (☒☒☒☒☒) ☒☒☒ ☒☒☒.
- ST_PixelAsCentroids - 2.1.0 ☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒ (☒☒☒☒☒) ☒ ☒☒☒☒☒☒ X, Y ☒☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.
- ST_PixelAsPoint - 2.1.0 ☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.
- ST_PixelAsPoints - 2.1.0 ☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒ X, Y ☒☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.
- ST_PixelOfValue - 2.1.0 ☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒ columnx, rowy ☒☒ ☒☒☒☒☒☒.
- ST_PointFromGeoHash - 2.1.0 ☒☒☒☒☒☒☒☒☒☒☒☒. GeoHash ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.
- ST_RasterToWorldCoord - 2.1.0 ☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒ X, Y(☒☒, ☒☒) ☒☒☒☒☒☒☒☒. ☒☒☒☒ 1 ☒☒☒☒☒☒☒.
- ST_Resize - 2.1.0 ☒☒☒☒☒☒☒☒☒☒☒☒. GDAL 1.6.1 ☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒/☒ ☒☒☒☒☒☒☒.
- ST_Roughness - 2.1.0 ☒☒☒☒☒☒☒☒☒☒☒☒. DEM ☒☒☒☒" ☒☒☒ (roughness)" ☒☒☒☒☒☒☒☒ ☒☒☒☒.
- ST_SetValues - 2.1.0 ☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒ ☒.
- ST_Simplify - 2.1.0 ☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒-☒☒ (Douglas-Peucker) ☒☒☒☒☒☒☒☒☒ TopoGeometry ☒" ☒☒☒☒" ☒☒☒☒☒☒☒☒☒☒.
- ST_Summary - 2.1.0 ☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.
- ST_TPI - 2.1.0 ☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒ (Topographic Position Index) ☒☒☒☒☒ ☒☒☒☒☒☒☒.
- ST_TRI - 2.1.0 ☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒ (Terrain Ruggedness Index) ☒☒☒☒☒ ☒☒☒☒☒☒☒.

- ST_Tile - 2.1.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_Touches - 2.1.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠ rastA ⊠⊠⊠⊠ rastB ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ TRUE ⊠⊠⊠⊠⊠⊠⊠.
- ST_Union - 2.1.0 ⊠⊠⊠⊠ ST_Union(rast, unionarg) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠ 1 ⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_Within - 2.1.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠ rastB ⊠⊠⊠⊠⊠⊠⊠⊠ rastA ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠ rastA ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ rastB ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_WorldToRasterCoord - 2.1.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠ X, Y(⊠⊠, ⊠⊠) ⊠⊠⊠⊠⊠⊠ ⊠⊠⊠.
- UpdateRasterSRID - 2.1.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ SRID ⊠⊠⊠⊠⊠⊠⊠.
- clearTopoGeom - 2.1 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. Clears the content of a topo geometry.
- postgis_sfcgal_version - 2.1.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠ SFCGAL ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

Functions enhanced in PostGIS 2.1

- ST_AddBand - ⊠⊠⊠⊠: 2.1.0 ⊠⊠⊠⊠ addbandarg ⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (⊠) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠.
- ST_AddBand - ⊠⊠⊠⊠: 2.1.0 ⊠⊠⊠⊠⊠⊠⊠⊠ DB ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (⊠) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠.
- ST_AsBinary/ST_AsWKB - ⊠⊠⊠⊠: 2.1.0 ⊠⊠⊠⊠ outasin ⊠⊠⊠⊠⊠⊠⊠. Return the Well-Known Binary (WKB) representation of the raster.
- ST_AsGML - ⊠⊠⊠⊠: 2.1.0 ⊠⊠⊠⊠ GML 3 ⊠⊠⊠ ID ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠ GML 2 ⊠⊠ GML 3 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_Aspect - ⊠⊠⊠⊠: 2.1.0 ⊠⊠⊠⊠ ST_MapAlgebra() ⊠⊠⊠⊠⊠, ⊠⊠⊠⊠ interpolate_nodata ⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (⊠⊠⊠⊠⊠⊠⊠) ⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_Boundary - ⊠⊠⊠⊠: 2.1.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠.
- ST_Clip - ⊠⊠⊠⊠: 2.1.0 ⊠⊠⊠⊠ C ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. Returns the raster clipped by the input geometry. If band number is not specified, all bands are processed. If crop is not specified or TRUE, the output raster is cropped. If touched is set to TRUE, then touched pixels are included, otherwise only if the center of the pixel is in the geometry it is included.
- ST_DWithin - Enhanced: 2.1.0 improved speed for geography. See Making Geography faster for details. Tests if two geometries are within a given distance
- ST_DWithin - Enhanced: 2.1.0 support for curved geometries was introduced. Tests if two geometries are within a given distance
- ST_Distance - Enhanced: 2.1.0 improved speed for geography. See Making Geography faster for details. ⊠⊠⊠⊠⊠⊠⊠ 3 ⊠⊠⊠⊠ (longest) ⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_Distance - ⊠⊠⊠⊠: 2.1.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠ 3 ⊠⊠⊠⊠ (longest) ⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_Distinct4ma - ⊠⊠⊠⊠: 2.1.0 ⊠⊠⊠⊠⊠⊠⊠ 2 ⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

- ST_DumpPoints - Enhanced: 2.1.0 Faster speed. Reimplemented as native-C. ☒☒☒☒☒☒☒☒☒☒☒ ☒☒☒☒☒☒☒☒☒.
- ST_HillShade - ☒☒☒☒: 2.1.0 ☒☒☒☒ ST_MapAlgebra() ☒☒☒☒☒, ☒☒☒☒ interpolate_nodata ☒ ☒☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒, ☒☒☒, ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒ ☒☒☒☒☒☒.
- ST_MakeValid - Enhanced: 2.1.0, added support for GEOMETRYCOLLECTION and MULTIPOINT. Attempts to make an invalid geometry valid without losing vertices.
- ST_Max4ma - ☒☒☒☒: 2.1.0 ☒☒☒☒☒☒☒ 2 ☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒ ☒☒☒☒☒☒☒.
- ST_Mean4ma - ☒☒☒☒: 2.1.0 ☒☒☒☒☒☒☒ 2 ☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒ ☒☒☒☒☒☒☒☒.
- ST_Min4ma - ☒☒☒☒: 2.1.0 ☒☒☒☒☒☒☒ 2 ☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒ ☒☒☒☒☒☒☒.
- ST_PixelAsPolygons - ☒☒☒☒: 2.1.0 ☒☒☒☒☒☒☒☒☒ exclude_nodata_value ☒☒☒☒☒☒☒☒☒☒. ☒☒ ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒ X, Y ☒☒☒☒☒☒☒☒☒☒☒☒☒☒.
- ST_Polygon - ☒☒☒☒: 2.1.0 ☒☒☒☒☒☒☒☒☒☒☒☒☒☒ (☒☒☒ C ☒☒☒☒☒☒☒☒☒☒☒☒☒☒). ☒☒☒☒ ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒. NODATA ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒ ☒☒☒☒☒☒☒☒☒.
- ST_Range4ma - ☒☒☒☒: 2.1.0 ☒☒☒☒☒☒☒ 2 ☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒ ☒☒☒☒☒☒☒☒☒☒☒.
- ST_SameAlignment - ☒☒☒☒: 2.1.0 ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒☒☒, ☒☒, ☒☒☒☒☒☒☒☒, ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒ (☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒ ☒) ☒☒☒☒☒☒☒, ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.
- ST_Segmentize - ☒☒☒☒: 2.1.0 ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒. Returns a modified geometry/geography having no segment longer than a given distance.
- ST_SetGeoReference - ☒☒☒☒: 2.1.0 ☒☒☒☒ ST_SetGeoReference(raster, double precision, ...) ☒ ☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒ 6 ☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒. GDAL ☒ ☒ ESRI ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒ GDAL ☒☒☒.
- ST_SetValue - ☒☒☒☒: 2.1.0 ☒☒☒☒ ST_SetValue() ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒ ☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒ ST_SetValues() ☒ geomval[] ☒☒☒☒☒☒☒☒☒ (wrapper) ☒☒☒. ☒☒☒ columnx, rowy ☒☒☒. ☒☒☒☒☒ 1 ☒☒☒☒☒☒☒, ☒☒☒☒☒☒☒☒☒☒☒ 1 ☒☒☒☒☒☒☒.
- ST_Slope - ☒☒☒☒: 2.1.0 ☒☒☒☒ ST_MapAlgebra() ☒☒☒☒☒☒, ☒☒☒☒ units, scale, interpolate_nodata ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒☒ (☒☒☒☒☒☒☒☒) ☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒ ☒☒☒☒☒☒☒☒.
- ST_StdDev4ma - ☒☒☒☒: 2.1.0 ☒☒☒☒☒☒☒ 2 ☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒ ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.
- ST_Sum4ma - ☒☒☒☒: 2.1.0 ☒☒☒☒☒☒☒ 2 ☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒ ☒☒☒☒☒☒☒☒☒☒☒☒.
- ST_Summary - ☒☒☒☒: 2.1.0 ☒☒. ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒ S ☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.
- ST_Transform - ☒☒☒☒: 2.1.0 ☒☒☒☒ ST_Transform(rast, alignto) ☒☒☒☒☒☒☒☒☒. ☒☒☒ ☒☒☒ ☒. NearestNeighbor, Bilinear, Cubic, CubicSpline, Lanczos ☒☒☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒ NearestNeighbor ☒☒☒.

- ST_Union - ☒☒☒☒: 2.1.0 ☒☒☒☒☒☒☒☒☒☒☒☒☒☒ (☒☒☒ C ☒☒☒☒☒☒☒☒☒☒☒☒☒☒). ☒☒☒☒☒☒☒☒☒ 1 ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.
- ST_Union - ☒☒☒☒: 2.1.0 ☒☒☒☒ ST_Union(rast) ☒☒ 1 ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒. PostGIS ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒ 1 ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.
- ST_Union - ☒☒☒☒: 2.1.0 ☒☒☒☒ ST_Union(rast, uniontype) ☒☒ 4 ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒ 1 ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.
- toTopoGeom - ☒☒☒☒: 2.1.0 ☒☒☒☒☒☒ TopoGeometry ☒☒☒☒☒☒☒☒☒☒☒☒☒☒. Converts a simple Geometry into a topo geometry.

Functions changed in PostGIS 2.1

- ST_Aspect - ☒☒☒☒: 2.1.0 ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒. 2.1.0 ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒ (☒☒☒☒☒☒☒) ☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.
- ST_EstimatedExtent - Changed: 2.1.0. Up to 2.0.x this was called ST_Estimated_Extent. Returns the estimated extent of a spatial table.
- ST_Force2D - ☒☒☒☒: 2.1.0 ☒☒☒☒, ☒ 2.0.x ☒☒☒☒☒☒☒☒☒☒☒ ST_Force_2D ☒☒☒☒. ☒☒☒"2 ☒☒☒☒" ☒☒☒☒☒☒☒.
- ST_Force3D - ☒☒☒☒: 2.1.0 ☒☒☒☒, ☒ 2.0.x ☒☒☒☒☒☒☒☒☒☒☒ ST_Force_3D ☒☒☒☒. ☒☒☒ XYZ ☒☒☒☒☒☒☒☒☒. ST_Force3DZ ☒☒☒☒☒☒☒.
- ST_Force3DM - ☒☒☒☒: 2.1.0 ☒☒☒☒, ☒ 2.0.x ☒☒☒☒☒☒☒☒☒☒☒ ST_Force_3DM ☒☒☒☒☒. ☒☒☒ XYM ☒☒☒☒☒☒☒☒☒.
- ST_Force3DZ - ☒☒☒☒: 2.1.0 ☒☒☒☒, ☒ 2.0.x ☒☒☒☒☒☒☒☒☒☒☒ ST_Force_3DZ ☒☒☒☒. ☒☒☒ XYZ ☒☒☒☒☒☒☒☒☒.
- ST_Force4D - ☒☒☒☒: 2.1.0 ☒☒☒☒, ☒ 2.0.x ☒☒☒☒☒☒☒☒☒☒☒ ST_Force_4D ☒☒☒☒. ☒☒☒ XYZM ☒☒☒☒☒☒☒☒☒.
- ST_ForceCollection - ☒☒☒☒: 2.1.0 ☒☒☒☒, ☒ 2.0.x ☒☒☒☒☒☒☒☒☒☒☒ ST_Force_Collection ☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒☒☒☒.
- ST_HillShade - ☒☒☒☒: 2.1.0 ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒. 2.1.0 ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒, ☒☒☒, ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.
- ST_LineInterpolatePoint - ☒☒☒☒: 2.1.0 ☒☒☒☒, ☒ 2.0.x ☒☒☒☒☒☒☒☒☒☒☒ ST_Line_Interpolate_Point ☒☒☒☒. Returns a point interpolated along a line at a fractional location.
- ST_LineLocatePoint - ☒☒☒☒: 2.1.0 ☒☒☒☒, ☒ 2.0.x ☒☒☒☒☒☒☒☒☒☒☒ ST_Line_Locate_Point ☒☒☒☒. Returns the fractional location of the closest point on a line to a point.
- ST_LineSubstring - ☒☒☒☒: 2.1.0 ☒☒☒☒, ☒ 2.0.x ☒☒☒☒☒☒☒☒☒☒☒ ST_Line_Substring ☒☒☒☒☒. Returns the part of a line between two fractional locations.
- ST_PixelAsCentroids - ☒☒☒☒: 2.1.1 ☒☒☒☒ exclude_nodata_value ☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒ (☒☒☒☒☒☒) ☒☒☒☒☒☒☒☒ X, Y ☒☒☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.
- ST_PixelAsPoints - ☒☒☒☒: 2.1.1 ☒☒☒☒ exclude_nodata_value ☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒ X, Y ☒☒☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.
- ST_PixelAsPolygons - ☒☒☒☒: 2.1.1 ☒☒☒☒ exclude_nodata_value ☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒ X, Y ☒☒☒☒☒☒☒☒☒☒☒☒☒☒.

- ST_Polygon - ☒☒☒☒: 2.1.0 ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒, ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒. NODATA ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.
- ST_RasterToWorldCoordX - ☒☒☒☒: 2.1.0 ☒☒☒☒☒☒☒ ST_Raster2WorldCoordX ☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒ X ☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒ 1 ☒☒☒☒☒☒☒.
- ST_RasterToWorldCoordY - ☒☒☒☒: 2.1.0 ☒☒☒☒☒☒☒ ST_Raster2WorldCoordY ☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒ Y ☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒ 1 ☒☒☒☒☒☒☒.
- ST_Rescale - ☒☒☒☒: 2.1.0 ☒☒☒☒ SRID ☒☒☒☒☒☒☒☒☒☒☒☒☒. Resample a raster by adjusting only its scale (or pixel size). New pixel values are computed using the NearestNeighbor (english or american spelling), Bilinear, Cubic, CubicSpline, Lanczos, Max or Min resampling algorithm. Default is NearestNeighbor.
- ST_Reskew - ☒☒☒☒: 2.1.0 ☒☒☒☒ SRID ☒☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒ (☒☒☒☒☒☒☒☒☒☒) ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒. NearestNeighbor(☒☒☒☒☒☒☒☒☒), Bilinear, Cubic, CubicSpline ☒☒ Lanczos ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒ NearestNeighbor ☒☒☒.
- ST_Segmentize - Changed: 2.1.0 As a result of the introduction of geography support, the usage ST_Segmentize('LINESTRING(1 2, 3 4)', 0.5) causes an ambiguous function error. The input needs to be properly typed as a geometry or geography. Use ST_GeomFromText, ST_GeogFromText or a cast to the required type (e.g. ST_Segmentize('LINESTRING(1 2, 3 4)'::geometry, 0.5)) Returns a modified geometry/geography having no segment longer than a given distance.
- ST_Slope - ☒☒☒☒: 2.1.0 ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒. 2.1.0 ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒ (☒☒☒☒☒☒) ☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒☒☒.
- ST_SnapToGrid - ☒☒☒☒: 2.1.0 ☒☒☒☒ SRID ☒☒☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒. NearestNeighbor(☒☒☒☒☒☒☒☒☒), Bilinear, Cubic, CubicSpline ☒☒ Lanczos ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒ NearestNeighbor ☒☒☒.
- ST_WorldToRasterCoordX - ☒☒☒☒: 2.1.0 ☒☒☒☒☒☒☒ ST_World2RasterCoordX ☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒ (pt) ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒ X, Y ☒☒☒☒ (xw, yw) ☒☒☒☒☒☒.
- ST_WorldToRasterCoordY - ☒☒☒☒: 2.1.0 ☒☒☒☒☒☒☒ ST_World2RasterCoordY ☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒ (pt) ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒ X, Y ☒☒☒☒ (xw, yw) ☒☒☒☒☒☒.

13.12.14 PostGIS Functions new or enhanced in 2.0

The functions given below are PostGIS functions that were added or enhanced.

Functions new in PostGIS 2.0

- && - 2.0.0 ☒☒☒☒☒☒☒☒☒☒☒☒. A ☒☒☒☒☒☒ B ☒☒☒☒☒☒☒☒☒☒☒☒ TRUE ☒☒☒☒☒☒.
- &&& - 2.0.0 ☒☒☒☒☒☒☒☒☒☒☒☒. A ☒ n ☒☒☒☒☒☒☒ B ☒ n ☒☒☒☒☒☒☒☒☒☒☒☒☒ TRUE ☒☒☒☒☒☒☒.
- <#> - Availability: 2.0.0. A ☒ B ☒☒☒☒☒☒☒☒ 2 ☒☒☒☒☒☒☒☒☒☒.
- <-> - Availability: 2.0.0 -- Weak KNN provides nearest neighbors based on geometry centroid distances instead of true distances. Exact results for points, inexact for all other types. A ☒ B ☒☒☒ 2 ☒☒☒☒☒☒☒☒☒☒.
- @ - 2.0.0 ☒☒☒☒ raster @ raster, raster @ geometry ☒☒☒☒☒☒☒☒☒. B ☒☒☒☒☒☒ A ☒☒☒☒☒☒☒☒☒☒☒☒ TRUE ☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.
- @ - 2.0.5 ☒☒☒☒ geometry @ raster ☒☒☒☒☒☒☒. B ☒☒☒☒☒☒ A ☒☒☒☒☒☒☒☒☒☒☒☒ TRUE ☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.

- AddEdge - 2.0.0 ☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒, ☒☒☒ (☒☒☒☒☒) ☒☒☒☒☒☒☒ ID(edgeid) ☒☒☒☒☒☒☒.
- AddFace - 2.0.0 ☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒ (face primitive) ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.
- AddNode - 2.0.0 ☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒ ID(nodeid) ☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒, ☒☒☒☒☒ ID ☒☒☒☒☒☒☒.
- AddOverviewConstraints - 2.0.0 ☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒ (overview) ☒☒☒☒☒☒☒.
- AddRasterConstraints - 2.0.0 ☒☒☒☒☒☒☒☒☒☒☒☒. Adds raster constraints to a loaded raster table for a specific column that constrains spatial ref, scaling, blocksize, alignment, bands, band type and a flag to denote if raster column is regularly blocked. The table must be loaded with data for the constraints to be inferred. Returns true if the constraint setting was accomplished and issues a notice otherwise.
- AsGML - 2.0.0 ☒☒☒☒☒☒☒☒☒☒☒☒. TopoGeometry ☒ GML ☒☒☒☒☒☒☒☒☒☒.
- CopyTopology - 2.0.0 ☒☒☒☒☒☒☒☒☒☒☒☒. Makes a copy of a topology (nodes, edges, faces, layers and TopoGeometries) into a new schema
- DropOverviewConstraints - 2.0.0 ☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒☒ (overview) ☒☒☒☒☒☒☒☒☒☒☒☒☒☒.
- DropRasterConstraints - 2.0.0 ☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒☒ PostGIS ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒.
- GetEdgeByPoint - 2.0.0 ☒☒☒☒☒☒☒☒☒☒☒☒. Finds the edge-id of an edge that intersects a given point.
- GetFaceByPoint - 2.0.0 ☒☒☒☒☒☒☒☒☒☒☒☒. Finds face intersecting a given point.
- GetNodeByPoint - 2.0.0 ☒☒☒☒☒☒☒☒☒☒☒☒. Finds the node-id of a node at a point location.
- GetNodeEdges - 2.0 ☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.
- GetRingEdges - 2.0.0 ☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒.
- GetTopoGeomElements - 2.0.0 ☒☒☒☒☒☒☒☒☒☒☒☒. Returns a set of topoelement objects containing the topological element_id,element_type of the given TopoGeometry (primitive elements).
- GetTopologySRID - 2.0.0 ☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒ topology.topology ☒☒☒☒☒☒☒☒☒☒ SRID ☒☒☒☒☒☒☒.
- Polygonize - 2.0.0 ☒☒☒☒☒☒☒☒☒☒☒☒. Finds and registers all faces defined by topology edges.
- ST_3DClosestPoint - 2.0.0 ☒☒☒☒☒☒☒☒☒☒☒☒. g2 ☒☒☒☒☒☒☒ g1 ☒☒☒☒☒ 3 ☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒ 3D ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.
- ST_3DDFullyWithin - 2.0.0 ☒☒☒☒☒☒☒☒☒☒☒☒. Tests if two 3D geometries are entirely within a given 3D distance
- ST_3DDWithin - 2.0.0 ☒☒☒☒☒☒☒☒☒☒☒☒. Tests if two 3D geometries are within a given 3D distance
- ST_3DDistance - 2.0.0 ☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒, ☒☒☒☒☒☒☒☒ (SRS ☒☒☒☒☒) 3 ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.
- ST_3DIntersects - 2.0.0 ☒☒☒☒☒☒☒☒☒☒☒☒. Tests if two geometries spatially intersect in 3D - only for points, linestrings, polygons, polyhedral surface (area)

- ST_3DLongestLine - 2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠ 3 ⊠⊠⊠⊠ (longest) ⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_3DMaxDistance - 2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠ (SRS ⊠⊠⊠⊠) 3 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_3DShortestLine - 2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠ 3 ⊠⊠⊠⊠ (shortest) ⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_AddEdgeModFace - 2.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_AddEdgeNewFaces - 2.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ 2 ⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_AsGDALRaster - 2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. GDAL 1.6.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. Return the raster tile in the designated GDAL Raster format. Raster formats are one of those supported by your compiled library. Use ST_GDALDrivers() to get a list of formats supported by your library.
- ST_AsJPEG - 2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. GDAL 1.6.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ JPEG(Joint Photographic Exports Group) ⊠⊠⊠ (⊠⊠⊠⊠⊠⊠) ⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠ 1 ⊠⊠⊠⊠⊠⊠ 3 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠ 3 ⊠⊠⊠⊠⊠⊠⊠⊠ 3 ⊠⊠⊠⊠⊠⊠ RGB ⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_AsLatLonText - 2.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_AsPNG - 2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. GDAL 1.6.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ PNG(Portable Network Graphics) ⊠⊠⊠ (⊠⊠⊠⊠⊠⊠) ⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠ 1 ⊠, 3 ⊠, ⊠⊠ 4 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠ 2 ⊠⊠⊠ 4 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠ 1 ⊠⊠⊠⊠⊠. ⊠⊠⊠ RGB ⊠⊠ RGBA ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_AsRaster - 2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. GDAL 1.6.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. PostGIS ⊠⊠⊠ PostGIS ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_AsTIFF - 2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. GDAL 1.6.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. Return the raster selected bands as a single TIFF image (byte array). If no band is specified or any of specified bands does not exist in the raster, then will try to use all bands.
- ST_AsX3D - 2.0.0 ⊠⊠⊠⊠ ISO-IEC-19776-1.2-X3DEncodings-XML ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠ X3D XML ⊠⊠⊠⊠⊠⊠⊠: ISO-IEC-19776-1.2-X3DEncodings-XML ⊠⊠⊠⊠⊠⊠⊠.
- ST_Aspect - 2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (⊠⊠⊠⊠⊠⊠⊠) ⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_Band - 2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_BandIsNoData - 2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠ NODATA ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_Clip - 2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. Returns the raster clipped by the input geometry. If band number is not specified, all bands are processed. If crop is not specified or TRUE, the output raster is cropped. If touched is set to TRUE, then touched pixels are included, otherwise only if the center of the pixel is in the geometry it is included.
- ST_CollectionHomogenize - 2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. Returns the simplest representation of a geometry collection.
- ST_ConcaveHull - 2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. Computes a possibly concave geometry that contains all input geometry vertices
- ST_Count - 2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ 1 ⊠⊠⊠. exclude_nodata_value ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, NODATA ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

- ST_CreateTopoGeo - 2.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_Distinct4ma - 2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠.
- ST_FlipCoordinates - 2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. Returns a version of a geometry with X and Y axis flipped.
- ST_GDALDrivers - 2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. GDAL 1.6.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. Returns a list of raster formats supported by PostGIS through GDAL. Only those formats with can_write=True can be used by ST_AsGDALRaster
- ST_GeomFromGeoJSON - 2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. JSON-C 0.9 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. GeoJSON ⊠⊠⊠⊠⊠⊠⊠⊠ PostGIS ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_GetFaceEdges - 2.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. aface ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠.
- ST_HasNoBand - 2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠ 1 ⊠⊠⊠⊠⊠⊠.
- ST_HillShade - 2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠, ⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_Histogram - 2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠ (bin; ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠) ⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_InterpolatePoint - 2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (M ⊠⊠) ⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_IsEmpty - 2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (width = 0, height = 0) ⊠⊠⊠⊠⊠ ⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_IsValidDetail - 2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. Returns a valid_detail row stating if a geometry is valid or if not a reason and a location.
- ST_IsValidReason - Availability: 2.0 version taking flags. Returns text stating if a geometry is valid, or a reason for invalidity.
- ST_MakeLine - 2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_MakeValid - 2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. Attempts to make an invalid geometry valid without losing vertices.
- ST_MapAlgebraExpr - 2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠ 1 ⊠⊠⊠: ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ PostgreSQL ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠ 1 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠ 1 ⊠⊠⊠⊠⊠⊠.
- ST_MapAlgebraExpr - 2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠ 2 ⊠⊠⊠: ⊠⊠⊠⊠⊠ 2 ⊠⊠⊠⊠⊠⊠ ⊠ PostgreSQL ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠ 1 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠ 1 ⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠ (⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠) ⊠⊠⊠⊠⊠⊠⊠. extenttype ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. extenttype ⊠⊠⊠ INTERSECTION, UNION, FIRST, SECOND ⊠⊠⊠⊠⊠⊠⊠.
- ST_MapAlgebraFct - 2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠ 1 ⊠⊠⊠: ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ PostgreSQL ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠ 1 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠ 1 ⊠⊠⊠⊠⊠⊠.

- ST_MapAlgebraFct - 2.0.0 ☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒ 2 ☒☒☒: ☒☒☒☒☒ 2 ☒☒☒☒☒☒☒☒ PostgreSQL ☒☒☒☒☒☒☒☒☒☒☒, ☒☒☒☒☒☒☒☒☒☒, ☒☒ 1 ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒ ☒☒☒☒☒☒☒☒☒☒, ☒☒ 1 ☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒ INTERSECTION ☒ ☒☒.
- ST_MapAlgebraFctNgb - 2.0.0 ☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒ 1 ☒☒☒: ☒☒☒☒☒ PostgreSQL ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒ (Map Algebra Nearest Neighbor) ☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒☒☒ (neighborhood) ☒☒☒☒ PostgreSQL ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.
- ST_Max4ma - 2.0.0 ☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.
- ST_Mean4ma - 2.0.0 ☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.
- ST_Min4ma - 2.0.0 ☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.
- ST_ModEdgeHeal - 2.0 ☒☒☒☒☒☒☒☒☒☒☒☒. Heals two edges by deleting the node connecting them, modifying the first edge and deleting the second edge. Returns the id of the deleted node.
- ST_MoveIsoNode - 2.0.0 ☒☒☒☒☒☒☒☒☒☒☒☒. Moves an isolated node in a topology from one point to another. If new apoint geometry exists as a node an error is thrown. Returns description of move.
- ST_NewEdgeHeal - 2.0 ☒☒☒☒☒☒☒☒☒☒☒☒. Heals two edges by deleting the node connecting them, deleting both edges, and replacing them with an edge whose direction is the same as the first edge provided.
- ST_Node - 2.0.0 ☒☒☒☒☒☒☒☒☒☒☒☒. Nodes a collection of lines.
- ST_NumPatches - 2.0.0 ☒☒☒☒☒☒☒☒☒☒☒☒. Returns the number of faces in a PolyhedralSurface or TIN.
- ST_OffsetCurve - 2.0 ☒☒☒☒☒☒☒☒☒☒☒☒. Returns an offset line at a given distance and side from an input line.
- ST_PatchN - 2.0.0 ☒☒☒☒☒☒☒☒☒☒☒☒. Returns the Nth face of a PolyhedralSurface or TIN.
- ST_Perimeter - ☒☒☒☒: 2.0.0 ☒☒☒☒☒☒☒☒☒☒☒☒☒. Returns the length of the boundary of a polygonal geometry or geography.
- ST_PixelAsPolygon - 2.0.0 ☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.
- ST_PixelAsPolygons - 2.0.0 ☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒ X, Y ☒☒☒☒☒☒☒☒☒☒☒☒☒☒.
- ST_Project - 2.0.0 ☒☒☒☒☒☒☒☒☒☒☒☒. Returns a point projected from a start point by a distance and bearing (azimuth).
- ST_Quantile - 2.0.0 ☒☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒ (population) ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒ (quantile) ☒☒☒☒☒☒☒. ☒☒☒, ☒☒☒☒☒ 25%, 50%, 75% ☒☒☒ (percentile) ☒☒☒☒☒☒☒☒☒☒☒☒☒☒.
- ST_Range4ma - 2.0.0 ☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.
- ST_Reclass - 2.0.0 ☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒. nband ☒☒☒☒☒☒☒☒☒☒☒☒. nband ☒☒☒☒☒☒☒☒☒☒☒☒☒☒ 1 ☒☒☒☒☒☒. ☒☒☒☒☒ ☒☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒: ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒ 16BUI ☒☒☒ 8BUI ☒☒ ☒☒☒☒☒☒☒☒.
- ST_RelateMatch - 2.0.0 ☒☒☒☒☒☒☒☒☒☒☒☒. Tests if a DE-9IM Intersection Matrix matches an Intersection Matrix pattern

- ST_RemEdgeModFace - 2.0 ☒☒☒☒☒☒☒☒☒☒☒☒. Removes an edge, and if the edge separates two faces deletes one face and modifies the other face to cover the space of both.
- ST_RemEdgeNewFace - 2.0 ☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒, ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒, ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.
- ST_Resample - 2.0.0 ☒☒☒☒☒☒☒☒☒☒☒☒☒☒. GDAL 1.6.1 ☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒, ☒☒☒☒☒☒, ☒☒☒☒☒☒☒☒☒☒, ☒☒.
- ST_Rescale - 2.0.0 ☒☒☒☒☒☒☒☒☒☒☒☒☒☒. GDAL 1.6.1 ☒☒☒☒☒☒☒☒☒☒☒. Resample a raster by adjusting only its scale (or pixel size). New pixel values are computed using the NearestNeighbor (english or american spelling), Bilinear, Cubic, CubicSpline, Lanczos, Max or Min resampling algorithm. Default is NearestNeighbor.
- ST_Reskew - 2.0.0 ☒☒☒☒☒☒☒☒☒☒☒☒☒☒. GDAL 1.6.1 ☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒ (☒☒☒☒☒☒☒☒☒☒) ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒. NearestNeighbor(☒☒☒☒☒☒☒☒☒), Bilinear, Cubic, CubicSpline ☒☒ Lanczos ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒ NearestNeighbor ☒☒☒.
- ST_SameAlignment - 2.0.0 ☒☒☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒☒, ☒☒, ☒☒☒☒☒☒☒☒, ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒ (☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒) ☒☒☒☒☒☒☒, ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.
- ST_SetBandIsNoData - 2.0.0 ☒☒☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒ isnodata ☒☒☒☒☒☒☒☒☒☒☒☒☒☒.
- ST_SharedPaths - 2.0.0 ☒☒☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒/☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.
- ST_Slope - 2.0.0 ☒☒☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒ (☒☒☒☒☒☒☒) ☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.
- ST_Snap - 2.0.0 ☒☒☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒.
- ST_SnapToGrid - 2.0.0 ☒☒☒☒☒☒☒☒☒☒☒☒☒☒. GDAL 1.6.1 ☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒. NearestNeighbor(☒☒☒☒☒☒☒☒☒), Bilinear, Cubic, CubicSpline ☒☒ Lanczos ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒ NearestNeighbor ☒☒☒.
- ST_Split - Availability: 2.0.0 requires GEOS Returns a collection of geometries created by splitting a geometry by another geometry.
- ST_StdDev4ma - 2.0.0 ☒☒☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.
- ST_Sum4ma - 2.0.0 ☒☒☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.
- ST_SummaryStats - 2.0.0 ☒☒☒☒☒☒☒☒☒☒☒☒☒☒. Returns summarystats consisting of count, sum, mean, stddev, min, max for a given raster band of a raster or raster coverage. Band 1 is assumed if no band is specified.
- ST_Transform - 2.0.0 ☒☒☒☒☒☒☒☒☒☒☒☒☒☒. GDAL 1.6.1 ☒☒☒☒☒☒☒☒☒☒☒. ☒☒. NearestNeighbor, Bilinear, Cubic, CubicSpline, Lanczos ☒☒☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒ NearestNeighbor ☒☒☒.
- ST_UnaryUnion - 2.0.0 ☒☒☒☒☒☒☒☒☒☒☒☒☒☒. Computes the union of the components of a single geometry.
- ST_Union - 2.0.0 ☒☒☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒ 1 ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.

- ST_ValueCount - 2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠) ⊠ ⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ 1 ⊠⊠⊠. ⊠⊠⊠⊠⊠⊠ NODATA ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- TopoElementArray_Agg - 2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. Returns a topoelementarray for a set of element_id, type arrays (topoelements).
- TopoGeo_AddLineString - 2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. Adds a linestring to an existing topology using a tolerance and possibly splitting existing edges/faces.
- TopoGeo_AddPoint - 2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (split) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- TopoGeo_AddPolygon - 2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. Adds a polygon to an existing topology using a tolerance and possibly splitting existing edges/faces. Returns face identifiers.
- TopologySummary - 2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. Takes a topology name and provides summary totals of types of objects in topology.
- toTopoGeom - 2.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. Converts a simple Geometry into a topo geometry.
- ~ - 2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. A ⊠⊠⊠⊠⊠⊠⊠ B ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ TRUE ⊠⊠⊠⊠⊠⊠⊠. ⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ~= - 2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. A ⊠⊠⊠⊠⊠⊠⊠ B ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ TRUE ⊠⊠⊠⊠⊠⊠⊠.

Functions enhanced in PostGIS 2.0

- && - ⊠⊠⊠⊠⊠: 2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (polyhedral surface) ⊠⊠⊠⊠⊠⊠⊠. A ⊠ 2D ⊠⊠⊠⊠⊠⊠ B ⊠ 2D ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ TRUE ⊠⊠⊠⊠⊠⊠⊠.
- AddGeometryColumn - ⊠⊠⊠⊠⊠: 2.0.0 ⊠⊠. use_typmod ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠ typmod ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- Box2D - ⊠⊠⊠⊠⊠: 2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠ TIN ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. Returns a BOX2D representing the 2D extent of a geometry.
- Box3D - ⊠⊠⊠⊠⊠: 2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠ TIN ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. Returns a BOX3D representing the 3D extent of a geometry.
- CreateTopology - Enhanced: 2.0 added support for hasZ. Creates a new topology schema and registers it in the topology.topology table.
- GeometryType - ⊠⊠⊠⊠⊠: 2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠ TIN ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ST_Geometry ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- Populate_Geometry_Columns - ⊠⊠⊠⊠⊠: 2.0.0 ⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ use_typmod ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. Ensures geometry columns are defined with type modifiers or have appropriate spatial constraints.
- ST_3DExtent - ⊠⊠⊠⊠⊠: 2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠ TIN ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. Aggregate function that returns the 3D bounding box of geometries.
- ST_Affine - ⊠⊠⊠⊠⊠: 2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠ TIN ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. Apply a 3D affine transformation to a geometry.
- ST_Area - ⊠⊠⊠⊠⊠: 2.0.0 ⊠⊠⊠⊠⊠ 2 ⊠⊠⊠⊠⊠⊠⊠⊠⊠ (polyhedral surface) ⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_AsBinary - ⊠⊠⊠⊠⊠: 2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠ TIN ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. Return the OGC/ISO Well-Known Binary (WKB) representation of the geometry/geography without SRID meta data.

- ST_AsBinary - ⊠⊠⊠⊠: 2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. Return the OGC/ISO Well-Known Binary (WKB) representation of the geometry/geography without SRID meta data.
- ST_AsBinary - ⊠⊠⊠⊠: 2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. Return the OGC/ISO Well-Known Binary (WKB) representation of the geometry/geography without SRID meta data.
- ST_AsEWKB - ⊠⊠⊠⊠: 2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠ TIN ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. Return the Extended Well-Known Binary (EWKB) representation of the geometry with SRID meta data.
- ST_AsEWKT - ⊠⊠⊠⊠: 2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠ TIN ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠ WKT(Well-Known Text) ⊠⊠⊠⊠ SRID ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_AsGML - ⊠⊠⊠⊠: 2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠ GML 3 ⊠⊠⊠’4’ ⊠⊠⊠⊠⊠⊠⊠⊠. GML 3 ⊠⊠⊠⊠⊠⊠⊠⊠ TIN ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠’32’ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠ GML 2 ⊠⊠ GML 3 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_AsKML - ⊠⊠⊠⊠: 2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠. ⊠⊠⊠ GML 2 ⊠⊠ GML 3 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_Azimuth - ⊠⊠⊠⊠: 2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠ 2 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_Dimension - ⊠⊠⊠⊠: 2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (polyhedral surface) ⊠ TIN ⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ST_Geometry ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_Dump - ⊠⊠⊠⊠: 2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠ TIN ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. Returns a set of geometry_dump rows for the components of a geometry.
- ST_DumpPoints - ⊠⊠⊠⊠: 2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠ TIN ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_Expand - ⊠⊠⊠⊠: 2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠ TIN ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. Returns a bounding box expanded from another bounding box or a geometry.
- ST_Extent - ⊠⊠⊠⊠: 2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠ TIN ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. Aggregate function that returns the bounding box of geometries.
- ST_Force2D - ⊠⊠⊠⊠: 2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (polyhedral surface) ⊠⊠⊠⊠⊠⊠. ⊠⊠⊠”2 ⊠⊠⊠⊠” ⊠⊠⊠⊠⊠⊠.
- ST_Force3D - ⊠⊠⊠⊠: 2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (polyhedral surface) ⊠⊠⊠⊠⊠⊠. ⊠⊠⊠ XYZ ⊠⊠⊠ ⊠⊠⊠⊠⊠. ST_Force3DZ ⊠⊠⊠⊠⊠⊠.
- ST_Force3DZ - ⊠⊠⊠⊠: 2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (polyhedral surface) ⊠⊠⊠⊠⊠⊠. ⊠⊠⊠ XYZ ⊠⊠ ⊠⊠⊠⊠⊠⊠.
- ST_ForceCollection - ⊠⊠⊠⊠: 2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (polyhedral surface) ⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_ForceRHR - ⊠⊠⊠⊠: 2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (polyhedral surface) ⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠ (orientation) ⊠⊠⊠⊠⊠⊠ (Right-Hand Rule) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_GMLToSQL - ⊠⊠⊠⊠: 2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (polyhedral surface) ⊠ TIN ⊠⊠⊠⊠⊠⊠. Returns a geometry from GML. Alias for ST_GeomFromGML.
- ST_GMLToSQL - ⊠⊠⊠⊠: 2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ SRID ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. Returns a geometry from GML. Alias for ST_GeomFromGML.
- ST_GeomFromEWKB - ⊠⊠⊠⊠: 2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (polyhedral surface) ⊠ TIN ⊠⊠⊠⊠⊠⊠. EWKB(Extended Well-Known Binary) ⊠⊠⊠⊠⊠⊠ ST_Geometry ⊠⊠⊠⊠⊠⊠⊠.
- ST_GeomFromEWKT - ⊠⊠⊠⊠: 2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (polyhedral surface) ⊠ TIN ⊠⊠⊠⊠⊠⊠. EWKT(Extended Well-Known Text) ⊠⊠⊠⊠⊠⊠ ST_Geometry ⊠⊠⊠⊠⊠⊠⊠.

- ST_GeomFromGML - ⊠⊠⊠⊠: 2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (polyhedral surface) ⊠ TIN ⊠⊠⊠⊠⊠⊠. ⊠ ⊠⊠ GML ⊠⊠⊠⊠⊠⊠⊠⊠ PostGIS ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_GeomFromGML - ⊠⊠⊠⊠: 2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ SRID ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠ GML ⊠⊠⊠⊠⊠⊠⊠⊠ PostGIS ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_GeometryN - ⊠⊠⊠⊠: 2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠ TIN ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ST_Geometry ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_GeometryType - ⊠⊠⊠⊠: 2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (polyhedral surface) ⊠⊠⊠⊠⊠⊠. ST_Geometry ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_IsClosed - ⊠⊠⊠⊠: 2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (polyhedral surface) ⊠⊠⊠⊠⊠⊠. LINESTRING ⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ TRUE ⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠ (⊠⊠⊠⊠⊠⊠⊠) ⊠⊠⊠⊠ TRUE ⊠⊠ ⊠⊠⊠⊠.
- ST_MakeEnvelope - ⊠⊠⊠⊠: 2.0 ⊠⊠⊠⊠ SRID ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (envelope) ⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠ SRID ⊠⊠⊠⊠ SRS ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_MakeValid - Enhanced: 2.0.1, speed improvements Attempts to make an invalid geometry valid without losing vertices.
- ST_NPoints - ⊠⊠⊠⊠: 2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (polyhedral surface) ⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠ (⊠⊠⊠⊠) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_NumGeometries - ⊠⊠⊠⊠: 2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠ TIN ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_Relate - Enhanced: 2.0.0 - added support for specifying boundary node rule. Tests if two geometries have a topological relationship matching an Intersection Matrix pattern, or computes their Intersection Matrix
- ST_Rotate - ⊠⊠⊠⊠: 2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠ TIN ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. Rotates a geometry about an origin point.
- ST_Rotate - Enhanced: 2.0.0 additional parameters for specifying the origin of rotation were added. Rotates a geometry about an origin point.
- ST_RotateX - ⊠⊠⊠⊠: 2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠ TIN ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. Rotates a geometry about the X axis.
- ST_RotateY - ⊠⊠⊠⊠: 2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠ TIN ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. Rotates a geometry about the Y axis.
- ST_RotateZ - ⊠⊠⊠⊠: 2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠ TIN ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. Rotates a geometry about the Z axis.
- ST_Scale - ⊠⊠⊠⊠: 2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠ TIN ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. Scales a geometry by given factors.
- ST_ShiftLongitude - ⊠⊠⊠⊠: 2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (polyhedral surface) ⊠ TIN ⊠⊠⊠⊠⊠⊠. Shifts the longitude coordinates of a geometry between -180..180 and 0..360.
- ST_Summary - ⊠⊠⊠⊠: 2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠.
- ST_Transform - ⊠⊠⊠⊠: 2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (polyhedral surface) ⊠⊠⊠⊠⊠⊠. Return a new geometry with coordinates transformed to a different spatial reference system.
- ST_Value - ⊠⊠⊠⊠: 2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠ exclude_nodata_value ⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠ columnx, rowy ⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠ 1 ⊠⊠⊠⊠⊠⊠, ⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠ 1 ⊠⊠⊠⊠⊠⊠. exclude_nodata_value ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, nodata ⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. exclude_nodata_value ⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.

- ValidateTopology - ☒☒☒☒: 2.0.0 ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒, ☒☒☒☒☒☒☒ ☒☒☒☒☒☒☒ (false positive) ☒☒☒☒☒☒☒. Returns a set of validatetopology_returntype objects detailing issues with topology.

Functions changed in PostGIS 2.0

- AddGeometryColumn - ☒☒☒☒: 2.0.0 ☒☒☒☒ geometry_columns ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒ ☒☒☒☒ geometry_columns ☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒, ☒☒ ☒ PostgreSQL ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒ WGS84 POINT ☒☒☒☒☒☒☒☒☒☒☒☒ ☒☒☒☒☒☒☒☒☒☒☒☒☒☒: ALTER TABLE some_table ADD COLUMN geom geometry(Point,4326); ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.
- AddGeometryColumn - ☒☒☒☒: 2.0.0 ☒☒. ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒, ☒☒☒☒ use_typmod ☒ ☒☒☒☒, ☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.
- AddGeometryColumn - ☒☒☒☒: 2.0.0 ☒☒. ☒☒☒☒☒☒☒☒ geometry_columns ☒☒☒☒☒☒☒☒☒☒ ☒, ☒☒ typmod ☒☒☒☒☒☒☒☒☒☒☒☒☒☒, ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒ typmod ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒ geometry_columns ☒☒☒☒☒☒ ☒☒☒☒☒, ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒ typmod ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒ ☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.
- Box3D - ☒☒☒☒: 2.0.0 ☒☒☒☒☒☒☒☒ BOX3D ☒☒ BOX2D ☒☒☒☒☒☒☒☒. BOX2D ☒☒☒☒☒☒☒☒☒ ☒☒☒☒☒, 2.0.0 ☒☒☒☒ BOX3D ☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒ BOX3D ☒☒☒☒☒☒☒☒☒☒.
- DropGeometryColumn - Changed: 2.0.0. Retained for backward compatibility after geometry_columns became a view of the system catalogs. ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.
- DropGeometryTable - Changed: 2.0.0. Retained for backward compatibility after geometry_columns became a view of the system catalogs. ☒☒☒☒ geometry_columns ☒☒☒☒☒☒☒☒☒☒☒.
- Populate_Geometry_Columns - ☒☒☒☒: 2.0.0 ☒☒. ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒ ☒☒☒☒☒☒☒☒☒☒. ☒ use_typmod ☒☒☒☒☒☒☒☒☒☒, ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒. Ensures geometry columns are defined with type modifiers or have appropriate spatial constraints.
- ST_3DExtent - Changed: 2.0.0 In prior versions this used to be called ST_Extent3D Aggregate function that returns the 3D bounding box of geometries.
- ST_3DLength - ☒☒☒☒: 2.0.0 ☒☒☒☒☒☒☒☒ ST_Length3D ☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒ ☒☒☒☒☒.
- ST_3DMakeBox - Changed: 2.0.0 In prior versions this used to be called ST_MakeBox3D Creates a BOX3D defined by two 3D point geometries.
- ST_3DPerimeter - ☒☒☒☒: 2.0.0 ☒☒☒☒☒☒☒☒ ST_Perimeter3D ☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒ ☒☒☒☒☒☒☒☒.
- ST_AsBinary - Changed: 2.0.0 Inputs to this function can not be unknown -- must be geometry. Constructs such as ST_AsBinary('POINT(1 2)') are no longer valid and you will get an n ST_AsBinary(unknown) is not unique error. Code like that needs to be changed to ST_AsBinary('POINT(1 2)'::geometry);. If that is not possible, then install legacy.sql. Return the OGC/ISO Well-Known Binary (WKB) representation of the geometry/geography without SRID meta data.
- ST_AsGML - ☒☒☒☒: 2.0.0 ☒☒☒☒☒☒☒☒☒☒☒☒ (named arg) ☒☒☒☒☒☒☒☒☒☒. ☒☒☒ GML 2 ☒ ☒ GML 3 ☒☒☒☒☒☒☒☒☒☒☒.
- ST_AsGeoJSON - ☒☒☒☒: 2.0.0 ☒☒☒☒☒☒☒☒☒☒☒ (default arg) ☒☒☒☒☒☒☒☒☒ (named arg) ☒☒ ☒☒☒☒. Return a geometry or feature in GeoJSON format.
- ST_AsSVG - ☒☒☒☒: 2.0.0 ☒☒☒☒☒☒☒☒☒☒☒ (default arg) ☒☒☒☒☒☒☒☒☒☒☒☒ (named arg) ☒☒☒☒☒☒. Returns SVG path data for a geometry.

- ST_EndPoint - ⊠⊠⊠⊠: 2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. PostGIS ⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. 2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ NULL ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ 2.0 ⊠⊠⊠⊠ NULL ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. Returns the last point of a LineString, CircularLineString, or NURBSCurve.
- ST_GDALDrivers - ⊠⊠⊠⊠: 2.0.6, 2.1.3 ⊠⊠ - GUC ⊠⊠⊠⊠⊠⊠⊠ gdal_enabled_drivers ⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. Returns a list of raster formats supported by PostGIS through GDAL. Only those formats with can_write=True can be used by ST_AsGDALRaster
- ST_GeomFromText - ⊠⊠⊠⊠: PostGIS 2.0.0 ⊠⊠⊠⊠⊠⊠⊠ ST_GeomFromText('GEOMETRYCOLLECTION ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. PostGIS 2.0.0 ⊠⊠⊠⊠⊠, SQL/MM ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠. ⊠⊠⊠ ST_GeomFromText('GEOMETRYCOLLECTION EMPTY') ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. WKT ⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠ ST_Geometry ⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_GeometryN - ⊠⊠⊠⊠: 2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ NULL ⊠⊠⊠⊠⊠⊠⊠⊠⊠. 2.0.0 ⊠⊠⊠⊠ ST_GeometryN(..,1) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ST_Geometry ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_IsEmpty - ⊠⊠⊠⊠: PostGIS 2.0.0 ⊠⊠⊠⊠⊠⊠⊠ ST_GeomFromText('GEOMETRYCOLLECTION(EMPT ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. PostGIS 2.0.0 ⊠⊠⊠⊠⊠, SQL/MM ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠. Tests if a geometry is empty.
- ST_Length - ⊠⊠⊠⊠: 2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. 2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠/⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠/⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. 2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠ 0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ST_Perimeter ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_LocateAlong - ⊠⊠⊠⊠: 2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠ ST_Locate_Along_Measure ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. Returns the point(s) on a geometry that match a measure value.
- ST_LocateBetween - ⊠⊠⊠⊠: 2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠ ST_Locate_Along_Measure ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. Returns the portions of a geometry that match a measure range.
- ST_ModEdgeSplit - ⊠⊠⊠⊠: 2.0 ⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠ ST_ModEdgesSplit ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_NumGeometries - ⊠⊠⊠⊠: 2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ NULL ⊠⊠⊠⊠⊠⊠ ⊠⊠. 2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ 1 ⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_NumInteriorRings - ⊠⊠⊠⊠: 2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_PointN - ⊠⊠⊠⊠: 2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. PostGIS ⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. 2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ NULL ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ST_LineString ⊠⊠ ST_CircularString ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_ScaleX - ⊠⊠⊠⊠: 2.0.0 ⊠⊠ WKTRaster ⊠⊠⊠⊠⊠ ST_PixelSizeX ⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠ X ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_ScaleY - ⊠⊠⊠⊠: 2.0.0 ⊠⊠ WKTRaster ⊠⊠⊠⊠⊠ ST_PixelSizeY ⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠ Y ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_SetScale - ⊠⊠⊠⊠: 2.0.0 ⊠⊠ WKTRaster ⊠⊠⊠⊠⊠ ST_SetPixelSize ⊠⊠⊠⊠⊠⊠⊠⊠⊠. 2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. X ⊠ Y ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠/⊠⊠⊠⊠/⊠ ⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_StartPoint - ⊠⊠⊠⊠: 2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. PostGIS ⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. 2.0.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠ NULL ⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ 2.0 ⊠⊠⊠⊠ NULL ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. Returns the first point of a LineString, CircularLineString, or NURBSCurve.

13.12.15 PostGIS Functions new or enhanced in 1.5

The functions given below are PostGIS functions that were added or enhanced.

Functions new in PostGIS 1.5

- && - 1.5.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. A ⊠ 2D ⊠⊠⊠⊠⊠ B ⊠ 2D ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ TRUE ⊠⊠ ⊠⊠⊠⊠.
- PostGIS_LibXML_Version - 1.5 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. Returns the version number of the libxml2 library.
- ST_AddMeasure - 1.5.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. Interpolates measures along a linear geometry.
- ST_AsBinary - 1.5.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. Return the OGC/ISO Well-Known Binary (WKB) representation of the geometry/geography without SRID meta data.
- ST_AsGML - 1.5.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠ GML 2 ⊠⊠ GML 3 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_AsGeoJSON - 1.5.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. Return a geometry or feature in GeoJSON format.
- ST_AsText - 1.5.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠/⊠⊠⊠⊠ WKT(Well-Known Text) ⊠⊠⊠⊠ SRID ⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_Buffer - Availability: 1.5 - ST_Buffer was enhanced to support different endcaps and join types. These are useful for example to convert road linestrings into polygon roads with flat or square edges instead of rounded edges. Thin wrapper for geography was added. Computes a geometry covering all points within a given distance from a geometry.
- ST_ClosestPoint - 1.5.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. Returns the 2D point on g1 that is closest to g2. This is the first point of the shortest line from one geometry to the other.
- ST_CollectionExtract - 1.5.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. Given a geometry collection, returns a multi-geometry containing only elements of a specified type.
- ST_Covers - 1.5.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. Tests if every point of B lies in A
- ST_DFullyWithin - 1.5.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. Tests if a geometry is entirely inside a distance of another
- ST_DWithin - Availability: 1.5.0 support for geography was introduced Tests if two geometries are within a given distance
- ST_Distance - 1.5.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠ 3 ⊠⊠⊠⊠ (longest) ⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_DistanceSphere - 1.5 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. 1.5 ⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. PostGIS 1.5 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_DistanceSpheroid - 1.5 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. 1.5 ⊠⊠⊠⊠⊠⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. PostGIS 1.5 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_DumpPoints - 1.5.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_Envelope - 1.5.0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, float4 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. ⊠ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ (double precision; float8) ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠.
- ST_Expand - Availability: 1.5.0 behavior changed to output double precision instead of float4 coordinates. Returns a bounding box expanded from another bounding box or a geometry.
- ST_GMLToSQL - 1.5 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. LibXML2 1.6 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠. Returns a geometry from GML. Alias for ST_GeomFromGML.

- ST_GeomFromGML - 1.5 ☒☒☒☒☒☒☒☒☒☒☒☒☒☒. LibXML2 1.6 ☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒ GML ☒☒☒☒☒☒☒☒ PostGIS ☒☒☒☒☒☒☒☒☒☒☒.
- ST_GeomFromKML - Availability: 1.5, requires libxml2 2.6+ ☒☒☒ KML ☒☒☒☒☒☒☒☒ PostGIS ☒☒☒☒☒☒☒☒☒☒.
- ST_HausdorffDistance - 1.5.0 ☒☒☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒ 3 ☒☒☒☒ (shortest) ☒☒☒☒☒☒☒☒.
- ST_Intersection - Availability: 1.5 support for geography data type was introduced. Computes a geometry representing the shared portion of geometries A and B.
- ST_Intersects - Availability: 1.5 support for geography was introduced. Tests if two geometries intersect (they have at least one point in common)
- ST_Length - 1.5.0 ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.
- ST_LongestLine - 1.5.0 ☒☒☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒ 3 ☒☒☒☒ (longest) ☒☒☒☒☒☒☒☒☒.
- ST_MakeEnvelope - 1.5 ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒ SRID ☒☒☒☒☒ SRS ☒☒☒☒☒☒☒☒☒☒☒.
- ST_MaxDistance - 1.5.0 ☒☒☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒ 2 ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.
- ST_ShortestLine - 1.5.0 ☒☒☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒ 2 ☒☒☒☒☒☒☒☒☒☒☒☒☒.
- ~= - 1.5.0 ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒. A ☒☒☒☒☒☒☒ B ☒☒☒☒☒☒☒☒☒☒☒☒ TRUE ☒☒☒☒☒☒☒.

13.12.16 PostGIS Functions new or enhanced in 1.4

The functions given below are PostGIS functions that were added or enhanced.

Functions new in PostGIS 1.4

- Populate_Geometry_Columns - 1.4.0 ☒☒☒☒☒☒☒☒☒☒☒☒☒☒. Ensures geometry columns are defined with type modifiers or have appropriate spatial constraints.
- ST_Collect - 1.4.0 ☒☒☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒ ST_MakeLine ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒ ST_MakeLine ☒☒☒☒☒☒☒☒☒☒☒☒☒. Creates a GeometryCollection or Multi* geometry from a set of geometries.
- ST_ContainsProperly - 1.4.0 ☒☒☒☒☒☒☒☒☒☒☒☒☒☒. Tests if every point of B lies in the interior of A
- ST_GeoHash - 1.4.0 ☒☒☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒ GeoHash ☒☒☒☒☒☒☒☒☒☒.
- ST_IsValidReason - Availability: 1.4 Returns text stating if a geometry is valid, or a reason for invalidity.
- ST_LineCrossingDirection - Availability: 1.4 Returns a number indicating the crossing behavior of two LineStrings
- ST_LocateBetweenElevations - 1.4.0 ☒☒☒☒☒☒☒☒☒☒☒☒☒☒. Returns the portions of a geometry that lie in an elevation (Z) range.
- ST_MakeLine - 1.4.0 ☒☒☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒ ST_MakeLine ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒ ST_MakeLine ☒☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒, ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒.
- ST_MinimumBoundingCircle - 1.4.0 ☒☒☒☒☒☒☒☒☒☒☒☒☒☒. Returns the smallest circle polygon that contains a geometry.
- ST_Union - Availability: 1.4.0 - ST_Union was enhanced. ST_Union(geomarray) was introduced and also faster aggregate collection in PostgreSQL. Computes a geometry representing the point-set union of the input geometries.

13.12.17 PostGIS Functions new or enhanced in 1.3

The functions given below are PostGIS functions that were added or enhanced.

Functions new in PostGIS 1.3

- ST_AsGML - 1.3.2 ☒☒☒☒☒☒☒☒☒☒☒☒☒. ☒☒☒ GML 2 ☒☒ GML 3 ☒☒☒☒☒☒☒☒☒☒.
- ST_AsGeoJSON - 1.3.4 ☒☒☒☒☒☒☒☒☒☒☒☒☒. Return a geometry or feature in GeoJSON format.
- ST_CurveToLine - Availability: 1.3.0 Converts a geometry containing curves to a linear geometry.
- ST_LineToCurve - Availability: 1.3.0 Converts a linear geometry to a curved geometry.
- ST_SimplifyPreserveTopology - 1.3.3 ☒☒☒☒☒☒☒☒☒☒☒☒☒. Returns a simplified and valid representation of a geometry, using the Douglas-Peucker algorithm.

Chapter 14

Reporting Problems

14.1 Reporting Software Bugs

Reporting bugs effectively is a fundamental way to help PostGIS development. The most effective bug report is that enabling PostGIS developers to reproduce it, so it would ideally contain a script triggering it and every information regarding the environment in which it was detected. Good enough info can be extracted running `SELECT postgis_full_version()` [for PostGIS] and `SELECT version()` [for postgresql].

If you aren't using the latest release, it's worth taking a look at its release changelog first, to find out if your bug has already been fixed.

Using the PostGIS bug tracker will ensure your reports are not discarded, and will keep you informed on its handling process. Before reporting a new bug please query the database to see if it is a known one, and if it is please add any new information you have about it.

You might want to read Simon Tatham's paper about How to Report Bugs Effectively before filing a new report.

14.2 Reporting Documentation Issues

The documentation should accurately reflect the features and behavior of the software. If it doesn't, it could be because of a software bug or because the documentation is in error or deficient.

Documentation issues can also be reported to the PostGIS bug tracker.

If your revision is trivial, just describe it in a new bug tracker issue, being specific about its location in the documentation.

If your changes are more extensive, a patch is definitely preferred. This is a four step process on Unix (assuming you already have git installed):

Before editing or building the manual, read the repository's documentation development guide. It covers the executable example, generated figure, validation, HTML, and PDF workflows used by the manual.

1. Clone the PostGIS' git repository. On Unix, type:

   git clone https://git.osgeo.org/gitea/postgis/postgis.git

   This will be stored in the directory postgis

2. Make your changes to the documentation with your favorite text editor. On Unix, type (for example):

   vim doc/postgis.xml

   Note that the documentation is written in DocBook XML rather than HTML, so if you are not familiar with it please follow the example of the rest of the documentation.

3. Make a patch file containing the differences from the master copy of the documentation. On Unix, type:

   git diff doc/postgis.xml > doc.patch

4. Attach the patch to a new issue in bug tracker.

Chapter 15

Credits and Acknowledgments

PostGIS is built and sustained by a worldwide community. This chapter records the people, organizations, funders, and upstream projects that have contributed code, documentation, infrastructure, testing, support, and funding.

15.1 Project Steering Committee

The PostGIS Project Steering Committee (PSC) coordinates the general direction, release cycles, documentation, and outreach efforts for the PostGIS project. In addition the PSC provides general user support, accepts and approves patches from the general PostGIS community and votes on miscellaneous issues involving PostGIS such as developer commit access, new PSC members or significant API changes.

Regina Obe CI and website maintenance, Windows production and experimental builds, documentation, alignment of PostGIS with PostgreSQL releases, X3D support, management functions.

Darafei Praliaskouski Index improvements, bug fixing and geometry/geography function improvements, SFCGAL, raster, GitHub curation, and ci maintenance.

Paul Ramsey (Chair) Co-founder of PostGIS project. General bug fixing, geography support, geography and geometry index support (2D, 3D, nD index and anything spatial index), underlying geometry internal structures, GEOS functionality integration and alignment with GEOS releases, alignment of PostGIS with PostgreSQL releases, loader/dumper, and Shapefile GUI loader.

Sandro Santilli Bug fixes and maintenance, ci maintenance, git mirror management, management functions, integration of new GEOS functionality and alignment with GEOS releases, topology support, and raster framework and low level API functions.

15.2 Core Contributors Present

Nicklas Avén Distance function enhancements (including 3D distance and relationship functions) and additions, Tiny WKB (TWKB) output format and general user support

Loïc Bartoletti SFCGAL enhancements and maintenance and ci support

Dan Baston Geometry clustering function additions, other geometry algorithm enhancements, GEOS enhancements and general user support

Martin Davis GEOS enhancements and documentation

Björn Harrtell MapBox Vector Tile, GeoBuf, and Flatgeobuf functions. Gitea testing and GitLab experimentation.

Aliaksandr Kalenik Geometry Processing, PostgreSQL gist, general bug fixing

Laurenţiu Nicola Bug fixes, maintenance, stability, and performance

15.3 Core Contributors Past

Raúl Marín Rodríguez Prior PSC Member, MVT support, Bug fixing, Performance and stability improvements, GitHub curation, alignment of PostGIS with PostgreSQL releases

Bborie Park Prior PSC Member. Raster development, integration with GDAL, raster loader, user support, general bug fixing, testing on various OS (Slackware, Mac, Windows, and more)

Mark Cave-Ayland Prior PSC Member. Coordinated bug fixing and maintenance effort, spatial index selectivity and binding, loader/dumper, and Shapefile GUI Loader, integration of new and new function enhancements.

Jorge Arévalo Raster development, GDAL driver support, loader

Olivier Courtin (Emeritus) Input/output XML (KML,GML)/GeoJSON functions, 3D support and bug fixes.

Chris Hodgson Prior PSC Member. General development, site and buildbot maintenance, OSGeo incubation management

Mateusz Loskot CMake support for PostGIS, built original raster loader in python and low level raster API functions

Kevin Neufeld Prior PSC Member. Documentation and documentation support tools, buildbot maintenance, advanced user support on PostGIS newsgroup, and PostGIS maintenance function enhancements.

Dave Blasby The original developer/Co-founder of PostGIS. Dave wrote the server side objects, index bindings, and many of the server side analytical functions.

Jeff Lounsbury Original development of the Shapefile loader/dumper.

Mark Leslie Ongoing maintenance and development of core functions. Enhanced curve support. Shapefile GUI loader.

Pierre Racine Architect of PostGIS raster implementation. Raster overall architecture, prototyping, programming support

David Zwarg Raster development (mostly map algebra analytic functions)

15.4 PostGIS Raster Origins

PostGIS raster discussions date to 2002 and included early PGCHIP experiments. Xing Lin developed the **geotiff2pgraster** PGRaster prototype during Google Summer of Code 2007. The WKT Raster project began in 2008, Jorge Arévalo developed the first GDAL driver prototype during Google Summer of Code 2009, and the 2010 FOSS4G code sprint agreed to make WKT Raster the PostGIS raster type. Raster support shipped as part of PostGIS 2.0 in 2012.

The early implementation team included Pierre Racine, Mateusz Loskot, Sandro Santilli, Jorge Arévalo, Bborie Park, David Zwarg, and Regina Obe. Their work established the raster type, loader, GDAL integration, raster/vector operations, map algebra, documentation, and testing that became the current raster subsystem.

Other recorded contributions include Jeff Adams's early sprint work, Qing Liu's raster distance analysis, Maurício Carvalho Mathias de Paulo's QGIS raster plugin and related paper with Lubia Vinhas, and Ignacio Brodin's gvSIG raster plugin and presentation with Jorge Sanz. Bryce L. Nordgren created early raster coverage and seamless-architecture diagrams. Pierre Racine wrote the original raster/vector tutorial and created its walkthrough and early API diagrams.

The historical project planning record credits Steve Cumming and Université Laval with funding Pierre Racine's work from June 2008 through July 2012; Martin Daly and Cadcorp with funding about six months of Mateusz Loskot's work through June 2009; Tyler Erickson and Michigan Tech Research Institute with a $1,200 USD contribution; Miguel Lizondo and Deimos Space with support for about two years of Jorge Arévalo's work through June 2011; and the Center for Vectorborne Diseases at UC Davis with funding Bborie Park's work from March 2010 through July 2012.

15.5 Other Contributors

Individual Contributors

@amwps290 (GitHub user)
dovkoy (GitHub user)
@kannes (GitHub user: Hannes)
@sunhm89 (GitHub user)
@wewindy (GitHub user: Chen)
Adrien Berchet
Alex Bodnaru
Alex Mayrhofer
Andrea Peri
Andreas Forø Tollefsen
Andreas Neumann
Andreas Schild
Andrew Gierth
Andrew Harvey
Andrew Kroh
Anne Ghisla
Antoine Bajolet
Ariel Mashraki
Arno Renevier
Arthur Bazin
Arthur Lesuisse
Artur Zakirov
Ayo Adesugba
Barbara Phillipot
Baris Ergun
Bas Couwenberg
Ben Jubb
Benjamin Morel
Bernd Reiß
Bernhard Reiter
Bilal Kahraman
Bill Mill
Björn Esser
Brian Hamlin
Bruce Rindahl
Bruno Wolff III
Bryce L. Nordgren
Carl Anderson
Carlos López Quintanilla
Charles Karney
Charlie Savage
Chris Mayo
Christian Schroeder
Christoph Berg
Christoph Moench-Tegeder
Dan Faudemer
Dan Levy
Dane Springmeyer
Daniel Nylander
Dapeng Wang
Daryl Herzmann
Dave Fuhry
David G. Johnston
David Garnier
David Skea
David Techer
Dennis Tighe
Denys Kovshun
Devrim Gündüz
Dian M Fay
Dmitry Vasilyev
Edouard Choinière
Eduin Carrillo
Egor Ivkov
Fabrice Fontaine
Florentin Drogeanu
Florian Weimer
François Bonzon
Frank Warmerdam
George Silva
Gerald Fenoy
Gino Lucrezi
Giovanni Zezza
Grant Hutchins
Greg Troxel
Guillaume Lelarge
Giuseppe Broccolo
Han Wang
Hannes Janetzek
Hans Lemuet
Haribabu Kommi
Harrissou Sant-anna
Havard Tveite
Ignacio Brodin
IIDA Tetsushi
Ingvild Nystuen
Jackie Leng
James Addison
James Marca
Jan Katins
Jan Tojnar
Jason Smith
Jean Felder
Jeff Adams
Jelte Fennema
Jim Jones
Joe Conway
John Carmack
Jonne Savolainen
Jorge Gustavo Rocha
Jorge Sanz
Jose Carlos Martinez Llari
Jostein Leira
Jörg Habenicht
Juan Sebastian Jimenez Aldana
Juergen E. Fischer
Julien Monticolo
Julien Rouhaud
Kashif Rasul
Klaus Foerster
Kris Jurka
Kristian Thy
Lars Aksel Opsahl
Laurenţiu Nicola
Laurenz Albe
Lars Roessiger
Lauri Kajan
Leo Hsu
Loic Dachary
Louis Descoteaux
Luca Delucchi
Luca S. Percich
Lucas C. Villa Real
Ludovic Hirlimann
Lubia Vinhas
Maksim Korotkov
Manuel Morillo
Maria Arias de Reyna
Matt Bret
Matthias
Maurício
Maxim Ko
Maxim Ko
Maxime G
Maxime v
Maxime S
Megan Ma
Mehmet I
Michael F
Michal Pá
Mike Toev
Mitchell H
Nathan W
Nathaniel
Nicolas R
Nikita Shu
Nikolai Be
Norman B
Norman V
Patricia T
Paul Norm
Paulo Ces
Pavel Sele
Pawel Ost
Peter Ngu
Pierre Ca
Qing Liu
Rafal Mag
Ralph Ma
Rémi Cura
Richard G
Robert Co
Rodrigo S
Roger Cre
Ron Maye
Sam Peter
Sarath Ku
Sergei Sh
Sergey Fe
Shinichi S
Shoaib Bu
Silvio Gro
Simon Gre
Stefan Co
Steffen M
Stepan Ku
Stephen F
Stephen W
Steven Ot
Svyatoslav
Talha Rizv
Temuri Do
Teramoto
Thomas M
Thomas P
Tobias Bu
Tom Glanc
Tom van T
Victor Col
Vincent Bi
Vincent M
Vincent Pi

Corporate Sponsors These are corporate entities that have contributed developer time, hosting, or direct monetary funding to the PostGIS project. In alphabetical order:

- Aiven
- Arrival 3D
- Associazione Italiana per l'Informazione Geografica Libera (GFOSS.it)
- AusVet
- Avencia
- Azavea
- Boundless
- Cadcorp
- Camptocamp
- Carto
- Crunchy Data
- City of Boston (DND)
- City of Helsinki
- Clever Elephant Solutions
- Cooperativa Alveo
- Deimos Space
- Faunalia
- Geographic Data BC
- Google
- HighGo
- Hunter Systems Group
- IITM Pravartak Security Team
- INIA-CSIC
- ISciences, LLC
- Jirotech (formerly LISAsoft)
- Koordinates
- Kontur
- Lidwala Consulting Engineers
- Logical Tracking & Tracing International AG
- Maponics
- Maumap
- Michigan Tech Research Institute
- Natural Resources Canada
- Norwegian Forest and Landscape Institute
- Norwegian Institute of Bioeconomy Research (NIBIO)
- OSGeo
- Oslandia
- Palantir Technologies
- Paragon Corporation
- PlanetScale
- Postgres Pro

- R3 GIS
- Refractions Research
- Regione Toscana - SITA
- Safe Software
- Sirius Corporation plc
- Stadt Uster
- UC Davis Center for Vectorborne Diseases
- Université Laval
- U.S. Census Bureau
- U.S. Department of State (HIU)
- Zonar Systems

Crowd Funding Campaigns Crowd funding campaigns are campaigns we run to get badly wanted features funded that can service a large number of people. Each campaign is specifically focused on a particular feature or set of features. Each sponsor chips in a small fraction of the needed funding and with enough people/organizations contributing, we have the funds to pay for the work that will help many. If you have an idea for a feature you think many others would be willing to co-fund, please post to the PostGIS newsgroup your thoughts and together we can make it happen.

PostGIS has experimented with community fundraising for features in the past. We used PledgeBank and we got two successful campaigns out of it.

postgistopology - 10 plus sponsors each contributed $250 USD to build toTopoGeometry function and beef up topology support in 2.0.0. It happened.

postgis64windows - 20 someodd sponsors each contributed $100 USD to pay for the work needed to work out PostGIS 64-bit issues on windows. It happened.

External Libraries and Platforms PostgreSQL, the extensible database system on which PostGIS is built. PostgreSQL provides the SQL engine, transactions, query planner, extension framework, and index access methods used by PostGIS.

The GEOS geometry operations library.

The PROJ coordinate transformation and geodesic calculation library.

The GDAL Geospatial Data Abstraction Library used to power much of the raster functionality introduced in PostGIS 2. In kind, improvements needed in GDAL to support PostGIS are contributed back to the GDAL project.

LibXML2 for XML-based spatial formats, and JSON-C for GeoJSON input.

The GNU Multiple Precision Arithmetic Library (GMP) for precision-sensitive topology operations.

protobuf-c for Mapbox Vector Tile and Geobuf output.

The iconv API, supplied by the C library or GNU libiconv, for character encoding conversion in the shapefile loader.

SFCGAL for advanced 2D and 3D functions, together with the CGAL algorithms that SFCGAL exposes.

Bundled Libraries and Source Code PostGIS ships selected source from FlatGeobuf and FlatBuffers to support FlatGeobuf input and output.

Wagyu and Mapbox Geometry provide polygon clipping and validation for Mapbox Vector Tile output.

Ryu provides round-trip-safe floating-point formatting, and uthash provides in-process hash tables.

The shapefile loader and dumper include code derived from ShapeLib, originally by Frank Warmerdam and Even Rouault. They also include public-domain `getopt` code from AT&T.

`liblwgeom/lookup3.c` contains the public-domain `lookup3` hash function by Bob Jenkins.

Appendix A

Release Notes

A.1 PostGIS 3.7.0

This version requires GEOS 3.10+, PostgreSQL 14-19beta3, Proj 6.1+, and libgmp. To take advantage of all postgis extension features, GEOS 3.15+ is needed. To take advantage of all postgis_sfcgal extension features SFCGAL 2.3+ is needed. Python 3 is a new requirement to build the documentation.

Visual examples in the docs are off by default. To build docs with them, configure with `--with-visual-examples` and make sure you have a running PostgreSQL instance; they are then generated automatically when building the docs (or explicitly with `make -C regress visual-examples`).

A.1.1 Breaking Changes

#5688, [topology] topology building functions now interpret tolerance 0 (old default) to really mean 0 and want -1 (new default) to mean ”use topology precision” (Sandro Santilli)

#6024, Remove support for PostgreSQL 12 and 13 (Regina Obe)

#6040, ST_GeoHash round trip through ST_GeomFromGeoHash (Paul Ramsey)

#6053, [address_standardizer] Extension removed and moved to https://github.com/postgis/address_standa (Paul Ramsey)

#6052, [tiger_geocoder] Extension removed and moved to https://git.osgeo.org/postgis/postgis_tiger_geoco (Regina Obe)

#6079, Remove support for GEOS < 3.10 (Sandro Santilli)

#1793, Remove ST_MemCollect aggregate (Darafei Praliaskouski)

GH-848, CG_AlphaShape now returns a MultiPolygon (Jean Felder)

[docs] Python 3 and an SVG rasterizer (rsvg-convert preferred, GraphicsMagick or ImageMagick supported) now required for doc building (Darafei Praliaskouski)

A.1.2 New Features

#1124, #2935, #3033, #4658, #4659, [loader] Rework loader arguments: shp2pgsql can create UNLOGGED tables and choose the feature id column name, shp2pgsql --drop-table can emit DROP TABLE before prepare output, raster2pgsql/shp2pgsql expose loader actions as long options, both loaders accept long aliases for existing options, and

-if-not-exists makes creation actions idempotent (Darafei Praliaskouski)

#4208, Add single-geometry variants of ST_MaxDistance and ST_LongestLine (Darafei Praliaskouski)

[topology] FindVertexSegmentPairsBelowDistance function (Sandro Santilli)

ST_CoverageEdges, returns MultiLinestring of distinct shared edges in polygonal coverage (Paul Ramsey)

ST_MinimumSpanningTree, window function to calculate MST (Paul Ramsey)

#5993, [topology] Add max_edges parameter to TopoGeo_AddLinestring (Sandro Santilli)

#6001, support MultiLineString in ST_MakeLine (Paul Ramsey)

#2858, ST_MMin and ST_MMax (Paul Ramsey)

#5941, [raster] Support GDT_Float16 pixel type (Darafei Praliaskouski)

#5992, Optimize GiST index for repeated edge subdivision in topology (Darafei Praliaskouski)

#5702, Allow the compiler to detect the parallelism -flto=auto (Darafei Praliaskouski)

#4798, ST_AsGeoJSON warns about duplicate property keys (Darafei Praliaskouski)

#4222, [raster] ST_DumpAsPolygons honours PostgreSQL interrupts (Darafei Praliaskouski)

#5889, [topology] Include representative locations in topology build errors (Darafei Praliaskouski)

#2614, Use GEOSPreparedDistance and caching to accelerate ST_DWithin (Paul Ramsey)

GH-839, ST_Multi support for TIN and surfaces (Loïc Bartoletti)

#4398, Add ST_CatmullSmoothing smoothes but retains original vertices (Paul Ramsey)

#4560, ST_3DInterpolatePoint for M interpolation from XYZ inputs (Paul Ramsey)

#1291, ST_MakePolygon support for curved rings (Darafei Praliaskouski)

GT-270, Add NURBSCurve (Loïc Bartoletti)

#2863, Add ST_XSize, ST_YSize, ST_ZSize, and ST_MSize dimension helpers (Darafei Praliaskouski)

GT-806, #6124 --with-visual-examples opt-in configure switch to build dynamic images (Regina Obe)

A.1.3 Enhancements

#2045, pgsql2shp query dumps no longer require temporary table privileges (Darafei Praliaskouski)

#1577, [loader] Report unsupported .zip archive input before trying to open shapefile sidecar files (Darafei Praliaskouski)

#3158, Move geometry and geography typmod bit helpers out of the public liblwgeom header (Darafei Praliaskouski)

#3776, [raster] Keep distinct 32-bit integer pixel values separate in ST_DumpAsPolygons (Darafei Praliaskouski)

#4678, [raster] Move st_sum4ma neighborhood callback to C (Darafei Praliaskouski)

#2116, [raster] Add ST_Value nearest-neighbor boundary options (Darafei Praliaskouski)

#2804, #4315, [raster] Support two-argument ST_MapAlgebra callbacks and pass callback call data as actual arguments (Darafei Praliaskouski)

#6062, [topology] Stop using recursive snapping, for improved robustness (Sandro Santilli)

#6065, Improved winding order computation robustness for rings having collapsed elements (Sandro Santilli)

#2137, [raster] Reuse matching source raster overviews in raster2pgsql -l (Darafei Praliaskouski)

#4749, Use point-in-polygon predicate fast paths for point-only GeometryCollections (Darafei Praliaskouski)

#2807, [raster] Preserve missing NODATA values in ST_MapAlgebra outputs (Darafei Praliaskouski)

#2832, [raster] Avoid padding single-tile raster2pgsql overviews when padding is not requested (Darafei Praliaskouski)

#2623, Generate a pgsql2shp GID field when exporting geometry-only shapefiles (Darafei Praliaskouski)

#2850, Preserve pgsql2shp numeric precision and scale in DBF metadata (Darafei Praliaskouski)

Build PostgreSQL extension modules with `-fno-semantic-interposition` when available so LTO can optimize same-DSO calls to exported PostGIS entry points directly instead of routing through the module PLT; in a local PostgreSQL 18 `pgbench` microbenchmark over 100k four-point groups this reduced `ST_Collect` aggregate latency by about 4% (Darafei Praliaskouski)

#5973, Install PostGIS utilities under `pg_config --bindir` by default and document `pg_config`-derived install paths more accurately (Darafei Praliaskouski)

#2808, Clarify postgis_full_version() output when installed component scripts are newer than core scripts (Darafei Praliaskouski)

#2838, Emit valid X3D Coordinate nodes for 2D polygon output (Darafei Praliaskouski)

GH-899, [raster] Honor PostgreSQL interrupts in long-running GDAL progress callbacks (Darafei Praliaskouski)

#6073, #6074, Stabilize geography LRS endpoint interpolation and pole centroid regressions on 32-bit systems (Darafei Praliaskouski)

#2583, Preserve Z/ZM dimensional qualifiers in ST_AsEWKT output for TIN (Darafei Praliaskouski)

#4616, Detect pgsql2shp query keywords case-insensitively (Darafei Praliaskouski)

#1986, ST_GeomFromGML support for GML ArcString and curved polygon rings (Darafei Praliaskouski)

#4749, Use point-in-polygon predicate fast paths for point-only GeometryCollections, including ST_Disjoint (Darafei Praliaskouski)

#3676, Mark geometry and geography btree comparison support functions as leakproof (Darafei Praliaskouski)

GH-875, Allow ST_OrderingEquals joins to use hash and merge join plans (Darafei Praliaskouski)

#1168, [raster] Support curved geometry inputs in ST_AsRaster (Darafei Praliaskouski)

#1416, ST_AsX3D outputs NURBSCURVE as X3D NurbsCurve and linearizes other curve geometries (Darafei Praliaskouski)

#3705, ST_AsX3D outputs valid coordinate nodes for 2D multi-geometries and GeometryCollections (Darafei Praliaskouski)

GT-484, Use column-level SELECT privileges for metadata views (Arthur Bazin)

#1408, #2386, #2557, #2805, #2898, #3743, #4323, #4332, #4385, #5109, #5532, #5593, #5638, #5645, #5950, #6022, #6082, GH-1161, GT-584, Numerous doc enhancements (Darafei Praliaskouski)

A.1.4 Bug Fixes

GH-1175, Guard missing operands before reading node tags in spatial selectivity estimation (Maksim Korotkov)

#6118, Reject empty geometry type modifiers without an out-of-bounds read (Egor Ivkov)

Fix WKB and TWKB parser resource exhaustion on malformed input (Darafei Praliaskouski)

#6038, Avoid stale relation lookups in geometry_columns after topology objects are dropped (Darafei Praliaskouski)

#5487, Improve error detail for repeated upgrades after an incomplete deprecated-function cleanup (Darafei Praliaskouski)

#5975, Initialize skipped union-find cluster ids deterministically (Darafei Praliaskouski)

#5904, [topology] Avoid MinGW format warnings when logging topology ids (Darafei Praliaskouski)

GH-888, [sfcgal] Avoid stale detoasted geometry access in CG_Visibility empty-input handling for SFCGAL < 2.2 (Darafei Praliaskouski)

#3179, Mark truncated libpgcommon PostgreSQL messages instead of silently dropping the tail (Darafei Praliaskouski)

#6003, Avoid size_t formats in liblwgeom logging/error wrappers for MinGW builds (Darafei Praliaskouski)

GH-886, ST_ClusterRelateWin validates NULL relate matrices before detoasting them (Darafei Praliaskouski)

GH-889, ST_Relate pattern inversion avoids out-of-bounds access on short DE-9IM patterns (Darafei Praliaskouski)

GH-890, Avoid GIDX BRIN summary merges across mismatched dimensions (Darafei Praliaskouski)

#5959, Prevent histogram target overflow when analysing massive tables (Darafei Praliaskouski)

#6080, Avoid out-of-bounds hex WKB lookup for high-bit input bytes (Darafei Praliaskouski)

#4828, geometry_columns handles NOT VALID SRID checks without errors (Darafei Praliaskouski)

#6048, [raster] ST_Clip no longer crashes when clipping sparse band selections (Darafei Praliaskouski)

GH-894, [raster] Fix Float16 import, mixed nodata warping, band replacement cleanup, GDAL credential redaction, and Python GDAL compatibility (Darafei Praliaskouski)