Introduction to PostGIS

Release 1.1

Mark Leslie, Paul Ramsey, et. al

March 26, 2024

CONTENTS

CHAPTER

ONE

WELCOME

1.1 Workshop Conventions

These sections conform to a number of conventions to make it easier to follow the conversation. This section gives a brief overview of what to expect in the way of typographic conventions, as well as a short overview of the structure of each workbook.

1.1.1 Directions

Directions for you, the workshop attendee, will be noted by **bold** font.

For example:

> Click **Next** to continue.

1.1.2 Code

SQL query examples will be displayed in an offset box

```
SELECT postgis_full_version();
```

These examples can be entered into the query window or command line interface.

1.1.3 Notes

Notes are used to provide information that is useful but not critical to the overall understanding of the topic.

Note: If you haven't eaten an apple today, the doctor may be on the way.

1.1.4 Functions

Where function names are defined in the text, they will be rendered in a **bold** font.

For example:

> **ST_Touches(geometry A, geometry B)** returns TRUE if either of the geometries' boundaries intersect

1.1.5 Files, Tables and Column Names

File names, paths, table names and column names will be shown in `fixed-width` font.

For example:

> Select the `name` column in the `nyc_streets` table.

1.1.6 Menus and Form elements

Menus/submenus and form elements such as fields or check boxes and other on-screen artifacts are displayed in *italics*.

For example:

> Click on the *File* > *New* menu. Check the box that says *Confirm*.

1.1.7 Workflow

Sections are designed to be progressive. Each section will start with the assumption that you have completed and understood the previous section in the series and will build on that knowledge. A single section will progress through a handful of ideas and provide working examples wherever possible. At the end of a section, where appropriate, we have included a handful of exercises to allow you to try out the ideas we've presented. In some cases the section will include "Things To Try". These tasks contain more complex problems than the exercises and is designed to challenge participants with advanced knowledge.

CHAPTER

TWO

INTRODUCTION

2.1 What is a Spatial Database?

PostGIS is a spatial database. Oracle Spatial and SQL Server (2008 and later) are also spatial databases. But what does that mean; what is it that makes an ordinary database a spatial database?

The short answer, is…

Spatial databases store and manipulate spatial objects like any other object in the database.

The following briefly covers the evolution of spatial databases, and then reviews three aspects that associate *spatial* data with a database – data types, indexes, and functions.

1. **Spatial data types** refer to shapes such as point, line, and polygon;
2. Multi-dimensional **spatial indexing** is used for efficient processing of spatial operations;
3. **Spatial functions**, posed in *SQL*, are for querying of spatial properties and relationships.

Combined, spatial data types, indexes, and functions provide a flexible structure for optimized performance and analysis.

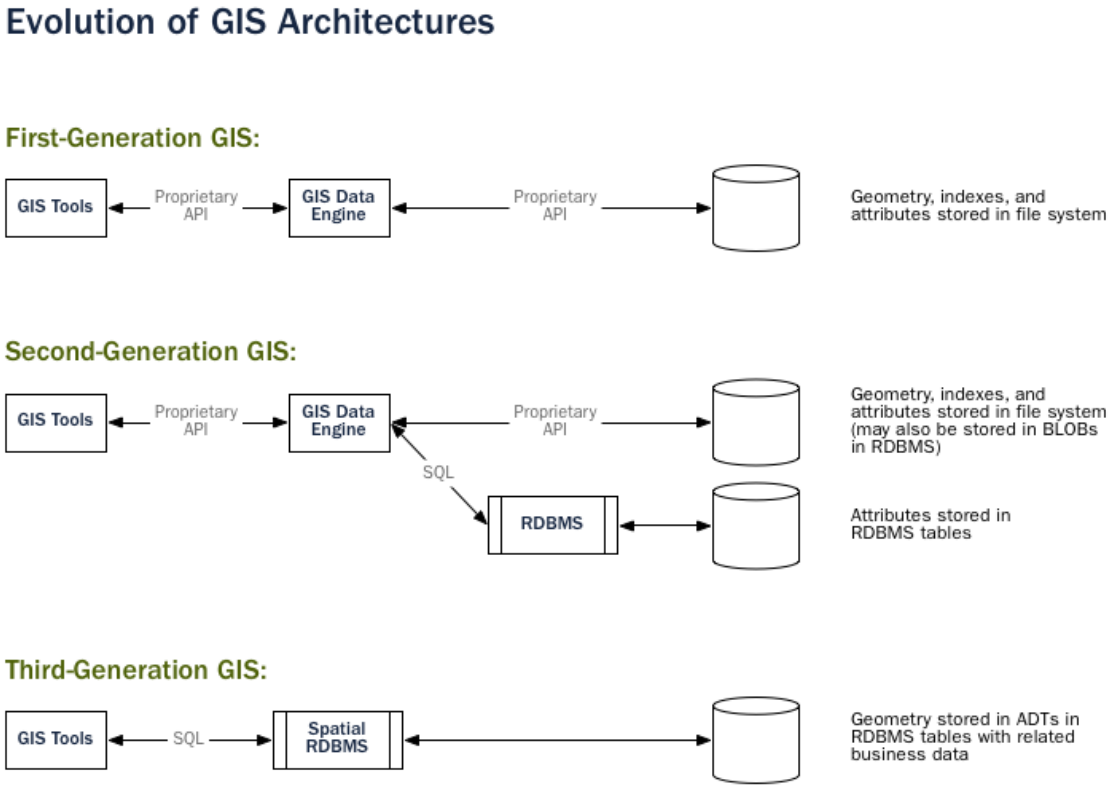
2.1.1 In the Beginning

In legacy first-generation *GIS* implementations, all spatial data is stored in flat files and special *GIS* software is required to interpret and manipulate the data. These first-generation management systems are designed to meet the needs of users where all required data is within the user's organizational domain. They are proprietary, self-contained systems specifically built for handling spatial data.

Second-generation spatial systems store some data in relational databases (usually the "attribute" or non-spatial parts) but still lack the flexibility afforded with direct integration.

True spatial databases were born when people started to treat spatial features as first class database objects.

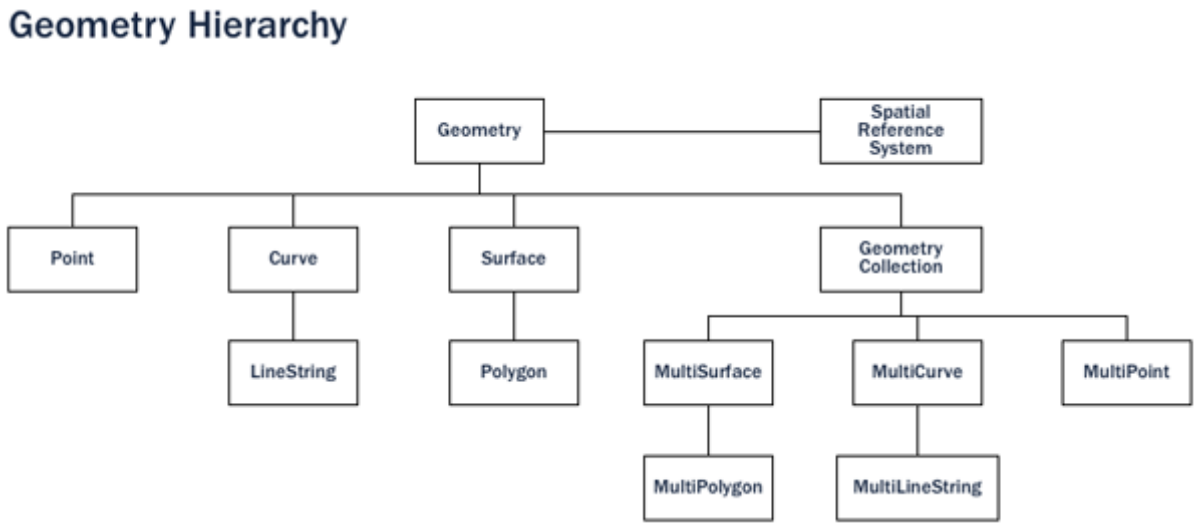
Spatial databases fully integrate spatial data with a relational database. The system orientation changes from GIS-centric to database-centric.



Note: A spatial database management system may be used in applications besides the geographic world. Spatial databases are used to manage data related to the anatomy of the human body, large-scale integrated circuits, molecular structures, and electro-magnetic fields, among others.

2.1.2 Spatial Data Types

An ordinary database has strings, numbers, and dates. A spatial database adds additional (spatial) types for representing **geographic features**. These spatial data types abstract and encapsulate spatial structures such as boundary and dimension. In many respects, spatial data types can be understood simply as shapes.



Spatial data types are organized in a type hierarchy. Each sub-type inherits the structure (attributes) and the behavior (methods or functions) of its super-type.

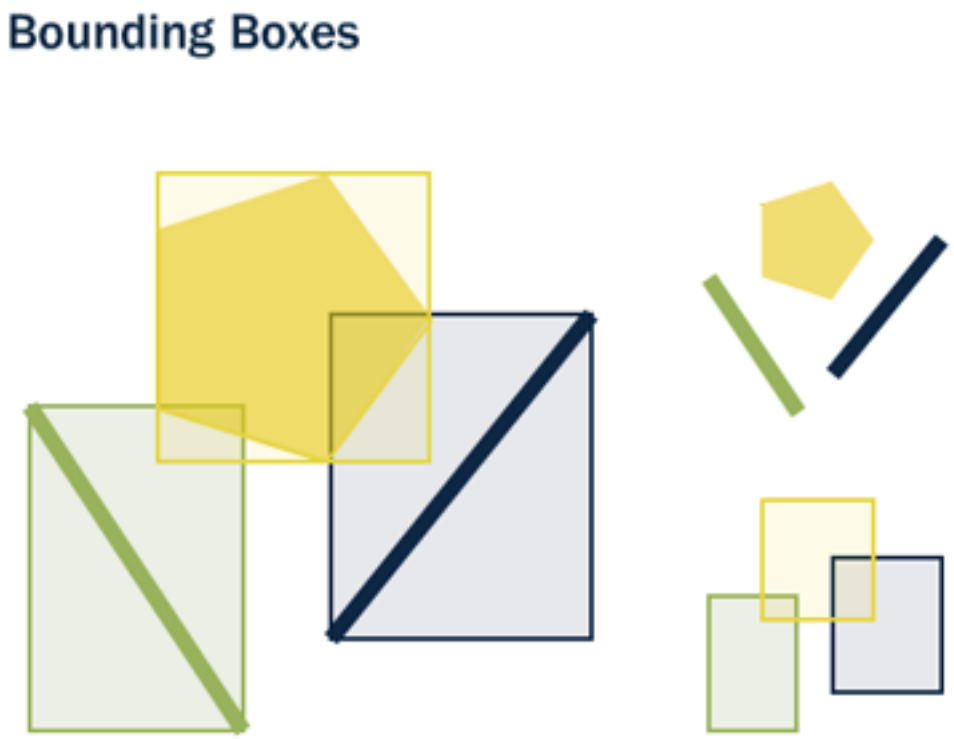
2.1.3 Spatial Indexes and Bounding Boxes

An ordinary database provides **indexes** to allow for fast and random access to subsets of data. Indexing for standard types (numbers, strings, dates) is usually done with B-tree indexes.

A B-tree partitions the data using the natural sort order to put the data into a hierarchical tree. The natural sort order of numbers, strings, and dates is simple to determine – every value is less than, greater than or equal to every other value.

But because polygons can overlap, can be contained in one another, and are arrayed in a two-dimensional (or more) space, a B-tree cannot be used to efficiently index them. Real spatial databases provide a "spatial index" that instead answers the question "which objects are within this particular bounding box?".

A **bounding box** is the smallest rectangle – parallel to the coordinate axes – capable of containing a given feature.



Bounding boxes are used because answering the question "is A inside B?" is very computationally intensive for polygons but very fast in the case of rectangles. Even the most complex polygons and linestrings can be represented by a simple bounding box.

Indexes have to perform quickly in order to be useful. So instead of providing exact results, as B-trees do, spatial indexes provide approximate results. The question "what lines are inside this polygon?" will be instead interpreted by a spatial index as "what lines have bounding boxes that are contained inside this polygon's bounding box?"

The actual spatial indexes implemented by various databases vary widely. The most common implementations are the R-Tree and Quadtree (used in PostGIS), but there are also grid-based indexes and GeoHash indexes implemented in other spatial databases.

2.1.4 Spatial Functions

For manipulating data during a query, an ordinary database provides **functions** such as concatenating strings, performing hash operations on strings, doing mathematics on numbers, and extracting information from dates.

A spatial database provides a complete set of functions for analyzing geometric components, determining spatial relationships, and manipulating geometries. These spatial functions serve as the building block for any spatial project.

The majority of all spatial functions can be grouped into one of the following five categories:

1. **Conversion**: Functions that *convert* between geometries and external data formats.
2. **Management**: Functions that *manage* information about spatial tables and PostGIS administration.
3. **Retrieval**: Functions that *retrieve* properties and measurements of a Geometry.
4. **Comparison**: Functions that *compare* two geometries with respect to their spatial relation.

5. **Generation**: Functions that *generate* new geometries from others.

The list of possible functions is very large, but a common set of functions is defined by the *OGC SFSQL* and implemented (along with additional useful functions) by PostGIS.

2.2 What is PostGIS?

PostGIS turns the PostgreSQL Database Management System into a spatial database by adding support for the three features: spatial types, spatial indexes, and spatial functions. Because it is built on PostgreSQL, PostGIS automatically inherits important "enterprise" features as well as open standards for implementation.

2.2.1 But what is PostgreSQL?

PostgreSQL is a powerful relational database management system (RDBMS). It is released under a BSD-style license and is thus free and open source software. As with many other open source programs, PostgreSQL is not controlled by any single company, but has a global community of developers and companies to develop it.

PostgreSQL was designed from the very start with type extension in mind – the ability to add new data types, functions and indexes at run-time. Because of this, the PostGIS extension can be developed by a separate development team, yet still integrate very tightly into the core PostgreSQL database.

Why choose PostgreSQL?

A common question from people familiar with open source databases is, "Why wasn't PostGIS built on MySQL?".

PostgreSQL has:

- Proven reliability and transactional integrity by default (ACID)
- Careful support for SQL standards (full SQL92)
- Pluggable type extension and function extension
- Community-oriented development model
- No limit on column sizes ("TOAST"able tuples) to support big GIS objects
- Generic index structure (GiST) to allow R-Tree index
- Easy to add custom functions

Combined, PostgreSQL provides a very easy development path to add new spatial types. In the proprietary world, only Illustra (now Informix Universal Server) allowed such easy extension. This is no coincidence; Illustra is a proprietary re-working of the original PostgreSQL code base from the 1980's.

Because the development path for adding types to PostgreSQL was so straightforward, it made sense to start there. When MySQL released basic spatial types in version 4.1, the PostGIS team took a look at their code, and the exercise reinforced the original decision to use PostgreSQL.

Because MySQL spatial objects had to be hacked on top of the string type as a special case, the MySQL code was spread over the entire code base. Development of PostGIS 0.1 took under a month. Doing a "MyGIS" 0.1 would have taken a lot longer, and as such, might never have seen the light of day.

2.2.2 Why not files?

The Shapefile (and other formats like the Esri File Geodatabase and the GeoPackage) have been a standard way of storing and interacting with spatial data since GIS software was first written. However, these "flat" files have the following disadvantages:

- **Files require special software to read and write.** SQL is an abstraction for random data access and analysis. Without that abstraction, you will need to write all the access and analysis code yourself.
- **Concurrent users can cause corruption and slowdowns.** While it's possible to write extra code to ensure that multiple writes to the same file do not corrupt the data, by the time you have solved the problem and also solved the associated performance problem, you will have written the better part of a database system. Why not just use a standard database?
- **Complicated questions require complicated software to answer.** Complicated and interesting questions (spatial joins, aggregations, etc) that are expressible in one line of SQL in the database take hundreds of lines of specialized code to answer when programming against files.

Most users of PostGIS are setting up systems where multiple applications will be expected to access the data, so having a standard SQL access method simplifies deployment and development. Some users are working with large data sets; with files, they might be segmented into multiple files, but in a database they can be stored as a single large table.

In summation, the combination of support for multiple users, complex ad hoc queries, and performance on large data sets are what sets spatial databases apart from file-based systems.

2.2.3 A brief history of PostGIS

In the May of 2001, Refractions Research released the first version of PostGIS. PostGIS 0.1 had objects, indexes and a handful of functions. The result was a database suitable for storage and retrieval, but not analysis.

As the number of functions increased, the need for an organizing principle became clear. The "Simple Features for SQL" (*SFSQL*) specification from the Open Geospatial Consortium provided such structure with guidelines for function naming and requirements.

With PostGIS support for simple analysis and spatial joins, Mapserver became the first external application to provide visualization of data in the database.

Over the next several years the number of PostGIS functions grew, but its power remained limited. Many of the most interesting functions (e.g., ST_Intersects(), ST_Buffer(), ST_Union()) were very difficult to code. Writing them from scratch promised years of work.

Fortunately a second project, the "Geometry Engine, Open Source" or GEOS, came along. The GEOS library provides the necessary algorithms for implementing the *SFSQL* specification. By linking in GEOS, PostGIS provided complete support for *SFSQL* by version 0.8.

As PostGIS data capacity grew, another issue surfaced: the representation used to store geometry proved relatively inefficient. For small objects like points and short lines, the metadata in the representation had as much as a 300% overhead. For performance reasons, it was necessary to put the representation on a diet. By shrinking the metadata header and required dimensions, overhead greatly reduced. In PostGIS 1.0, this new, faster, lightweight representation became the default.

Recent releases of PostGIS continue to add features and performance improvements, as well as support for new features in the PostgreSQL core system.

2.2.4 Who uses PostGIS?

For a complete list of case studies, see the PostGIS case studies page.

Institut Geographique National, France

IGN is the national mapping agency of France, and uses PostGIS to store the high resolution topographic map of the country, "BDUni". BDUni has more than 100 million features, and is maintained by a staff of over 100 field staff who verify observations and add new mapping to the database daily. The IGN installation uses the database transactional system to ensure consistency during update processes, and a warm standby system to maintain uptime in the event of a system failure.

RedFin

RedFin is a real estate agency with a web-based service for exploring properties and estimate values. Their system was originally built on MySQL, but they found that moving to PostgreSQL and PostGIS provided huge benefits in performance and reliability.

2.2.5 What applications support PostGIS?

PostGIS has become a widely used spatial database, and the number of third-party programs that support storing and retrieving data using it has increased as well. The programs that support PostGIS include both open source and proprietary software on both server and desktop systems.

The following table shows a list of some of the software that leverages PostGIS:

Open/Free	Closed/Proprietary/Paid services
• Loading/Extracting – Shp2Pgsql – ogr2ogr – Dxf2PostGIS – GeoKettle • Web-Based – Mapserver – GeoServer /geoNode – pg_tileserv – pg_featureserv – Deegree – Carto – QGIS Server – MapGuide Open Source (using FDO) • Desktop – QGIS – OpenJUMP – GRASS – pgAdmin – DBeaver – GvSIG – SAGA – uDig	• Loading/Extracting – Safe FME Desktop Translator/Converter – Dbt • Web-Based – Cadcorp GeognoSIS – ESRI ArcGIS Server / Online • Services / DbaaS – Aiven for PostgreSQL – Amazon RDS / Aurora for PostgreSQL – Carto – Crunchy Bridge – Microsoft Azure for PostgreSQL – Google Cloud SQL for PostgreSQL • Desktop – Cadcorp SIS – ESRI Desktop/Pro – GeoConcept – Global Mapper – Manifold – MapInfo – Microimages TNTmips GIS

CHAPTER

THREE

INSTALLATION

To explore the PostgreSQL/PostGIS database, and learn about writing spatial queries in SQL, we will need some software, either installed locally or available remotely on the cloud.

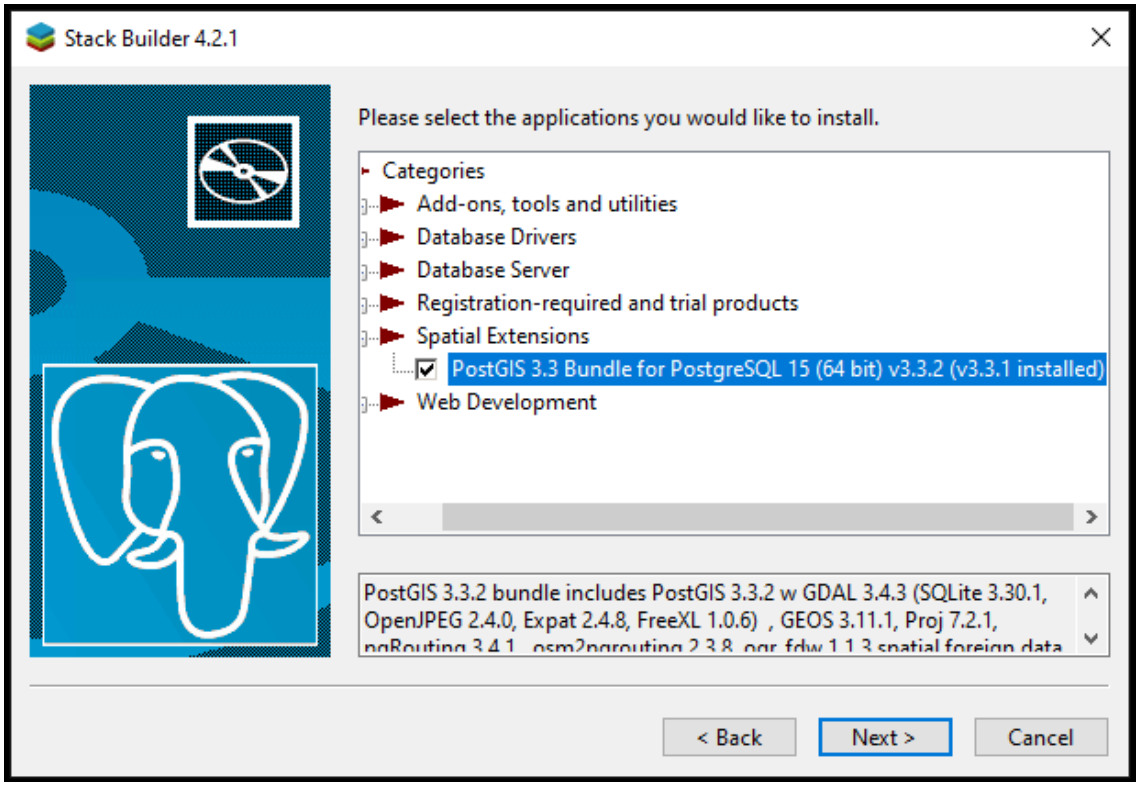
- There are instructions below on how to access PostgreSQL for installation on Windows or MacOS. PostgreSQL for Windows and MacOS either include PostGIS or have an easy way to add it on.
- There are instructions below on how to install PgAdmin. PgAdmin is a graphical database explorer and SQL editor which provides a "user facing" interface to the database engine that does all the world.

For always up-to-date directions on installing PostgreSQL, go to the PostgreSQL download page and select the operating system you are using.

3.1 PostgreSQL for Microsoft Windows

For a Windows install:

1. Go to the Windows PostgreSQL download page.
2. Select the latest version of PostgreSQL and save the installer to disk.
3. Run the installer and accept the defaults.
4. Find and run the "StackBuilder" program that was installed with the database.
5. Select the "Spatial Extensions" section and choose latest "PostGIS ..Bundle" option.

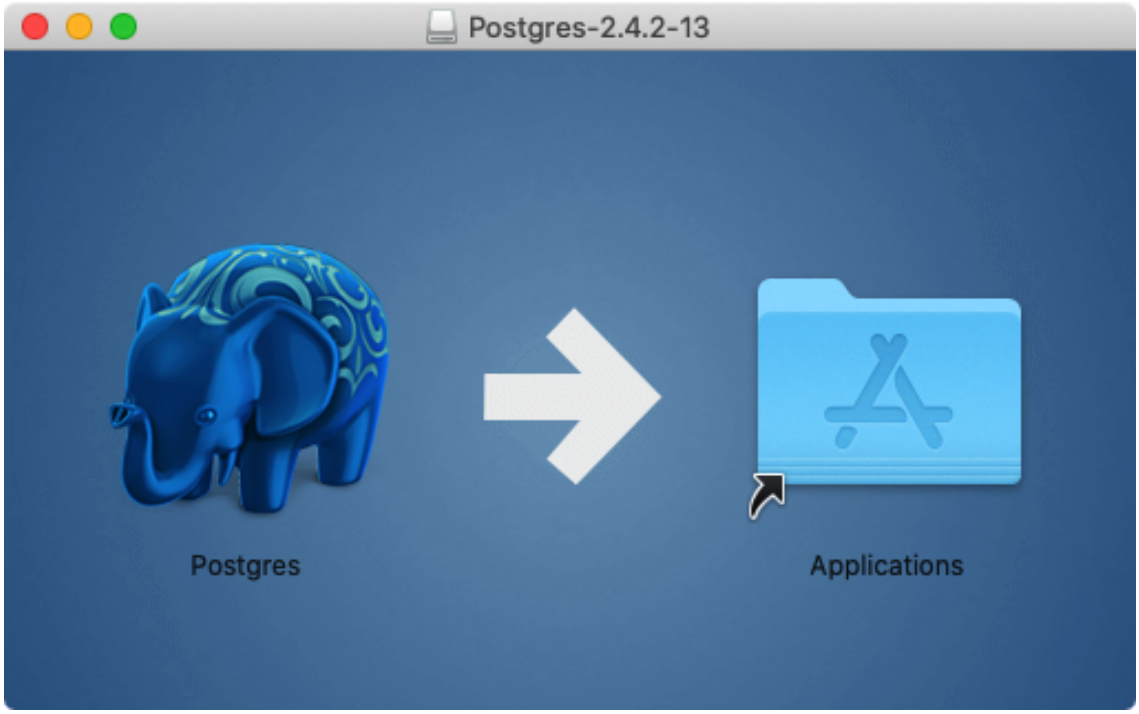


6. Accept the defaults and install.

3.2 PostgreSQL for Apple MacOS

For a MacOS install:

1. Go to the Postgres.app site, and download the latest release.
2. Open the disk image, and drag the **Postgres** icon into the **Applications** folder.



3. In the **Applications** folder, double-click the **Postgres** icon to start the server.
4. Click the **Initialize** button to create a new blank database instance.

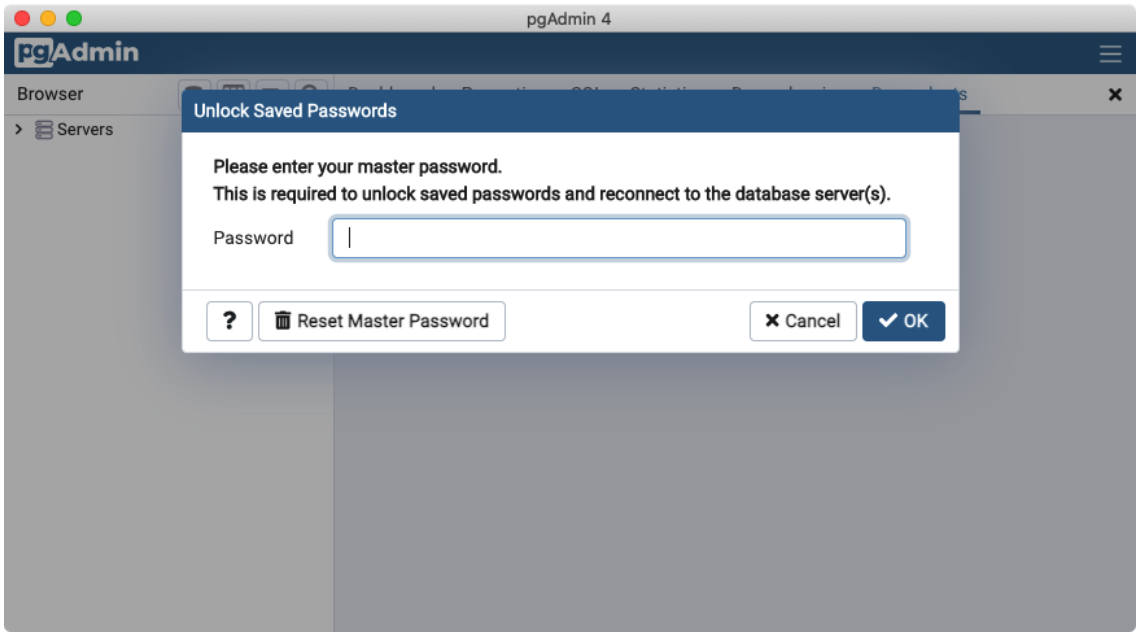
5. In the **Applications** folder, go to the **Utilities** folder and open **Terminal**.
6. Add the command-line utilities to your *PATH* for convenience.

```
sudo mkdir -p /etc/paths.d
echo /Applications/Postgres.app/Contents/Versions/latest/bin |␣
↪sudo tee /etc/paths.d/postgresapp
```

3.3 PgAdmin for Windows and MacOS

PgAdmin is available for multiple platforms, at https://www.pgadmin.org/download/.

1. Download and install the latest version for your platform.
2. Start PgAdmin!



CHAPTER

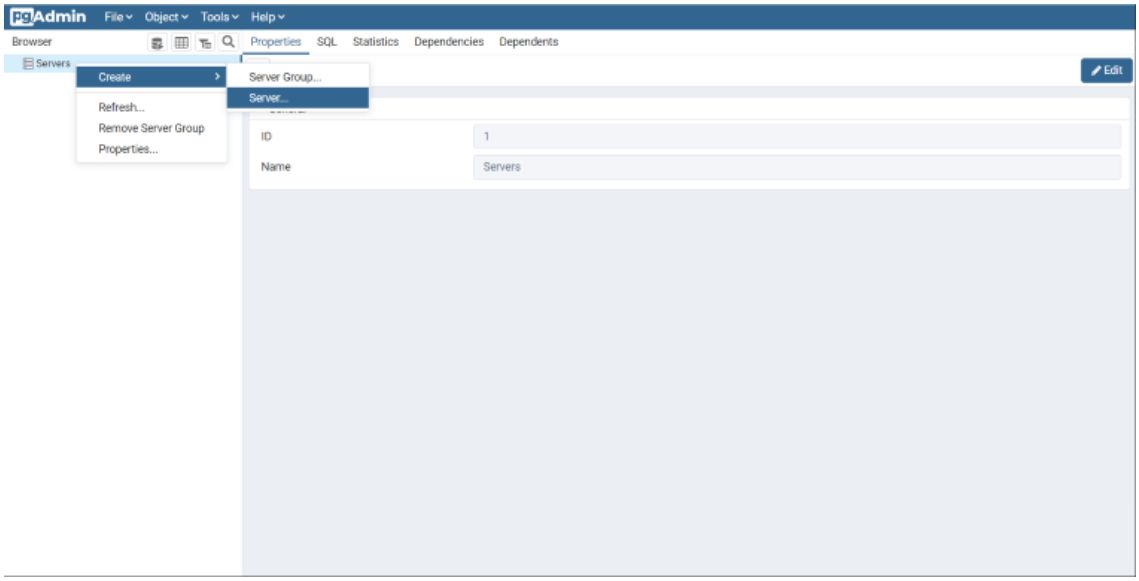
FOUR

CREATING A SPATIAL DATABASE

4.1 PgAdmin

PostgreSQL has a number of administrative front-ends. The primary one is psql, a command-line tool for entering SQL queries. Another popular PostgreSQL front-end is the free and open source graphical tool pgAdmin. All queries done in pgAdmin can also be done on the command line with `psql`. pgAdmin also includes a geometry viewer you can use to spatial view PostGIS queries.

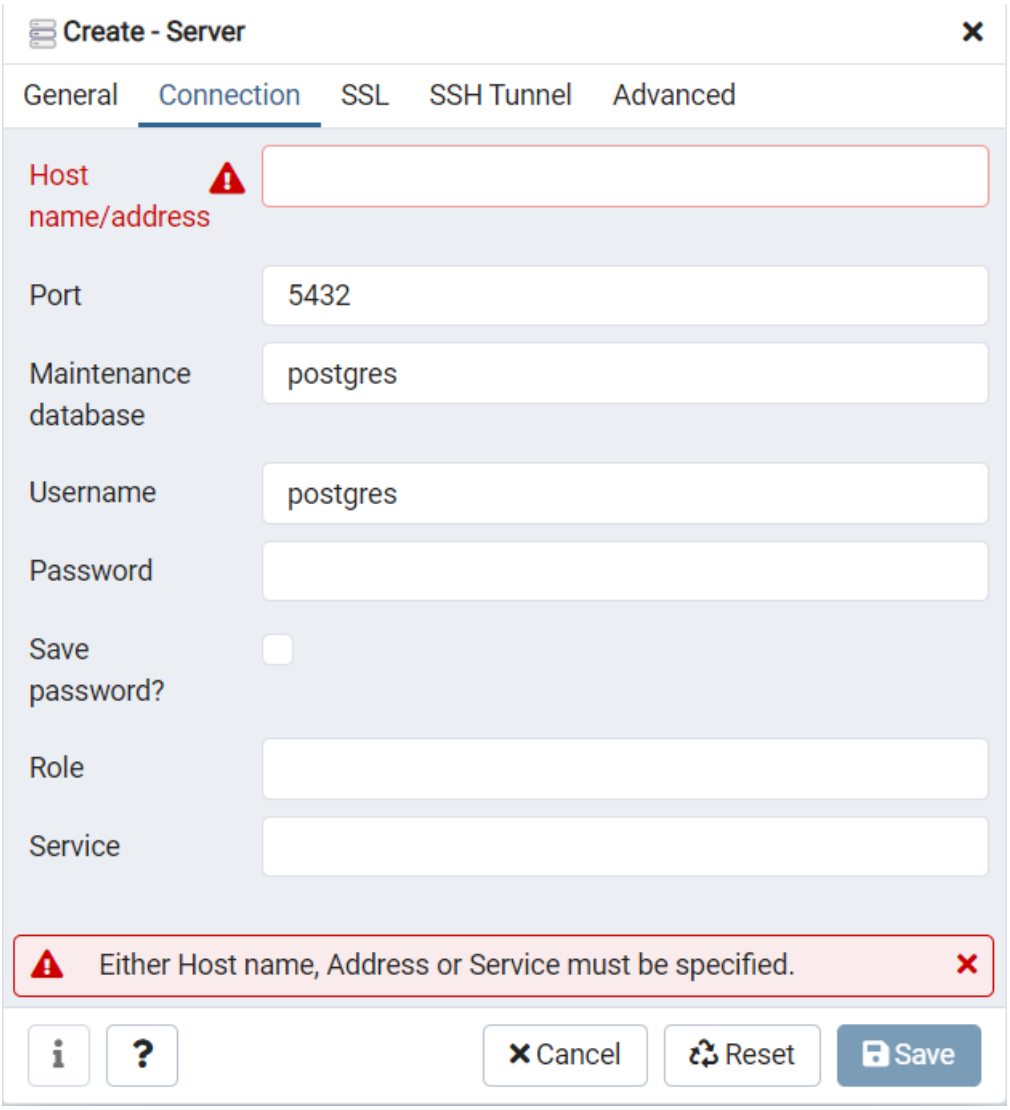
1. Find pgAdmin and start it up.



2. If this is the first time you have run pgAdmin, you probably don't have any servers configured. Right click the `Servers` item in the Browser panel.

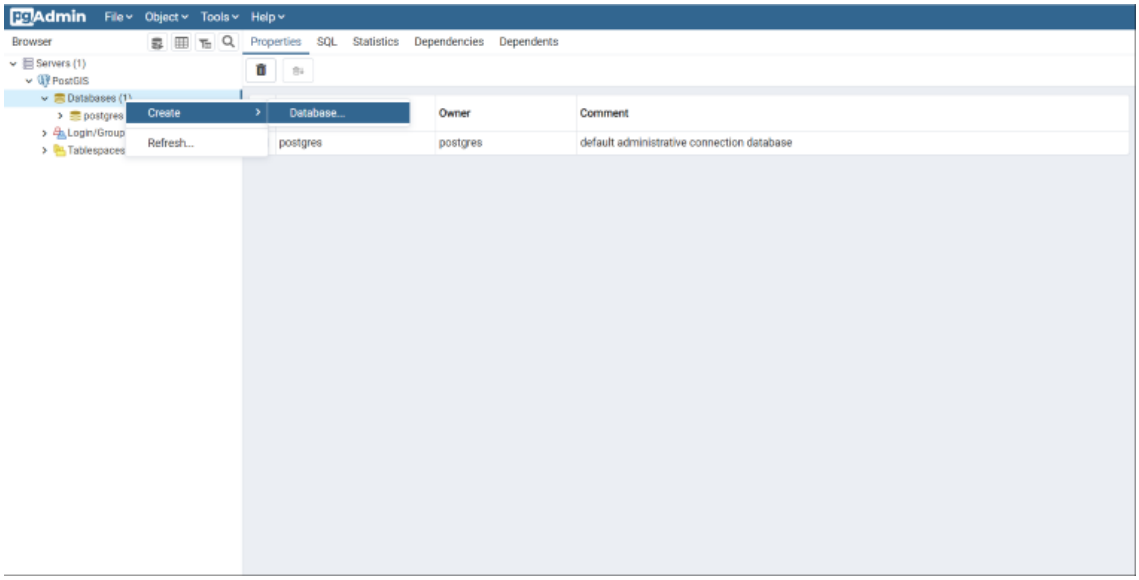
   We'll name our server **PostGIS**. In the Connection tab, enter the `Host name/address`. If you're working with a local PostgreSQL install, you'll be able to use `localhost`. If you're using a cloud service, you should be able to retrieve the host name from your account.

   Leave **Port** set at `5432`, and both **Maintenance database** and **Username** as `postgres`. The **Password** should be what you specified with a local install or with your cloud service.



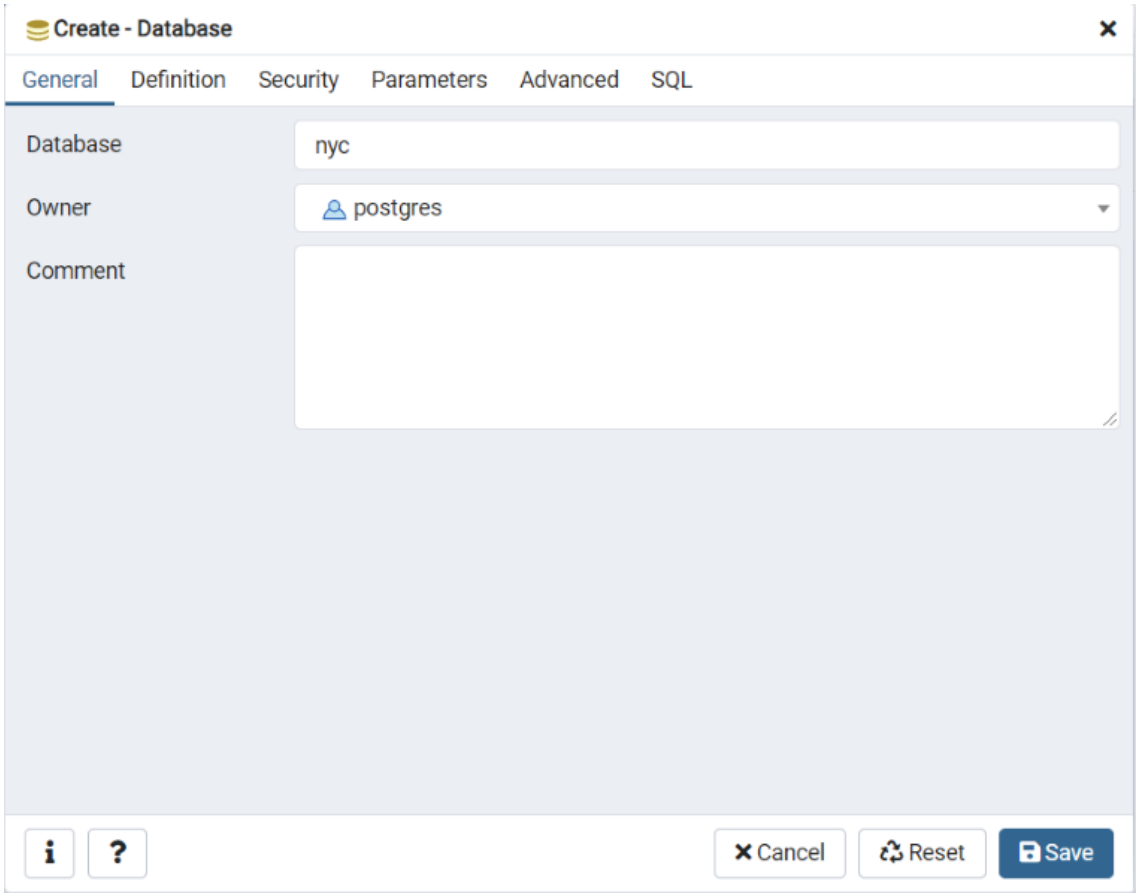
4.2 Creating a Database

1. Open the Databases tree item and have a look at the available databases. The `postgres` database is the user database for the default postgres user and is not too interesting to us.
2. Right-click on the `Databases` item and select `New Database`.

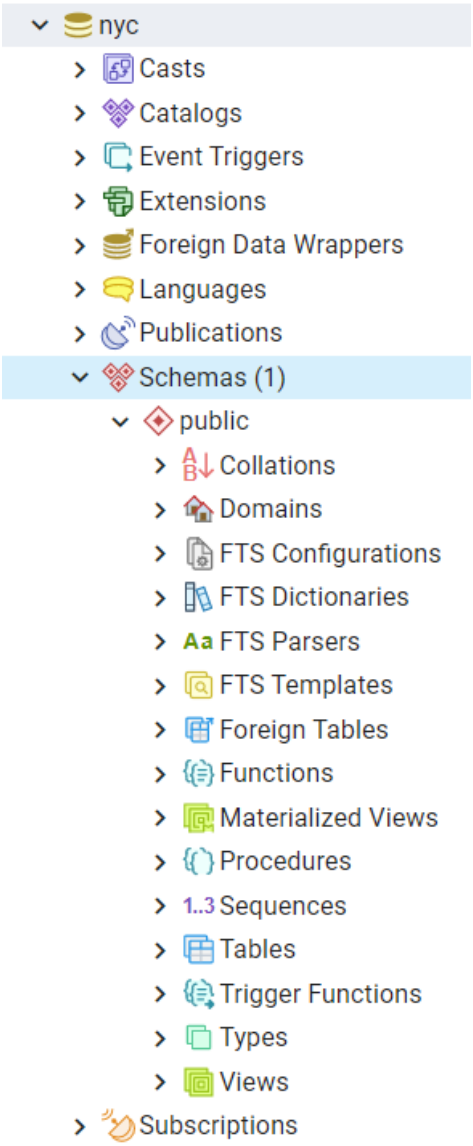


3. Fill in the `Create Database` form as shown below and click **OK**.

Name	`nyc`
Owner	`postgres`



4. Select the new `nyc` database and open it up to display the tree of objects. You'll see the `public` schema.



5. Click on the SQL query button indicated below (or go to *Tools* > *Query Tool*).

6. Enter the following query into the query text field to load the PostGIS spatial extension:

```
CREATE EXTENSION postgis;
```

7. Click the **Play** button in the toolbar (or press **F5**) to "Execute the query."
8. Now confirm that PostGIS is installed by running a PostGIS function:

```
SELECT postgis_full_version();
```

You have successfully created a PostGIS spatial database!!

4.3 Function List

PostGIS_Full_Version: Reports full PostGIS version and build configuration info.

CHAPTER

FIVE

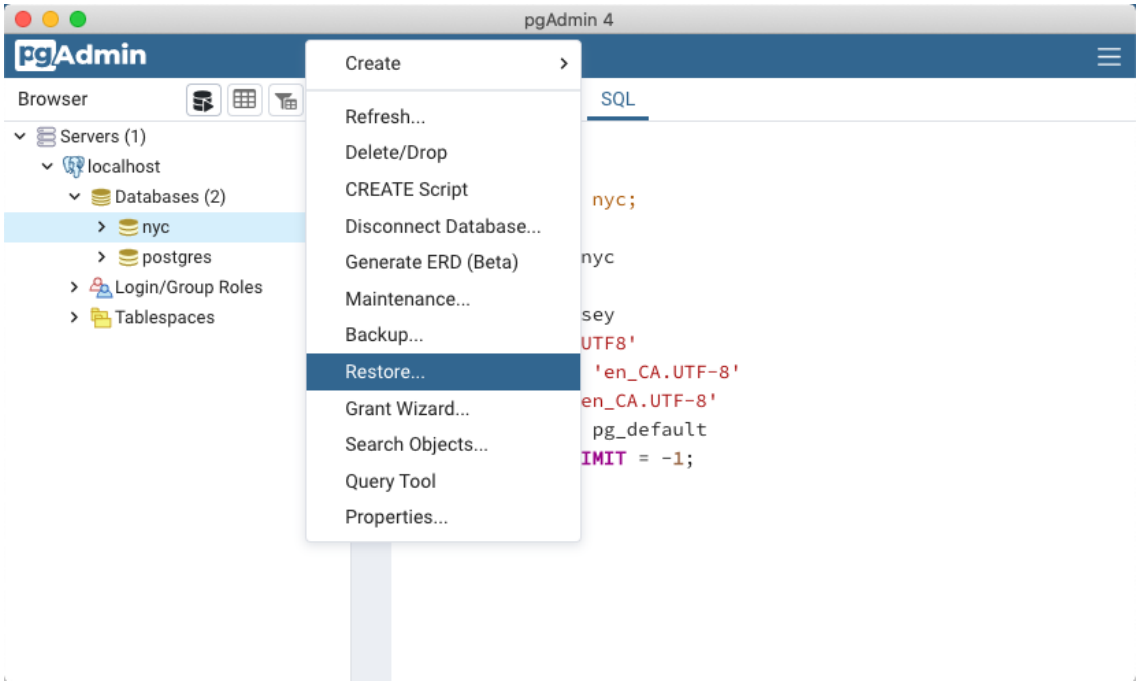
LOADING SPATIAL DATA

Supported by a wide variety of libraries and applications, PostGIS provides many options for loading data.

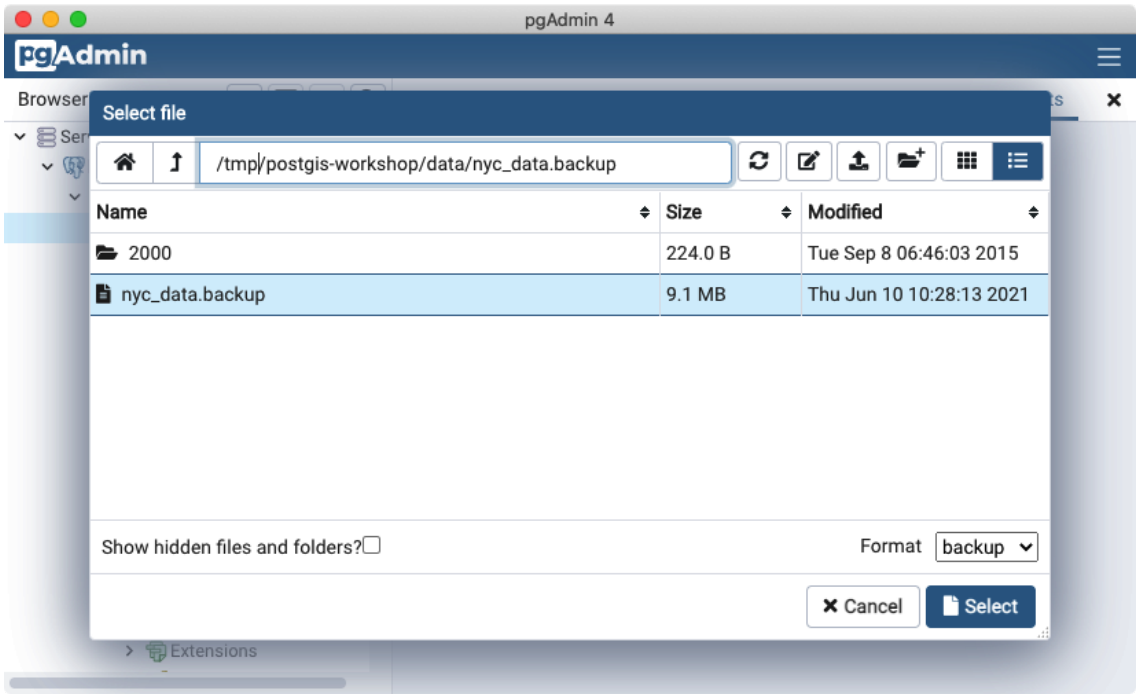
We will first load our working data from a database backup file, then review some standard ways of loading different GIS data formats using common tools.

5.1 Loading the Backup File

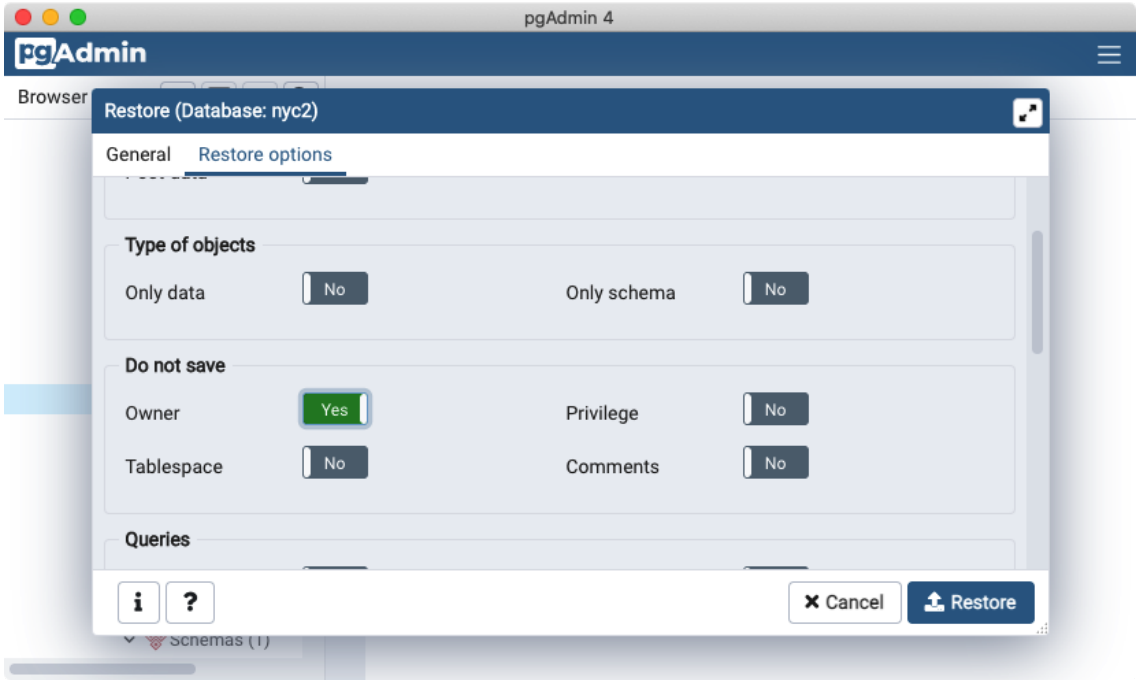
1. In the PgAdmin browser, **right-click** on the **nyc** database icon, and then select the **Restore...** option.



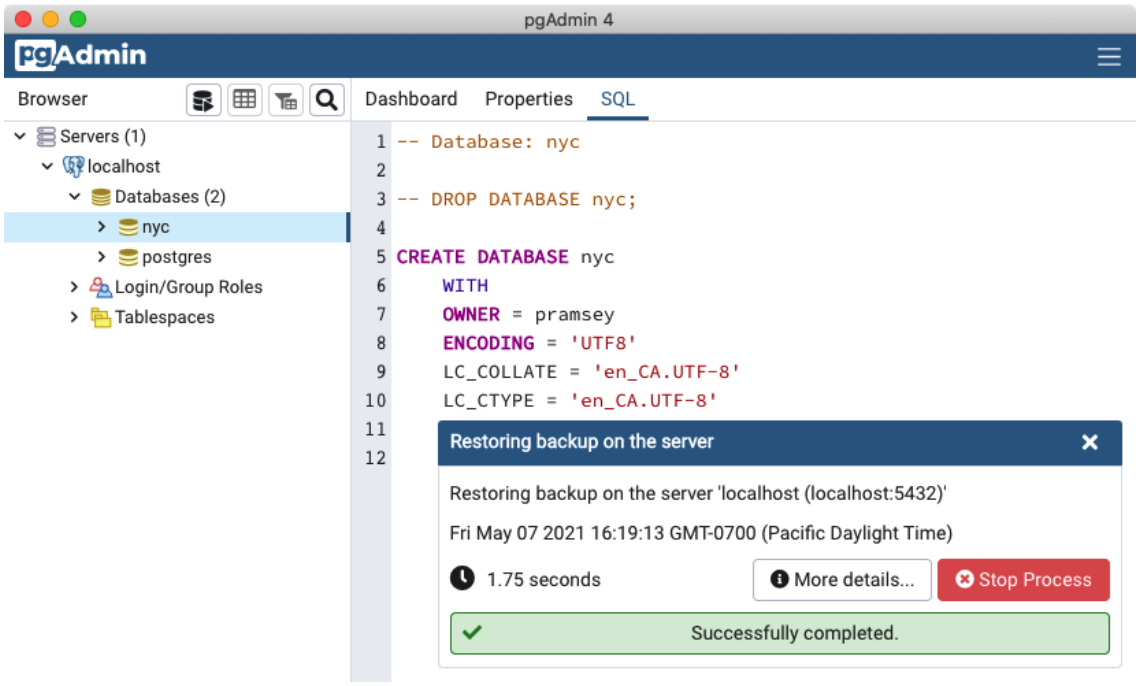
2. Browse to the location of your workshop data data directory (available in the workshop data bundle), and select the `nyc_data.backup` file.



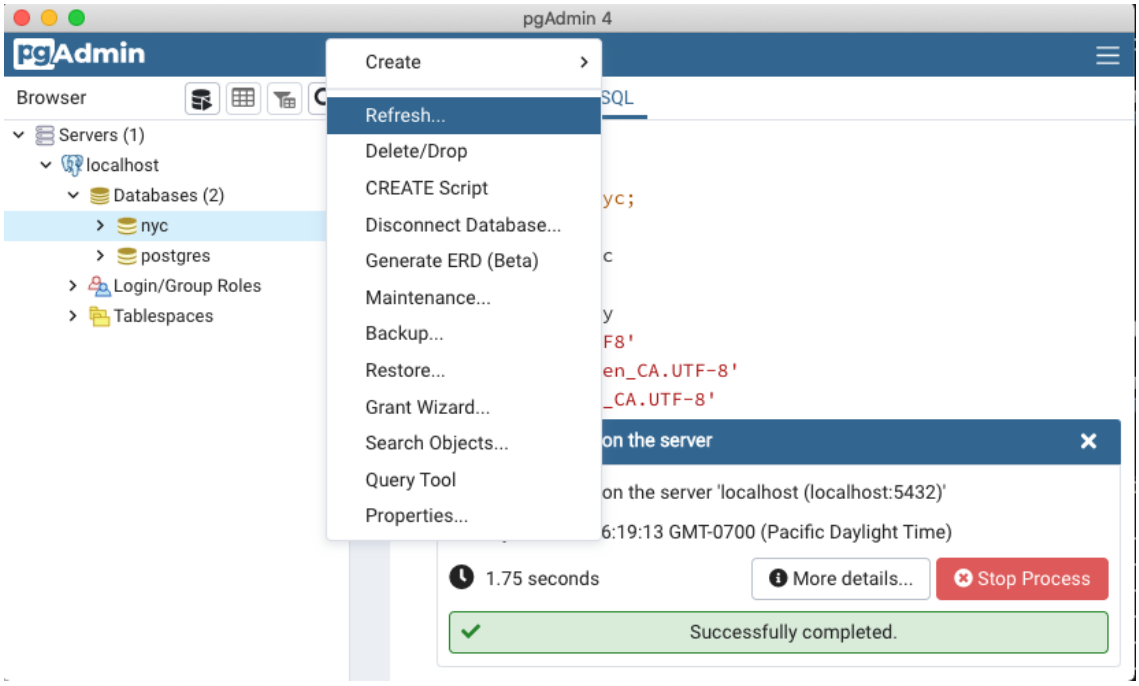
3. Click on the **Restore options** tab, scroll down to the **Do not save** section and toggle **Owner** to **Yes**.



4. Click the **Restore** button. The database restore should run to completion without errors.



5. After the load is complete, right click the **nyc** database, and select the **Refresh** option to update the client information about what tables exist in the database.



Note: If you want to practice loading data from the native spatial formats, instead of using the PostgreSQL db backup files just covered, the next couple of sections will guide you thru loading using various command-line tools and QGIS DbManager. Note you can skip these sections, if you have already loaded the data with pgAdmin.

5.2 Loading with ogr2ogr

`ogr2ogr` is a commandline utility for converting data between GIS data formats, including common file formats and common spatial databases.

Windows:

- Builds of ogr2ogr can be downloaded from GIS Internals.
- ogr2ogr is included as part of QGIS Install and accessible via OSGeo4W Shell -
- Builds of ogr2ogr can be downloaded from MS4W.

MacOS:

- If you installed Postgres.app, then you will find `ogr2ogr` in the `/Applications/Postgres.app/Contents/Versions/*/bin` directory.
- Finally, if you have installed HomeBrew you can install the **gdal** package to get access to `ogr2ogr`

Linux:

- If you installed QGIS from packages, `ogr2ogr` should be installed and on your PATH already as part of the **gdal** or *libgdal** packages.

The postgis workshop data directory includes a `2000/` sub-directory, which contains shape files from the 2000 census, that were obsoleted by data from the 2010 census. We can practice data loading using those files, to avoid creating name collisions with the data we already loaded using the backup file. Be sure to be in the `2000/` sub-directory with the shell when doing these instructions:

```
export PGPASSWORD=mydatabasepassword
```

Rather than passing the password in the connection string, we put it in the environment, so it won't be visible in the process list while the command runs.

Note that on Windows, you will need to use `set` instead of `export`.

```
ogr2ogr \
  -nln nyc_census_blocks_2000 \
  -nlt PROMOTE_TO_MULTI \
  -lco GEOMETRY_NAME=geom \
  -lco FID=gid \
  -lco PRECISION=NO \
  Pg:"dbname=nyc host=localhost user=pramsey port=5432" \
  nyc_census_blocks_2000.shp
```

For more visual clarity, these lines are displayed with `\`, but they should be written in one line on your shell.

The `ogr2ogr` has a **huge** number of options, and we're only using a handful of them here. Here is a line-by-line explanation of the command.

```
ogr2ogr \
```

The executable name! You may need to ensure the executable location is in your *PATH* or use the full path to the executable, depending on your setup.

```
-nln nyc_census_blocks_2000 \
```

The **nln** option stands for "new layer name", and sets the table name that will be created in the target database.

```
-nlt PROMOTE_TO_MULTI \
```

The **nlt** option stands for "new layer type". For shape file input in particular, the new layer type is often a "multi-part geometry", so the system needs to be told in advance to use "MultiPolygon" instead of "Polygon" for the geometry type.

```
-lco GEOMETRY_NAME=geom \
-lco FID=gid \
-lco PRECISION=NO \
```

The **lco** option stands for "layer create option". Different drivers have different create options, and we are using three options for the PostgreSQL driver here.

- **GEOMETRY_NAME** sets the column name for the geometry column. We prefer "geom" over the default, so that our tables match the standard column names in the workshop.
- **FID** sets the primary key column name. Again we prefer "gid" which is the standard used in the workshop.
- **PRECISION** controls how numeric fields are represented in the database. The default when loading a shape file is to use the database "numeric" type, which is more precise but sometimes harder to work with than simple number types like "integer" and "double precision". We use "NO" to turn off the "numeric" type.

```
Pg:"dbname=nyc host=localhost user=pramsey port=5432" \
```

The order of arguments in `ogr2ogr` is, roughly: executable, then options, then **destination** location, then **source location**. So this is the destination, the connection string for our PostgreSQL database. The "Pg:" portion is the driver name, and then the connection string is contained in quotation marks (because it might have embedded spaces).

```
nyc_census_blocks_2000.shp
```

The source data set in this case is the shape file we are reading. It is possible to read multiple layers in one invocation by putting the connection string here, and then following it with a list of layer names, but in this case we have just the one shape file to load.

5.3 Shapefiles? What's that?

You may be asking yourself – "What's this shapefile thing?" A "shapefile" commonly refers to a collection of files with `.shp`, `.shx`, `.dbf`, and other extensions on a common prefix name (e.g., nyc_census_blocks). The actual shapefile relates specifically to files with the `.shp` extension. However, the `.shp` file alone is incomplete for distribution without the required supporting files.

Mandatory files:

- `.shp`—shape format; the feature geometry itself
- `.shx`—shape index format; a positional index of the feature geometry

- `.dbf`—attribute format; columnar attributes for each shape, in dBase III

Optional files include:

- `.prj`—projection format; the coordinate system and projection information, a plain text file describing the projection using well-known text format

The `shp2pgsql` utility makes shape data usable in PostGIS by converting it from binary data into a series of SQL commands that are then run in the database to load the data.

5.4 Loading with shp2pgsql

The `shp2pgsql` converts Shape files into SQL. It is a conversion utility that is part of the PostGIS code base and ships with PostGIS packages. If you installed PostgreSQL locally on your computer, you may find that `shp2pgsql` has been installed along with it, and it is available in the executable directory of your installation.

Unlike `ogr2ogr`, `shp2pgsql` does not connect directly to the destination database, it just emits the SQL equivalent to the input shape file. It is up to the user to pass the SQL to the database, either with a "pipe" or by saving the SQL to file and then loading it.

Here is an example invocation, loading the same data as before:

```
export PGPASSWORD=mydatabasepassword

shp2pgsql \
  -D \
  -I \
  -s 26918 \
  nyc_census_blocks_2000.shp \
  nyc_census_blocks_2000 \
  | psql dbname=nyc user=postgres host=localhost
```

Here is a line-by-line explanation of the command.

```
shp2pgsql \
```

The executable program! It reads the source data file, and emits SQL which can be directed to a file or piped to `psql` to load directly into the database.

```
-D \
```

The **D** flag tells the program to generate "dump format" which is much faster to load than the default "insert format".

```
-I \
```

The **I** flag tells the program to create a spatial index on the table after loading is complete.

```
-s 26918 \
```

The **s** flag tells the program what the "spatial reference identifier (SRID)" of the data is. The source data for this workshop is all in "UTM 18", for which the SRID is **26918** (see below).

```
nyc_census_blocks_2000.shp \
```

The source shape file to read.

```
nyc_census_blocks_2000 \
```

The table name to use when creating the destination table.

```
| psql dbname=nyc user=postgres host=localhost
```

The utility program is generating a stream of SQL. The "|" operator takes that stream and uses it as input to the `psql` database terminal program. The arguments to `psql` are just the connection string for the destination database.

5.5 SRID 26918? What's with that?

Most of the import process is self-explanatory, but even experienced GIS professionals can trip over an **SRID**.

An "SRID" stands for "Spatial Reference IDentifier." It defines all the parameters of our data's geographic coordinate system and projection. An SRID is convenient because it packs all the information about a map projection (which can be quite complex) into a single number.

You can see the definition of our workshop map projection by looking it up either in an online database,

- https://epsg.io/26918

or directly inside PostGIS with a query to the `spatial_ref_sys` table.

```
SELECT srtext FROM spatial_ref_sys WHERE srid = 26918;
```

Note: The PostGIS `spatial_ref_sys` table is an *OGC*-standard table that defines all the spatial reference systems known to the database. The data shipped with PostGIS, lists over 3000 known spatial reference systems and details needed to transform/re-project between them.

In both cases, you see a textual representation of the **26918** spatial reference system (pretty-printed here for clarity):

```
PROJCS["NAD83 / UTM zone 18N",
  GEOGCS["NAD83",
    DATUM["North_American_Datum_1983",
      SPHEROID["GRS 1980",6378137,298.257222101,AUTHORITY["EPSG","7019"]],
      AUTHORITY["EPSG","6269"]],
    PRIMEM["Greenwich",0,AUTHORITY["EPSG","8901"]],
    UNIT["degree",0.01745329251994328,AUTHORITY["EPSG","9122"]],
    AUTHORITY["EPSG","4269"]],
  UNIT["metre",1,AUTHORITY["EPSG","9001"]],
  PROJECTION["Transverse_Mercator"],
  PARAMETER["latitude_of_origin",0],
  PARAMETER["central_meridian",-75],
  PARAMETER["scale_factor",0.9996],
  PARAMETER["false_easting",500000],
```

(continues on next page)

(continued from previous page)

```
  PARAMETER["false_northing",0],
  AUTHORITY["EPSG","26918"],
  AXIS["Easting",EAST],
  AXIS["Northing",NORTH]]
```

If you open up the `nyc_neighborhoods.prj` file from the data directory, you'll see the same projection definition.

Data you receive from local agencies—such as New York City—will usually be in a local projection noted by "state plane" or "UTM". Our projection is "Universal Transverse Mercator (UTM) Zone 18 North" or EPSG:26918.

5.6 Things to Try: View data using QGIS

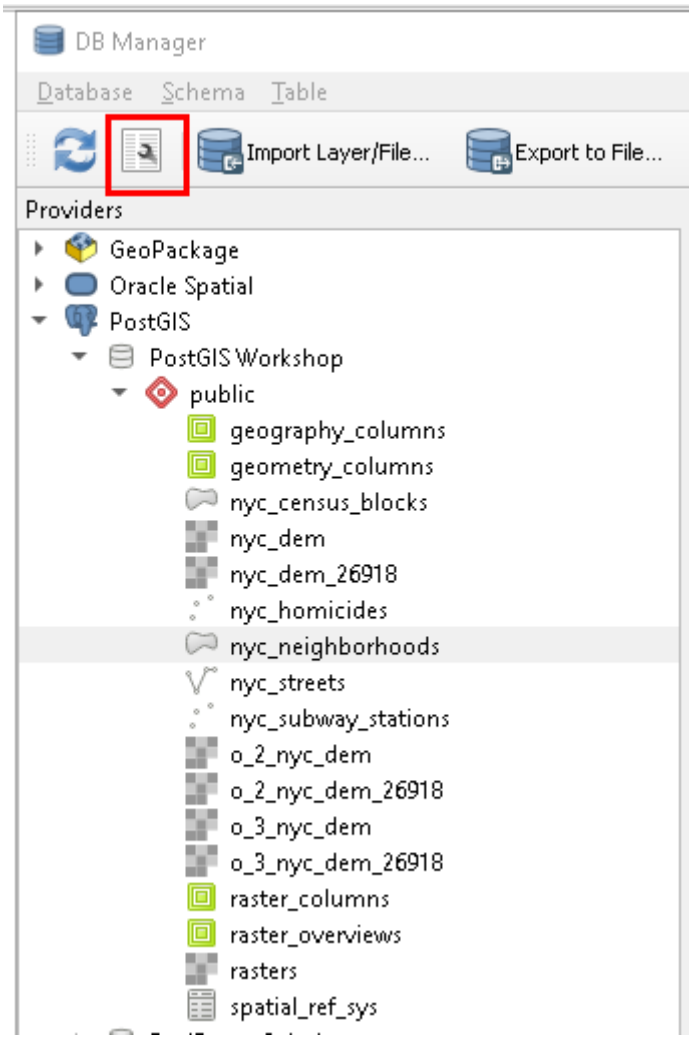
QGIS, is a desktop GIS viewer/editor for quickly looking at data. You can view a number of data formats including flat shapefiles and a PostGIS database. Its graphical interface allows for easy exploration of your data, as well as simple testing and fast styling.

Try using this software to connect your PostGIS database. The application can be downloaded from https://qgis.org

You'll first want to create a connection to a PostGIS database using menu **Layer->Add Layer->PostGIS Layers->New** and then filling in the prompts. Once you have a connection, you can add Layers by clicking connect and selecting a table to display.

5.7 Loading data using QGIS DbManager

QGIS comes with a tool called DbManager that allows you to connect to various different kinds of databases, including a PostGIS enabled one. After you have a PostGIS Database connection configured, go to **Database->DbManager** and expand to your database as shown below:



From there you can use the **`Import Layer/File`** menu option to load numerous different spatial formats. In addition to being able to load data from many spatial formats and export data to many formats, you can also add ad-hoc queries to the canvas or define views in your database, using the highlighted wrench icon.

CHAPTER

SIX

ABOUT OUR DATA

The data for this workshop is four shapefiles for New York City, and one attribute table of sociodemographic variables. We've loaded our shapefiles as PostGIS tables and will add sociodemographic data later in the workshop.

The following describes the number of records and table attributes for each of our datasets. These attribute values and relationships are fundamental to our future analysis.

To explore the nature of your tables in pgAdmin, right-click a highlighted table and select **Properties**. You will find a summary of table properties, including a list of table attributes within the **Columns** tab.

6.1 nyc_census_blocks

A census block is the smallest geography for which census data is reported. All higher level census geographies (block groups, tracts, metro areas, counties, etc) can be built from unions of census blocks. We have attached some demographic data to our collection of blocks.

Number of records: 38794

blkid	A 15-digit code that uniquely identifies every census **block**. Eg: 360050001009000
popn_total	Total number of people in the census block
popn_white	Number of people self-identifying as "White" in the block
popn_black	Number of people self-identifying as "Black" in the block
popn_nativ	Number of people self-identifying as "Native American" in the block
popn_asian	Number of people self-identifying as "Asian" in the block
popn_other	Number of people self-identifying with other categories in the block
boroname	Name of the New York borough. Manhattan, The Bronx, Brooklyn, Staten Island, Queens
geom	Polygon boundary of the block

Note: To get census data into GIS, you need to join two pieces of information: the actual data (text), and the boundary files (spatial). There are many options for getting the data, including downloading data and boundaries from the Census Bureau's American FactFinder.

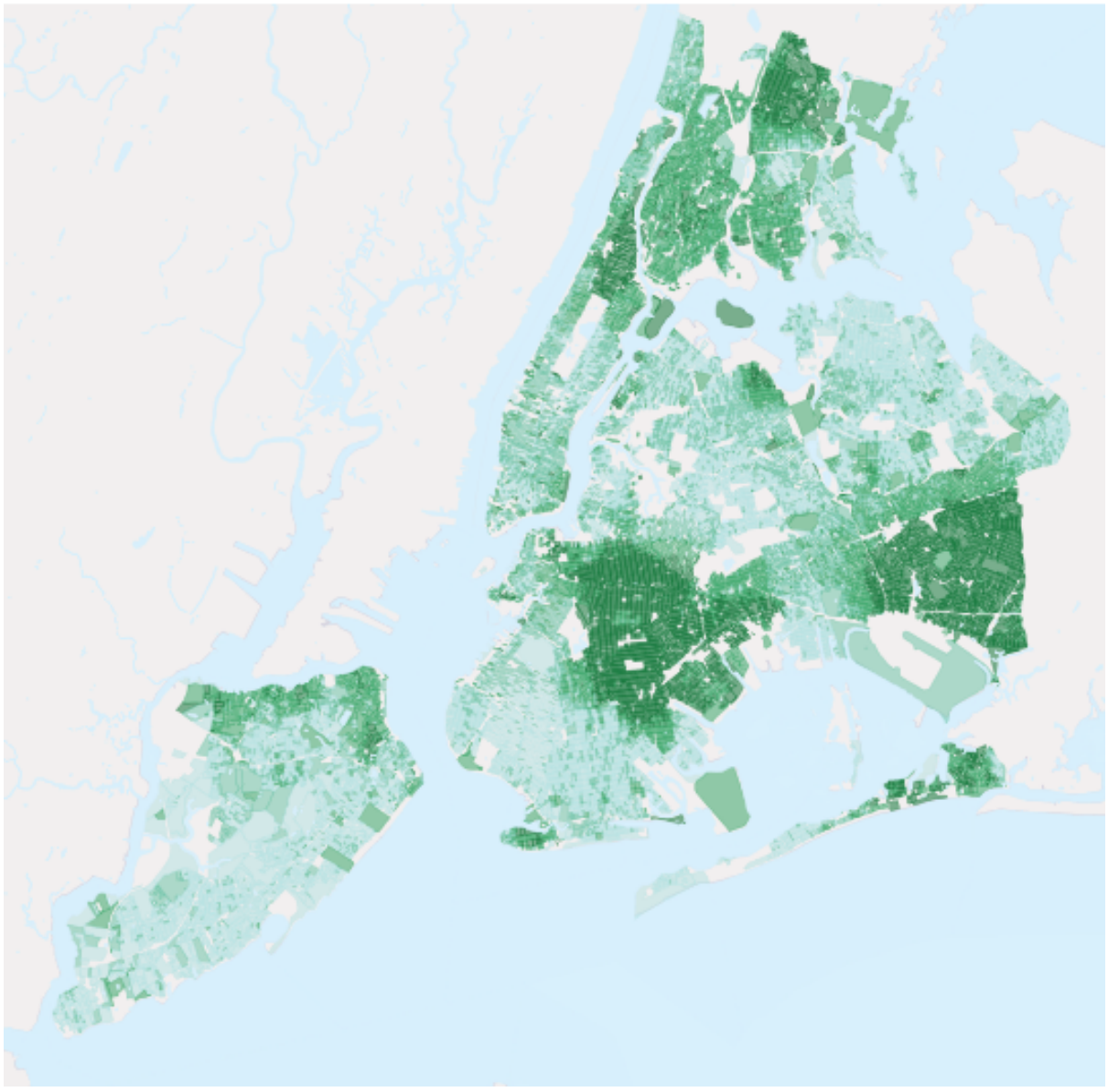


Fig. 1: *Black population as a percentage of Total Population*

6.2 nyc_neighborhoods

New York has a rich history of neighborhood names and extent. Neighborhoods are social constructs that do not follow lines laid down by the government. For example, the Brooklyn neighborhoods of Carroll Gardens, Red Hook, and Cobble Hill were once collectively known as "South Brooklyn." And now, depending on which real estate agent you talk to, the same four blocks in the-neighborhood-formerly-known-as-Red-Hook can be referred to as Columbia Heights, Carroll Gardens West, or Red Hook!

Number of records: 129

name	Name of the neighborhood
boroname	Name of the New York borough. Manhattan, The Bronx, Brooklyn, Staten Island, Queens
geom	Polygon boundary of the neighborhood

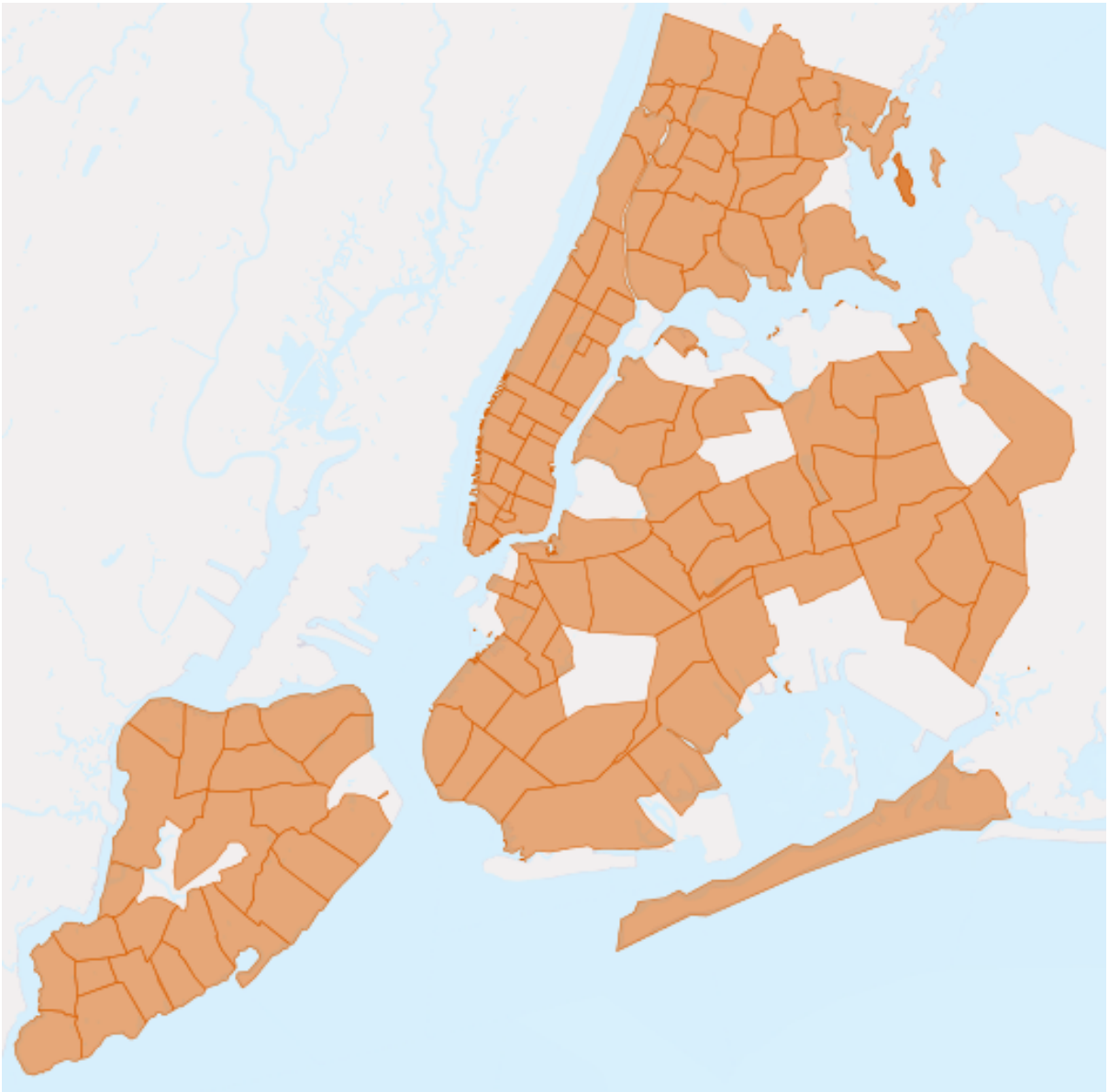


Fig. 2: *The neighborhoods of New York City*

6.3 nyc_streets

The street centerlines form the transportation network of the city. These streets have been flagged with types in order to distinguish between such thoroughfares as back alleys, arterial streets, freeways, and smaller streets. Desirable areas to live might be on residential streets rather than next to a freeway.

Number of records: 19091

name	Name of the street
oneway	Is the street one-way? “yes” = yes, “” = no
type	Road type (primary, secondary, residential, motorway)
geom	Linear centerline of the street

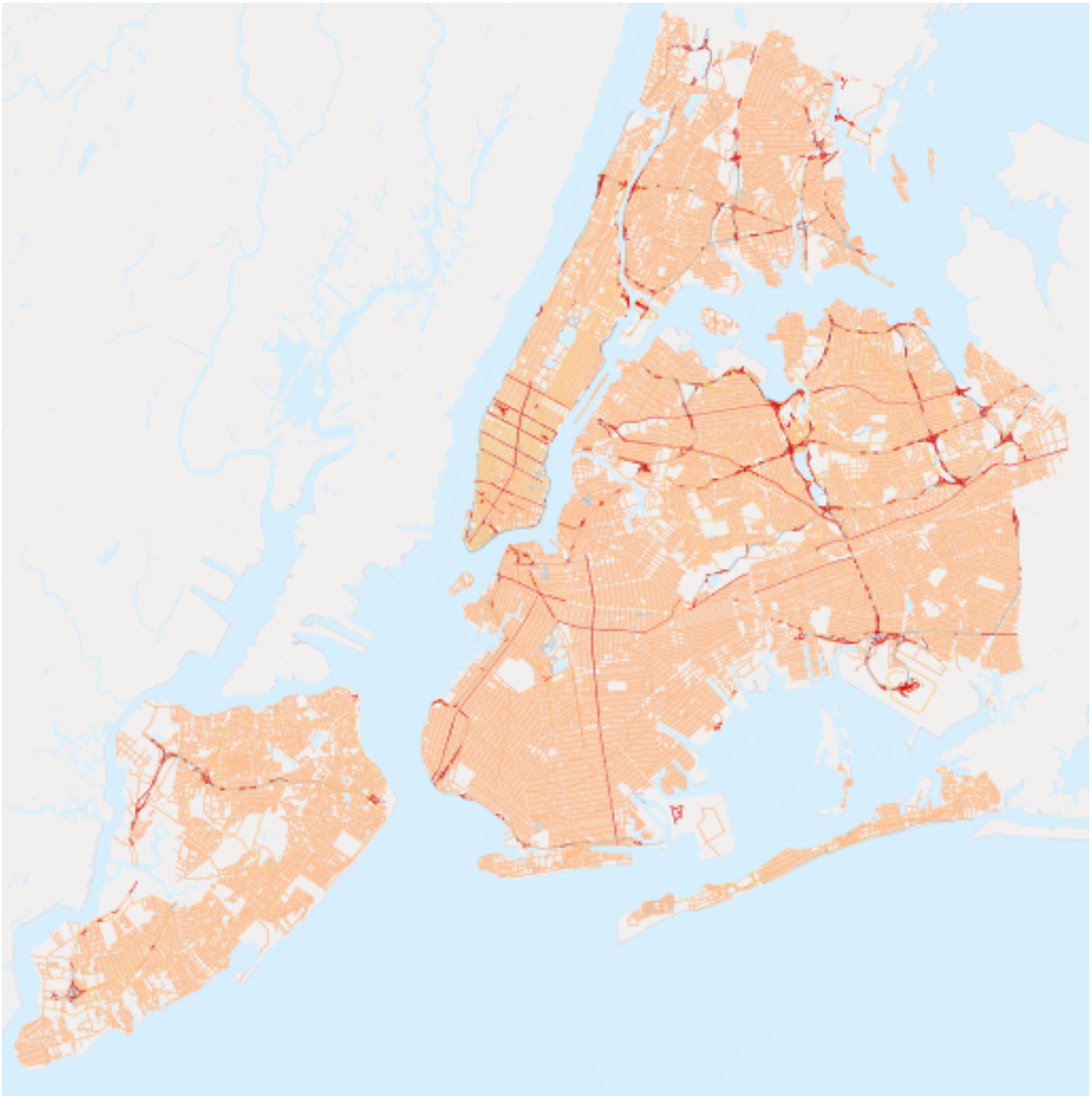


Fig. 3: *The streets of New York City. Major roads are in red.*

6.4 nyc_subway_stations

The subway stations link the upper world where people live to the invisible network of subways beneath. As portals to the public transportation system, station locations help determine how easy it is for different people to enter the subway system.

Number of records: 491

name	Name of the station
borough	Name of the New York borough. Manhattan, The Bronx, Brooklyn, Staten Island, Queens
routes	Subway lines that run through this station
transfers	Lines you can transfer to via this station
express	Stations where express trains stop, "express" = yes, "" = no
geom	Point location of the station

6.5 nyc_census_sociodata

There is a rich collection of social-economic data collected during the census process, but only at the larger geography level of census tract. Census blocks combine to form census tracts (and block groups). We have collected some social-economic at a census tract level to answer some of these more interesting questions about New York City.

Note: The `nyc_census_sociodata` is a data table. We will need to connect it to Census geographies before conducting any spatial analysis.

---

tractid	An 11-digit code that uniquely identifies every census **tract**. ("36005000100")
transit_total	Number of workers in the tract
transit_private	Number of workers in the tract who use private automobiles / motorcycles
transit_public	Number of workers in the tract who take public transit
transit_walk	Number of workers in the tract who walk
transit_other	Number of workers in the tract who use other forms like walking / biking
transit_none	Number of workers in the tract who work from home
transit_time_mins	Total number of minutes spent in transit by all workers in the tract (minutes)
family_count	Number of families in the tract
family_income_median	Median family income in the tract (dollars)
family_income_mean	Average family income in the tract (dollars)
family_income_aggregate	Total income of all families in the tract (dollars)
edu_total	Number of people with educational history
edu_no_highschool_dipl	Number of people with no high school diploma
edu_highschool_dipl	Number of people with high school diploma and no further education
edu_college_dipl	Number of people with college diploma and no further education
edu_graduate_dipl	Number of people with graduate school diploma

Fig. 4: *Point locations for New York City subway stations*

CHAPTER

SEVEN

SIMPLE SQL

SQL, or "Structured Query Language", is a means of asking questions of, and updating data in, relational databases. You have already seen SQL when we created our first database. Recall:

```
SELECT postgis_full_version();
```

But that was a question about the database. Now that we've loaded data into our database, let's use SQL to ask questions of the data! For example,

> "What are the names of all the neighborhoods in New York City?"

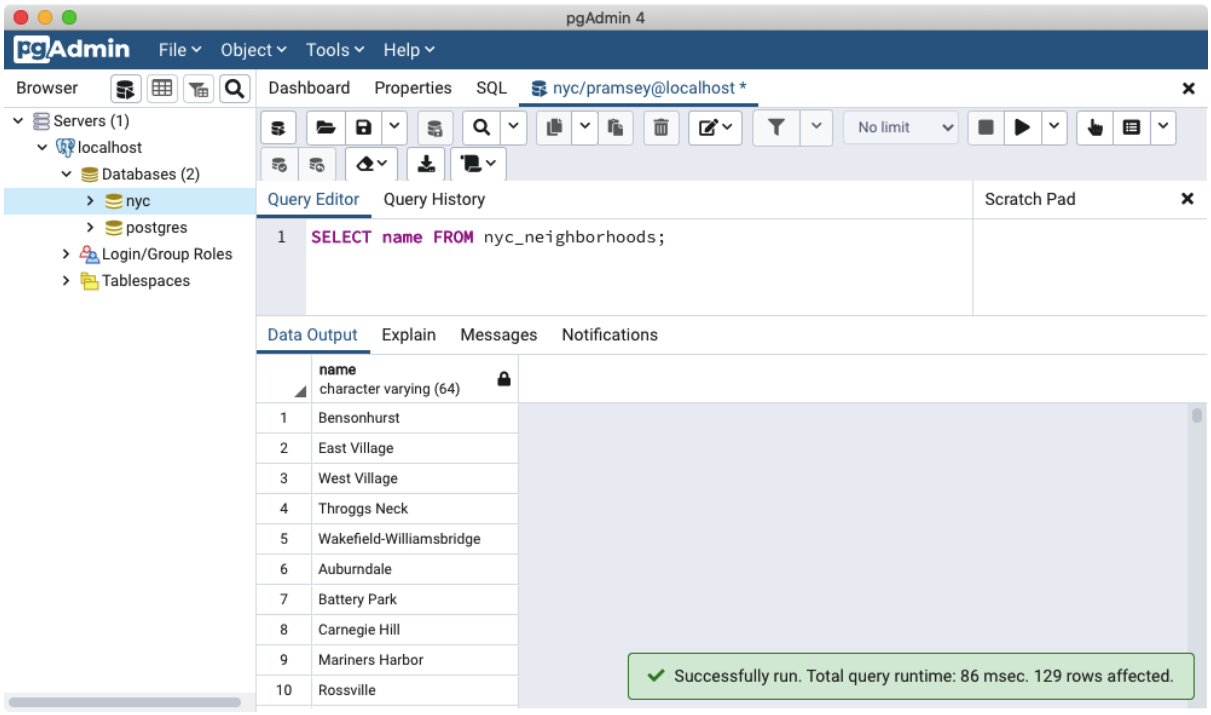
Open up the SQL query window in pgAdmin by clicking the "Query Tool" button.

then enter the following query in to the query window

```
SELECT name FROM nyc_neighborhoods;
```

and click the **Execute Query** button (the green triangle).

The query will run for a few (milli)seconds and return the 129 results.



But what exactly happened here? To understand, let's begin with the four "verbs" of SQL,

- `SELECT`, returns rows in response to a query
- `INSERT`, adds new rows to a table
- `UPDATE`, alters existing rows in a table
- `DELETE`, removes rows from a table

We will be working almost exclusively with `SELECT` in order to ask questions of tables using spatial functions.

7.1 SELECT queries

A select query is generally of the form:

```
SELECT some_columns FROM some_data_source WHERE some_condition;
```

Note: For a synopsis of all `SELECT` parameters, see the PostgresSQL documentation.

The `some_columns` are either column names or functions of column values. The `some_data_source` is either a single table, or a composite table created by joining two tables on a key or condition. The `some_condition` is a filter that restricts the number of rows to be returned.

> "What are the names of all the neighborhoods in Brooklyn?"

We return to our `nyc_neighborhoods` table with a filter in hand. The table contains all the neighborhoods in New York, but we only want the ones in Brooklyn.

```
SELECT name
  FROM nyc_neighborhoods
  WHERE boroname = 'Brooklyn';
```

The query will run for even fewer (milli)seconds and return the 23 results.

Sometimes we will need to apply a function to the results of our query. For example,

> "What is the number of letters in the names of all the neighborhoods in Brooklyn?"

Fortunately, PostgreSQL has a string length function, **char_length(string)**.

```
SELECT char_length(name)
  FROM nyc_neighborhoods
  WHERE boroname = 'Brooklyn';
```

Often, we are less interested in the individual rows than in a statistic that applies to all of them. So knowing the lengths of the neighborhood names might be less interesting than knowing the average length of the names. Functions that take in multiple rows and return a single result are called "aggregate" functions.

PostgreSQL has a series of built-in aggregate functions, including the general purpose **avg()** for average values and **stddev()** for standard deviations.

> "What is the average number of letters and standard deviation of number of letters in the names of all the neighborhoods in Brooklyn?"

```
SELECT avg(char_length(name)), stddev(char_length(name))
  FROM nyc_neighborhoods
  WHERE boroname = 'Brooklyn';
```

```
         avg          |       stddev
----------------------+--------------------
 11.7391304347826087 | 3.9105613559407395
```

The aggregate functions in our last example were applied to every row in the result set. What if we want the summaries to be carried out over smaller groups within the overall result set? For that we add a `GROUP BY` clause. Aggregate functions often need an added `GROUP BY` statement to group the result-set by one or more columns.

> "What is the average number of letters in the names of all the neighborhoods in New York City, reported by borough?"

```
SELECT boroname, avg(char_length(name)), stddev(char_length(name))
  FROM nyc_neighborhoods
  GROUP BY boroname;
```

We include the `boroname` column in the output result so we can determine which statistic applies to which borough. In an aggregate query, you can only output columns that are either (a) members of the grouping clause or (b) aggregate functions.

```
   boroname     |         avg          |       stddev
----------------+----------------------+--------------------
 Brooklyn       | 11.7391304347826087 | 3.9105613559407395
 Manhattan      | 11.8214285714285714 | 4.3123729948325257
 The Bronx      | 12.0416666666666667 | 3.6651017740975152
```

(continues on next page)

(continued from previous page)

```
Queens        | 11.6666666666666667 | 5.0057438272815975
Staten Island | 12.2916666666666667 | 5.2043390480959474
```

7.2 Function List

avg(expression): PostgreSQL aggregate function that returns the average value of a numeric column.

char_length(string): PostgreSQL string function that returns the number of character in a string.

stddev(expression): PostgreSQL aggregate function that returns the standard deviation of input values.

CHAPTER

EIGHT

SIMPLE SQL EXERCISES

Using the `nyc_census_blocks` table, answer the following questions (don't peek at the answers!).

Here is some helpful information to get started. Recall from the *About Our Data* section our `nyc_census_blocks` table definition.

blkid	A 15-digit code that uniquely identifies every census **block**. ("360050001009000")
popn_total	Total number of people in the census block
popn_white	Number of people self-identifying as "white" in the block
popn_black	Number of people self-identifying as "black" in the block
popn_nativ	Number of people self-identifying as "native american" in the block
popn_asian	Number of people self-identifying as "asias" in the block
popn_other	Number of people self-identifying with other categories in the block
hous_total	Number of housing units in the block
hous_own	Number of owner-occupied housing units in the block
hous_rent	Number of renter-occupied housing units in the block
boroname	Name of the New York borough. Manhattan, The Bronx, Brooklyn, Staten Island, Queens
geom	Polygon boundary of the block

And, here are some common SQL aggregation functions you might find useful:

- avg() - the average (mean) of the values in a set of records
- sum() - the sum of the values in a set of records
- count() - the number of records in a set of records

Now the questions:

- **How many records are in the nyc_streets table?**

```
SELECT Count(*)
FROM nyc_streets;
```

```
19091
```

- **How many streets in NYC start with 'B'?**

```
SELECT Count(*)
  FROM nyc_streets
  WHERE name LIKE 'B%';
```

```
1282
```

- **What is the population of the City of New York?**

```
SELECT Sum(popn_total) AS population
  FROM nyc_census_blocks;
```

```
8175032
```

Note: What is this `AS`? You can give a table or a column another name by using an alias. Aliases can make queries easier to both write and to read. So instead of our outputted column name as `sum` we write it **AS** the more readable `population`.

- **What is the population of the Bronx?**

```
SELECT Sum(popn_total) AS population
  FROM nyc_census_blocks
  WHERE boroname = 'The Bronx';
```

```
1385108
```

- **How many "neighborhoods" are in each borough?**

```
SELECT boroname, count(*)
  FROM nyc_neighborhoods
  GROUP BY boroname;
```

```
   boroname     | count
----------------+-------
 Queens         |    30
 Brooklyn       |    23
 Staten Island  |    24
 The Bronx      |    24
 Manhattan      |    28
```

- **For each borough, what percentage of the population is white?**

```
SELECT
  boroname,
  100.0 * Sum(popn_white)/Sum(popn_total) AS white_pct
FROM nyc_census_blocks
GROUP BY boroname;
```

```
   boroname     |    white_pct
----------------+------------------
 Brooklyn       | 42.8011737932687
 Manhattan      | 57.4493039480463
 The Bronx      | 27.9037446899448
 Queens         |  39.722077394591
 Staten Island | 72.8942034860154
```

8.1 Function List

avg(expression): PostgreSQL aggregate function that returns the average value of a numeric column.

count(expression): PostgreSQL aggregate function that returns the number of records in a set of records.

sum(expression): PostgreSQL aggregate function that returns the sum of records in a set of records.

CHAPTER

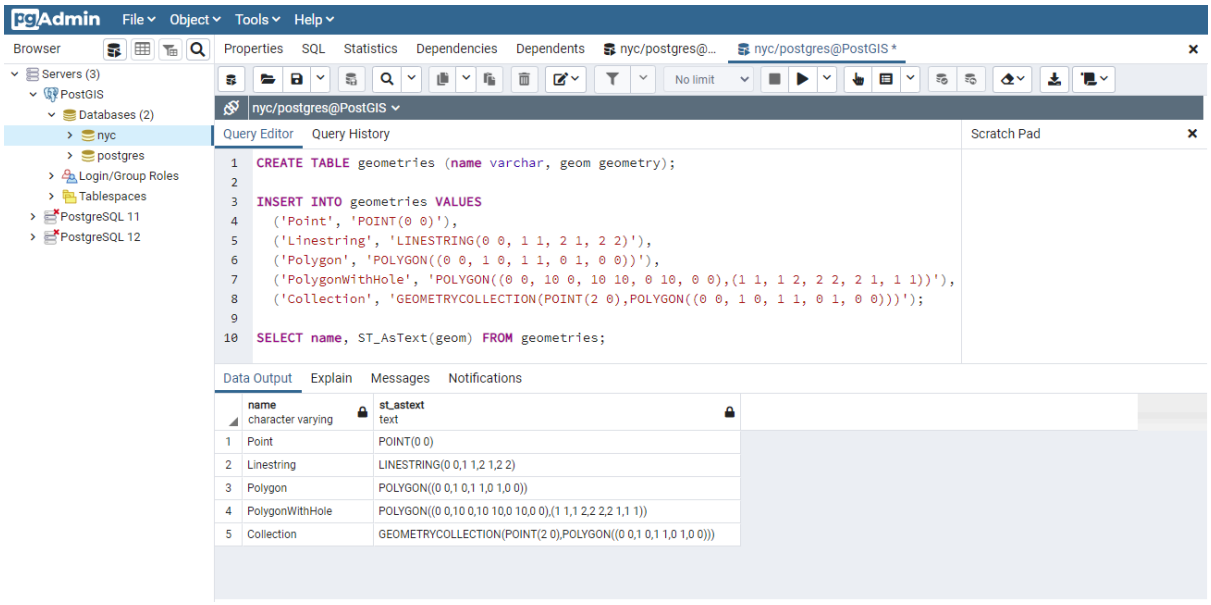
NINE

GEOMETRIES

9.1 Introduction

In the previous *section*, we loaded a variety of data. Before we start playing with our data lets have a look at some simpler examples. In pgAdmin, once again select the **nyc** database and open the SQL query tool. Paste this example SQL code into the pgAdmin SQL Editor window (removing any text that may be there by default) and then execute.

```
CREATE TABLE geometries (name varchar, geom geometry);

INSERT INTO geometries VALUES
  ('Point', 'POINT(0 0)'),
  ('Linestring', 'LINESTRING(0 0, 1 1, 2 1, 2 2)'),
  ('Polygon', 'POLYGON((0 0, 1 0, 1 1, 0 1, 0 0))'),
  ('PolygonWithHole', 'POLYGON((0 0, 10 0, 10 10, 0 10, 0 0),(1 1, 1 2, 2
↪2, 2 1, 1 1))'),
  ('Collection', 'GEOMETRYCOLLECTION(POINT(2 0),POLYGON((0 0, 1 0, 1 1, 0
↪1, 0 0)))');

SELECT name, ST_AsText(geom) FROM geometries;
```

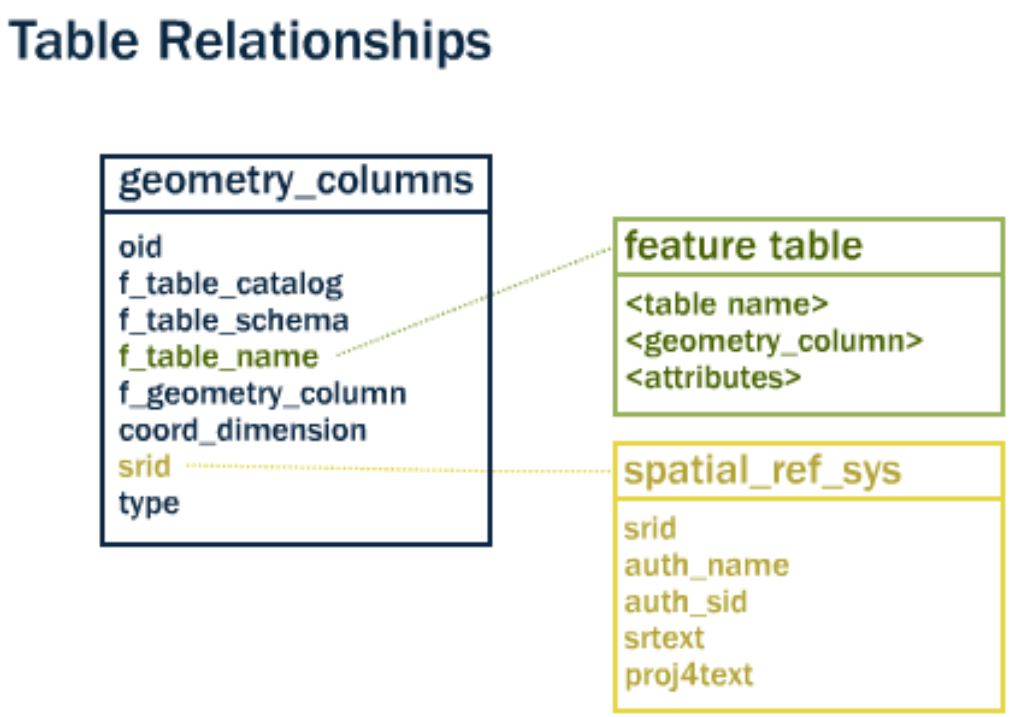


The above example CREATEs a table (**geometries**) then INSERTs five geometries: a point, a line, a polygon, a polygon with a hole, and a collection. Finally, the inserted rows are SELECTed and displayed in the Output pane.

9.2 Metadata Tables

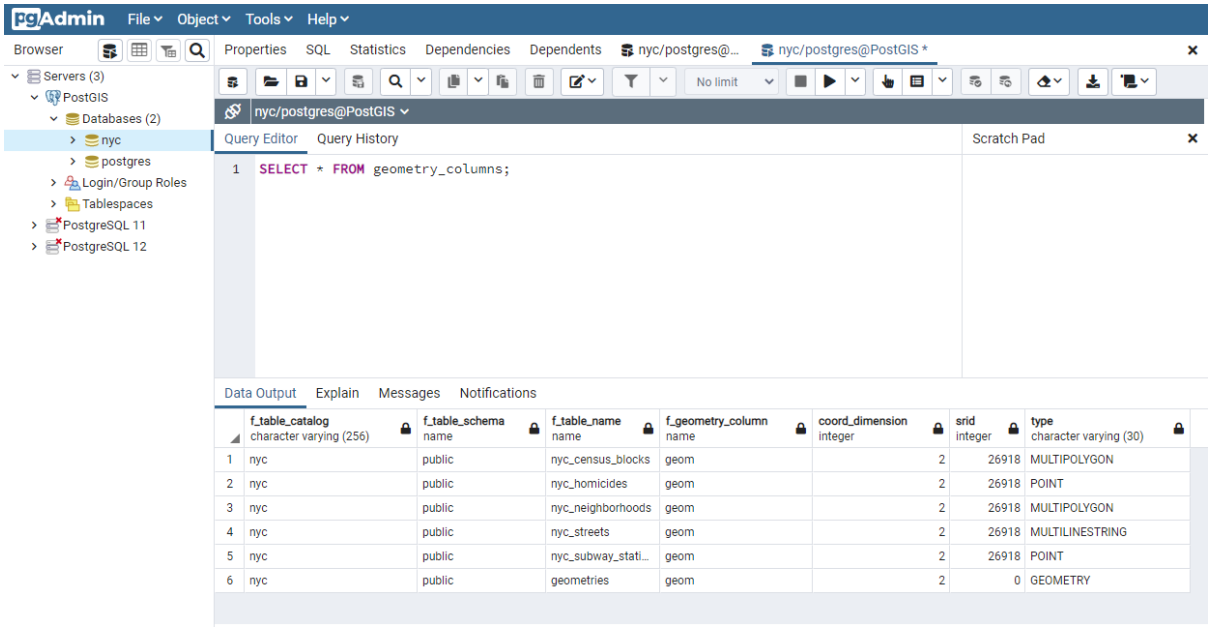
In conformance with the Simple Features for SQL (*SFSQL*) specification, PostGIS provides two tables to track and report on the geometry types available in a given database.

- The first table, `spatial_ref_sys`, defines all the spatial reference systems known to the database and will be described in greater detail later.
- The second table (actually, a view), `geometry_columns`, provides a listing of all "features" (defined as an object with geometric attributes), and the basic details of those features.



Let's have a look at the `geometry_columns` table in our database. Paste this command in the Query Tool as before:

```
SELECT * FROM geometry_columns;
```



- `f_table_catalog`, `f_table_schema`, and `f_table_name` provide the fully qualified name of the feature table containing a given geometry. Because PostgreSQL doesn't make use of catalogs, `f_table_catalog` will tend to be empty.
- `f_geometry_column` is the name of the column that geometry containing column – for feature tables with multiple geometry columns, there will be one record for each.
- `coord_dimension` and `srid` define the the dimension of the geometry (2-, 3- or 4-dimensional) and the Spatial Reference system identifier that refers to the `spatial_ref_sys` table respectively.
- The `type` column defines the type of geometry as described below; we've seen Point and Linestring types so far.

By querying this table, GIS clients and libraries can determine what to expect when retrieving data and can perform any necessary projection, processing or rendering without needing to inspect each geometry.

Note:

Do some or all of your `nyc` tables not have an `srid` of 26918? It's easy to fix by updating the table.

```
ALTER TABLE nyc_neighborhoods
  ALTER COLUMN geom
  TYPE Geometry(MultiPolygon, 26918)
  USING ST_SetSRID(geom, 26918);
```

9.3 Representing Real World Objects

The Simple Features for SQL (*SFSQL*) specification, the original guiding standard for PostGIS development, defines how a real world object is represented. By taking a continuous shape and digitizing it at a fixed resolution we achieve a passable representation of the object. SFSQL only handled 2-dimensional representations. PostGIS has extended that to include 3- and 4-dimensional representations; more recently the SQL-Multimedia Part 3 (*SQL/MM*) specification has officially defined their own representation.

Our example table contains a mixture of different geometry types. We can collect general information about each object using functions that read the geometry metadata.

- **ST_GeometryType(geometry)** returns the type of the geometry
- **ST_NDims(geometry)** returns the number of dimensions of the geometry
- **ST_SRID(geometry)** returns the spatial reference identifier number of the geometry

```
SELECT name, ST_GeometryType(geom), ST_NDims(geom), ST_SRID(geom)
  FROM geometries;
```

```
      name        |    st_geometrytype     | st_ndims | st_srid
------------------+------------------------+----------+---------
 Point            | ST_Point               |        2 |       0
 Polygon          | ST_Polygon             |        2 |       0
 PolygonWithHole  | ST_Polygon             |        2 |       0
 Collection       | ST_GeometryCollection  |        2 |       0
 Linestring       | ST_LineString          |        2 |       0
```

9.3.1 Points

A spatial **point** represents a single location on the Earth. This point is represented by a single coordinate (including either 2-, 3- or 4-dimensions). Points are used to represent objects when the exact details, such as shape and size, are not important at the target scale. For example, cities on a map of the world can be described as points, while a map of a single state might represent cities as polygons.

```
SELECT ST_AsText(geom)
  FROM geometries
  WHERE name = 'Point';
```

```
POINT(0 0)
```

Some of the specific spatial functions for working with points are:

- **ST_X(geometry)** returns the X ordinate
- **ST_Y(geometry)** returns the Y ordinate

So, we can read the ordinates from a point like this:

```
SELECT ST_X(geom), ST_Y(geom)
  FROM geometries
  WHERE name = 'Point';
```

The New York City subway stations (`nyc_subway_stations`) table is a data set represented as points. The following SQL query will return the geometry associated with one point (in the **`ST_AsText`** column).

```
SELECT name, ST_AsText(geom)
  FROM nyc_subway_stations
  LIMIT 1;
```

9.3.2 Linestrings

A **linestring** is a path between locations. It takes the form of an ordered series of two or more points. Roads and rivers are typically represented as linestrings. A linestring is said to be **closed** if it starts and ends on the same point. It is said to be **simple** if it does not cross or touch itself (except at its endpoints if it is closed). A linestring can be both **closed** and **simple**.

The street network for New York (`nyc_streets`) was loaded earlier in the workshop. This dataset contains details such as name, and type. A single real world street may consist of many linestrings, each representing a segment of road with different attributes.

The following SQL query will return the geometry associated with one linestring (in the **`ST_AsText`** column).

```
SELECT ST_AsText(geom)
  FROM geometries
  WHERE name = 'Linestring';
```

```
LINESTRING(0 0, 1 1, 2 1, 2 2)
```

Some of the specific spatial functions for working with linestrings are:

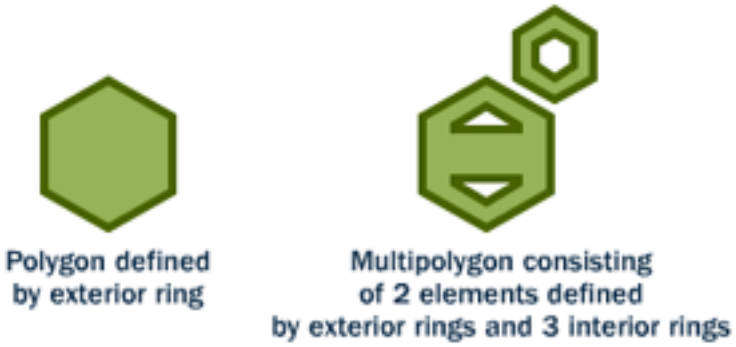
- **`ST_Length(geometry)`** returns the length of the linestring
- **`ST_StartPoint(geometry)`** returns the first coordinate as a point
- **`ST_EndPoint(geometry)`** returns the last coordinate as a point
- **`ST_NPoints(geometry)`** returns the number of coordinates in the linestring

So, the length of our linestring is:

```
SELECT ST_Length(geom)
  FROM geometries
  WHERE name = 'Linestring';
```

```
3.41421356237309
```

9.3.3 Polygons



A polygon is a representation of an area. The outer boundary of the polygon is represented by a ring. This ring is a linestring that is both closed and simple as defined above. Holes within the polygon are also represented by rings.

Polygons are used to represent objects whose size and shape are important. City limits, parks, building footprints or bodies of water are all commonly represented as polygons when the scale is sufficiently high to see their area. Roads and rivers can sometimes be represented as polygons.

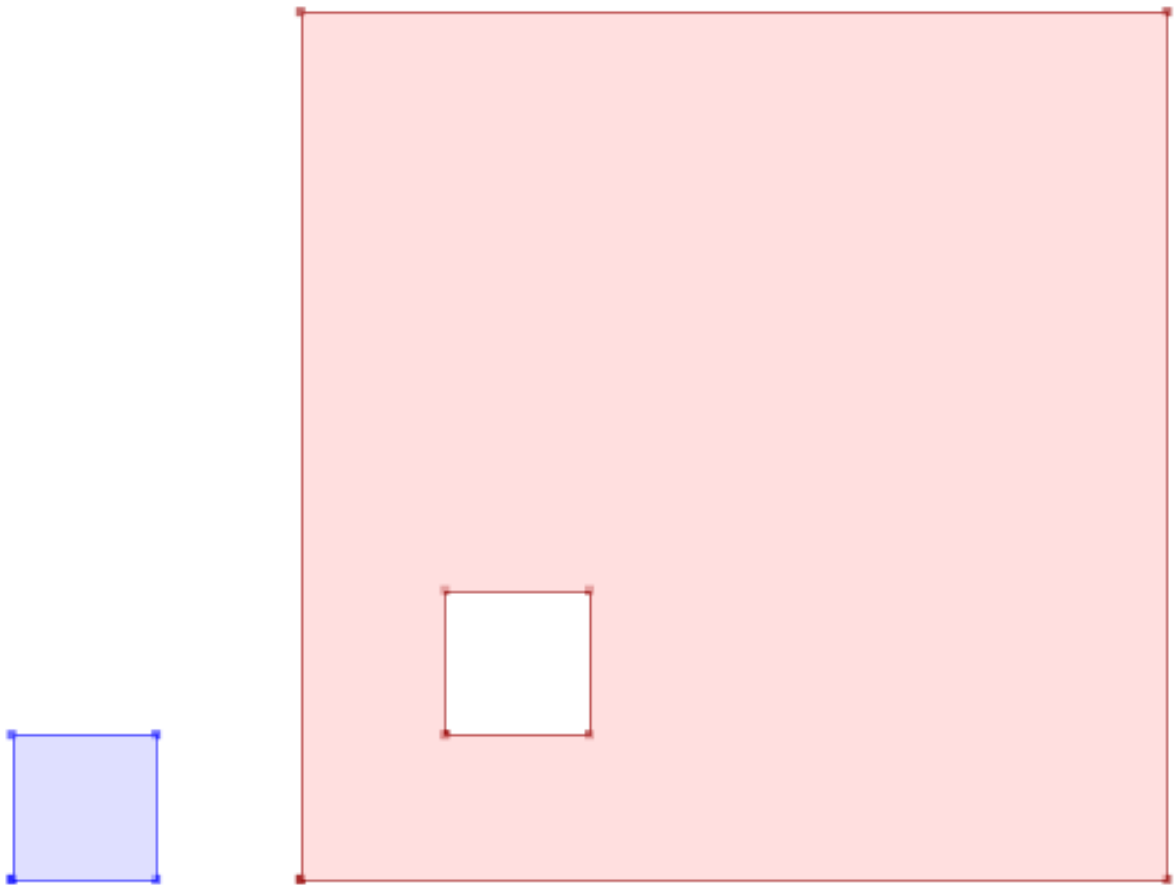
The following SQL query will return the geometry associated with one polygon (in the **`ST_AsText`** column).

```
SELECT ST_AsText(geom)
  FROM geometries
  WHERE name LIKE 'Polygon%';
```

Note: Rather than using an = sign in our `WHERE` clause, we are using the `LIKE` operator to carry out a string matching operation. **You may be used to the ``*`` symbol as a "glob" for pattern matching, but in SQL the ``%`` symbol is used**, along with the `LIKE` operator to tell the system to do globbing.

```
POLYGON((0 0, 1 0, 1 1, 0 1, 0 0))
POLYGON((0 0, 10 0, 10 10, 0 10, 0 0),(1 1, 1 2, 2 2, 2 1, 1 1))
```

The first polygon has only one ring. The second one has an interior "hole". Most graphics systems include the concept of a "polygon", but GIS systems are relatively unique in allowing polygons to explicitly have holes.



Some of the specific spatial functions for working with polygons are:

- **ST_Area(geometry)** returns the area of the polygons
- **ST_NRings(geometry)** returns the number of rings (usually 1, more if there are holes)
- **ST_ExteriorRing(geometry)** returns the outer ring as a linestring
- **ST_InteriorRingN(geometry,n)** returns a specified interior ring as a linestring
- **ST_Perimeter(geometry)** returns the length of all the rings

We can calculate the area of our polygons using the area function:

```
SELECT name, ST_Area(geom)
  FROM geometries
  WHERE name LIKE 'Polygon%';
```

```
Polygon             1
PolygonWithHole     99
```

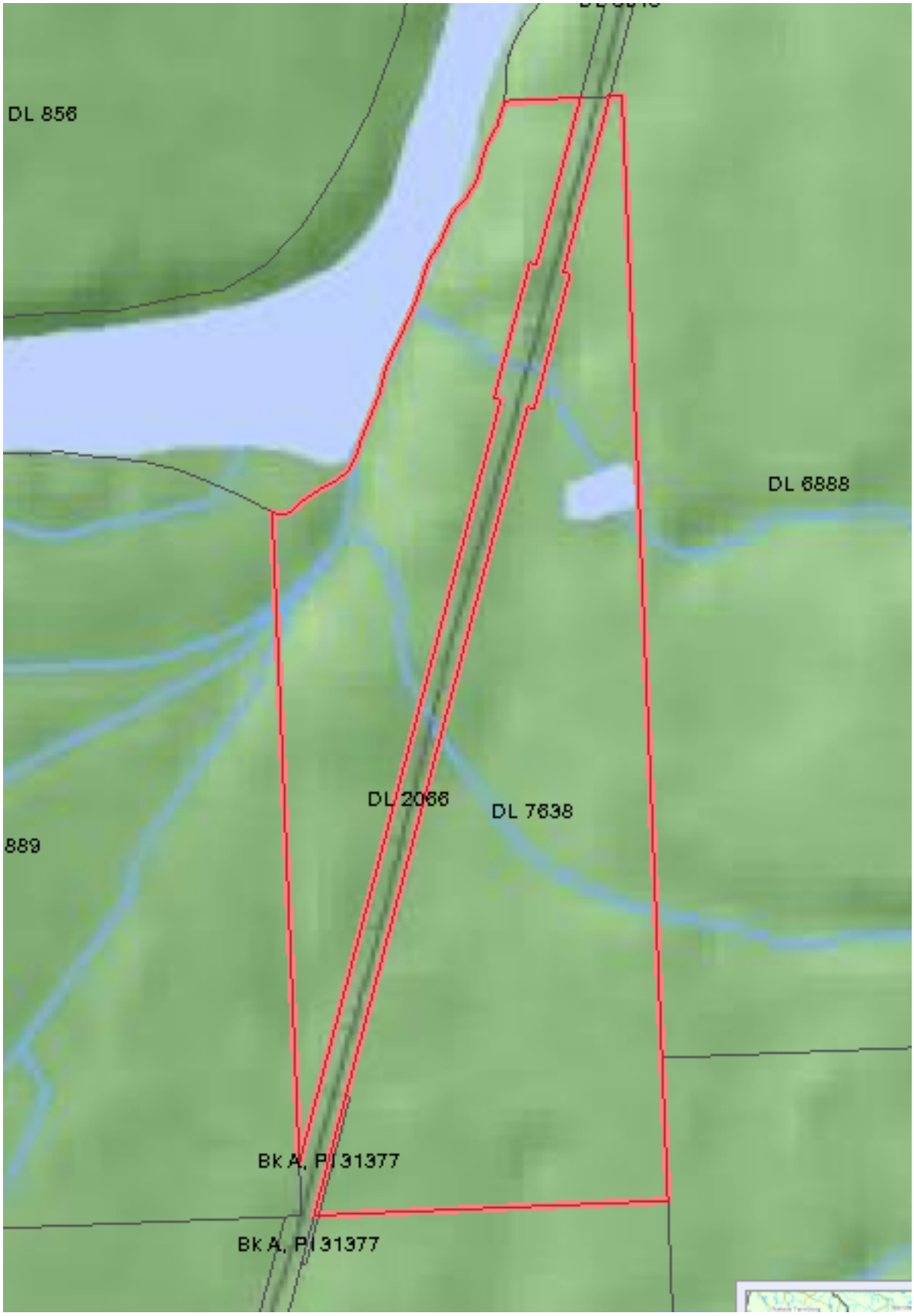
Note that the polygon with a hole has an area that is the area of the outer shell (a 10x10 square) minus the area of the hole (a 1x1 square).

9.3.4 Collections

There are four collection types, which group multiple simple geometries into sets.

- **MultiPoint**, a collection of points
- **MultiLineString**, a collection of linestrings
- **MultiPolygon**, a collection of polygons
- **GeometryCollection**, a heterogeneous collection of any geometry (including other collections)

Collections are another concept that shows up in GIS software more than in generic graphics software. They are useful for directly modeling real world objects as spatial objects. For example, how to model a lot that is split by a right-of-way? As a **MultiPolygon**, with a part on either side of the right-of-way.



Our example collection contains a polygon and a point:

```
SELECT name, ST_AsText(geom)
  FROM geometries
  WHERE name = 'Collection';
```

```
GEOMETRYCOLLECTION(POINT(2 0),POLYGON((0 0, 1 0, 1 1, 0 1, 0 0)))
```

Some of the specific spatial functions for working with collections are:

- **ST_NumGeometries(geometry)** returns the number of parts in the collection
- **ST_GeometryN(geometry,n)** returns the specified part
- **ST_Area(geometry)** returns the total area of all polygonal parts
- **ST_Length(geometry)** returns the total length of all linear parts

9.4 Geometry Input and Output

Within the database, geometries are stored on disk in a format only used by the PostGIS program. In order for external programs to insert and retrieve useful geometries, they need to be converted into a format that other applications can understand. Fortunately, PostGIS supports emitting and consuming geometries in a large number of formats:

- Well-known text (*WKT*)
  - **ST_GeomFromText(text, srid)** returns `geometry`
  - **ST_AsText(geometry)** returns `text`
  - **ST_AsEWKT(geometry)** returns `text`
- Well-known binary (*WKB*)
  - **ST_GeomFromWKB(bytea)** returns `geometry`
  - **ST_AsBinary(geometry)** returns `bytea`
  - **ST_AsEWKB(geometry)** returns `bytea`
- Geographic Mark-up Language (*GML*)
  - **ST_GeomFromGML(text)** returns `geometry`
  - **ST_AsGML(geometry)** returns `text`
- Keyhole Mark-up Language (*KML*)
  - **ST_GeomFromKML(text)** returns `geometry`
  - **ST_AsKML(geometry)** returns `text`
- *GeoJSON*

  - **ST_AsGeoJSON(geometry)** returns `text`

- Scalable Vector Graphics (*SVG*)
  - **ST_AsSVG(geometry)** returns `text`

The most common use of a constructor is to turn a text representation of a geometry into an internal representation:

Note that in addition to a text parameter with a geometry representation, we also have a numeric parameter providing the *SRID* of the geometry.

The following SQL query shows an example of *WKB* representation (the call to **encode()** is required to convert the binary output into an ASCII form for printing):

```
SELECT encode(
  ST_AsBinary(ST_GeometryFromText('LINESTRING(0 0,1 0)')),
  'hex');
```

```
0102000000020000000000000000000000000000000000000000000000000000f03f0000000000000000000
```

For the purposes of this workshop we will continue to use WKT to ensure you can read and understand the geometries we're viewing. However, most actual processes, such as viewing data in a GIS application, transferring data to a web service, or processing data remotely, WKB is the format of choice.

Since WKT and WKB were defined in the *SFSQL* specification, they do not handle 3- or 4-dimensional geometries. For these cases PostGIS has defined the Extended Well Known Text (EWKT) and Extended Well Known Binary (EWKB) formats. These provide the same formatting capabilities of WKT and WKB with the added dimensionality.

Here is an example of a 3D linestring in WKT:

```
SELECT ST_AsText(ST_GeometryFromText('LINESTRING(0 0 0,1 0 0,1 1 2)'));
```

```
LINESTRING Z (0 0 0,1 0 0,1 1 2)
```

Note that the text representation changes! This is because the text input routine for PostGIS is liberal in what it consumes. It will consume

- hex-encoded EWKB,
- extended well-known text, and
- ISO standard well-known text.

On the output side, the **ST_AsText** function is conservative, and only emits ISO standard well-known text.

In addition to the **ST_GeometryFromText** function, there are many other ways to create geometries from well-known text or similar formatted inputs:

```
-- Using ST_GeomFromText with the SRID parameter
SELECT ST_GeomFromText('POINT(2 2)',4326);

-- Using ST_GeomFromText without the SRID parameter
SELECT ST_SetSRID(ST_GeomFromText('POINT(2 2)'),4326);

-- Using a ST_Make* function
```

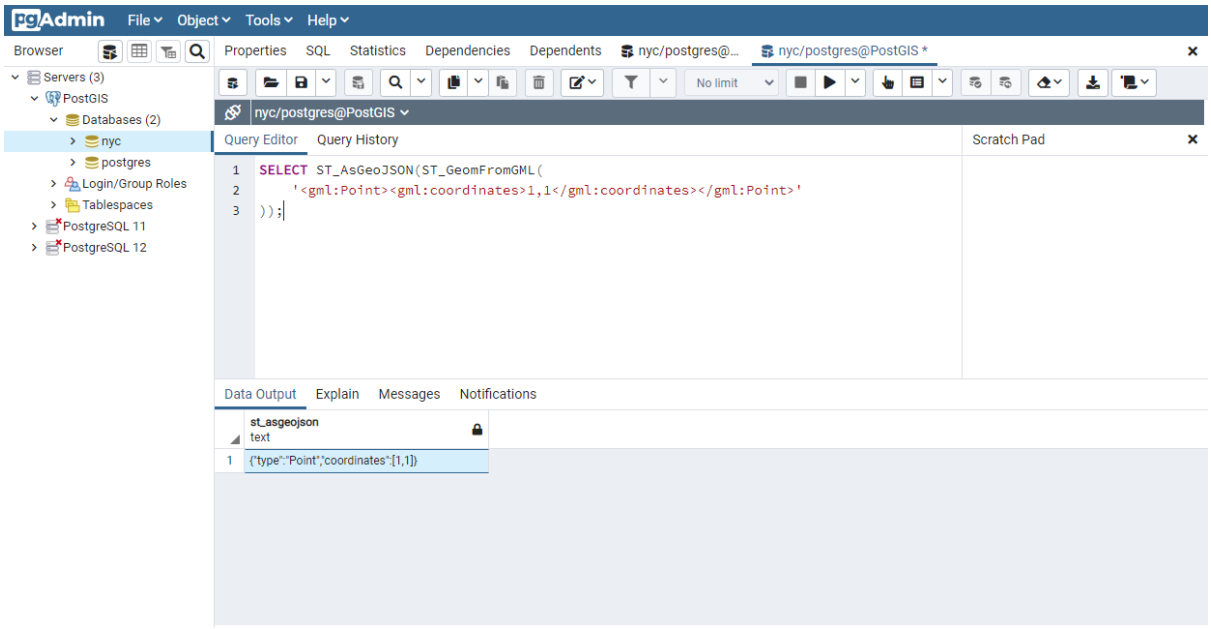
(continues on next page)

(continued from previous page)

```
SELECT ST_SetSRID(ST_MakePoint(2, 2), 4326);

-- Using PostgreSQL casting syntax and ISO WKT
SELECT ST_SetSRID('POINT(2 2)'::geometry, 4326);

-- Using PostgreSQL casting syntax and extended WKT
SELECT 'SRID=4326;POINT(2 2)'::geometry;
```

In addition to emitters for the various forms (WKT, WKB, GML, KML, JSON, SVG), PostGIS also has consumers for four (WKT, WKB, GML, KML). Most applications use the WKT or WKB geometry creation functions, but the others work too. Here's an example that consumes GML and output JSON:

```
SELECT ST_AsGeoJSON(ST_GeomFromGML('<gml:Point><gml:coordinates>1,1</
↪gml:coordinates></gml:Point>'));
```



9.5 Casting from Text

The *WKT* strings we've see so far have been of type 'text' and we have been converting them to type 'geometry' using PostGIS functions like **`ST_GeomFromText()`**.

PostgreSQL includes a short form syntax that allows data to be converted from one type to another, the casting syntax, *oldata::newtype*. So for example, this SQL converts a double into a text string.

```
SELECT 0.9::text;
```

Less trivially, this SQL converts a *WKT* string into a geometry:

```
SELECT 'POINT(0 0)'::geometry;
```

One thing to note about using casting to create geometries: unless you specify the SRID, you will get a geometry with an unknown SRID. You can specify the SRID using the "extended" well-known text form, which includes an SRID block at the front:

```
SELECT 'SRID=4326;POINT(0 0)'::geometry;
```

It's very common to use the casting notation when working with *WKT*, as well as *geometry* and *geography* columns (see *Geography*).

9.6 Function List

ST_Area: Returns the area of the surface if it is a polygon or multi-polygon. For "geometry" type area is in SRID units. For "geography" area is in square meters.

ST_AsText: Returns the Well-Known Text (WKT) representation of the geometry/geography without SRID metadata.

ST_AsBinary: Returns the Well-Known Binary (WKB) representation of the geometry/geography without SRID meta data.

ST_EndPoint: Returns the last point of a LINESTRING geometry as a POINT.

ST_AsEWKB: Returns the Well-Known Binary (WKB) representation of the geometry with SRID meta data.

ST_AsEWKT: Returns the Well-Known Text (WKT) representation of the geometry with SRID meta data.

ST_AsGeoJSON: Returns the geometry as a GeoJSON element.

ST_AsGML: Returns the geometry as a GML version 2 or 3 element.

ST_AsKML: Returns the geometry as a KML element. Several variants. Default version=2, default precision=15.

ST_AsSVG: Returns a Geometry in SVG path data given a geometry or geography object.

ST_ExteriorRing: Returns a line string representing the exterior ring of the POLYGON geometry. Return NULL if the geometry is not a polygon. Will not work with MULTIPOLYGON

ST_GeometryN: Returns the 1-based Nth geometry if the geometry is a GEOMETRYCOLLECTION, MULTIPOINT, MULTILINESTRING, MULTICURVE or MULTIPOLYGON. Otherwise, return NULL.

ST_GeomFromGML: Takes as input GML representation of geometry and outputs a PostGIS geometry object.

ST_GeomFromKML: Takes as input KML representation of geometry and outputs a PostGIS geometry object

ST_GeomFromText: Returns a specified ST_Geometry value from Well-Known Text representation (WKT).

ST_GeomFromWKB: Creates a geometry instance from a Well-Known Binary geometry representation (WKB) and optional SRID.

ST_GeometryType: Returns the geometry type of the ST_Geometry value.

ST_InteriorRingN: Returns the Nth interior linestring ring of the polygon geometry. Return NULL if the geometry is not a polygon or the given N is out of range.

ST_Length: Returns the 2d length of the geometry if it is a linestring or multilinestring. geometry are in units of spatial reference and geography are in meters (default spheroid)

ST_NDims: Returns coordinate dimension of the geometry as a small int. Values are: 2,3 or 4.

ST_NPoints: Returns the number of points (vertexes) in a geometry.

ST_NRings: If the geometry is a polygon or multi-polygon returns the number of rings.

ST_NumGeometries: If geometry is a GEOMETRYCOLLECTION (or MULTI*) returns the number of geometries, otherwise return NULL.

ST_Perimeter: Returns the length measurement of the boundary of an ST_Surface or ST_MultiSurface value. (Polygon, Multipolygon)

ST_SRID: Returns the spatial reference identifier for the ST_Geometry as defined in spatial_ref_sys table.

ST_StartPoint: Returns the first point of a LINESTRING geometry as a POINT.

ST_X: Returns the X coordinate of the point, or NULL if not available. Input must be a point.

ST_Y: Returns the Y coordinate of the point, or NULL if not available. Input must be a point.

CHAPTER

TEN

GEOMETRY EXERCISES

Here's a reminder of all the functions we have seen so far. They should be useful for the exercises!

- **`sum(expression)`** aggregate to return a sum for a set of records
- **`count(expression)`** aggregate to return the size of a set of records
- **`ST_GeometryType(geometry)`** returns the type of the geometry
- **`ST_NDims(geometry)`** returns the number of dimensions of the geometry
- **`ST_SRID(geometry)`** returns the spatial reference identifier number of the geometry
- **`ST_X(point)`** returns the X ordinate
- **`ST_Y(point)`** returns the Y ordinate
- **`ST_Length(linestring)`** returns the length of the linestring
- **`ST_StartPoint(geometry)`** returns the first coordinate as a point
- **`ST_EndPoint(geometry)`** returns the last coordinate as a point
- **`ST_NPoints(geometry)`** returns the number of coordinates in the linestring
- **`ST_Area(geometry)`** returns the area of the polygons
- **`ST_NRings(geometry)`** returns the number of rings (usually 1, more if there are holes)
- **`ST_ExteriorRing(polygon)`** returns the outer ring as a linestring
- **`ST_InteriorRingN(polygon, integer)`** returns a specified interior ring as a linestring
- **`ST_Perimeter(geometry)`** returns the length of all the rings
- **`ST_NumGeometries(multi/geomcollection)`** returns the number of parts in the collection
- **`ST_GeometryN(geometry, integer)`** returns the specified part of the collection
- **`ST_GeomFromText(text)`** returns `geometry`
- **`ST_AsText(geometry)`** returns WKT `text`
- **`ST_AsEWKT(geometry)`** returns EWKT `text`
- **`ST_GeomFromWKB(bytea)`** returns `geometry`
- **`ST_AsBinary(geometry)`** returns WKB `bytea`
- **`ST_AsEWKB(geometry)`** returns EWKB `bytea`
- **`ST_GeomFromGML(text)`** returns `geometry`

- **ST_AsGML(geometry)** returns GML `text`
- **ST_GeomFromKML(text)** returns `geometry`
- **ST_AsKML(geometry)** returns KML `text`
- **ST_AsGeoJSON(geometry)** returns JSON `text`
- **ST_AsSVG(geometry)** returns SVG `text`

Also remember the tables we have available:

- `nyc_census_blocks`
  - blkid, popn_total, boroname, geom
- `nyc_streets`
  - name, type, geom
- `nyc_subway_stations`
  - name, geom
- `nyc_neighborhoods`
  - name, boroname, geom

10.1 Exercises

- **What is the area of the 'West Village' neighborhood?**

```
SELECT ST_Area(geom)
  FROM nyc_neighborhoods
  WHERE name = 'West Village';
```

```
1044614.5296486
```

> **Note:** The area is given in square meters. To get an area in hectares, divide by 10000. To get an area in acres, divide by 4047.

- **What is the geometry type of 'Pelham St'? The length?**

```
SELECT
   ST_GeometryType(geom),
   ST_Length(geom)
  FROM nyc_streets
  WHERE name = 'Pelham St';
```

```
ST_MultiLineString
50.323
```

- **What is the GeoJSON representation of the 'Broad St' subway station?**

```
SELECT
 ST_AsGeoJSON(geom)
FROM nyc_subway_stations
WHERE name = 'Broad St';
```

```
{"type":"Point",
 "crs":{"type":"name","properties":{"name":"EPSG:26918"}},
 "coordinates":[583571.905921312,4506714.341192182]}
```

- **What is the total length of streets (in kilometers) in New York City?** (Hint: The units of measurement of the spatial data are meters, there are 1000 meters in a kilometer.)

```
SELECT Sum(ST_Length(geom)) / 1000
  FROM nyc_streets;
```

```
10418.9047172
```

- **What is the area of Manhattan in acres?** (Hint: both `nyc_census_blocks` and `nyc_neighborhoods` have a `boroname` in them.)

```
SELECT Sum(ST_Area(geom)) / 4047
  FROM nyc_neighborhoods
  WHERE boroname = 'Manhattan';
```

```
13965.3201224118
```

or...

```
SELECT Sum(ST_Area(geom)) / 4047
  FROM nyc_census_blocks
  WHERE boroname = 'Manhattan';
```

```
14601.3987215548
```

- **What is the most westerly subway station?**

```
SELECT ST_X(geom), name
  FROM nyc_subway_stations
  ORDER BY ST_X(geom)
  LIMIT 1;
```

```
Tottenville
```

- **How long is 'Columbus Cir' (aka Columbus Circle)?**

```
SELECT ST_Length(geom)
  FROM nyc_streets
  WHERE name = 'Columbus Cir';
```

```
308.34199
```

- **What is the length of streets in New York City, summarized by type?**

```
SELECT type, Sum(ST_Length(geom)) AS length
FROM nyc_streets
GROUP BY type
ORDER BY length DESC;
```

```
                      type                       |      length
-------------------------------------------------+------------------
 residential                                     | 8629870.33786606
 motorway                                        | 403622.478126363
 tertiary                                        | 360394.879051303
 motorway_link                                   | 294261.419479668
 secondary                                       | 276264.303897926
 unclassified                                    | 166936.371604458
 primary                                         | 135034.233017947
 footway                                         | 71798.4878378096
 service                                         |  28337.635038596
 trunk                                           | 20353.5819826076
 cycleway                                        | 8863.75144825929
 pedestrian                                      | 4867.05032825026
 construction                                    | 4803.08162103562
 residential; motorway_link                      | 3661.57506293745
 trunk_link                                      | 3202.18981240201
 primary_link                                    | 2492.57457083536
 living_street                                   | 1894.63905457332
 primary; residential; motorway_link; residential | 1367.76576941335
 undefined                                       |  380.53861910346
 steps                                           | 282.745221342127
 motorway_link; residential                      |  215.07778911517
```

Note: The `ORDER BY length DESC` clause sorts the result by length in descending order. The result is that most prevalent types are first in the list.

CHAPTER

ELEVEN

SPATIAL RELATIONSHIPS

So far we have only used spatial functions that measure (**ST_Area**, **ST_Length**), serialize (**ST_GeomFromText**) or deserialize (**ST_AsGML**) geometries. What these functions have in common is that they only work on one geometry at a time.

Spatial databases are powerful because they not only store geometry, they also have the ability to compare *relationships between geometries*.

Questions like "Which are the closest bike racks to a park?" or "Where are the intersections of subway lines and streets?" can only be answered by comparing geometries representing the bike racks, streets, and subway lines.

The OGC standard defines the following set of methods to compare geometries.

11.1 ST_Equals

ST_Equals(geometry A, geometry B) tests the spatial equality of two geometries.

ST_Equals returns TRUE if two geometries of the same type have identical x,y coordinate values, i.e. if the second shape is equal (identical) to the first shape.

First, let's retrieve a representation of a point from our `nyc_subway_stations` table. We'll take just the entry for 'Broad St'.

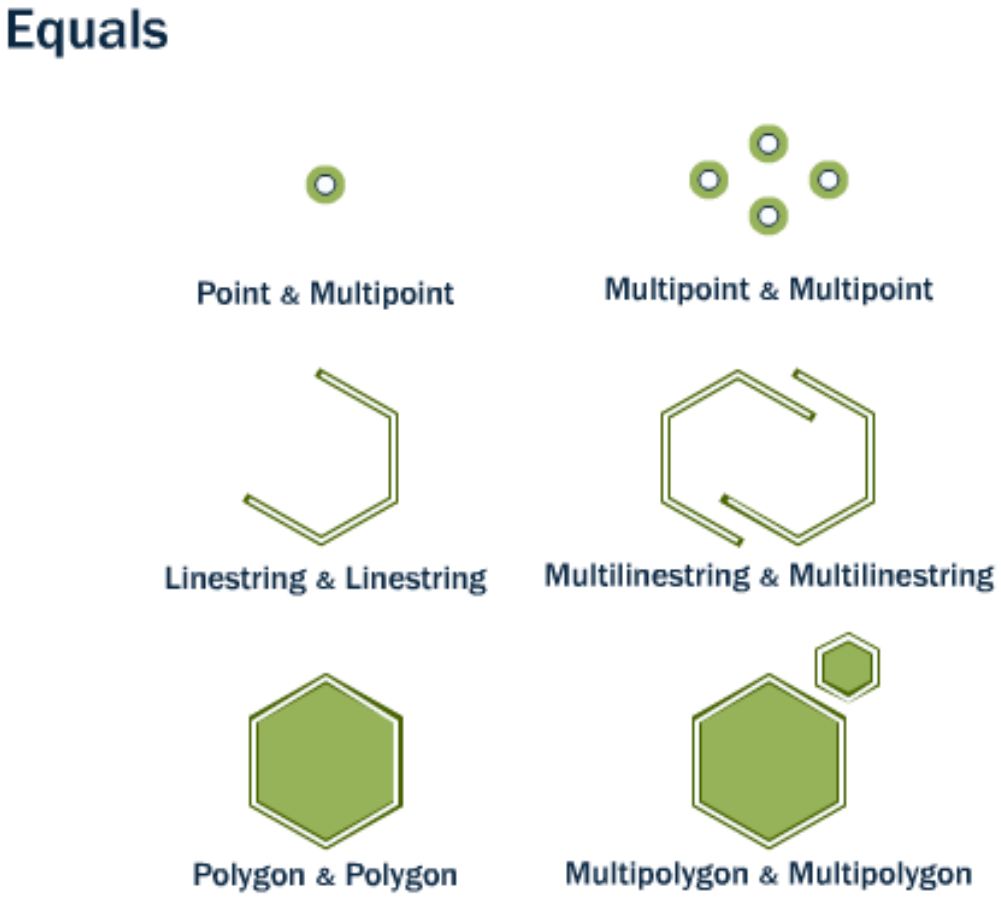
```
SELECT name, geom
FROM nyc_subway_stations
WHERE name = 'Broad St';
```

```
   name    |                        geom
-----------+---------------------------------------------------
 Broad St  | 0101000020266900000EEBD4CF27CF2141BC17D69516315141
```

Then, plug the geometry representation back into an **ST_Equals** test:

```
SELECT name
FROM nyc_subway_stations
WHERE ST_Equals(
  geom,
  '0101000020266900000EEBD4CF27CF2141BC17D69516315141');
```

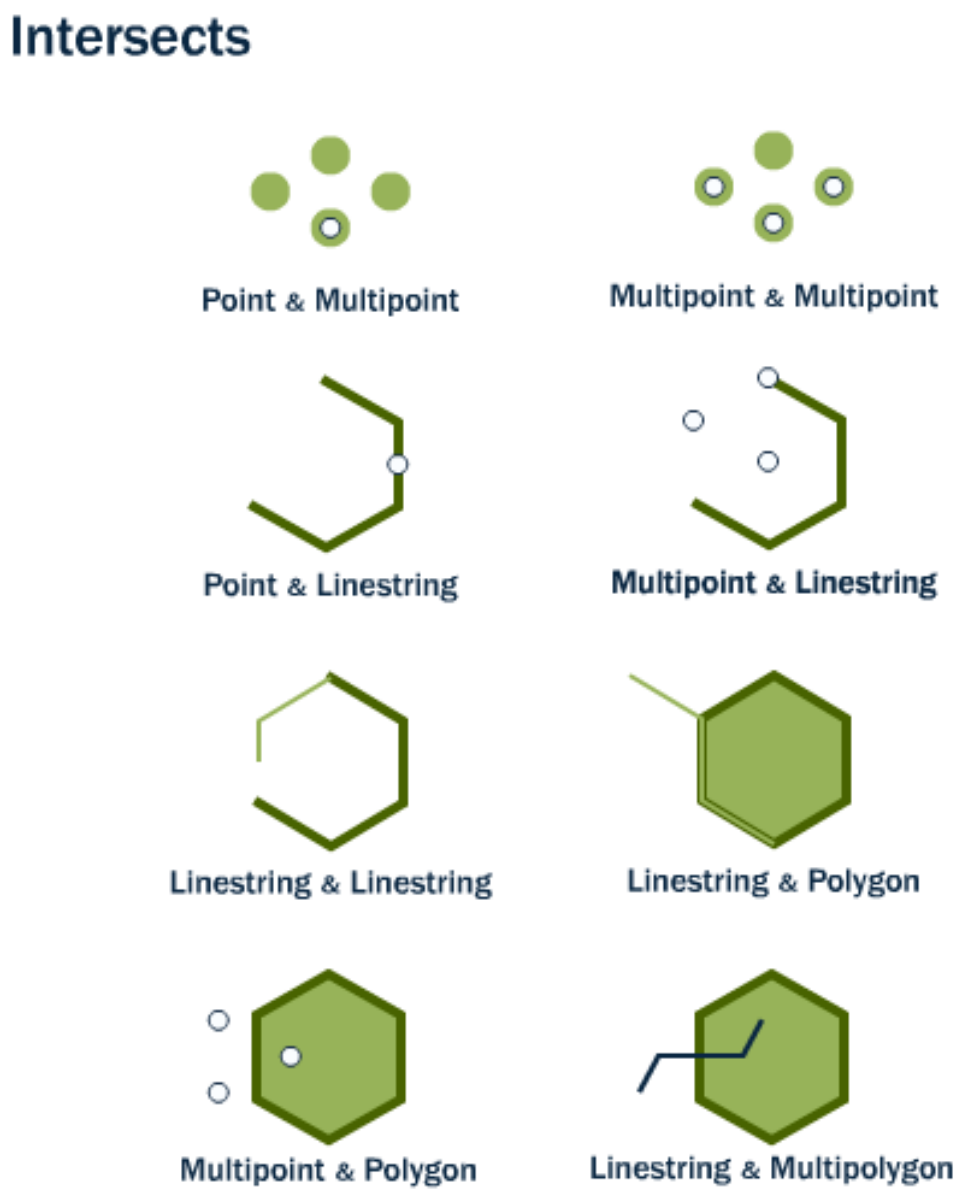
```
Broad St
```


Equals
Point & Multipoint
Multipoint & Multipoint
Linestring & Linestring
Multilinestring & Multilinestring
Polygon & Polygon
Multipolygon & Multipolygon

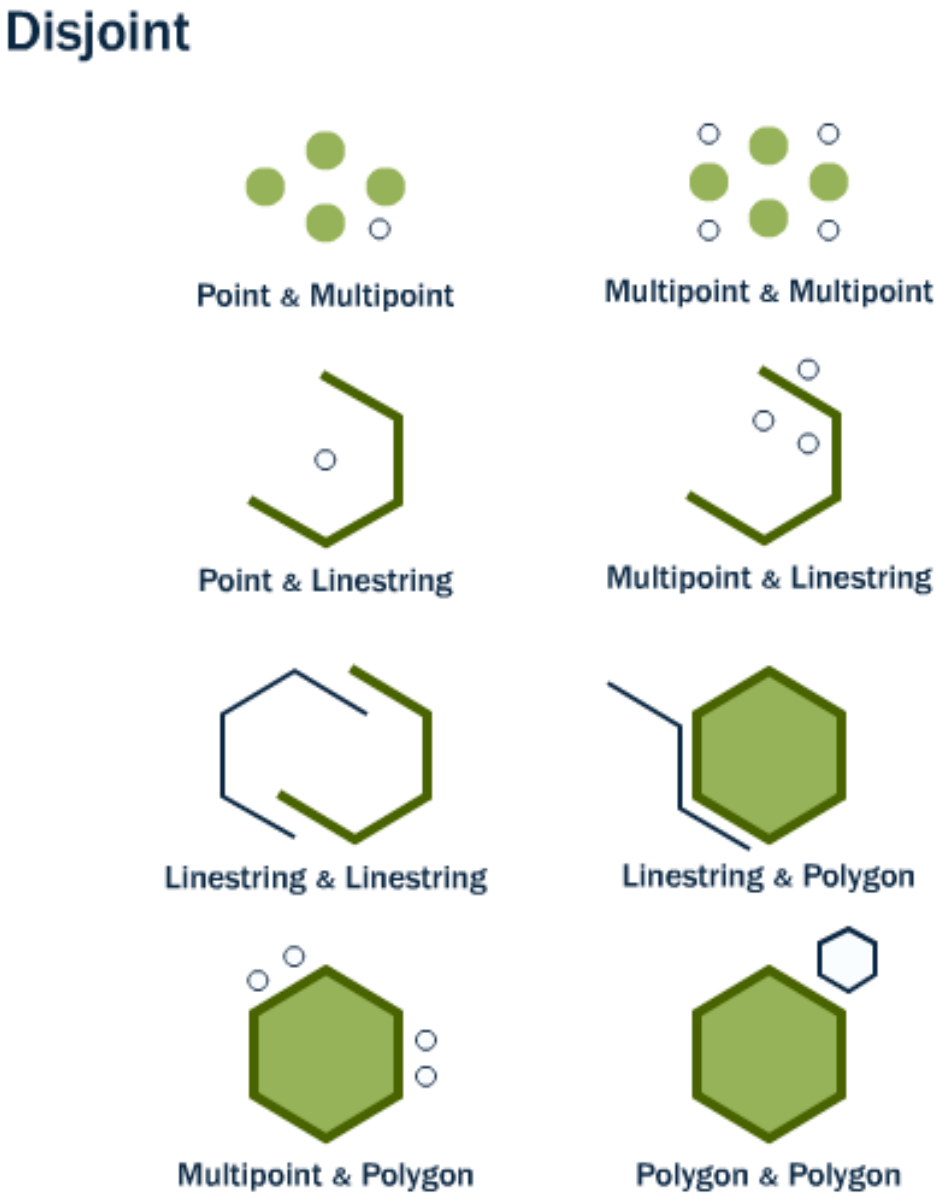
Note: The representation of the point was not very human readable (`0101000020266900000EEBD4CF27CF2141BC17D69516315141`) but it was an exact representation of the coordinate values. For a test like equality, using the exact coordinates is necessary.

11.2 ST_Intersects, ST_Disjoint, ST_Crosses and ST_Overlaps

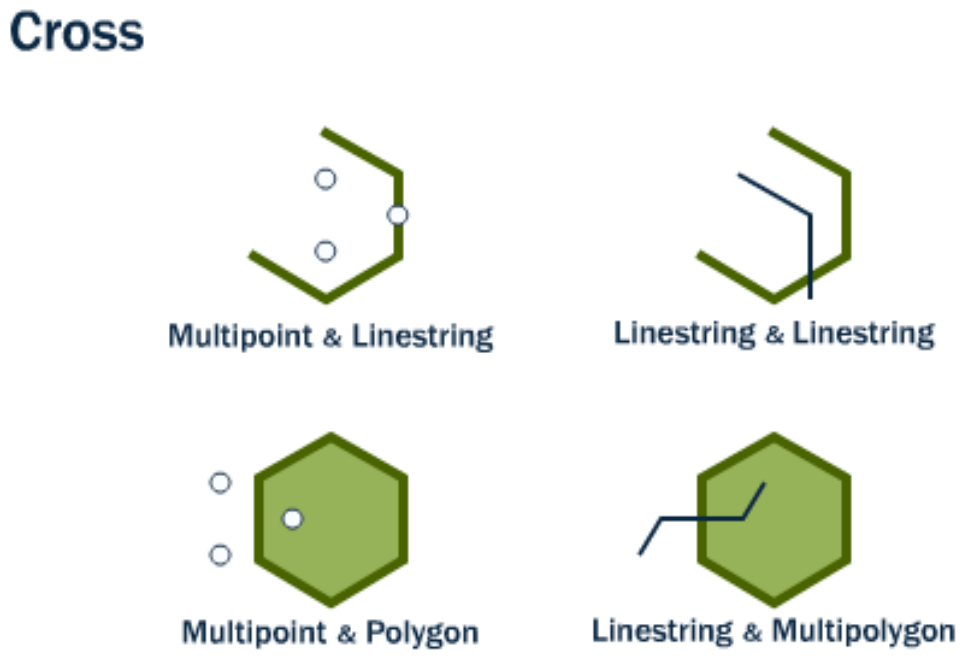
`ST_Intersects`, **`ST_Crosses`**, and **`ST_Overlaps`** test whether the interiors of the geometries intersect.



`ST_Intersects(geometry A, geometry B)` returns t (TRUE) if the two shapes have any space in common, i.e., if their boundaries or interiors intersect.


Disjoint
Point & Multipoint
Multipoint & Multipoint
Point & Linestring
Multipoint & Linestring
Linestring & Linestring
Linestring & Polygon
Multipoint & Polygon
Polygon & Polygon

The opposite of ST_Intersects is **ST_Disjoint(geometry A , geometry B)**. If two geometries are disjoint, they do not intersect, and vice-versa. In fact, it is often more efficient to test "not intersects" than to test "disjoint" because the intersects tests can be spatially indexed, while the disjoint test cannot.



For multipoint/polygon, multipoint/linestring, linestring/linestring, linestring/polygon, and linestring/multipolygon comparisons, **ST_Crosses(geometry A, geometry B)** returns t (TRUE) if the intersection results in a geometry whose dimension is one less than the maximum dimension of the two source geometries and the intersection set is interior to both source geometries.

ST_Overlaps(geometry A, geometry B) compares two geometries of the same dimension and returns TRUE if their intersection set results in a geometry different from both but of the same dimension.

Let's take our Broad Street subway station and determine its neighborhood using the **ST_Intersects** function:

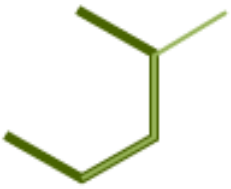
```
SELECT name, ST_AsText(geom)
FROM nyc_subway_stations
WHERE name = 'Broad St';
```

```
POINT(583571 4506714)
```

```
SELECT name, boroname
FROM nyc_neighborhoods
WHERE ST_Intersects(geom, ST_GeomFromText('POINT(583571 4506714)',26918));
```

Overlap

Multipoint & Multipoint



Linestring & Linestring

Polygon & Polygon

```
        name         | boroname
---------------------+-----------
 Financial District | Manhattan
```

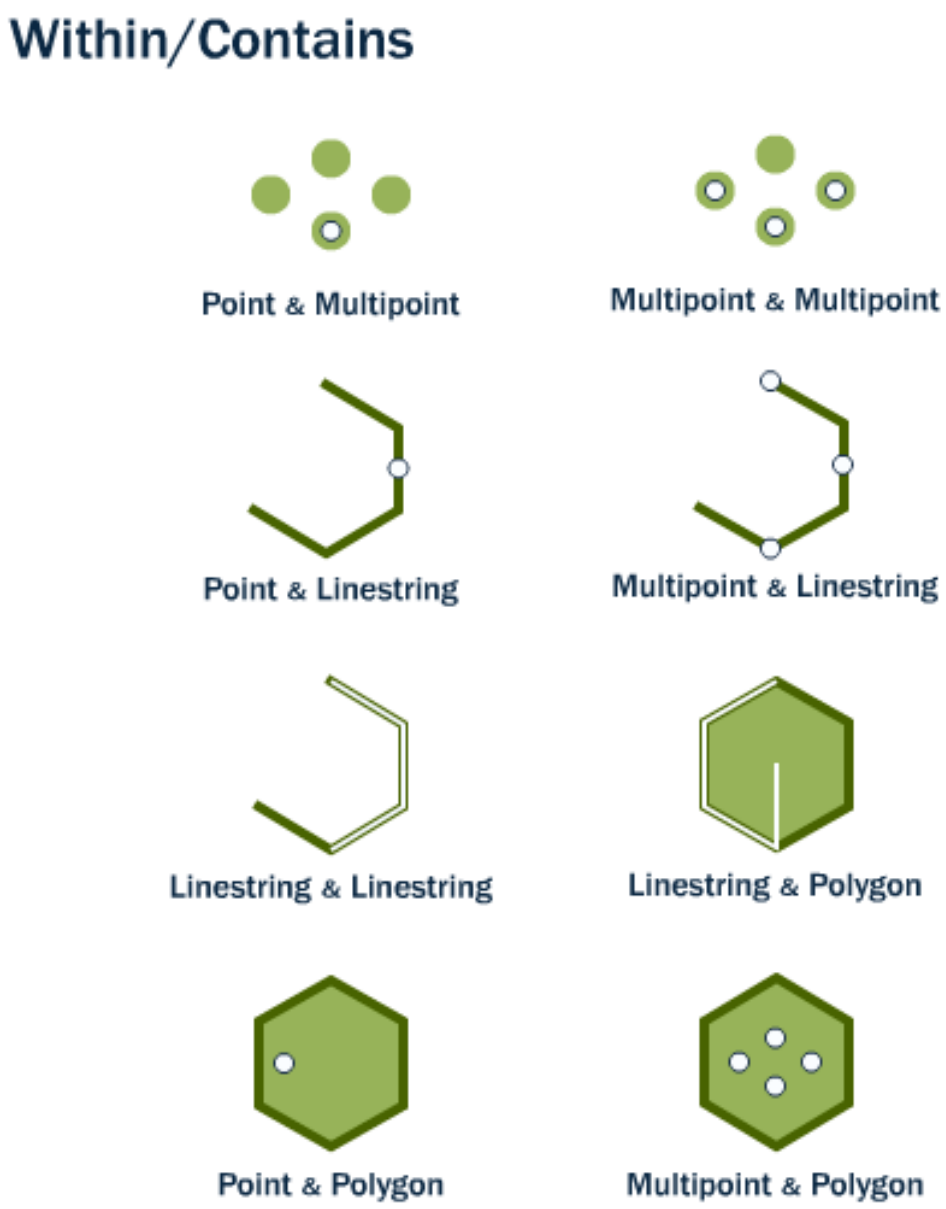
11.3 ST_Touches

ST_Touches tests whether two geometries touch at their boundaries, but do not intersect in their interiors

ST_Touches(geometry A, geometry B) returns TRUE if either of the geometries' boundaries intersect or if only one of the geometry's interiors intersects the other's boundary.

11.4 ST_Within and ST_Contains

ST_Within and **ST_Contains** test whether one geometry is fully within the other.



ST_Within(geometry A , geometry B) returns TRUE if the first geometry is completely within the second geometry. ST_Within tests for the exact opposite result of ST_Contains.

ST_Contains(geometry A, geometry B) returns TRUE if the second geometry is completely contained by the first geometry.

11.5 ST_Distance and ST_DWithin

An extremely common GIS question is "find all the stuff within distance X of this other stuff".

The **`ST_Distance(geometry A, geometry B)`** calculates the *shortest* distance between two geometries and returns it as a float. This is useful for actually reporting back the distance between objects.

```
SELECT ST_Distance(
  ST_GeometryFromText('POINT(0 5)'),
  ST_GeometryFromText('LINESTRING(-2 2, 2 2)'));
```

```
3
```

For testing whether two objects are within a distance of one another, the **`ST_DWithin`** function provides an index-accelerated true/false test. This is useful for questions like "how many trees are within a 500 meter buffer of the road?". You don't have to calculate an actual buffer, you just have to test the distance relationship.

Using our Broad Street subway station again, we can find the streets nearby (within 10 meters of) the subway stop:

```
SELECT name
FROM nyc_streets
```

(continues on next page)

(continued from previous page)

```
WHERE ST_DWithin(
        geom,
        ST_GeomFromText('POINT(583571 4506714)',26918),
        10
      );
```

```
     name
--------------
   Wall St
   Broad St
   Nassau St
```

And we can verify the answer on a map. The Broad St station is actually at the intersection of Wall, Broad and Nassau Streets.

11.6 Function List

ST_Contains(geometry A, geometry B): Returns true if and only if no points of B lie in the exterior of A, and at least one point of the interior of B lies in the interior of A.

ST_Crosses(geometry A, geometry B): Returns TRUE if the supplied geometries have some, but not all, interior points in common.

ST_Disjoint(geometry A , geometry B): Returns TRUE if the Geometries do not "spatially intersect" - if they do not share any space together.

ST_Distance(geometry A, geometry B): Returns the 2-dimensional cartesian minimum distance (based on spatial ref) between two geometries in projected units.

ST_DWithin(geometry A, geometry B, radius): Returns true if the geometries are within the specified distance (radius) of one another.

ST_Equals(geometry A, geometry B): Returns true if the given geometries represent the same geometry. Directionality is ignored.

ST_Intersects(geometry A, geometry B): Returns TRUE if the Geometries/Geography "spatially intersect" - (share any portion of space) and FALSE if they don't (they are Disjoint).

ST_Overlaps(geometry A, geometry B): Returns TRUE if the Geometries share space, are of the same dimension, but are not completely contained by each other.

ST_Touches(geometry A, geometry B): Returns TRUE if the geometries have at least one point in common, but their interiors do not intersect.

ST_Within(geometry A , geometry B): Returns true if the geometry A is completely inside geometry B

CHAPTER

TWELVE

SPATIAL RELATIONSHIPS EXERCISES

Here's a reminder of the functions we saw in the last section. They should be useful for the exercises!

- **`sum(expression)`** aggregate to return a sum for a set of records
- **`count(expression)`** aggregate to return the size of a set of records
- **`ST_Contains(geometry A, geometry B)`** returns true if geometry A contains geometry B
- **`ST_Crosses(geometry A, geometry B)`** returns true if geometry A crosses geometry B
- **`ST_Disjoint(geometry A , geometry B)`** returns true if the geometries do not "spatially intersect"
- **`ST_Distance(geometry A, geometry B)`** returns the minimum distance between geometry A and geometry B
- **`ST_DWithin(geometry A, geometry B, radius)`** returns true if geometry A is radius distance or less from geometry B
- **`ST_Equals(geometry A, geometry B)`** returns true if geometry A is the same as geometry B
- **`ST_Intersects(geometry A, geometry B)`** returns true if geometry A intersects geometry B
- **`ST_Overlaps(geometry A, geometry B)`** returns true if geometry A and geometry B share space, but are not completely contained by each other.
- **`ST_Touches(geometry A, geometry B)`** returns true if the boundary of geometry A touches geometry B
- **`ST_Within(geometry A, geometry B)`** returns true if geometry A is within geometry B

Also remember the tables we have available:

- `nyc_census_blocks`
  - blkid, popn_total, boroname, geom
- `nyc_streets`
  - name, type, geom
- `nyc_subway_stations`
  - name, geom

- nyc_neighborhoods
  - name, boroname, geom

12.1 Exercises

- **What is the geometry value for the street named 'Atlantic Commons'?**

```
SELECT ST_AsText(geom)
  FROM nyc_streets
  WHERE name = 'Atlantic Commons';
```

```
MULTILINESTRING((586781.701577724 4504202.15314339,586863.51964484␣
↪4504215.9881701))
```

- **What neighborhood and borough is Atlantic Commons in?**

```
SELECT name, boroname
FROM nyc_neighborhoods
WHERE ST_Intersects(
  geom,
  ST_GeomFromText('LINESTRING(586782 4504202,586864 4504216)', 26918)
);
```

```
    name     | boroname
-------------+----------
 Fort Green | Brooklyn
```

Note: "Hey, why did you change from a 'MULTILINESTRING' to a 'LINESTRING'?" Spatially they describe the same shape, so going from a single-item multi-geometry to a singleton saves a few keystrokes.

More importantly, we also rounded the coordinates to make them easier to read, which does actually change results: we couldn't use the ST_Touches() predicate to find out which roads join Atlantic Commons, because the coordinates are not exactly the same anymore.

- **What streets does Atlantic Commons join with?**

```
SELECT name
FROM nyc_streets
WHERE ST_DWithin(
  geom,
  ST_GeomFromText('LINESTRING(586782 4504202,586864 4504216)', 26918),
  0.1
);
```

```
     name
------------------
 Cumberland St
 Atlantic Commons
```

- **Approximately how many people live on (within 50 meters of) Atlantic Commons?**

```
SELECT Sum(popn_total)
  FROM nyc_census_blocks
  WHERE ST_DWithin(
   geom,
   ST_GeomFromText('LINESTRING(586782 4504202,586864 4504216)',␣
↪26918),
   50
  );
```

```
1438
```

CHAPTER

THIRTEEN

SPATIAL JOINS

Spatial joins are the bread-and-butter of spatial databases. They allow you to combine information from different tables by using spatial relationships as the join key. Much of what we think of as "standard GIS analysis" can be expressed as spatial joins.

In the previous section, we explored spatial relationships using a two-step process: first we extracted a subway station point for 'Broad St'; then, we used that point to ask further questions such as "what neighborhood is the 'Broad St' station in?"

Using a spatial join, we can answer the question in one step, retrieving information about the subway station and the neighborhood that contains it:

```
SELECT
  subways.name AS subway_name,
  neighborhoods.name AS neighborhood_name,
  neighborhoods.boroname AS borough
FROM nyc_neighborhoods AS neighborhoods
JOIN nyc_subway_stations AS subways
ON ST_Contains(neighborhoods.geom, subways.geom)
WHERE subways.name = 'Broad St';
```

```
 subway_name | neighborhood_name   |  borough
-------------+---------------------+-----------
 Broad St    | Financial District | Manhattan
```

We could have joined every subway station to its containing neighborhood, but in this case we wanted information about just one. Any function that provides a true/false relationship between two tables can be used to drive a spatial join, but the most commonly used ones are: **`ST_Intersects`**, **`ST_Contains`**, and **`ST_DWithin`**.

13.1 Join and Summarize

The combination of a `JOIN` with a `GROUP BY` provides the kind of analysis that is usually done in a GIS system.

For example: **"What is the population and racial make-up of the neighborhoods of Manhattan?"** Here we have a question that combines information from about population from the census with the boundaries of neighborhoods, with a restriction to just one borough of Manhattan.

```
SELECT
  neighborhoods.name AS neighborhood_name,
  Sum(census.popn_total) AS population,
```

(continues on next page)

(continued from previous page)

```
  100.0 * Sum(census.popn_white) / Sum(census.popn_total) AS white_pct,
  100.0 * Sum(census.popn_black) / Sum(census.popn_total) AS black_pct
FROM nyc_neighborhoods AS neighborhoods
JOIN nyc_census_blocks AS census
ON ST_Intersects(neighborhoods.geom, census.geom)
WHERE neighborhoods.boroname = 'Manhattan'
GROUP BY neighborhoods.name
ORDER BY white_pct DESC;
```

```
  neighborhood_name   | population | white_pct | black_pct
----------------------+------------+-----------+-----------
 Carnegie Hill        |      18763 |      90.1 |       1.4
 North Sutton Area    |      22460 |      87.6 |       1.6
 West Village         |      26718 |      87.6 |       2.2
 Upper East Side      |     203741 |      85.0 |       2.7
 Soho                 |      15436 |      84.6 |       2.2
 Greenwich Village    |      57224 |      82.0 |       2.4
 Central Park         |      46600 |      79.5 |       8.0
 Tribeca              |      20908 |      79.1 |       3.5
 Gramercy             |     104876 |      75.5 |       4.7
 Murray Hill          |      29655 |      75.0 |       2.5
 Chelsea              |      61340 |      74.8 |       6.4
 Upper West Side      |     214761 |      74.6 |       9.2
 Midtown              |      76840 |      72.6 |       5.2
 Battery Park         |      17153 |      71.8 |       3.4
 Financial District   |      34807 |      69.9 |       3.8
 Clinton              |      32201 |      65.3 |       7.9
 East Village         |      82266 |      63.3 |       8.8
 Garment District     |      10539 |      55.2 |       7.1
 Morningside Heights  |      42844 |      52.7 |      19.4
 Little Italy         |      12568 |      49.0 |       1.8
 Yorkville            |      58450 |      35.6 |      29.7
 Inwood               |      50047 |      35.2 |      16.8
 Washington Heights   |     169013 |      34.9 |      16.8
 Lower East Side      |      96156 |      33.5 |       9.1
 East Harlem          |      60576 |      26.4 |      40.4
 Hamilton Heights     |      67432 |      23.9 |      35.8
 Chinatown            |      16209 |      15.2 |       3.8
 Harlem               |     134955 |      15.1 |      67.1
```

What's going on here? Notionally (the actual evaluation order is optimized under the covers by the database) this is what happens:

1. The `JOIN` clause creates a virtual table that includes columns from both the neighborhoods and census tables.
2. The `WHERE` clause filters our virtual table to just rows in Manhattan.
3. The remaining rows are grouped by the neighborhood name and fed through the aggregation function to **`Sum()`** the population values.
4. After a little arithmetic and formatting (e.g., `GROUP BY`, `ORDER BY`) on the final numbers, our query spits out the percentages.

Note: The `JOIN` clause combines two `FROM` items. By default, we are using an `INNER JOIN`, but there are four other types of joins. For further information see the join_type definition in the PostgreSQL

documentation.

We can also use distance tests as a join key, to create summarized "all items within a radius" queries. Let's explore the racial geography of New York using distance queries.

First, let's get the baseline racial make-up of the city.

```
SELECT
  100.0 * Sum(popn_white) / Sum(popn_total) AS white_pct,
  100.0 * Sum(popn_black) / Sum(popn_total) AS black_pct,
  Sum(popn_total) AS popn_total
FROM nyc_census_blocks;
```

```
    white_pct      |    black_pct      | popn_total
-------------------+-------------------+------------
 44.0039500762811 | 25.5465789002416 |    8175032
```

So, of the 8M people in New York, about 44% are recorded as "white" and 26% are recorded as "black".

Duke Ellington once sang that "You / must take the A-train / To / go to Sugar Hill way up in Harlem." As we saw earlier, Harlem has far and away the highest African-American population in Manhattan (80.5%). Is the same true of Duke's A-train?

First, note that the contents of the `nyc_subway_stations` table `routes` field is what we are interested in to find the A-train. The values in there are a little complex.

```
SELECT DISTINCT routes FROM nyc_subway_stations;
```

```
A,C,G
4,5
D,F,N,Q
5
E,F
E,J,Z
R,W
```

Note: The `DISTINCT` keyword eliminates duplicate rows from the result. Without the `DISTINCT` keyword, the query above identifies 491 results instead of 73.

So to find the A-train, we will want any row in `routes` that has an 'A' in it. We can do this a number of ways, but today we will use the fact that **`strpos(routes,'A')`** will return a non-zero number only if 'A' is in the `routes` field.

```
SELECT DISTINCT routes
FROM nyc_subway_stations AS subways
WHERE strpos(subways.routes,'A') > 0;
```

```
A,B,C
A,C
A
A,C,G
A,C,E,L
A,S
```

(continues on next page)

(continued from previous page)

```
A,C,F
A,B,C,D
A,C,E
```

Let's summarize the racial make-up of within 200 meters of the A-train line.

```
SELECT
  100.0 * Sum(popn_white) / Sum(popn_total) AS white_pct,
  100.0 * Sum(popn_black) / Sum(popn_total) AS black_pct,
  Sum(popn_total) AS popn_total
FROM nyc_census_blocks AS census
JOIN nyc_subway_stations AS subways
ON ST_DWithin(census.geom, subways.geom, 200)
WHERE strpos(subways.routes,'A') > 0;
```

```
    white_pct      |    black_pct      | popn_total
-------------------+-------------------+------------
 45.5901255900202 | 22.0936235670937 |     189824
```

So the racial make-up along the A-train isn't radically different from the make-up of New York City as a whole.

13.2 Advanced Join

In the last section we saw that the A-train didn't serve a population that differed much from the racial make-up of the rest of the city. Are there any trains that have a non-average racial make-up?

To answer that question, we'll add another join to our query, so that we can simultaneously calculate the make-up of many subway lines at once. To do that, we'll need to create a new table that enumerates all the lines we want to summarize.

```
CREATE TABLE subway_lines ( route char(1) );
INSERT INTO subway_lines (route) VALUES
  ('A'),('B'),('C'),('D'),('E'),('F'),('G'),
  ('J'),('L'),('M'),('N'),('Q'),('R'),('S'),
  ('Z'),('1'),('2'),('3'),('4'),('5'),('6'),
  ('7');
```

Now we can join the table of subway lines onto our original query.

```
SELECT
  lines.route,
  100.0 * Sum(popn_white) / Sum(popn_total) AS white_pct,
  100.0 * Sum(popn_black) / Sum(popn_total) AS black_pct,
  Sum(popn_total) AS popn_total
FROM nyc_census_blocks AS census
JOIN nyc_subway_stations AS subways
ON ST_DWithin(census.geom, subways.geom, 200)
JOIN subway_lines AS lines
ON strpos(subways.routes, lines.route) > 0
GROUP BY lines.route
ORDER BY black_pct DESC;
```

```
 route | white_pct | black_pct | popn_total
-------+-----------+-----------+------------
 S     |      39.8 |      46.5 |      33301
 3     |      42.7 |      42.1 |     223047
 5     |      33.8 |      41.4 |     218919
 2     |      39.3 |      38.4 |     291661
 C     |      46.9 |      30.6 |     224411
 4     |      37.6 |      27.4 |     174998
 B     |      40.0 |      26.9 |     256583
 A     |      45.6 |      22.1 |     189824
 J     |      37.6 |      21.6 |     132861
 Q     |      56.9 |      20.6 |     127112
 Z     |      38.4 |      20.2 |      87131
 D     |      39.5 |      19.4 |     234931
 L     |      57.6 |      16.8 |     110118
 G     |      49.6 |      16.1 |     135012
 6     |      52.3 |      15.7 |     260240
 1     |      59.1 |      11.3 |     327742
 F     |      60.9 |       7.5 |     229439
 M     |      56.5 |       6.4 |     174196
 E     |      66.8 |       4.7 |      90958
 R     |      58.5 |       4.0 |     196999
 N     |      59.7 |       3.5 |     147792
 7     |      35.7 |       3.5 |     102401
```

As before, the joins create a virtual table of all the possible combinations available within the constraints of the `JOIN ON` restrictions, and those rows are then fed into a `GROUP` summary. The spatial magic is in the `ST_DWithin` function, that ensures only census blocks close to the appropriate subway stations are included in the calculation.

13.3 Function List

ST_Contains(geometry A, geometry B): Returns true if and only if no points of B lie in the exterior of A, and at least one point of the interior of B lies in the interior of A.

ST_DWithin(geometry A, geometry B, radius): Returns true if the geometries are within the specified distance of one another.

ST_Intersects(geometry A, geometry B): Returns TRUE if the Geometries/Geography "spatially intersect" - (share any portion of space) and FALSE if they don't (they are Disjoint).

round(v numeric, s integer): PostgreSQL math function that rounds to s decimal places

strpos(string, substring): PostgreSQL string function that returns an integer location of a specified substring.

sum(expression): PostgreSQL aggregate function that returns the sum of records in a set of records.

CHAPTER

FOURTEEN

SPATIAL JOINS EXERCISES

Here's a reminder of some of the functions we have seen. Hint: they should be useful for the exercises!

- **`sum(expression)`**: aggregate to return a sum for a set of records
- **`count(expression)`**: aggregate to return the size of a set of records
- **`ST_Area(geometry)`** returns the area of the polygons
- **`ST_AsText(geometry)`** returns WKT `text`
- **`ST_Contains(geometry A, geometry B)`** returns the true if geometry A contains geometry B
- **`ST_Distance(geometry A, geometry B)`** returns the minimum distance between geometry A and geometry B
- **`ST_DWithin(geometry A, geometry B, radius)`** returns the true if geometry A is radius distance or less from geometry B
- **`ST_GeomFromText(text)`** returns `geometry`
- **`ST_Intersects(geometry A, geometry B)`** returns the true if geometry A intersects geometry B
- **`ST_Length(linestring)`** returns the length of the linestring
- **`ST_Touches(geometry A, geometry B)`** returns the true if the boundary of geometry A touches geometry B
- **`ST_Within(geometry A, geometry B)`** returns the true if geometry A is within geometry B

Also remember the tables we have available:

- `nyc_census_blocks`
  - name, popn_total, boroname, geom
- `nyc_streets`
  - name, type, geom
- `nyc_subway_stations`
  - name, routes, geom
- `nyc_neighborhoods`
  - name, boroname, geom

14.1 Exercises

- **What subway station is in 'Little Italy'? What subway route is it on?**

```
SELECT s.name, s.routes
FROM nyc_subway_stations AS s
JOIN nyc_neighborhoods AS n
ON ST_Contains(n.geom, s.geom)
WHERE n.name = 'Little Italy';
```

```
   name     | routes
------------+--------
 Spring St | 6
```

- **What are all the neighborhoods served by the 6-train?** (Hint: The `routes` column in the `nyc_subway_stations` table has values like 'B,D,6,V' and 'C,6')

```
SELECT DISTINCT n.name, n.boroname
FROM nyc_subway_stations AS s
JOIN nyc_neighborhoods AS n
ON ST_Contains(n.geom, s.geom)
WHERE strpos(s.routes,'6') > 0;
```

```
        name         | boroname
---------------------+-----------
 Midtown             | Manhattan
 Hunts Point         | The Bronx
 Gramercy            | Manhattan
 Little Italy        | Manhattan
 Financial District  | Manhattan
 South Bronx         | The Bronx
 Yorkville           | Manhattan
 Murray Hill         | Manhattan
 Mott Haven          | The Bronx
 Upper East Side     | Manhattan
 Chinatown           | Manhattan
 East Harlem         | Manhattan
 Greenwich Village   | Manhattan
 Parkchester         | The Bronx
 Soundview           | The Bronx
```

Note: We used the `DISTINCT` keyword to remove duplicate values from our result set where there were more than one subway station in a neighborhood.

- **After 9/11, the 'Battery Park' neighborhood was off limits for several days. How many people had to be evacuated?**

```
SELECT Sum(popn_total)
FROM nyc_neighborhoods AS n
JOIN nyc_census_blocks AS c
ON ST_Intersects(n.geom, c.geom)
WHERE n.name = 'Battery Park';
```

```
17153
```

- **What neighborhood has the highest population density (persons/km2)?**

```
SELECT
  n.name,
  Sum(c.popn_total) / (ST_Area(n.geom) / 1000000.0) AS popn_per_sqkm
FROM nyc_census_blocks AS c
JOIN nyc_neighborhoods AS n
ON ST_Intersects(c.geom, n.geom)
GROUP BY n.name, n.geom
ORDER BY popn_per_sqkm DESC LIMIT 2;
```

```
      name       |  popn_per_sqkm
-----------------+------------------
 North Sutton Area | 68435.13283772678
 East Village      | 50404.48341332535
```

CHAPTER

FIFTEEN

SPATIAL INDEXING

Recall that spatial index is one of the three key features of a spatial database. Indexes make using a spatial database for large data sets possible. Without indexing, any search for a feature would require a "sequential scan" of every record in the database. Indexing speeds up searching by organizing the data into a search tree which can be quickly traversed to find a particular record.

Spatial indices are one of the greatest assets of PostGIS. In the previous example building spatial joins requires comparing whole tables with each other. This can get very costly: joining two tables of 10,000 records each without indexes would require 100,000,000 comparisons; with indexes the cost could be as low as 20,000 comparisons.

Our data load file already included spatial indexes for all the tables, so in order to demonstrate the efficacy of indexes we will have to first remove them.

Let's run a query on `nyc_census_blocks` **without** our spatial index.

Our first step is to **remove** the index.

```
DROP INDEX nyc_census_blocks_geom_idx;
```

Note: The `DROP INDEX` statement drops an existing index from the database system. For more information, see the PostgreSQL documentation.

Now, watch the "Timing" meter at the lower right-hand corner of the pgAdmin query window and run the following. Our query searches through every single census block in order to identify blocks that contain subway stops that start with "B".

```
SELECT count(blocks.blkid)
 FROM nyc_census_blocks blocks
 JOIN nyc_subway_stations subways
 ON ST_Contains(blocks.geom, subways.geom)
 WHERE subways.name LIKE 'B%';
```

```
    count
---------------
    46
```

The `nyc_census_blocks` table is very small (only a few thousand records) so even without an index, the query only takes **300 ms** on my test computer.

Now add the spatial index back in and run the query again.

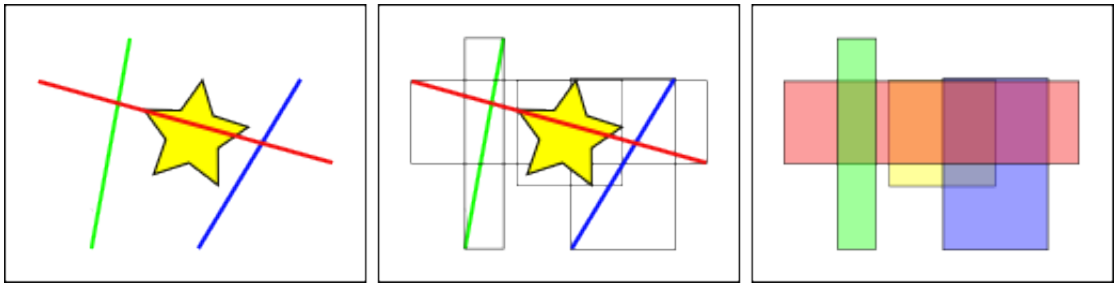
```
CREATE INDEX nyc_census_blocks_geom_idx
  ON nyc_census_blocks
  USING GIST (geom);
```

Note: The `USING GIST` clause tells PostgreSQL to use the generic index structure (GIST) when building the index. If you receive an error that looks like `ERROR: index row requires 11340 bytes, maximum size is 8191` when creating your index, you have likely neglected to add the `USING GIST` clause.

On my test computer the time drops to **50 ms**. The larger your table, the larger the relative speed improvement of an indexed query will be.

15.1 How Spatial Indexes Work

Standard database indexes create a hierarchical tree based on the values of the column being indexed. Spatial indexes are a little different – they are unable to index the geometric features themselves and instead index the bounding boxes of the features.



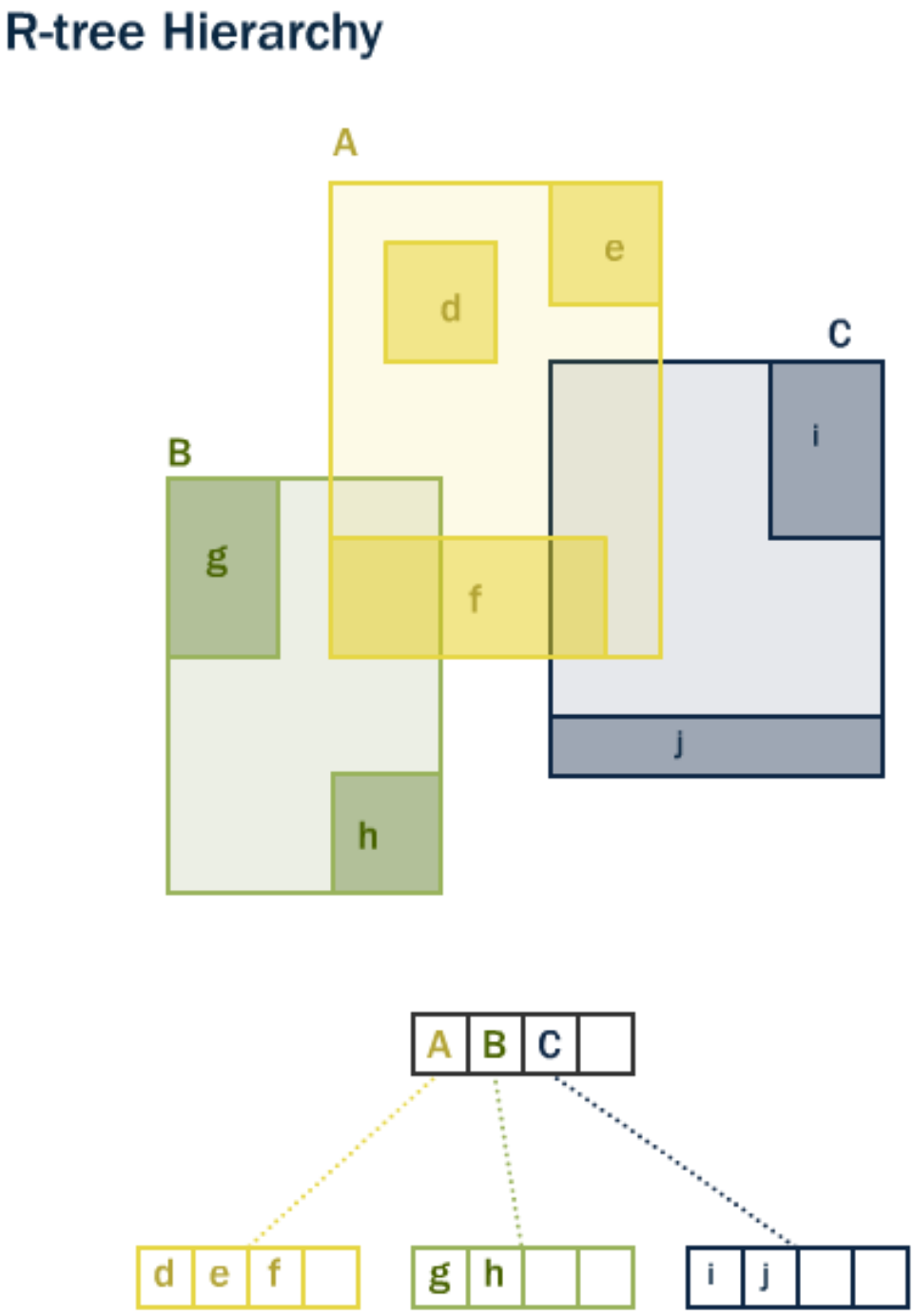
In the figure above, the number of lines that intersect the yellow star is **one**, the red line. But the bounding boxes of features that intersect the yellow box is **two**, the red and blue ones.

The way the database efficiently answers the question "what lines intersect the yellow star" is to first answer the question "what boxes intersect the yellow box" using the index (which is very fast) and then do an exact calculation of "what lines intersect the yellow star" **only for those features returned by the first test**.

For a large table, this "two pass" system of evaluating the approximate index first, then carrying out an exact test can radically reduce the amount of calculations necessary to answer a query.

Both PostGIS and Oracle Spatial share the same "R-Tree"[1] spatial index structure. R-Trees break up data into rectangles, and sub-rectangles, and sub-sub rectangles, etc. It is a self-tuning index structure that automatically handles variable data density, differing amounts of object overlap, and object size.

[1] http://postgis.net/docs/support/rtree.pdf


R-tree Hierarchy
A
B
C
d
e
g
f
i
j
h
A B C
d e f
g h
i j

15.2 Spatially Indexed Functions

Only a subset of functions will automatically make use of a spatial index, if one is available.

- ST_Intersects
- ST_Contains
- ST_Within
- ST_DWithin
- ST_ContainsProperly
- ST_CoveredBy
- ST_Covers
- ST_Overlaps
- ST_Crosses
- ST_DFullyWithin
- ST_3DIntersects
- ST_3DDWithin
- ST_3DDFullyWithin
- ST_LineCrossingDirection
- ST_OrderingEquals
- ST_Equals

The first four are the ones most commonly used in queries, and ST_DWithin is very important for doing "within a distance" or "within a radius" style queries while still getting a performance boost from the index.

In order to add index acceleration to other functions that are not in this list (most commonly, ST_Relate) add an index-only clause as descibed below.

15.3 Index-Only Queries

Most of the commonly used functions in PostGIS (**ST_Contains**, **ST_Intersects**, **ST_DWithin**, etc) include an index filter automatically. But some functions (e.g., **ST_Relate**) do not include an index filter.

To do a bounding-box search using the index (and no filtering), make use of the **&&** operator. For geometries, the **&&** operator means "bounding boxes overlap or touch" in the same way that for numbers the **=** operator means "values are the same".

Let's compare an index-only query for the population of the 'West Village' to a more exact query. Using **&&** our index-only query looks like the following:

```
SELECT Sum(popn_total)
FROM nyc_neighborhoods neighborhoods
JOIN nyc_census_blocks blocks
```

(continues on next page)

(continued from previous page)

```
ON neighborhoods.geom && blocks.geom
WHERE neighborhoods.name = 'West Village';
```

```
49821
```

Now let's do the same query using the more exact **ST_Intersects** function.

```
SELECT Sum(popn_total)
FROM nyc_neighborhoods neighborhoods
JOIN nyc_census_blocks blocks
ON ST_Intersects(neighborhoods.geom, blocks.geom)
WHERE neighborhoods.name = 'West Village';
```

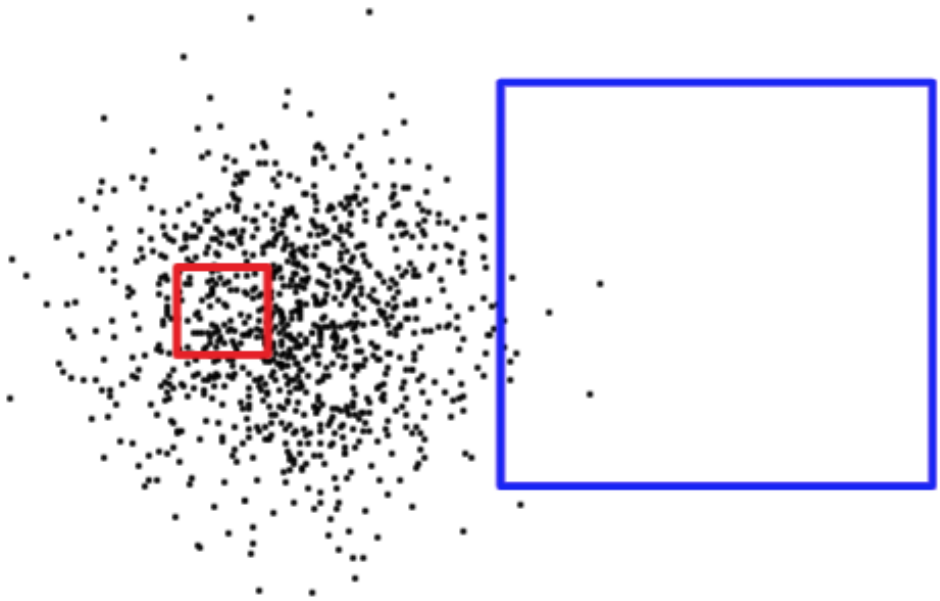
```
26718
```

A much lower answer! The first query summed up every block whose bounding box intersects the neighborhood's bounding box; the second query only summed up those blocks that intersect the neighborhood itself.

15.4 Analyzing

The PostgreSQL query planner intelligently chooses when to use or not to use indexes to evaluate a query. Counter-intuitively, it is not always faster to do an index search: if the search is going to return every record in the table, traversing the index tree to get each record will actually be slower than just sequentially reading the whole table from the start.

Knowing the size of the query rectangle is not enough to pin down whether a query will return a large number or small number of records. Below, the red square is small, but will return many more records than the blue square.



In order to figure out what situation it is dealing with (reading a small part of the table versus reading a large portion of the table), PostgreSQL keeps statistics about the distribution of data in each indexed

table column. By default, PostgreSQL gathers statistics on a regular basis. However, if you dramatically change the contents of your table within a short period of time, the statistics will not be up-to-date.

To ensure the statistics match your table contents, it is wise the to run the `ANALYZE` command after bulk data loads and deletes in your tables. This force the statistics system to gather data for all your indexed columns.

The `ANALYZE` command asks PostgreSQL to traverse the table and update its internal statistics used for query plan estimation (query plan analysis will be discussed later).

```
ANALYZE nyc_census_blocks;
```

15.5 Vacuuming

It's worth stressing that just creating an index is not enough to allow PostgreSQL to use it effectively. VACUUMing must be performed whenever a large number of UPDATEs, INSERTs or DELETEs are issued against a table. The `VACUUM` command asks PostgreSQL to reclaim any unused space in the table pages left by updates or deletes to records.

Vacuuming is so critical for the efficient running of the database that PostgreSQL provides an "autovacuum" facility by default.

Autovacuum both vacuums (recovers space) and analyzes (updates statistics) on your tables at sensible intervals determined by the level of activity. While this is essential for highly transactional databases, it is not advisable to wait for an autovacuum run after adding indices or bulk-loading data. Whenever a large batch update is performed, you should manually run `VACUUM`.

Vacuuming and analyzing the database can be performed separately as needed. Issuing `VACUUM` command will not update the database statistics; likewise issuing an `ANALYZE` command will not recover unused table rows. Both commands can be run against the entire database, a single table, or a single column.

```
VACUUM ANALYZE nyc_census_blocks;
```

15.6 Function List

geometry_a && geometry_b: Returns TRUE if A's bounding box overlaps B's.

geometry_a = geometry_b: Returns TRUE if A's bounding box is the same as B's.

ST_Intersects(geometry_a, geometry_b): Returns TRUE if the Geometries/Geography "spatially intersect" - (share any portion of space) and FALSE if they don't (they are Disjoint).

CHAPTER

SIXTEEN

PROJECTING DATA

The earth is not flat, and there is no simple way of putting it down on a flat paper map (or computer screen), so people have come up with all sorts of ingenious solutions, each with pros and cons. Some projections preserve area, so all objects have a relative size to each other; other projections preserve angles (conformal) like the Mercator projection; some projections try to find a good intermediate mix with only little distortion on several parameters. Common to all projections is that they transform the (spherical) world onto a flat Cartesian coordinate system, and which projection to choose depends on how you will be using the data.

We've already encountered projections when we *loaded our nyc data*. (Recall that pesky SRID 26918). Sometimes, however, you need to transform and re-project between spatial reference systems. PostGIS includes built-in support for changing the projection of data, using the **`ST_Transform(geometry, srid)`** function. For managing the spatial reference identifiers on geometries, PostGIS provides the **`ST_SRID(geometry)`** and **`ST_SetSRID(geometry, srid)`** functions.

We can confirm the SRID of our data with the **`ST_SRID`** function:

```
SELECT ST_SRID(geom) FROM nyc_streets LIMIT 1;
```

```
26918
```

And what is definition of "26918"? As we saw in "*loading data section*", the definition is contained in the `spatial_ref_sys` table. In fact, **two** definitions are there. The "well-known text" (*WKT*) definition is in the `srtext` column, and there is a second definition in "proj.4" format in the `proj4text` column.

```
SELECT * FROM spatial_ref_sys WHERE srid = 26918;
```

The PostGIS reprojection engine will attempt to find the best projection from the `spatial_ref_sys` table:

- **auth_name / auth_srid** If proj can find a valid "authority name" and "authority srid" in its internal catalogue, it will use that to generate a projection definition.
- **srtext** If proj can parse and form a definition object from the `srtext` it will use that.
- **proj4text** Finally, proj will attempt to process the `proj4text`.

All this redundancy means that all you need to create a new projection in PostGIS is either a valid `srtext` string or `proj4text` string. All the common authority name/code pairs are already loaded in the table by default.

If you have a choice when creating a custom projection, fill out the `srtext` column, since that column is also used by external programs like GeoServer, QGIS, and FME and others.

16.1 Comparing Data

Taken together, a coordinate and an SRID define a location on the globe. Without an SRID, a coordinate is just an abstract notion. A "Cartesian" coordinate plane is defined as a "flat" coordinate system placed on the surface of Earth. Because PostGIS functions work on such a plane, comparison operations require that both geometries be represented in the same SRID.

If you feed in geometries with differing SRIDs you will just get an error:

```
SELECT ST_Equals(
         ST_GeomFromText('POINT(0 0)', 4326),
         ST_GeomFromText('POINT(0 0)', 26918)
         );
```

```
ERROR:  ST_Equals: Operation on mixed SRID geometries (Point, 4326) !=␣
↪(Point, 26918)
```

Note: Be careful of getting too happy with using **`ST_Transform`** for on-the-fly conversion. Spatial indexes are built using SRID of the stored geometries. If comparison are done in a different SRID, spatial indexes are (often) not used. It is best practice to choose **one SRID** for all the tables in your database. Only use the transformation function when you are reading or writing data to external applications.

16.2 Transforming Data

If we return to our proj4 definition for SRID 26918, we can see that our working projection is UTM (Universal Transverse Mercator) of zone 18, with meters as the unit of measurement.

```
SELECT srtext FROM spatial_ref_sys WHERE srid = 26918;
```

```
PROJCS["NAD83 / UTM zone 18N",
  GEOGCS["NAD83",
    DATUM["North_American_Datum_1983",
      SPHEROID["GRS 1980",6378137,298.257222101,AUTHORITY["EPSG","7019"]],
      TOWGS84[0,0,0,0,0,0,0],
      AUTHORITY["EPSG","6269"]],
    PRIMEM["Greenwich",0,AUTHORITY["EPSG","8901"]],
    UNIT["degree",0.0174532925199433,AUTHORITY["EPSG","9122"]],
    AUTHORITY["EPSG","4269"]],
  PROJECTION["Transverse_Mercator"],
  PARAMETER["latitude_of_origin",0],
  PARAMETER["central_meridian",-75],
  PARAMETER["scale_factor",0.9996],
  PARAMETER["false_easting",500000],
  PARAMETER["false_northing",0],
  UNIT["metre",1,AUTHORITY["EPSG","9001"]],
  AXIS["Easting",EAST],AXIS["Northing",NORTH],
  AUTHORITY["EPSG","26918"]]
```

Let's convert some data from our working projection to geographic coordinates – also known as "longitude/latitude".

To convert data from one SRID to another, you must first verify that your geometry has a valid SRID. Since we have already confirmed a valid SRID, we next need the SRID of the projection to transform into. In other words, what is the SRID of geographic coordinates?

The most common SRID for geographic coordinates is 4326, which corresponds to "longitude/latitude on the WGS84 spheroid". You can see the definition here:

https://epsg.io/4326

You can also pull the definitions from the `spatial_ref_sys` table:

```
SELECT srtext FROM spatial_ref_sys WHERE srid = 4326;
```

```
GEOGCS["WGS 84",
  DATUM["WGS_1984",
    SPHEROID["WGS 84",6378137,298.257223563,AUTHORITY["EPSG","7030"]],
    AUTHORITY["EPSG","6326"]],
  PRIMEM["Greenwich",0,AUTHORITY["EPSG","8901"]],
  UNIT["degree",0.01745329251994328,AUTHORITY["EPSG","9122"]],
  AUTHORITY["EPSG","4326"]]
```

Let's convert the coordinates of the 'Broad St' subway station into geographics:

```
SELECT ST_AsText(ST_Transform(geom,4326))
FROM nyc_subway_stations
WHERE name = 'Broad St';
```

```
POINT(-74.01067146887341 40.70710481558761)
```

If you load data or create a new geometry without specifying an SRID, the SRID value will be 0. Recall in *Geometries*, that when we created our `geometries` table we didn't specify an SRID. If we query our database, we should expect all the `nyc_` tables to have an SRID of 26918, while the `geometries` table defaulted to an SRID of 0.

To view a table's SRID assignment, query the database's `geometry_columns` table.

```
SELECT f_table_name AS name, srid
FROM geometry_columns;
```

```
        name          | srid
----------------------+-------
 nyc_census_blocks    | 26918
 nyc_homicides        | 26918
 nyc_neighborhoods    | 26918
 nyc_streets          | 26918
 nyc_subway_stations  | 26918
 geometries           |     0
```

However, if you know what the SRID of the coordinates is supposed to be, you can set it post-facto, using **`ST_SetSRID`** on the geometry. Then you will be able to transform the geometry into other systems.

```
SELECT ST_AsText(
    ST_Transform(
      ST_SetSRID(geom,26918),
      4326)
```

(continues on next page)

(continued from previous page)

```
    )
FROM geometries;
```

16.3 Function List

ST_AsText: Returns the Well-Known Text (WKT) representation of the geometry/geography without SRID metadata.

ST_SetSRID(geometry, srid): Sets the SRID on a geometry to a particular integer value.

ST_SRID(geometry): Returns the spatial reference identifier for the ST_Geometry as defined in spatial_ref_sys table.

ST_Transform(geometry, srid): Returns a new geometry with its coordinates transformed to the SRID referenced by the integer parameter.

CHAPTER

SEVENTEEN

PROJECTION EXERCISES

Here's a reminder of some of the functions we have seen. Hint: they should be useful for the exercises!

- **`sum(expression)`** aggregate to return a sum for a set of records
- **`ST_Length(linestring)`** returns the length of the linestring
- **`ST_SRID(geometry)`** returns the SRID of the geometry
- **`ST_Transform(geometry, srid)`** converts geometries into different spatial reference systems
- **`ST_GeomFromText(text)`** returns `geometry`
- **`ST_AsText(geometry)`** returns WKT `text`
- **`ST_AsGML(geometry)`** returns GML `text`

Remember the online resources that are available to you:

- https://epsg.io/

Also remember the tables we have available:

- `nyc_census_blocks`
  - name, popn_total, boroname, geom
- `nyc_streets`
  - name, type, geom
- `nyc_subway_stations`
  - name, geom
- `nyc_neighborhoods`
  - name, boroname, geom

17.1 Exercises

- **What is the length of all streets in New York, as measured in UTM 18?**

```
SELECT Sum(ST_Length(geom))
  FROM nyc_streets;
```

```
10418904.7172
```

- **What is the WKT definition of SRID 2831?**

```
SELECT srtext FROM spatial_ref_sys
WHERE SRID = 2831;
```

Or, via https://epsg.io/2831

```
PROJCS["NAD83(HARN) / New York Long Island",
  GEOGCS["NAD83(HARN)",
    DATUM["NAD83 (High Accuracy Regional Network)",
      SPHEROID["GRS 1980", 6378137.0, 298.257222101,
        AUTHORITY["EPSG","7019"]],
      TOWGS84[-0.991, 1.9072, 0.5129, 0.0257899075194932, -0.
↪00965009896027040 2, -0.011659943232342112, 0.0],
      AUTHORITY["EPSG","6152"]],
    PRIMEM["Greenwich", 0.0,
      AUTHORITY["EPSG","8901"]],
    UNIT["degree", 0.017453292519943295],
    AXIS["Geodetic longitude", EAST],
    AXIS["Geodetic latitude", NORTH],
    AUTHORITY["EPSG","4152"]],
  PROJECTION["Lambert Conic Conformal (2SP)",
    AUTHORITY["EPSG","9802"]],
  PARAMETER["central_meridian", -74.0],
  PARAMETER["latitude_of_origin", 40.166666666666664],
  PARAMETER["standard_parallel_1", 41.03333333333333],
  PARAMETER["false_easting", 300000.0],
  PARAMETER["false_northing", 0.0],
  PARAMETER["scale_factor", 1.0],
  PARAMETER["standard_parallel_2", 40.666666666666664],
  UNIT["m", 1.0],
  AXIS["Easting", EAST],
  AXIS["Northing", NORTH],
  AUTHORITY["EPSG","2831"]]
```

- **What is the length of all streets in New York, as measured in SRID 2831?**

```
SELECT Sum(ST_Length(ST_Transform(geom,2831)))
  FROM nyc_streets;
```

```
10421993.706374
```

Note: The difference between the UTM 18 and the State Plane Long Island measurements is (10421993 - 10418904)/10418904, or 0.02%. Calculated on the spheroid using *Geography* the total street length is 10421999, which is closer to the State Plane value. This is not surprising,

since the State Plane Long Island projection is precisely calibrated for a very small area (New York City) while UTM 18 has to provide reasonable results for a large regional area.

- **How many streets cross the 74th meridian?**

```
SELECT Count(*)
FROM nyc_streets
WHERE ST_Intersects(
  ST_Transform(geom, 4326),
  'SRID=4326;LINESTRING(-74 20, -74 60)'
  );
```

```
223
```

The "74th meridian" is a fancy way of saying "a vertical line in geographics where the X value is -74". We can construct such a line and then compare it to the streets, projected into geographics also. Projecting the line into UTM and comparing it there will return a slightly different answer. To get the same answer, you need to "segmentize" it, so it has more points, before transforming.

```
SELECT Count(*)
FROM nyc_streets
WHERE ST_Intersects(
  geom,
  ST_Transform(ST_Segmentize('SRID=4326;LINESTRING(-74 20, -74 60)
↪'::geometry,0.001), 26918)
  );
```

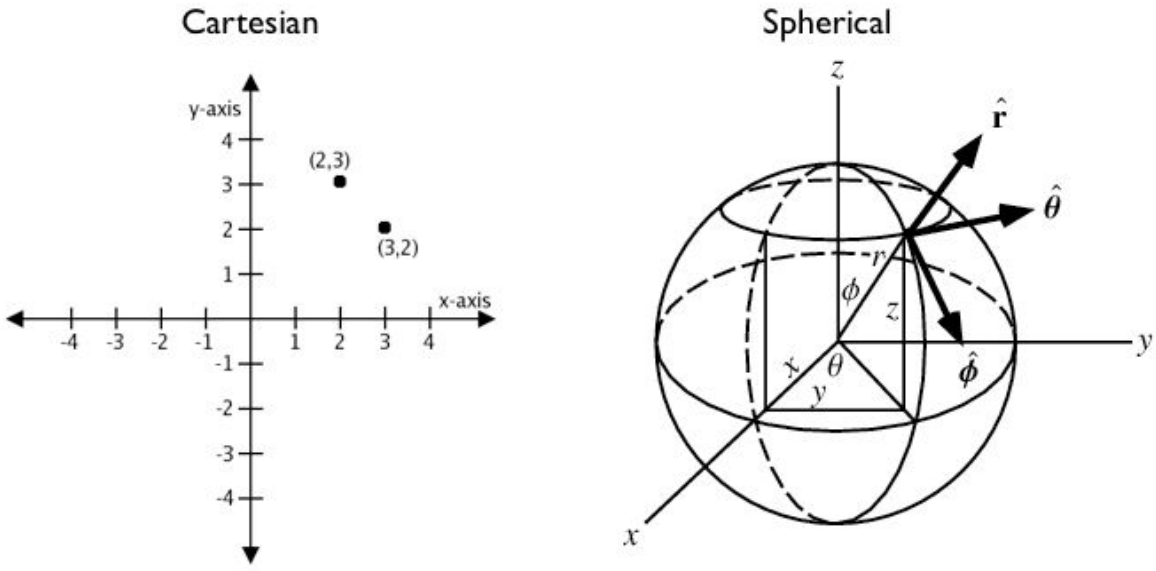
CHAPTER

EIGHTEEN

GEOGRAPHY

It is very common to have data in which the coordinate are "geographics" or "latitude/longitude".

Unlike coordinates in Mercator, UTM, or Stateplane, geographic coordinates are **not Cartesian coordinates**. Geographic coordinates do not represent a linear distance from an origin as plotted on a plane. Rather, these **spherical coordinates** describe angular coordinates on a globe. In spherical coordinates a point is specified by the angle of rotation from a reference meridian (longitude), and the angle from the equator (latitude).



You can treat geographic coordinates as approximate Cartesian coordinates and continue to do spatial calculations. However, measurements of distance, length and area will be **nonsensical**. Since spherical coordinates measure **angular** distance, the units are in "degrees." Further, the approximate results from indexes and true/false tests like intersects and contains can become terribly wrong. The distance between points get larger as problem areas like the poles or the international dateline are approached.

For example, here are the coordinates of Los Angeles and Paris.

- Los Angeles: `POINT(-118.4079 33.9434)`
- Paris: `POINT(2.3490 48.8533)`

The following calculates the distance between Los Angeles and Paris using the standard PostGIS Cartesian **`ST_Distance(geometry, geometry)`**. Note that the SRID of 4326 declares a geographic spatial reference system.

```
SELECT ST_Distance(
  'SRID=4326;POINT(-118.4079 33.9434)'::geometry, -- Los Angeles (LAX)
  'SRID=4326;POINT(2.5559 49.0083)'::geometry     -- Paris (CDG)
  );
```

```
121.898285970107
```

Aha! 122! But, what does that mean?

The units for spatial reference 4326 are degrees. So our answer is 122 degrees. But (again), what does that mean?

On a sphere, the size of one "degree square" is quite variable, becoming smaller as you move away from the equator. Think of the meridians (vertical lines) on the globe getting closer to each other as you go towards the poles. So, a distance of 122 degrees doesn't *mean* anything. It is a nonsense number.

In order to calculate a meaningful distance, we must treat geographic coordinates not as approximate Cartesian coordinates but rather as true spherical coordinates. We must measure the distances between points as true paths over a sphere – a portion of a great circle.

PostGIS provides this functionality through the `geography` type.

Note: Different spatial databases have different approaches for "handling geographics"

- Oracle attempts to paper over the differences by transparently doing geographic calculations when the SRID is geographic.
- SQL Server uses two spatial types, "STGeometry" for Cartesian data and "STGeography" for geographics.
- Informix Spatial is a pure Cartesian extension to Informix, while Informix Geodetic is a pure geographic extension.
- Similar to SQL Server, PostGIS uses two types, "geometry" and "geography".

Using the `geography` instead of `geometry` type, let's try again to measure the distance between Los Angeles and Paris.

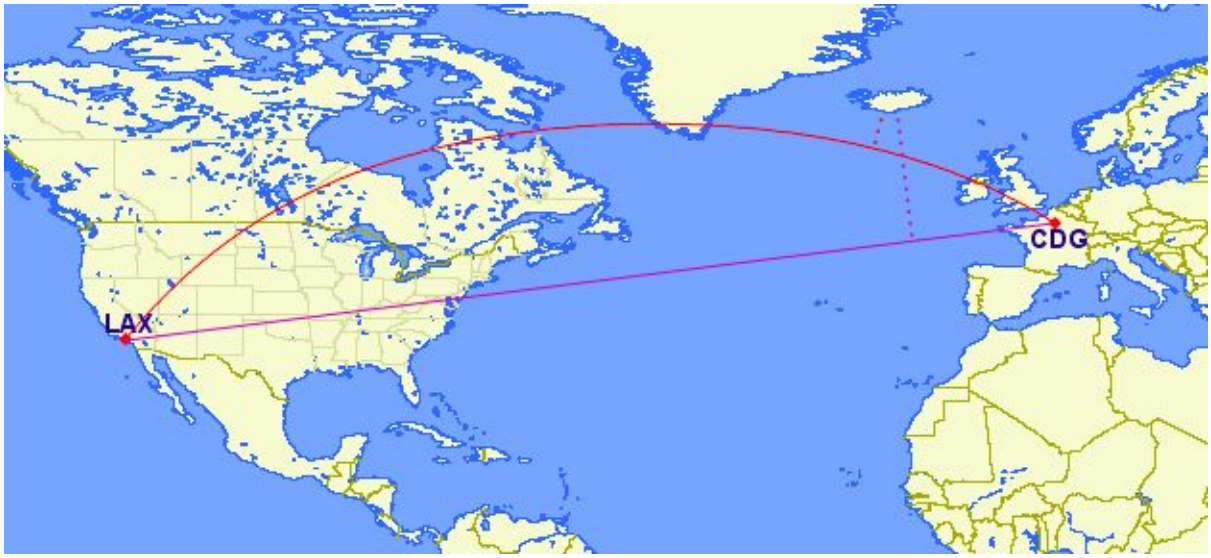
```
SELECT ST_Distance(
  'SRID=4326;POINT(-118.4079 33.9434)'::geography, -- Los Angeles (LAX)
  'SRID=4326;POINT(2.5559 49.0083)'::geography     -- Paris (CDG)
  );
```

```
9124665.27317673
```

A big number! All return values from `geography` calculations are in **meters**, so our answer is 9125km.

Older versions of PostGIS supported very basic calculations over the sphere using the **`ST_Distance_Spheroid(point, point, measurement)`** function. However, **`ST_Distance_Spheroid`** is substantially limited. The function only works on points and provides no support for indexing across the poles or international dateline.

The need to support non-point geometries becomes very clear when posing a question like "How close will a flight from Los Angeles to Paris come to Iceland?"



Working with geographic coordinates on a Cartesian plane (the purple line) yields a *very* wrong answer indeed! Using great circle routes (the red lines) gives the right answer. If we convert our LAX-CDG flight into a line string and calculate the distance to a point in Iceland using `geography` we'll get the right answer (recall) in meters.

```
SELECT ST_Distance(
  ST_GeographyFromText('LINESTRING(-118.4079 33.9434, 2.5559 49.0083)'), --
↪ LAX-CDG
  ST_GeographyFromText('POINT(-22.6056 63.9850)')                        --
↪ Iceland (KEF)
);
```

```
502454.906643729
```

So the closest approach to Iceland (as measured from its international airport) on the LAX-CDG route is a relatively small 502km.

The Cartesian approach to handling geographic coordinates breaks down entirely for features that cross the international dateline. The shortest great-circle route from Los Angeles to Tokyo crosses the Pacific Ocean. The shortest Cartesian route crosses the Atlantic and Indian Oceans.

```
SELECT ST_Distance(
  ST_GeometryFromText('Point(-118.4079 33.9434)'),  -- LAX
  ST_GeometryFromText('Point(139.733 35.567)'))     -- NRT (Tokyo/Narita)
    AS geometry_distance,
ST_Distance(
  ST_GeographyFromText('Point(-118.4079 33.9434)'), -- LAX
  ST_GeographyFromText('Point(139.733 35.567)'))    -- NRT (Tokyo/Narita)
    AS geography_distance;
```

```
 geometry_distance | geography_distance
-------------------+--------------------
  258.146005837336 |   8833954.76996256
```

18.1 Using Geography

In order to load geometry data into a geography table, the geometry first needs to be projected into EPSG:4326 (longitude/latitude), then it needs to be changed into geography. The **ST_Transform(geometry,srid)** function converts coordinates to geographics and the **Geography(geometry)** function or the ::geography suffix "casts" to geography.

```
CREATE TABLE nyc_subway_stations_geog AS
SELECT
  ST_Transform(geom,4326)::geography AS geog,
  name,
  routes
FROM nyc_subway_stations;
```

Building a spatial index on a geography table is exactly the same as for geometry:

```
CREATE INDEX nyc_subway_stations_geog_gix
ON nyc_subway_stations_geog USING GIST (geog);
```

The difference is under the covers: the geography index will correctly handle queries that cover the poles or the international date-line, while the geometry one will not.

Here's a query to find all the subway stations within 500 meters of the Empire State Building.

There are only a small number of native functions for the geography type:

- **ST_AsText(geography)** returns text
- **ST_GeographyFromText(text)** returns geography
- **ST_AsBinary(geography)** returns bytea
- **ST_GeogFromWKB(bytea)** returns geography
- **ST_AsSVG(geography)** returns text
- **ST_AsGML(geography)** returns text
- **ST_AsKML(geography)** returns text
- **ST_AsGeoJson(geography)** returns text
- **ST_Distance(geography, geography)** returns double

- **ST_DWithin(geography, geography, float8)** returns boolean
- **ST_Area(geography)** returns double
- **ST_Length(geography)** returns double
- **ST_Covers(geography, geography)** returns boolean
- **ST_CoveredBy(geography, geography)** returns boolean
- **ST_Intersects(geography, geography)** returns boolean
- **ST_Buffer(geography, float8)** returns geography[1]
- **ST_Intersection(geography, geography)** returns geography[1]

18.2 Creating a Geography Table

The SQL for creating a new table with a geography column is much like that for creating a geometry table. However, geography includes the ability to specify the object type directly at the time of table creation. For example:

```
CREATE TABLE airports (
    code VARCHAR(3),
    geog GEOGRAPHY(Point)
  );

INSERT INTO airports
  VALUES ('LAX', 'POINT(-118.4079 33.9434)');
INSERT INTO airports
  VALUES ('CDG', 'POINT(2.5559 49.0083)');
INSERT INTO airports
  VALUES ('KEF', 'POINT(-22.6056 63.9850)');
```

In the table definition, the GEOGRAPHY(Point) specifies our airport data type as points. The new geography fields don't get registered in the geometry_columns view. Instead, they are registered in a view called geography_columns.

```
SELECT * FROM geography_columns;
```

```
          f_table_name     | f_geography_column | srid |    type
---------------------------+--------------------+------+-----------
 nyc_subway_stations_geog | geog               |    0 | Geometry
 airports                 | geog               | 4326 | Point
```

Note: Some columns were omitted from the above output.

[1] The buffer and intersection functions are actually wrappers on top of a cast to geometry, and are not carried out natively in spherical coordinates. As a result, they may fail to return correct results for objects with very large extents that cannot be cleanly converted to a planar representation.

For example, the **ST_Buffer(geography,distance)** function transforms the geography object into a "best" projection, buffers it, and then transforms it back to geographics. If there is no "best" projection (the object is too large), the operation can fail or return a malformed buffer.

18.3 Casting to Geometry

While the basic functions for geography types can handle many use cases, there are times when you might need access to other functions only supported by the geometry type. Fortunately, you can convert objects back and forth from geography to geometry.

The PostgreSQL syntax convention for casting is to append `::typename` to the end of the value you wish to cast. So, `2::text` will convert a numeric two to a text string '2'. And `'POINT(0 0)'::geometry` will convert the text representation of point into a geometry point.

The **`ST_X(point)`** function only supports the geometry type. How can we read the X coordinate from our geographies?

```
SELECT code, ST_X(geog::geometry) AS longitude FROM airports;
```

```
 code | longitude
------+-----------
 LAX  | -118.4079
 CDG  |    2.5559
 KEF  |  -21.8628
```

By appending `::geometry` to our geography value, we convert the object to a geometry with an SRID of 4326. From there we can use as many geometry functions as strike our fancy. But, remember – now that our object is a geometry, the coordinates will be interpretted as Cartesian coordinates, not spherical ones.

18.4 Why (Not) Use Geography

Geographics are universally accepted coordinates – everyone understands what latitude/longitude mean, but very few people understand what UTM coordinates mean. Why not use geography all the time?

- First, as noted earlier, there are far fewer functions available (right now) that directly support the geography type. You may spend a lot of time working around geography type limitations.
- Second, the calculations on a sphere are computationally far more expensive than Cartesian calculations. For example, the Cartesian formula for distance (Pythagoras) involves one call to sqrt(). The spherical formula for distance (Haversine) involves two sqrt() calls, an arctan() call, four sin() calls and two cos() calls. Trigonometric functions are very costly, and spherical calculations involve a lot of them.

The conclusion?

If your data is geographically compact (contained within a state, county or city), **use the geometry type with a Cartesian projection** that makes sense with your data. See the http://epsg.io site and type in the name of your region for a selection of possible reference systems.

If you need to measure distance with a dataset that is geographically dispersed (covering much of the world), **use the geography type.** The application complexity you save by working in `geography` will offset any performance issues. And casting to `geometry` can offset most functionality limitations.

18.5 Function List

ST_Distance(geometry, geometry): For geometry type Returns the 2-dimensional Cartesian minimum distance (based on spatial ref) between two geometries in projected units. For geography type defaults to return spheroidal minimum distance between two geographies in meters.

ST_GeographyFromText(text): Returns a specified geography value from Well-Known Text representation or extended (WKT).

ST_Transform(geometry, srid): Returns a new geometry with its coordinates transformed to the SRID referenced by the integer parameter.

ST_X(point): Returns the X coordinate of the point, or NULL if not available. Input must be a point.

ST_Azimuth(geography_A, geography_B): Returns the direction from A to B in radians.

ST_DWithin(geography_A, geography_B, R): Returns true if A is within R meters of B.

CHAPTER

NINETEEN

GEOGRAPHY EXERCISES

Here's a reminder of all the functions we have seen so far. They should be useful for the exercises!

- **`Sum(number)`** adds up all the numbers in the result set
- **`ST_GeogFromText(text)`** returns a geography
- **`ST_Distance(geography, geography)`** returns the distance between geographies
- **`ST_Transform(geometry, srid)`** returns geometry, in the new projection
- **`ST_Length(geography)`** returns the length of the line
- **`ST_Intersects(geometry, geometry)`** returns true if the objects are not disjoint in planar space
- **`ST_Intersects(geography, geography)`** returns true if the objects are not disjoint in spheroidal space

Also remember the tables we have available:

- `nyc_streets`
  - name, type, geom
- `nyc_neighborhoods`
  - name, boroname, geom

19.1 Exercises

- **How far is New York from Seattle? What are the units of the answer?**

  Note: New York = `POINT(-74.0064 40.7142)` and Seattle = `POINT(-122.3331 47.6097)`

```
SELECT ST_Distance(
  'POINT(-74.0064 40.7142)'::geography,
  'POINT(-122.3331 47.6097)'::geography
  );
```

```
3875538.57141352
```

- **What is the total length of all streets in New York, calculated on the spheroid?**

```
SELECT Sum(
  ST_Length(Geography(
    ST_Transform(geom,4326)
  )))
FROM nyc_streets;
```

```
10421999.666
```

Note: The length calculated in the planar "UTM Zone 18" projection is 10418904.717, 0.02% different. UTM is good at preserving area and distance, within the zone boundaries.

- **Does 'POINT(1 2.0001)' intersect with 'POLYGON((0 0, 0 2, 2 2, 2 0, 0 0))' in geography? In geometry? Why the difference?**

```
SELECT ST_Intersects(
  'POINT(1 2.0001)'::geography,
  'POLYGON((0 0,0 2,2 2,2 0,0 0))'::geography
);

SELECT ST_Intersects(
  'POINT(1 2.0001)'::geometry,
  'POLYGON((0 0,0 2,2 2,2 0,0 0))'::geometry
);
```

```
true and false
```

Note: The upper edge of the square is a straight line in geometry, and passes **below** the point, so the square does not contain the point. The upper edge of the square is a great circle in geography, and passes **above** the point, so the square does contain the point.

CHAPTER

TWENTY

GEOMETRY CONSTRUCTING FUNCTIONS

All the functions we have seen so far work with geometries "as they are" and returns

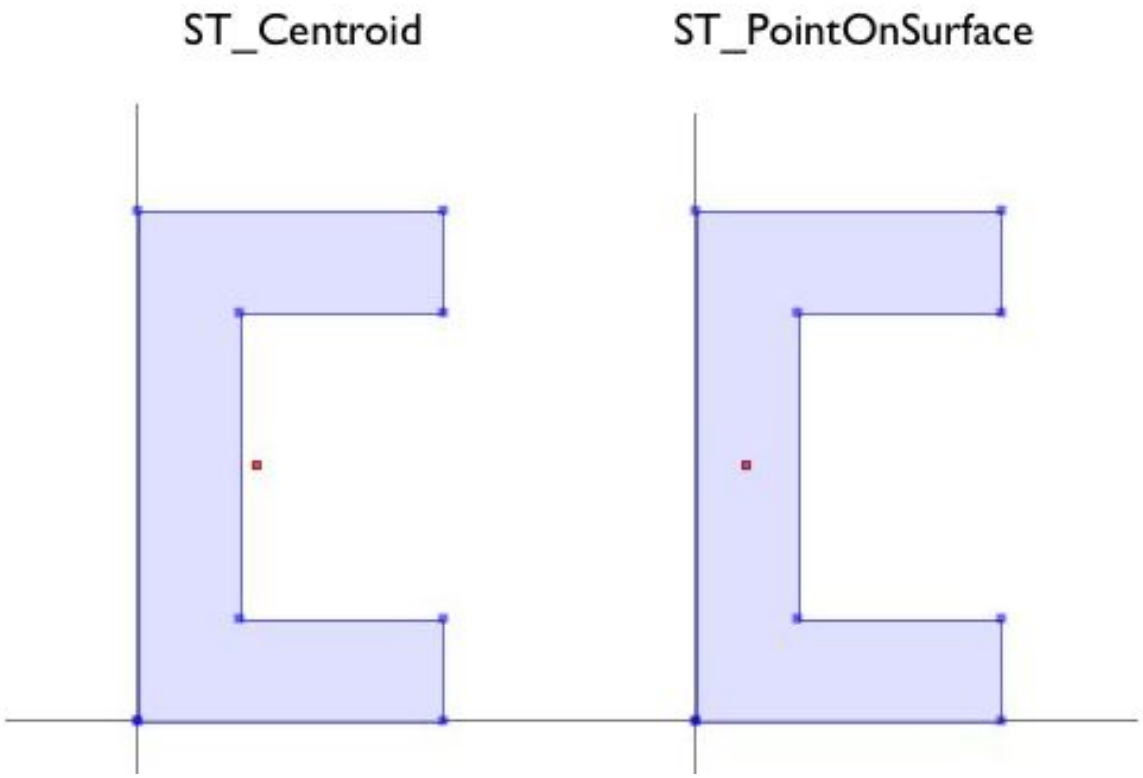
- analyses of the objects (**`ST_Length(geometry)`**, **`ST_Area(geometry)`**),
- serializations of the objects (**`ST_AsText(geometry)`**, **`ST_AsGML(geometry)`**),
- parts of the object (**`ST_RingN(geometry,n)`**) or
- true/false tests (**`ST_Contains(geometry,geometry)`**, **`ST_Intersects(geometry, geometry)`**).

"Geometry constructing functions" take geometries as inputs and output new shapes.

20.1 ST_Centroid / ST_PointOnSurface

A common need when composing a spatial query is to replace a polygon feature with a point representation of the feature. This is useful for spatial joins (as discussed in *Polygon/Polygon Joins*) because using **`ST_Intersects(geometry,geometry)`** on two polygon layers often results in double-counting: a polygon on a boundary will intersect an object on both sides; replacing it with a point forces it to be on one side or the other, not both.

- **`ST_Centroid(geometry)`** returns a point that is approximately on the center of mass of the input argument. This simple calculation is very fast, but sometimes not desirable, because the returned point is not necessarily in the feature itself. If the input feature has a convexity (imagine the letter 'C') the returned centroid might not be in the interior of the feature.
- **`ST_PointOnSurface(geometry)`** returns a point that is guaranteed to be inside the input argument. This makes it more useful for computing "proxy points" for spatial joins.

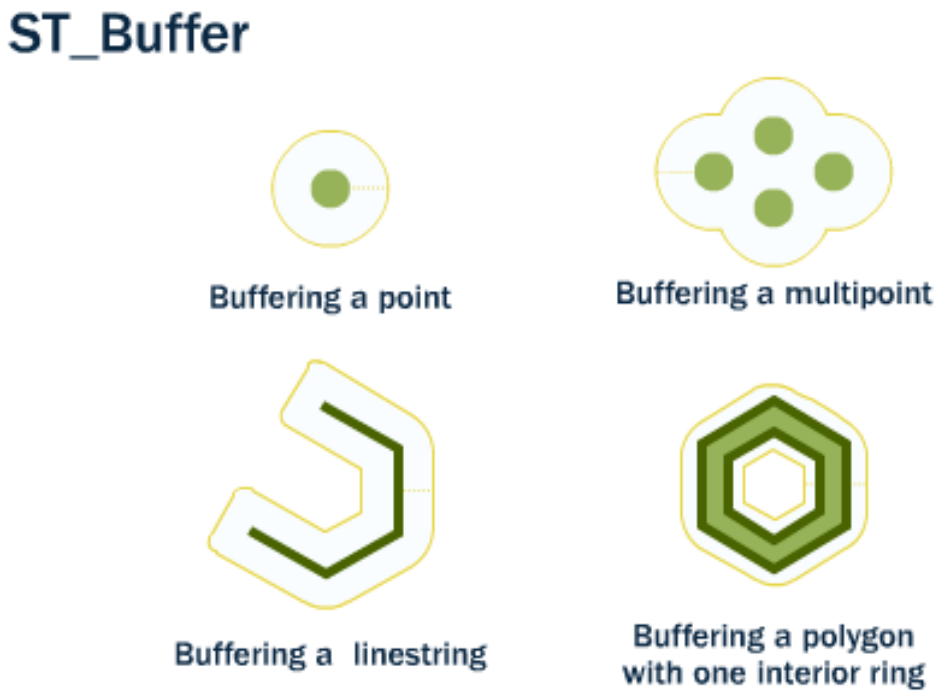



```
-- Compare the location of centroid and point-on-surface for a concave␣
↪geometry

SELECT ST_Intersects(geom, ST_Centroid(geom)) AS centroid_inside,
       ST_Intersects(geom, ST_PointOnSurface(geom)) AS pos_inside
FROM (VALUES
    ('POLYGON ((30 0, 30 10, 10 10, 10 40, 30 40, 30 50, 0 50, 0 0, 0 0,␣
↪30 0))'::geometry)
  ) AS t(geom);
```

```
 centroid_inside | pos_inside
-----------------+------------
 f               | t
```

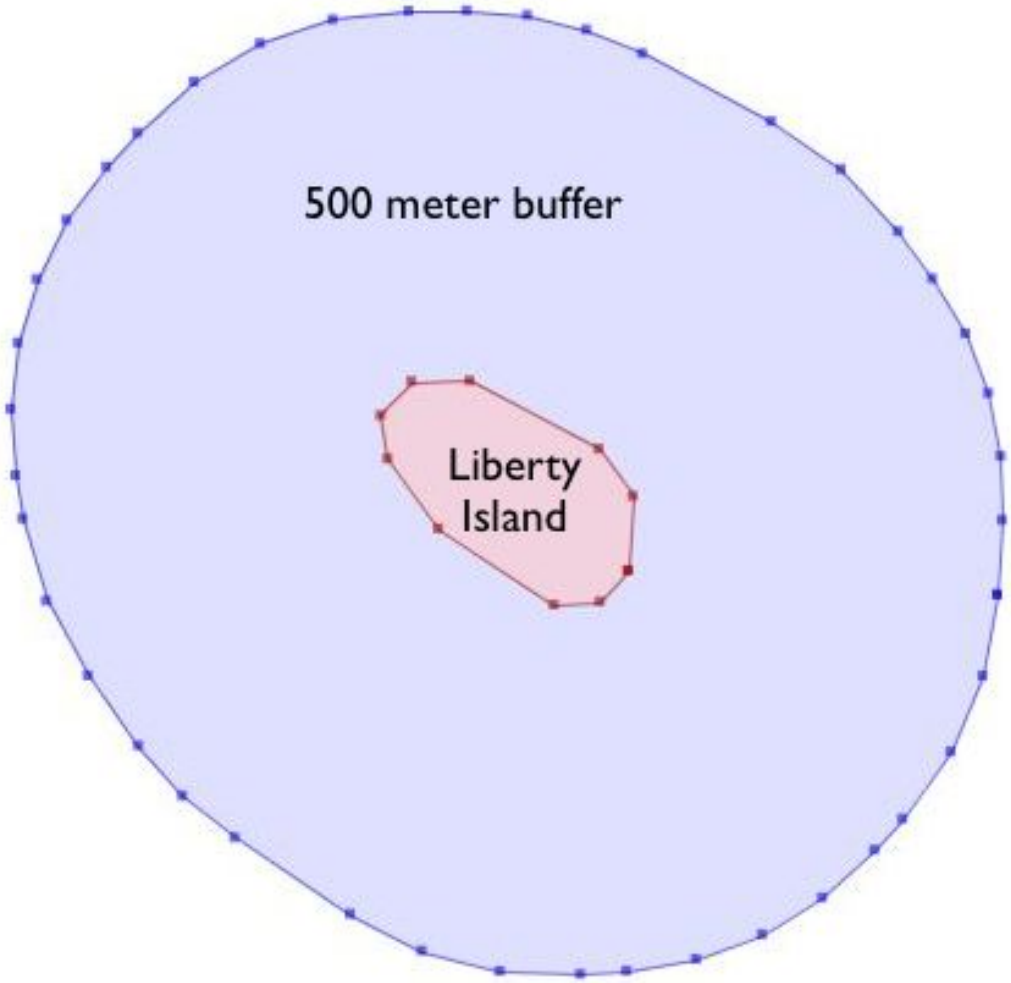
20.2 ST_Buffer

The buffering operation is common in GIS workflows, and is also available in PostGIS. **`ST_Buffer(geometry,distance)`** takes in a buffer distance and geometry type and outputs a polygon with a boundary the buffer distance away from the input geometry.

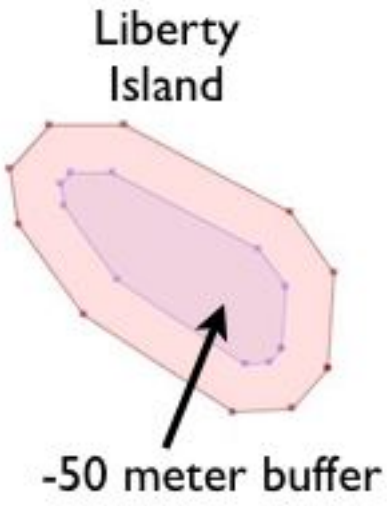


For example, if the US Park Service wanted to enforce a marine traffic zone around Liberty Island, they might build a 500 meter buffer polygon around the island. Liberty Island is a single census block in our `nyc_census_blocks` table, so we can easily extract and buffer it.

```
-- Make a new table with a Liberty Island 500m buffer zone
CREATE TABLE liberty_island_zone AS
SELECT ST_Buffer(geom,500)::geometry(Polygon,26918) AS geom
FROM nyc_census_blocks
WHERE blkid = '360610001001001';
```



The **ST_Buffer** function also accepts negative distances and builds inscribed polygons within polygonal inputs. For lines and points you will just get an empty return.

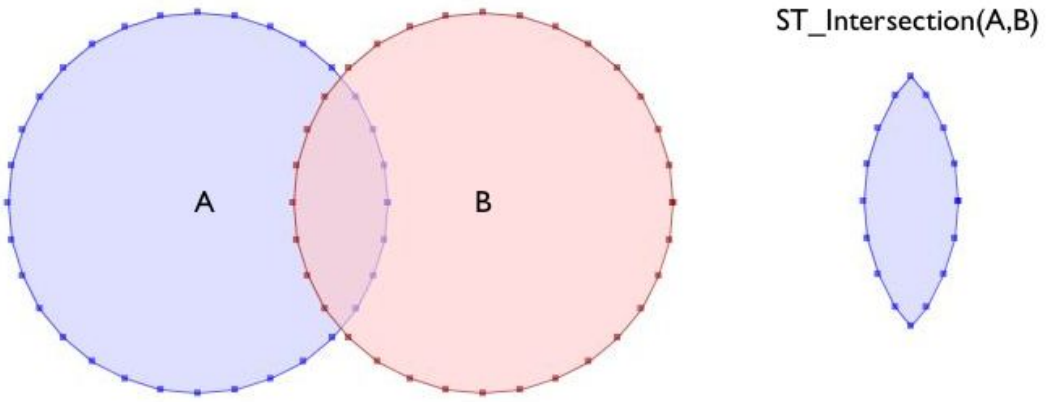


20.3 ST_Intersection

Another classic GIS operation – the "overlay" – creates a new coverage by calculating the intersection of two superimposed polygons. The resultant has the property that any polygon in either of the parents can be built by merging polygons in the resultant.

The **ST_Intersection(geometry A, geometry B)** function returns the spatial area (or line, or point) that both arguments have in common. If the arguments are disjoint, the function returns an empty geometry.

```
-- What is the area these two circles have in common?
-- Using ST_Buffer to make the circles!

SELECT ST_AsText(ST_Intersection(
  ST_Buffer('POINT(0 0)', 2),
  ST_Buffer('POINT(3 0)', 2)
));
```

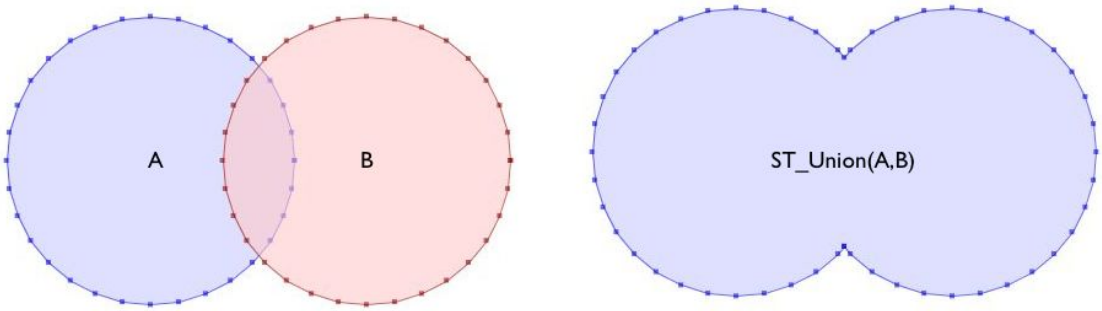


20.4 ST_Union

In the previous example we intersected geometries, creating a new geometry that had lines from both the inputs. The **ST_Union** function does the reverse; it takes inputs and removes common lines. There are two forms of the **ST_Union** function:

- **ST_Union(geometry, geometry)**: A two-argument version that takes in two geometries and returns the merged union. For example, our two-circle example from the previous section looks like this when you replace the intersection with a union.

```
-- What is the total area these two circles cover?
-- Using ST_Buffer to make the circles!

SELECT ST_AsText(ST_Union(
  ST_Buffer('POINT(0 0)', 2),
  ST_Buffer('POINT(3 0)', 2)
));
```



- **`ST_Union([geometry])`**: An aggregate version that takes in a set of geometries and returns the merged geometry for the entire group. The aggregate ST_Union can be used with the `GROUP BY` SQL statement to create carefully merged subsets of basic geometries. It is very powerful.

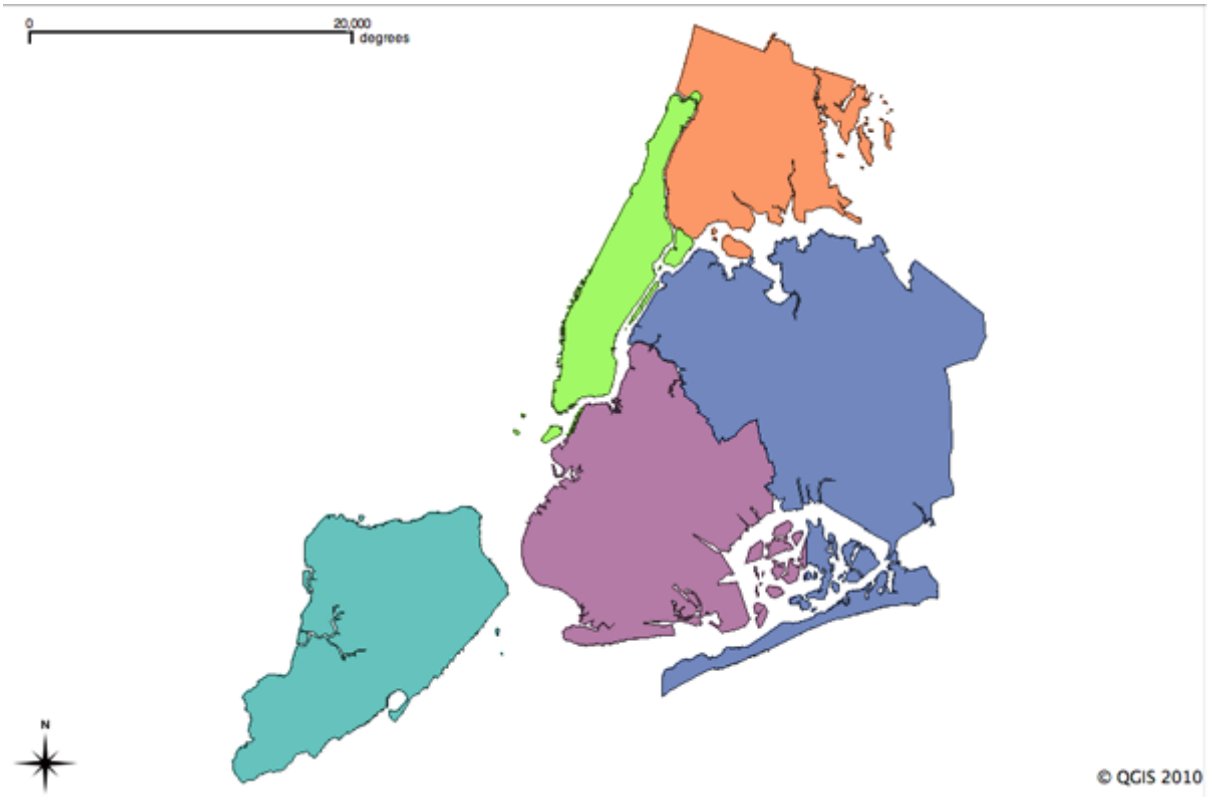
As an example of **`ST_Union`** aggregation, consider our `nyc_census_blocks` table. Census geography is carefully constructed so that larger geographies can be built up from smaller ones. So, we can create a census tracts map by merging the blocks that form each tract (as we do later in *Creating a Census Tracts Table*). Or, we can create a county map by merging blocks that fall within each county.

To carry out the merge, note that the unique key `blkid` actually embeds information about the higher level geographies. Here are the parts of the key for Liberty Island we used earlier:

```
360610001001001 = 36 061 000100 1 001

36     = State of New York
061    = New York County (Manhattan)
000100 = Census Tract
1      = Census Block Group
001    = Census Block
```

So, we can create a county map by merging all geometries that share the same first 5 digits of their `blkid`. Be patient; this is computationally expensive and can take a minute or two.

```
-- Create a nyc_census_counties table by merging census blocks
CREATE TABLE nyc_census_counties AS
SELECT
  ST_Union(geom)::Geometry(MultiPolygon,26918) AS geom,
  SubStr(blkid,1,5) AS countyid
FROM nyc_census_blocks
GROUP BY countyid;
```



An area test can confirm that our union operation did not lose any geometry. First, we calculate the area of each individual census block, and sum those areas grouping by census county id.

```
SELECT SubStr(blkid,1,5) AS countyid, Sum(ST_Area(geom)) AS area
FROM nyc_census_blocks
GROUP BY countyid
ORDER BY countyid;
```

```
 countyid |       area
----------+------------------
 36005    | 110196022.906506
 36047    | 181927497.678368
 36061    | 59091860.6261323
 36081    | 283194473.613692
 36085    | 150758328.111199
```

Then we calculate the area of each of our new county polygons from the county table:

```
SELECT countyid, ST_Area(geom) AS area
FROM nyc_census_counties
ORDER BY countyid;
```

```
 countyid |       area
----------+------------------
 36005    | 110196022.906507
 36047    | 181927497.678367
 36061    | 59091860.6261324
 36081    | 283194473.593646
 36085    | 150758328.111199
```

The same answer! We have successfully built an NYC county table from our census blocks data.

20.5 Function List

ST_Centroid(geometry): Returns a point geometry that represents the center of mass of the input geometry.

ST_PointOnSurface(geometry): Returns a point geometry that is guaranteed to be in the interior of the input geometry.

ST_Buffer(geometry, distance): For geometry: Returns a geometry that represents all points whose distance from this Geometry is less than or equal to distance. Calculations are in the Spatial Reference System of this Geometry. For geography: Uses a planar transform wrapper.

ST_Intersection(geometry A, geometry B): Returns a geometry that represents the shared portion of geomA and geomB. The geography implementation does a transform to geometry to do the intersection and then transform back to WGS84.

ST_Union(): Returns a geometry that represents the point set union of the Geometries.

ST_AsText(text): Returns the Well-Known Text (WKT) representation of the geometry/geography without SRID metadata.

substring(string [from int] [for int]): PostgreSQL string function to extract substring matching SQL regular expression.

sum(expression): PostgreSQL aggregate function that returns the sum of records in a set of records.

CHAPTER

TWENTYONE

GEOMETRY CONSTRUCTING EXERCISES

Here's a reminder of some of the functions we have seen. Hint: they should be useful for the exercises!

- **`sum(expression)`** aggregate to return a sum for a set of records
- **`ST_Area(geometry)`** returns the area of the geometry
- **`ST_Centroid(geometry)`** returns the `geometry` centroid
- **`ST_Transform(geometry, srid)`** converts `geometries` into different spatial reference systems
- **`ST_Buffer(geometry, radius)`** returns an expanded `geometry` shape
- **`ST_Contains(geometry1, geometry2)`** returns truw if geometry1 contains geometry2
- **`ST_Union(geometry[])`** returns the aggregate union of all geometries in the group
- **`ST_GeometryType(geometry)`** returns the type of the geometry
- **`ST_NumGeometries(geometry)`** returns the number of geometries in a collection or 1 for simple geometries
- **`ST_Intersection(geometry, geometry)`** returns the area that the two input geometries share in common

Remember the tables we have available:

- `nyc_census_blocks`
  - name, popn_total, boroname, geom
- `nyc_streets`
  - name, type, geom
- `nyc_subway_stations`
  - name, geom
- `nyc_neighborhoods`
  - name, boroname, geom

21.1 Exercises

- **How many census blocks don't contain their own centroid?**

```
SELECT Count(*)
  FROM nyc_census_blocks
  WHERE NOT
    ST_Contains(
      geom,
      ST_Centroid(geom)
    );
```

```
481
```

- **Union all the census blocks into a single output. What kind of geometry is it? How many parts does it have?**

```
CREATE TABLE nyc_census_blocks_merge AS
  SELECT ST_Union(geom) AS geom
  FROM nyc_census_blocks;

SELECT ST_GeometryType(geom)
  FROM nyc_census_blocks_merge;
```

```
ST_MultiPolygon
```

```
SELECT ST_NumGeometries(geom)
  FROM nyc_census_blocks_merge;
```

```
63
```

- **What is the area of a one unit buffer around the origin? How different is it from what you would expect? Why?**

```
SELECT ST_Area(ST_Buffer('POINT(0 0)', 1));
```

```
3.121445152258052
```

Note: A unit circle (circle with radius of one) should have an area of pi, 3.1415926... The difference is due to the linear stroking of the edges of the buffer. The buffer has a finite number of edges. Increasing the number of edges in the buffer will get the value closer to pi, but it will always be smaller due to the linearization.

- **The Brooklyn neighborhoods of 'Park Slope' and 'Carroll Gardens' are going to war! Construct a polygon delineating a 100 meter wide DMZ on the border between the neighborhoods. What is the area of the DMZ?**

```
CREATE TABLE brooklyn_dmz AS
  SELECT
    ST_Intersection(
      ST_Buffer(ps.geom, 50),
```

(continues on next page)

(continued from previous page)

```
      ST_Buffer(cg.geom, 50))
    AS geom
  FROM
    nyc_neighborhoods ps,
    nyc_neighborhoods cg
  WHERE ps.name = 'Park Slope'
  AND cg.name = 'Carroll Gardens';

SELECT ST_Area(geom) FROM brooklyn_dmz;
```

Note: It is easy to buffer both the neighborhoods of interest, but to get the intersection requires a self-join of the table, creating one relation (`ps`) with just the "Park Slope" record and another (`cg`) with just the "Carroll Gardens" record. Note that the area of the intersection is in square meters because we are still working in UTM 18 (EPSG:26918).

```
180990.964207547
```

CHAPTER

TWENTYTWO

MORE SPATIAL JOINS

In the last section we saw the **`ST_Centroid(geometry)`** and **`ST_Union([geometry])`** functions, and some simple examples. In this section we will do some more elaborate things with them.

22.1 Creating a Census Tracts Table

In the workshop `\data\` directory, is a file that includes attribute data, but no geometry, `nyc_census_sociodata.sql`. The table includes interesting socioeconomic data about New York: commute times, incomes, and education attainment. There is just one problem. The data are summarized by "census tract" and we have no census tract spatial data!

In this section we will

- Load the `nyc_census_sociodata.sql` table
- Create a spatial table for census tracts
- Join the attribute data to the spatial data
- Carry out some analysis using our new data

22.1.1 Loading nyc_census_sociodata.sql

1. Open the SQL query window in PgAdmin
2. Select **File->Open** from the menu and browse to the `nyc_census_sociodata.sql` file
3. Press the "Run Query" button
4. If you press the "Refresh" button in PgAdmin, the list of tables should now include at `nyc_census_sociodata` table

22.1.2 Creating a Census Tracts Table

As we saw in the previous section, we can build up higher level geometries from the census block by summarizing on substrings of the `blkid` key. In order to get census tracts, we need to summarize grouping on the first 11 characters of the `blkid`.

```
360610001001001 = 36 061 000100 1 001

36      = State of New York
061     = New York County (Manhattan)
```

(continues on next page)

(continued from previous page)

```
000100 = Census Tract
1      = Census Block Group
001    = Census Block
```

Create the new table using the **ST_Union** aggregate:

```
-- Make the tracts table
CREATE TABLE nyc_census_tract_geoms AS
SELECT
  ST_Union(geom) AS geom,
  SubStr(blkid,1,11) AS tractid
FROM nyc_census_blocks
GROUP BY tractid;

-- Index the tractid
CREATE INDEX nyc_census_tract_geoms_tractid_idx
  ON nyc_census_tract_geoms (tractid);
```

22.1.3 Join the Attributes to the Spatial Data

Join the table of tract geometries to the table of tract attributes with a standard attribute join

```
-- Make the tracts table
CREATE TABLE nyc_census_tracts AS
SELECT
  g.geom,
  a.*
FROM nyc_census_tract_geoms g
JOIN nyc_census_sociodata a
ON g.tractid = a.tractid;

-- Index the geometries
CREATE INDEX nyc_census_tract_gidx
  ON nyc_census_tracts USING GIST (geom);
```

22.1.4 Answer an Interesting Question

Answer an interesting question! "List top 10 New York neighborhoods ordered by the proportion of people who have graduate degrees."

```
SELECT
  100.0 * Sum(t.edu_graduate_dipl) / Sum(t.edu_total) AS graduate_pct,
  n.name, n.boroname
FROM nyc_neighborhoods n
JOIN nyc_census_tracts t
ON ST_Intersects(n.geom, t.geom)
WHERE t.edu_total > 0
GROUP BY n.name, n.boroname
ORDER BY graduate_pct DESC
LIMIT 10;
```

We sum up the statistics we are interested, then divide them together at the end. In order to avoid divide-by-zero errors, we don't bother bringing in tracts that have a population count of zero.

```
 graduate_pct |       name         | boroname
--------------+--------------------+-----------
         47.6 | Carnegie Hill      | Manhattan
         42.2 | Upper West Side    | Manhattan
         41.1 | Battery Park       | Manhattan
         39.6 | Flatbush           | Brooklyn
         39.3 | Tribeca            | Manhattan
         39.2 | North Sutton Area  | Manhattan
         38.7 | Greenwich Village  | Manhattan
         38.6 | Upper East Side    | Manhattan
         37.9 | Murray Hill        | Manhattan
         37.4 | Central Park       | Manhattan
```

Note: New York geographers will be wondering at the presence of "Flatbush" in this list of over-educated neighborhoods. The answer is discussed in the next section.

22.2 Polygon/Polygon Joins

In our interesting query (in *Answer an Interesting Question*) we used the **`ST_Intersects(geometry_a, geometry_b)`** function to determine which census tract polygons to include in each neighborhood summary. Which leads to the question: what if a tract falls on the border between two neighborhoods? It will intersect both, and so will be included in the summary statistics for **both**.

To avoid this kind of double counting there are two methods:

- The simple method is to ensure that each tract only falls in **one** summary area (using **`ST_Centroid(geometry)`**)
- The complex method is to divide crossing tracts at the borders (using **`ST_Intersection(geometry,geometry)`**)

Here is an example of using the simple method to avoid double counting in our graduate education query:

```
SELECT
  100.0 * Sum(t.edu_graduate_dipl) / Sum(t.edu_total) AS graduate_pct,
  n.name, n.boroname
FROM nyc_neighborhoods n
JOIN nyc_census_tracts t
ON ST_Contains(n.geom, ST_Centroid(t.geom))
WHERE t.edu_total > 0
GROUP BY n.name, n.boroname
ORDER BY graduate_pct DESC
LIMIT 10;
```

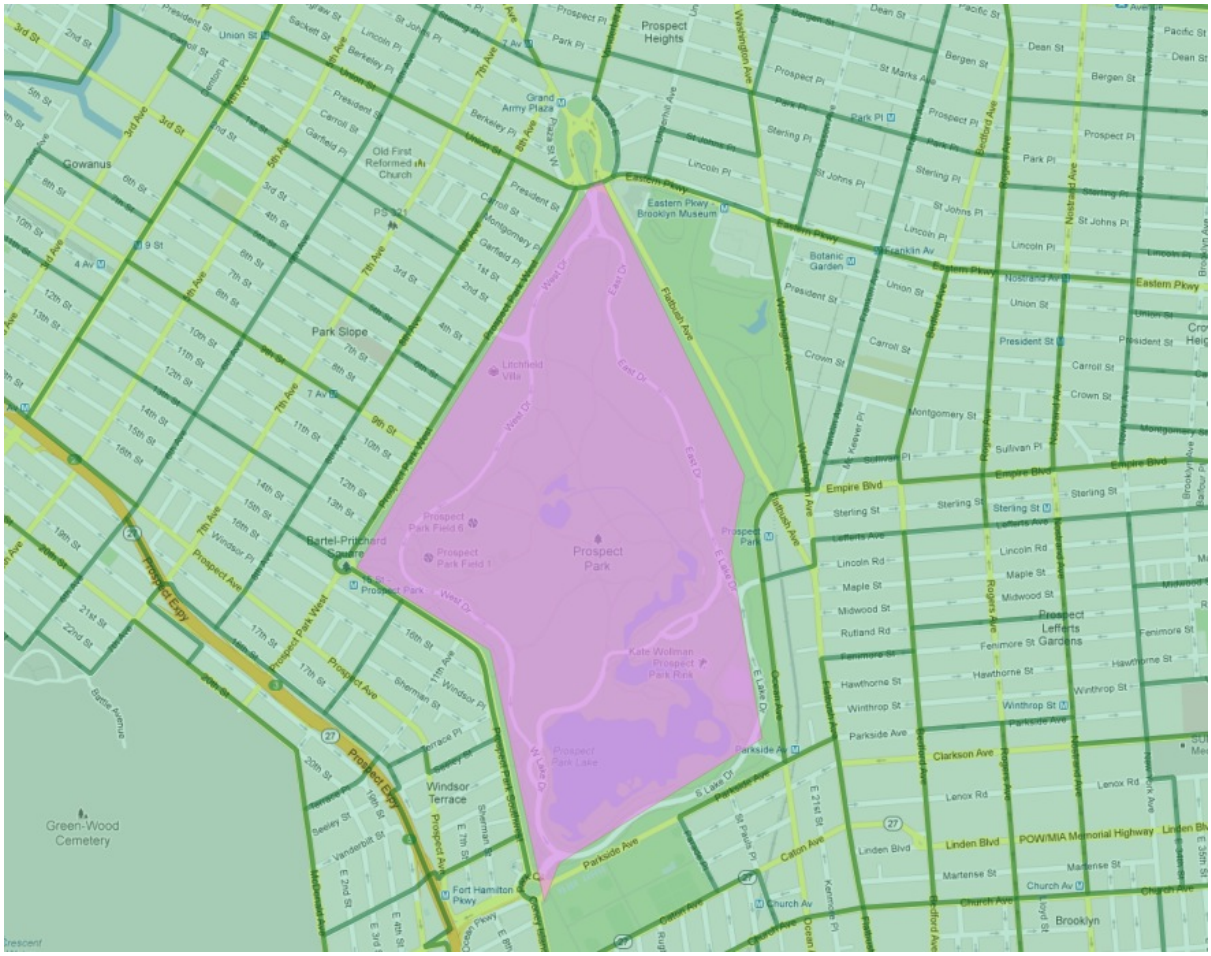
Note that the query takes longer to run now, because the **`ST_Centroid`** function has to be run on every census tract.

```
 graduate_pct |        name          | boroname
--------------+----------------------+-----------
         48.0 | Carnegie Hill        | Manhattan
         44.2 | Morningside Heights  | Manhattan
         42.1 | Greenwich Village    | Manhattan
         42.0 | Upper West Side      | Manhattan
         41.4 | Tribeca              | Manhattan
         40.7 | Battery Park         | Manhattan
         39.5 | Upper East Side      | Manhattan
         39.3 | North Sutton Area    | Manhattan
         37.4 | Cobble Hill          | Brooklyn
         37.4 | Murray Hill          | Manhattan
```

Avoiding double counting changes the results!

22.2.1 What about Flatbush?

In particular, the Flatbush neighborhood has dropped off the list. The reason why can be seen by looking more closely at the map of the Flatbush neighborhood in our table.



As defined by our data source, Flatbush is not really a neighborhood in the conventional sense, since it just covers the area of Prospect Park. The census tract for that area records, naturally, zero residents. However, the neighborhood boundary does scrape one of the expensive census tracts bordering the north side of the park (in the gentrified Park Slope neighborhood). When using polygon/polygon tests, this single tract was added to the otherwise empty Flatbush, resulting in the very high score for that query.

22.3 Large Radius Distance Joins

A query that is fun to ask is "How do the commute times of people near (within 500 meters) subway stations differ from those of people far away from subway stations?"

However, the question runs into some problems of double counting: many people will be within 500 meters of multiple subway stations. Compare the population of New York:

```
SELECT Sum(popn_total)
FROM nyc_census_blocks;
```

```
8175032
```

With the population of the people in New York within 500 meters of a subway station:

```
SELECT Sum(popn_total)
FROM nyc_census_blocks census
```

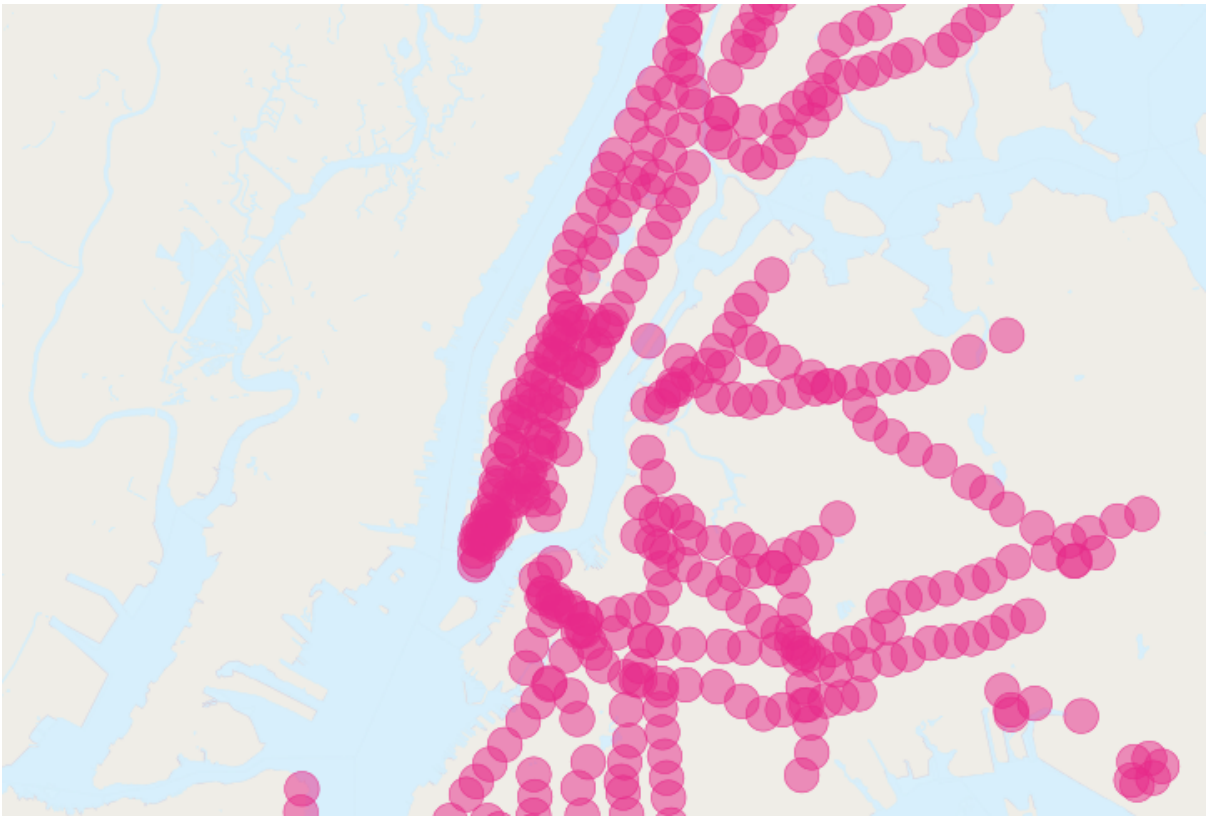
(continues on next page)

(continued from previous page)

```
JOIN nyc_subway_stations subway
ON ST_DWithin(census.geom, subway.geom, 500);
```

```
10855873
```

There's more people close to the subway than there are people! Clearly, our simple SQL is making a big double-counting error. You can see the problem looking at the picture of the buffered subways.



The solution is to ensure that we have only distinct census blocks before passing them into the summarization portion of the query. We can do that by breaking our query up into a subquery that finds the distinct blocks, wrapped in a summarization query that returns our answer:

```
WITH distinct_blocks AS (
  SELECT DISTINCT ON (blkid) popn_total
  FROM nyc_census_blocks census
  JOIN nyc_subway_stations subway
  ON ST_DWithin(census.geom, subway.geom, 500)
)
SELECT Sum(popn_total)
FROM distinct_blocks;
```

```
5005743
```

That's better! So a bit over half the population of New York is within 500m (about a 5-7 minute walk) of the subway.

CHAPTER

TWENTYTHREE

VALIDITY

In 90% of the cases the answer to the question, "why is my query giving me a 'TopologyException' error" is "one or more of the inputs are invalid". Which begs the question: what does it mean to be invalid, and why should we care?

23.1 What is Validity

Validity is most important for polygons, which define bounded areas and require a good deal of structure. Lines are very simple and cannot be invalid, nor can points.

Some of the rules of polygon validity feel obvious, and others feel arbitrary (and in fact, are arbitrary).

- Polygon rings must close.
- Rings that define holes should be inside rings that define exterior boundaries.
- Rings may not self-intersect (they may neither touch nor cross themselves).
- Rings may not touch other rings, except at a point.
- Elements of multi-polygons may not touch each other.

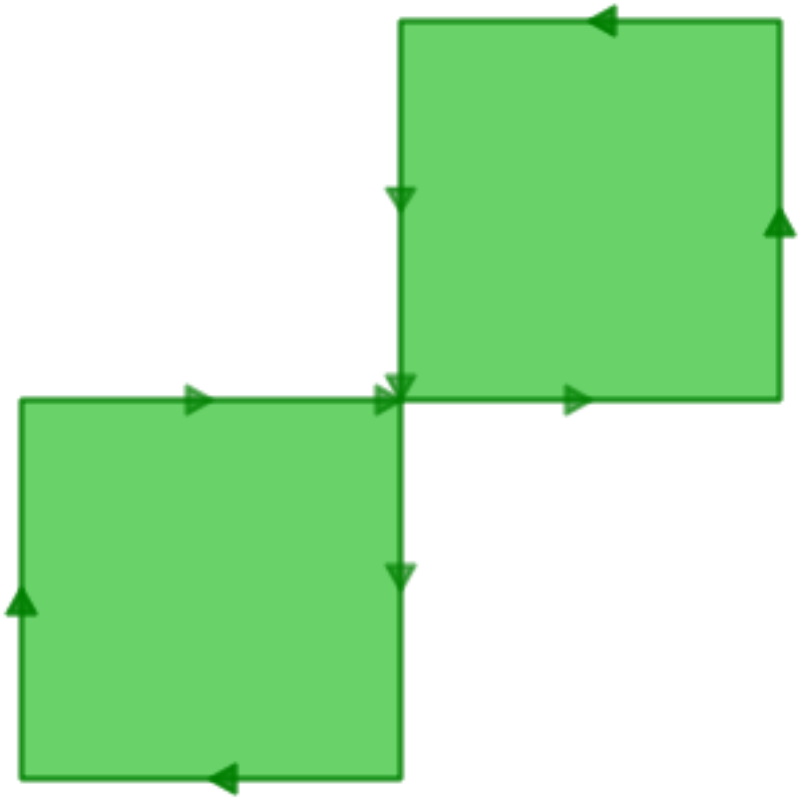
The last three rules are in the arbitrary category. There are other ways to define polygons that are equally self-consistent but the rules above are the ones used by the *OGC SFSQL* standard that PostGIS conforms to.

The reason the rules are important is because algorithms for geometry calculations depend on consistent structure in the inputs. It is possible to build algorithms that have no structural assumptions, but those routines tend to be very slow, because the first step in any structure-free routine is to *analyze the inputs and build structure into them*.

Here's an example of why structure matters. This polygon is invalid:

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

You can see the invalidity a little more clearly in this diagram:



The outer ring is actually a figure-eight, with a self-intersection in the middle. Note that the graphic routines successfully render the polygon fill, so that visually it is appears to be an "area": two one-unit squares, so a total area of two units of area.

Let's see what the database thinks the area of our polygon is:

```
SELECT ST_Area(ST_GeometryFromText(
         'POLYGON((0 0, 0 1, 1 1, 2 1, 2 2, 1 2, 1 1, 1 0, 0 0))'
       ));
```

```
 st_area
---------
       0
```

What's going on here? The algorithm that calculates area assumes that rings do not self-intersect. A well-behaved ring will always have the area that is bounded (the interior) on one side of the bounding line (it doesn't matter which side, just that it is on *one* side). However, in our (poorly behaved) figure-eight, the bounded area is to the right of the line for one lobe and to the left for the other. This causes the areas calculated for each lobe to cancel out (one comes out as 1, the other as -1) hence the "zero area" result.

23.2 Detecting Validity

In the previous example we had one polygon that we **knew** was invalid. How do we detect invalidity in a table with millions of geometries? With the **ST_IsValid(geometry)** function. Used against our figure-eight, we get a quick answer:

```
SELECT ST_IsValid(ST_GeometryFromText(
          'POLYGON((0 0, 0 1, 1 1, 2 1, 2 2, 1 2, 1 1, 1 0, 0 0))'
       ));
```

```
f
```

Now we know that the feature is invalid, but we don't know why. We can use the **ST_IsValidReason(geometry)** function to find out the source of the invalidity:

```
SELECT ST_IsValidReason(ST_GeometryFromText(
          'POLYGON((0 0, 0 1, 1 1, 2 1, 2 2, 1 2, 1 1, 1 0, 0 0))'
       ));
```

```
Self-intersection[1 1]
```

Note that in addition to the reason (self-intersection) the location of the invalidity (coordinate (1 1)) is also returned.

We can use the **ST_IsValid(geometry)** function to test our tables too:

```
-- Find all the invalid polygons and what their problem is
SELECT name, boroname, ST_IsValidReason(geom)
FROM nyc_neighborhoods
WHERE NOT ST_IsValid(geom);
```

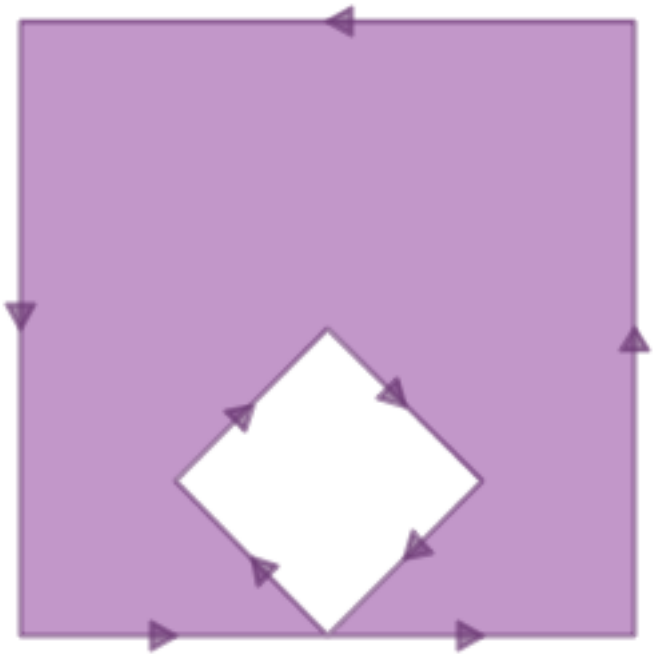
```
          name            |   boroname    |          st_isvalidreason
--------------------------+---------------+-----------------------------------
↪--------
 Howard Beach             | Queens        | Self-intersection[597264.08␣
↪4499924.54]
 Corona                   | Queens        | Self-intersection[595483.05␣
↪4513817.95]
 Steinway                 | Queens        | Self-intersection[593545.57␣
↪4514735.20]
 Red Hook                 | Brooklyn      | Self-intersection[584306.82␣
↪4502360.51]
```

23.3 Repairing Invalidity

Repairing invalidity involves stripping a polygon down to its simplest structures (rings), ensuring the rings follow the rules of validity, then building up new polygons that follow the rules of ring enclosure. Frequently the results are intuitive, but in the case of extremely ill-behaved inputs, the valid outputs may not conform to your intuition of how they should look. Recent versions of PostGIS include different algorithms for geometry repair: read the manual page carefully and choose the one you like best.

For example, here's a classic invalidity – the "banana polygon" – a single ring that encloses an area but bends around to touch itself, leaving a "hole" which is not actually a hole.

```
POLYGON((0 0, 2 0, 1 1, 2 2, 3 1, 2 0, 4 0, 4 4, 0 4, 0 0))
```



Running ST_MakeValid on the polygon returns a valid *OGC* polygon, consisting of an outer and inner ring that touch at one point.

```
SELECT ST_AsText(
         ST_MakeValid(
           ST_GeometryFromText('POLYGON((0 0, 2 0, 1 1, 2 2, 3 1, 2 0, 4 0,
↪ 4 4, 0 4, 0 0))')
         )
       );
```

```
POLYGON((0 0,0 4,4 4,4 0,2 0,0 0),(2 0,3 1,2 2,1 1,2 0))
```

Note: The "banana polygon" (or "inverted shell") is a case where the *OGC* topology model for valid geometry and the model used internally by ESRI differ. The ESRI model considers rings that touch to be invalid, and prefers the banana form for this kind of shape. The OGC model is the reverse. Neither is "correct", they are just different ways to model the same situation.

23.4 Bulk Validity Repair

Here's an example of SQL to flag invalid geometries for review while adding a repaired version to the table.

```
-- Column for old invalid form
ALTER TABLE nyc_neighborhoods
  ADD COLUMN geom_invalid geometry
  DEFAULT NULL;

-- Fix invalid and save the original
UPDATE nyc_neighborhoods
  SET geom = ST_MakeValid(geom),
      geom_invalid = geom
  WHERE NOT ST_IsValid(geom);

-- Review the invalid cases
SELECT geom, ST_IsValidReason(geom_invalid)
  FROM nyc_neighborhoods
  WHERE geom_invalid IS NOT NULL;
```

A good tool for visually repairing invalid geometry is OpenJump (http://openjump.org) which includes a validation routine under **Tools->QA->Validate Selected Layers**.

23.5 Function List

ST_IsValid(geometry A): Returns a boolean indiciting whether the geometery is valid.

ST_IsValidReason(geometry A): Returns a text string with the reason for the invalidity and a coordinate of invalidity.

ST_MakeValid(geometry A): Returns a geometry re-constructed to obey the validity rules.

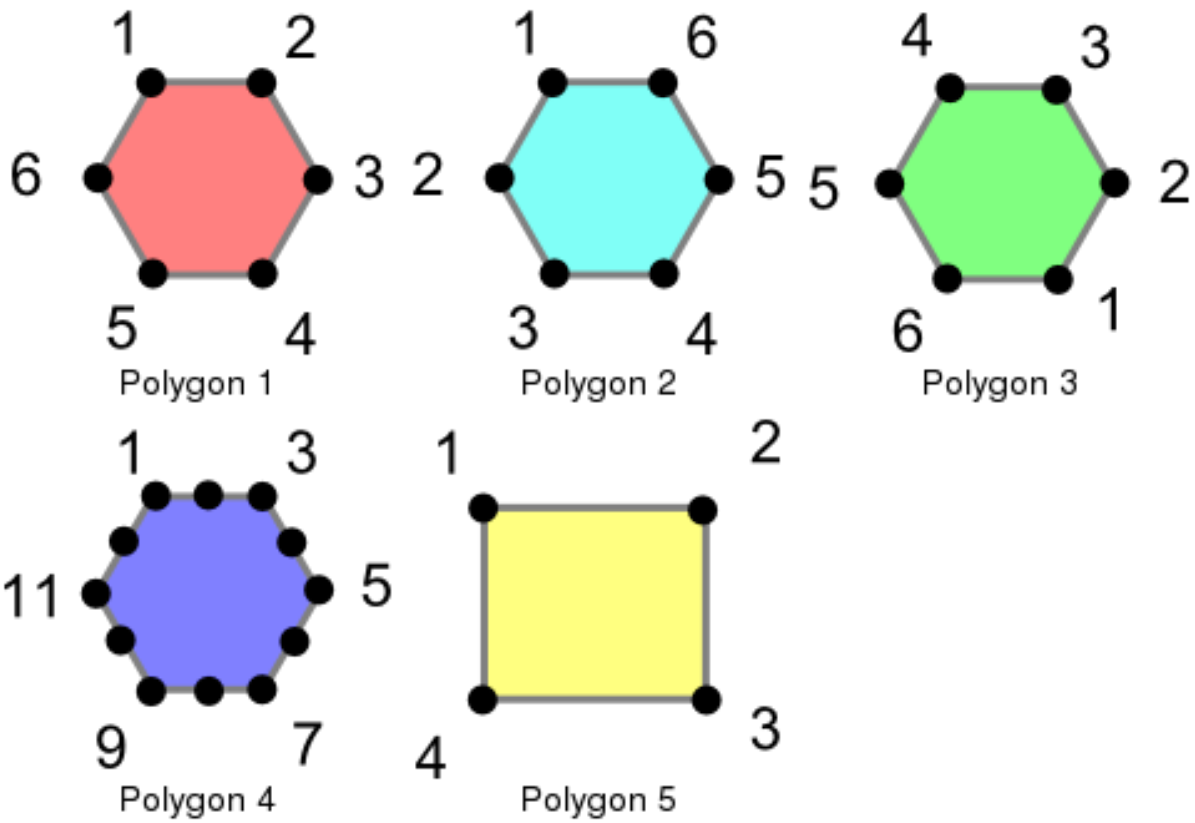
CHAPTER

TWENTYFOUR

EQUALITY

24.1 Equality

Determining equality when dealing with geometries can be tricky. PostGIS supports three different functions that can be used to determine different levels of equality, though for clarity we will use the definitions below. To illustrate these functions, we will use the following polygons.



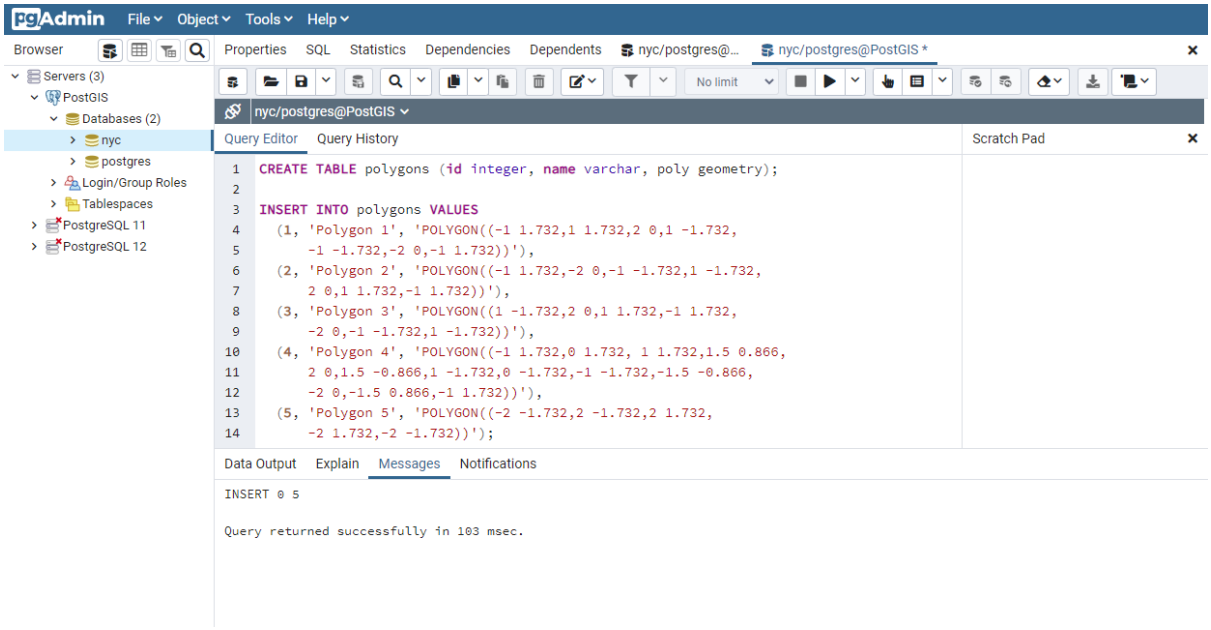
These polygons are loaded using the following commands.

```
CREATE TABLE polygons (id integer, name varchar, poly geometry);

INSERT INTO polygons VALUES
  (1, 'Polygon 1', 'POLYGON((-1 1.732,1 1.732,2 0,1 -1.732,
      -1 -1.732,-2 0,-1 1.732))'),
  (2, 'Polygon 2', 'POLYGON((-1 1.732,-2 0,-1 -1.732,1 -1.732,
      2 0,1 1.732,-1 1.732))'),
  (3, 'Polygon 3', 'POLYGON((1 -1.732,2 0,1 1.732,-1 1.732,
```

(continues on next page)

(continued from previous page)

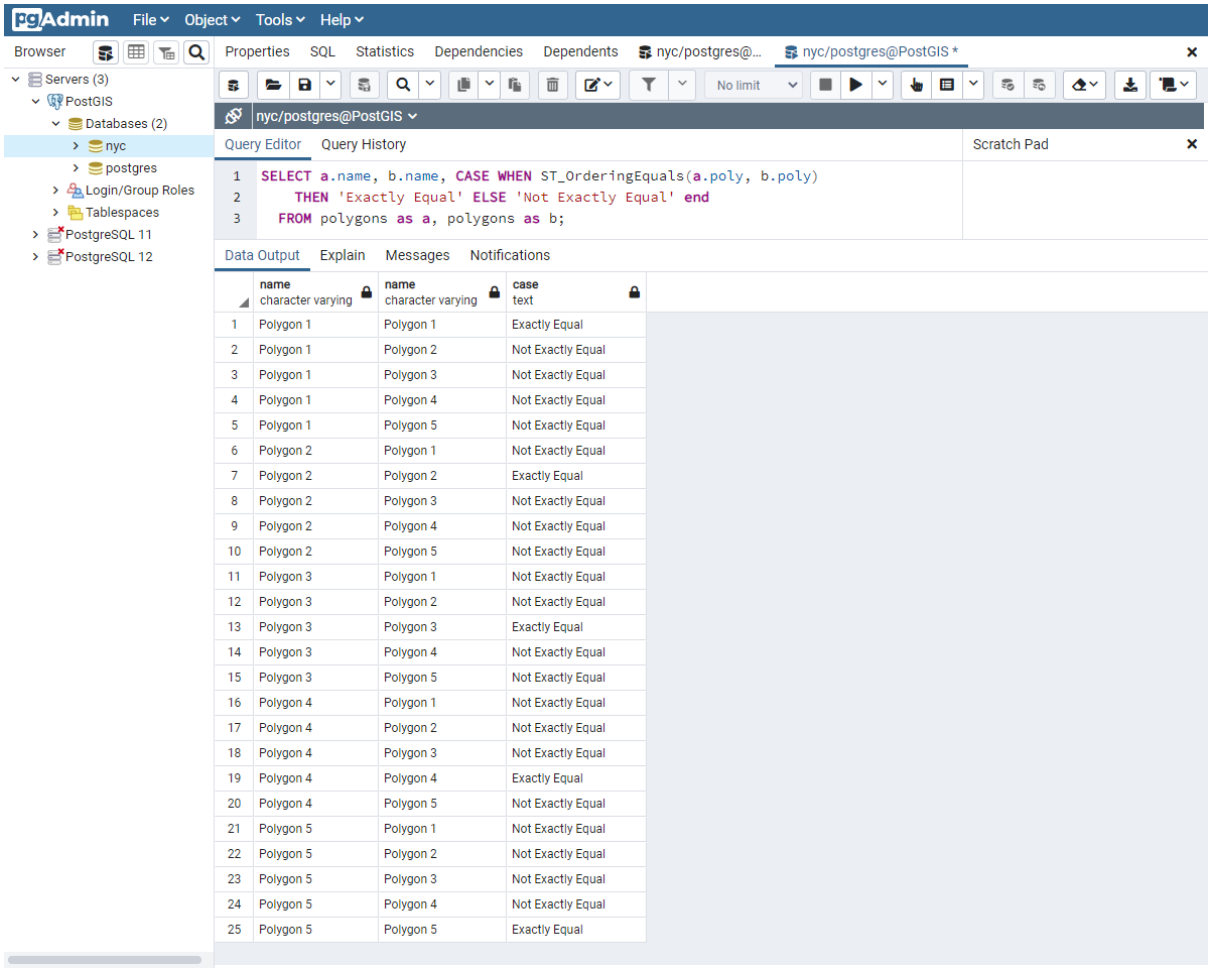
```
      -2 0,-1 -1.732,1 -1.732))'),
  (4, 'Polygon 4', 'POLYGON((-1 1.732,0 1.732, 1 1.732,1.5 0.866,
      2 0,1.5 -0.866,1 -1.732,0 -1.732,-1 -1.732,-1.5 -0.866,
      -2 0,-1.5 0.866,-1 1.732))'),
  (5, 'Polygon 5', 'POLYGON((-2 -1.732,2 -1.732,2 1.732,
      -2 1.732,-2 -1.732))');
```



24.1.1 Exactly Equal

Exact equality is determined by comparing two geometries, vertex by vertex, in order, to ensure they are identical in position. The following examples show how this method can be limited in its effectiveness.

```
SELECT a.name, b.name,
  CASE WHEN ST_OrderingEquals(a.poly, b.poly)
       THEN 'Exactly Equal'
       ELSE 'Not Exactly Equal' END
  FROM polygons AS a, polygons AS b;
```

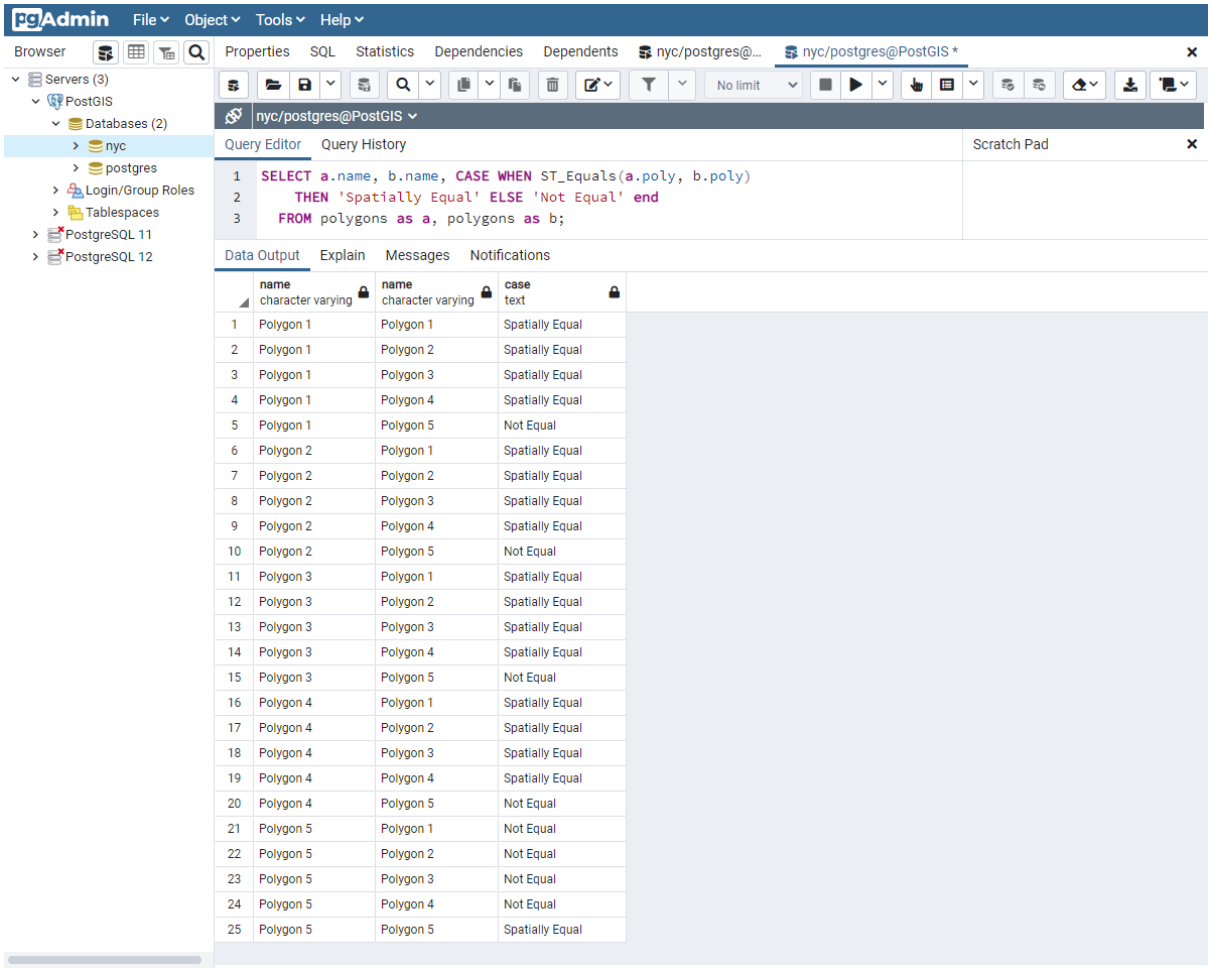


In this example, the polygons are only equal to themselves, not to other seemingly equivalent polygons (as in the case of Polygons 1 through 3). In the case of Polygons 1, 2, and 3, the vertices are in identical positions but are defined in differing orders. Polygon 4 has colinear (and thus redundant) vertices on the hexagon edges causing inequality with Polygon 1.

24.1.2 Spatially Equal

As we saw above, exact equality does not take into account the spatial nature of the geometries. There is an function, aptly named **ST_Equals**, available to test the spatial equality or equivalence of geometries.

```
SELECT a.name, b.name,
  CASE WHEN ST_Equals(a.poly, b.poly)
       THEN 'Spatially Equal'
       ELSE 'Not Equal' END
  FROM polygons AS a, polygons AS b;
```

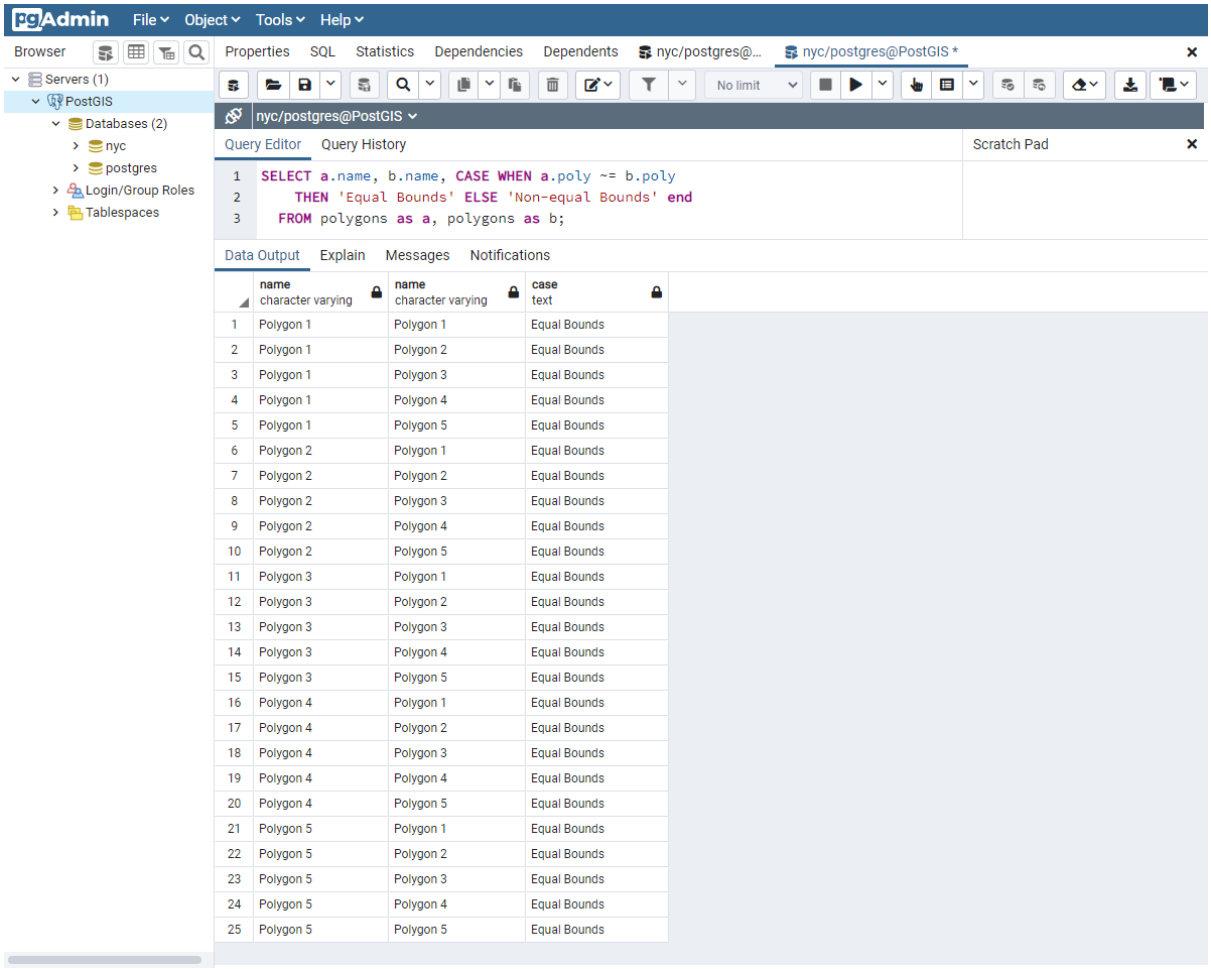


These results are more in line with our intuitive understanding of equality. Polygons 1 through 4 are considered equal, since they enclose the same area. Note that neither the direction of the polygon is drawn, the starting point for defining the polygon, nor the number of points used are important here. What is important is that the polygons contain the same space.

24.1.3 Equal Bounds

Exact equality requires, in the worst case, comparison of each and every vertex in the geometry to determine equality. This can be slow, and may not be appropriate for comparing huge numbers of geometries. To allow for speedier comparison, the equal bounds operator, **~=**, is provided. This operates only on the bounding box (rectangle), ensuring that the geometries occupy the same two dimensional extent, but not necessarily the same space.

```
SELECT a.name, b.name,
  CASE WHEN a.poly ~= b.poly
       THEN 'Equal Bounds'
       ELSE 'Non-equal Bounds' END
  FROM polygons AS a, polygons AS b;
```



As you can see, all of our spatially equal geometries also have equal bounds. Unfortunately, Polygon 5 is also returned as equal under this test, because it shares the same bounding box as the other geometries. Why is this useful, then? Although this will be covered in detail later, the short answer is that this enables the use of spatial indexing that can quickly reduce huge comparison sets into more manageable blocks when joining or filtering data.

CHAPTER

TWENTYFIVE

LINEAR REFERENCING

Linear referencing (sometimes called "dynamic segmentation") is a means of representing features that can be described by referencing a base set of linear features. Common examples of features that are modelled using linear referencing are:

- Highway assets, which are referenced using miles along a highway network
- Road maintenance operations, which are referenced as occurring along a road network between a pair of mile measurements.
- Aquatic inventories, where fish presence is recorded as existing between a pair of mileage-upstream measurements.
- Hydrologic characterizations ("reaches") of streams, recorded with a from- and to- mileage.

The benefit of linear referencing models is that the dependent spatial observations do not need to be separately recorded from the base observations, and updates to the base observation layer can be carried out knowing that the dependent observations will automatically track the new geometry.

Note: The Esri terminological convention for linear referencing is to have a base table of linear spatial features, and a non-spatial table of "events" which includes a foreign key reference to the spatial feature and a measure along the referenced feature. We will use the term "event table" to refer to the non-spatial tables we build.

25.1 Creating Linear References

If you have an existing point table that you want to reference to a linear network, use the **ST_LineLocatePoint** function, which takes a line and point, and returns the proportion along the line that the point can be found.

```
-- Simple example of locating a point half-way along a line
SELECT ST_LineLocatePoint('LINESTRING(0 0, 2 2)', 'POINT(1 1)');
-- Answer 0.5

-- What if the point is not on the line? It projects to closest point
SELECT ST_LineLocatePoint('LINESTRING(0 0, 2 2)', 'POINT(0 2)');
-- Answer 0.5
```

We can convert the **nyc_subway_stations** into an "event table" relative to the streets by using **ST_LineLocatePoint**.

```
-- All the SQL below is in aid of creating the new event table
CREATE TABLE nyc_subway_station_events AS
-- We first need to get a candidate set of maybe-closest
-- streets, ordered by id and distance...
WITH ordered_nearest AS (
SELECT
  ST_GeometryN(streets.geom,1) AS streets_geom,
  streets.gid AS streets_gid,
  subways.geom AS subways_geom,
  subways.gid AS subways_gid,
  ST_Distance(streets.geom, subways.geom) AS distance
FROM nyc_streets streets
  JOIN nyc_subway_stations subways
  ON ST_DWithin(streets.geom, subways.geom, 200)
ORDER BY subways_gid, distance ASC
)
-- We use the 'distinct on' PostgreSQL feature to get the first
-- street (the nearest) for each unique street gid. We can then
-- pass that one street into ST_LineLocatePoint along with
-- its candidate subway station to calculate the measure.
SELECT
  DISTINCT ON (subways_gid)
  subways_gid,
  streets_gid,
  ST_LineLocatePoint(streets_geom, subways_geom) AS measure,
  distance
FROM ordered_nearest;

-- Primary keys are useful for visualization softwares
ALTER TABLE nyc_subway_station_events ADD PRIMARY KEY (subways_gid);
```

Once we have an event table, it's fun to turn it back into a spatial view, so we can visualize the events relative to the original points they were derived from.

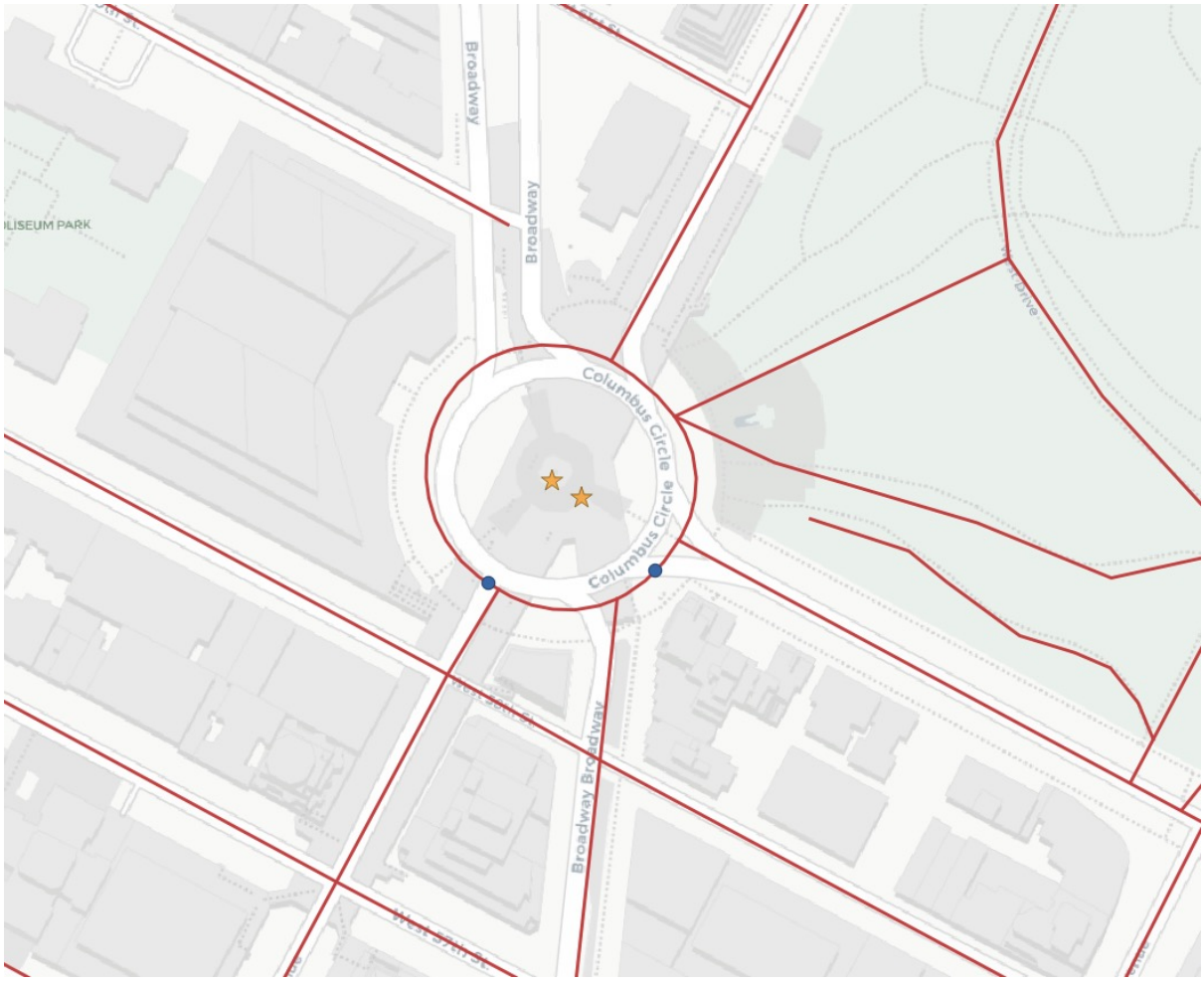
To go from a measure to a point, we use the **ST_LineInterpolatePoint** function. Here's our previous simple examples reversed:

```
-- Simple example of locating a point half-way along a line
SELECT ST_AsText(ST_LineInterpolatePoint('LINESTRING(0 0, 2 2)', 0.5));

-- Answer POINT(1 1)
```

And we can join the **nyc_subway_station_events** tables back to the **nyc_streets** table and use the **measure** attribute to generate the spatial event points, without referencing the original **nyc_subway_stations** table.

```
-- New view that turns events back into spatial objects
CREATE OR REPLACE VIEW nyc_subway_stations_lrs AS
SELECT
  events.subways_gid,
  ST_LineInterpolatePoint(ST_GeometryN(streets.geom, 1), events.measure)AS␣
↪geom,
  events.streets_gid
FROM nyc_subway_station_events events
JOIN nyc_streets streets
ON (streets.gid = events.streets_gid);
```

Viewing the original (red star) and event (blue circle) points with the streets, you can see how the events are snapped directly to the closest street lines.



Note: One surprising use of the linear referencing functions has nothing to do with linear referencing models. As shown above, it's possible to use the functions to snap points to linear features. For use cases like GPS tracks or other inputs that are expected to reference a linear network, snapping is a handy feature to have available.

25.2 Function List

- ST_LineInterpolatePoint(geometry A, double measure): Returns a point interpolated along a line.
- ST_LineLocatePoint(geometry A, geometry B): Returns a float between 0 and 1 representing the location of the closest point on LineString to the given Point.
- ST_LineSubstring(geometry A, double from, double to): Return a linestring being a substring of the input one starting and ending at the given fractions of total 2d length.
- ST_LocateAlong(geometry A, double measure): Return a derived geometry collection value with elements that match the specified measure.
- ST_LocateBetween(geometry A, double from, double to): Return a derived geometry collection value with elements that match the specified range of measures inclusively.

- ST_AddMeasure(geometry A, double from, double to): Return a derived geometry with measure elements linearly interpolated between the start and end points. If the geometry has no measure dimension, one is added.

CHAPTER

TWENTYSIX

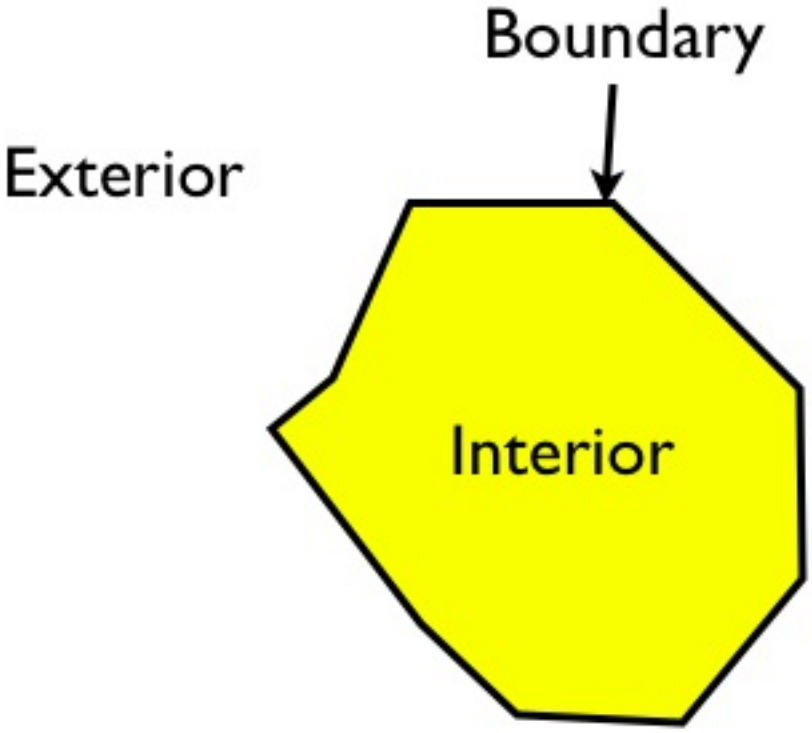
DIMENSIONALLY EXTENDED 9-INTERSECTION MODEL

The "Dimensionally Extended 9-Intersection Model" (DE9IM) is a framework for modelling how two spatial objects interact.

First, every spatial object has:

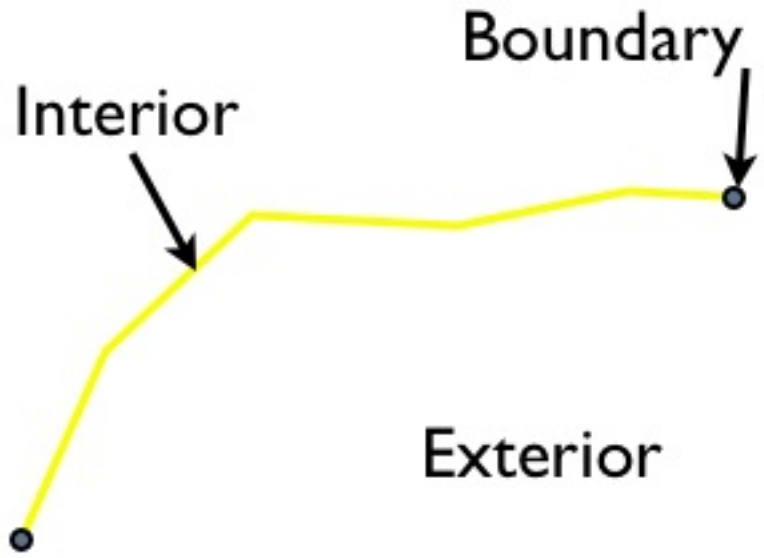
- An interior
- A boundary
- An exterior

For polygons, the interior, boundary and exterior are obvious:



The interior is the part bounded by the rings; the boundary is the rings themselves; the exterior is everything else in the plane.

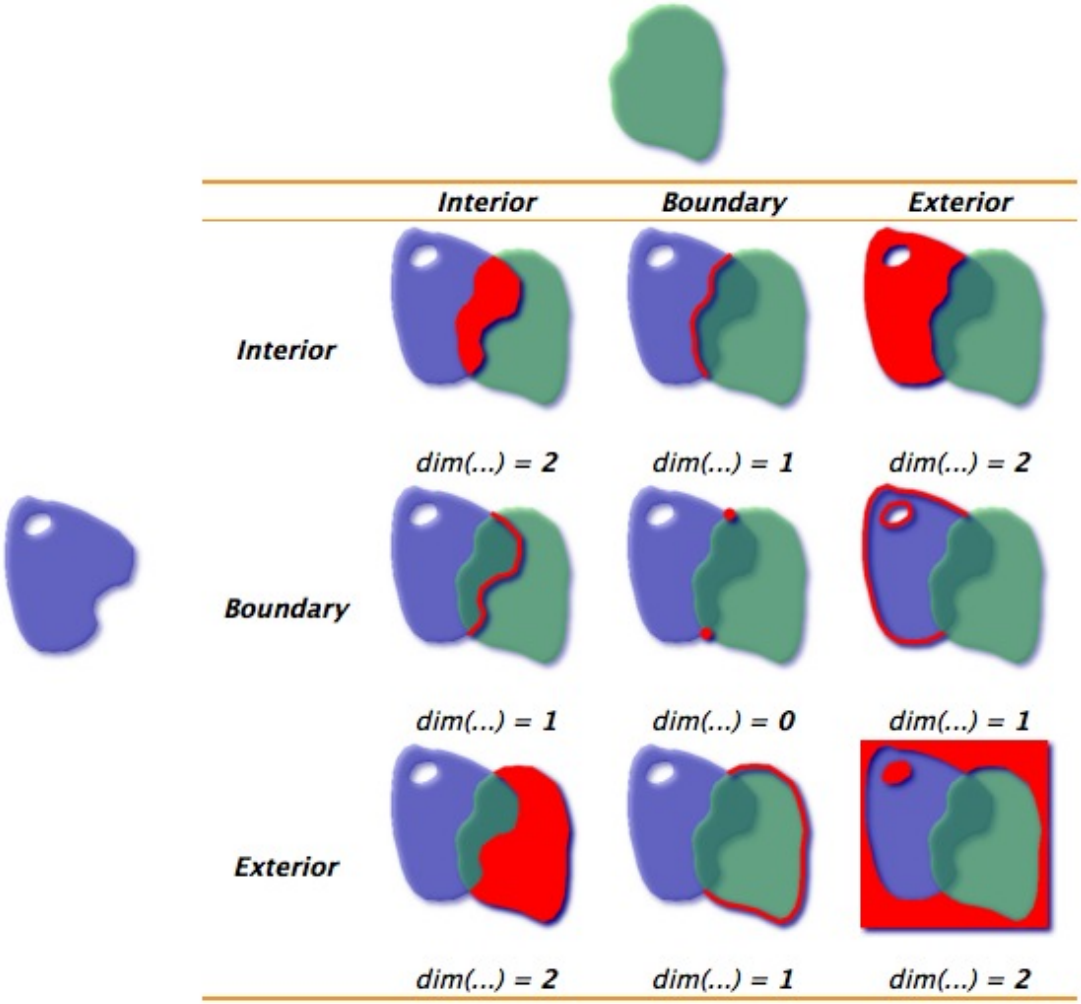
For linear features, the interior, boundary and exterior are less well-known:



The interior is the part of the line bounded by the ends; the boundary is the ends of the linear feature, and the exterior is everything else in the plane.

For points, things are even stranger: the interior is the point; the boundary is the empty set and the exterior is everything else in the plane.

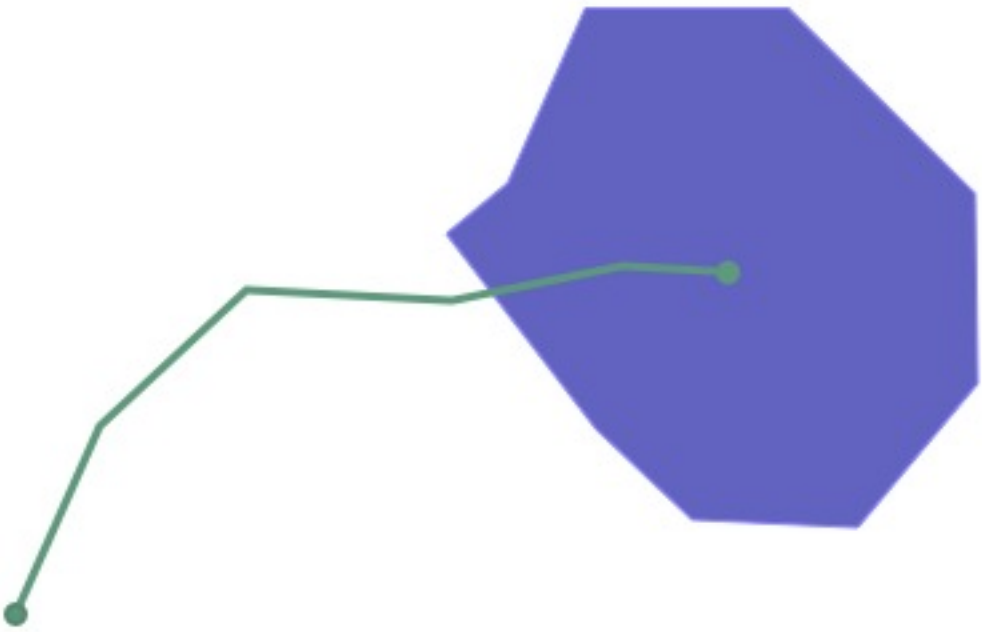
Using these definitions of interior, exterior and boundary, the relationships between any pair of spatial features can be characterized using the dimensionality of the nine possible intersections between the interiors/boundaries/exteriors of a pair of objects.



For the polygons in the example above, the intersection of the interiors is a 2-dimensional area, so that portion of the matrix is filled out with a "2". The boundaries only intersect at points, which are zero-dimensional, so that portion of the matrix is filled out with a 0.

When there is no intersection between components, the square the matrix is filled out with an "F".

Here's another example, of a linestring partially entering a polygon:



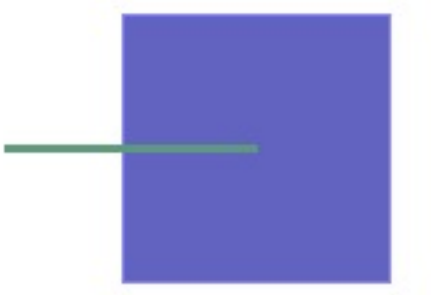
The DE9IM matrix for the interaction is this:

	I	B	E
I	1	0	1
B	0	F	0
E	2	1	2

Note that the boundaries of the two objects don't actually intersect at all (the end point of the line interacts with the interior of the polygon, not the boundary, and vice versa), so the B/B cell is filled in with an "F".

While it's fun to visually fill out DE9IM matrices, it would be nice if a computer could do it, and that's what the **ST_Relate** function is for.

The previous example can be simplified using a simple box and line, with the same spatial relationship as our polygon and linestring:



And we can generate the DE9IM information in SQL:

```
SELECT ST_Relate(
         'LINESTRING(0 0, 2 0)',
         'POLYGON((1 -1, 1 1, 3 1, 3 -1, 1 -1))'
       );
```

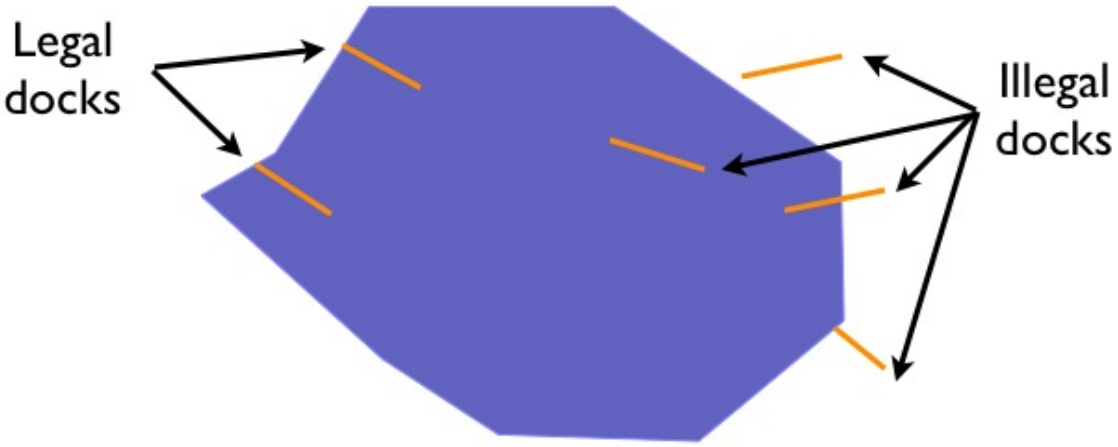
The answer (1010F0212) is the same as we calculated visually, but returned as a 9-character string, with the first row, second row and third row of the table appended together.

```
101
0F0
212
```

However, the power of DE9IM matrices is not in generating them, but in using them as a matching key to find geometries with very specific relationships to one another.

```
CREATE TABLE lakes ( id serial primary key, geom geometry );
CREATE TABLE docks ( id serial primary key, good boolean, geom geometry );

INSERT INTO lakes ( geom )
  VALUES ( 'POLYGON ((100 200, 140 230, 180 310, 280 310, 390 270, 400 210,
↪ 320 140, 215 141, 150 170, 100 200))');

INSERT INTO docks ( geom, good )
  VALUES
         ('LINESTRING (170 290, 205 272)',true),
         ('LINESTRING (120 215, 176 197)',true),
         ('LINESTRING (290 260, 340 250)',false),
         ('LINESTRING (350 300, 400 320)',false),
         ('LINESTRING (370 230, 420 240)',false),
         ('LINESTRING (370 180, 390 160)',false);
```

Suppose we have a data model that includes **Lakes** and **Docks**, and suppose further that Docks must be inside lakes, and must touch the boundary of their containing lake at one end. Can we find all the docks in our database that obey that rule?



Our legal docks have the following characteristics:

- Their interiors have a linear (1D) intersection with the lake interior
- Their boundaries have a point (0D) intersection with the lake interior
- Their boundaries **also** have a point (0D) intersection with the lake boundary
- Their interiors have no intersection (F) with the lake exterior

So their DE9IM matrix looks like this:

	I	B	E
I	I	F	F
B	0	0	F
E	2	I	2

So to find all the legal docks, we would want to find all the docks that intersect lakes (a super-set of **potential** candidates we use for our join key), and then find all the docks in that set which have the legal relate pattern.

```
SELECT docks.*
FROM docks JOIN lakes ON ST_Intersects(docks.geom, lakes.geom)
WHERE ST_Relate(docks.geom, lakes.geom, '1FF00F212');

-- Answer: our two good docks
```

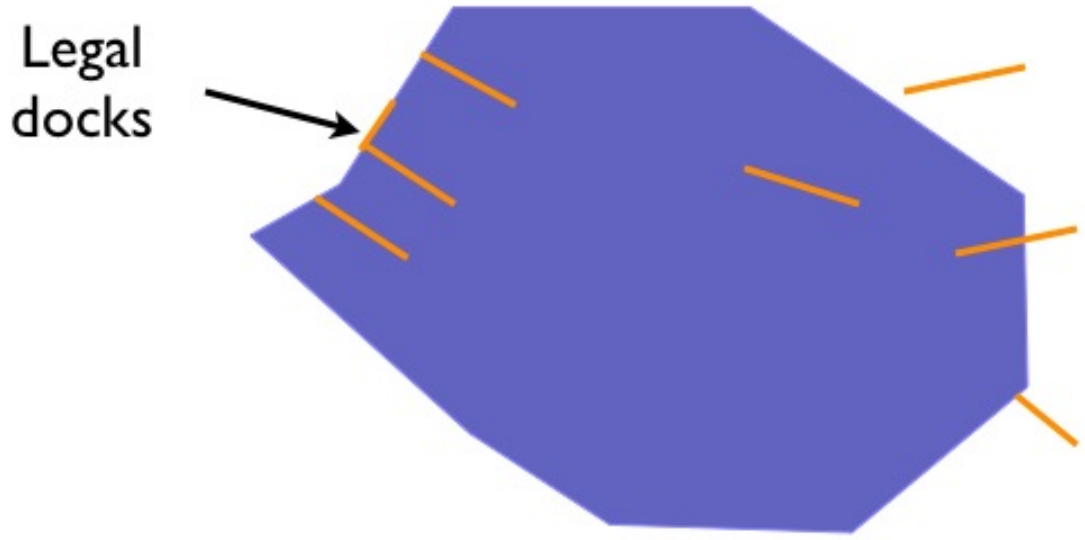
Note the use of the three-parameter version of **`ST_Relate`**, which returns true if the pattern matches or false if it does not. For a fully-defined pattern like this one, the three-parameter version is not needed – we could have just used a string equality operator.

However, for looser pattern searches, the three-parameter allows substitution characters in the pattern string:

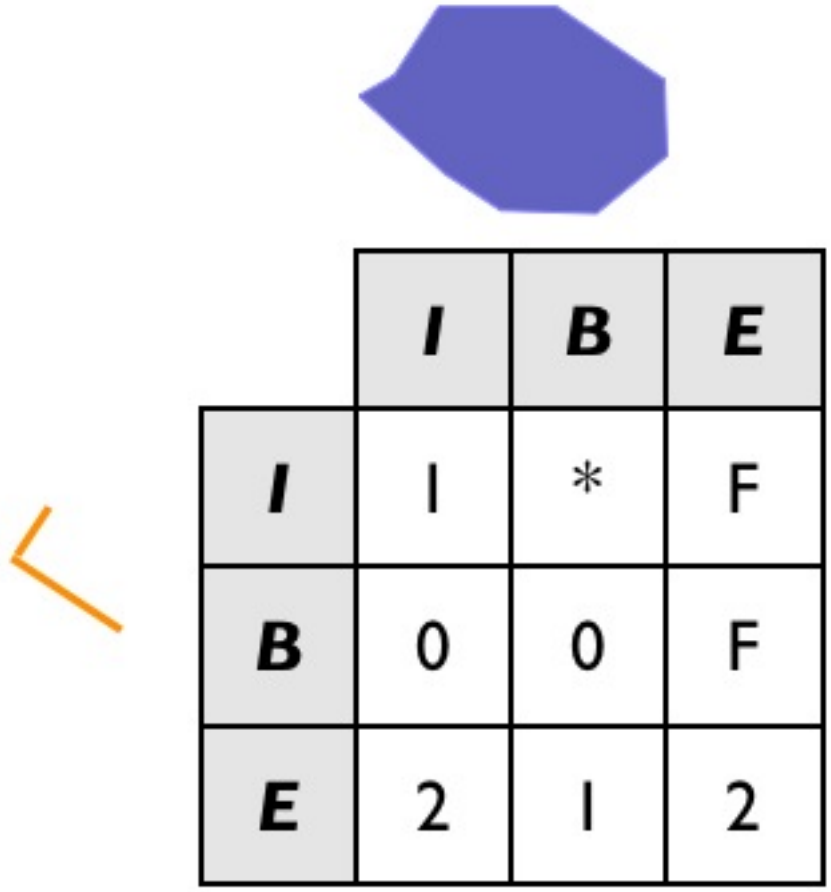
- "*" means "any value in this cell is acceptable"
- "T" means "any non-false value (0, 1 or 2) is acceptable"

So for example, one possible dock we did not include in our example graphic is a dock with a two-dimensional intersection with the lake boundary:

```
INSERT INTO docks ( geom, good )
  VALUES ('LINESTRING (140 230, 150 250, 210 230)',true);
```



If we are to include this case in our set of "legal" docks, we need to change the relate pattern in our query. In particular, the intersection of the dock interior lake boundary can now be either 1 (our new case) or F (our original case). So we use the "*" catchall in the pattern.



	I	B	E
I	1	*	F
B	0	0	F
E	2	1	2

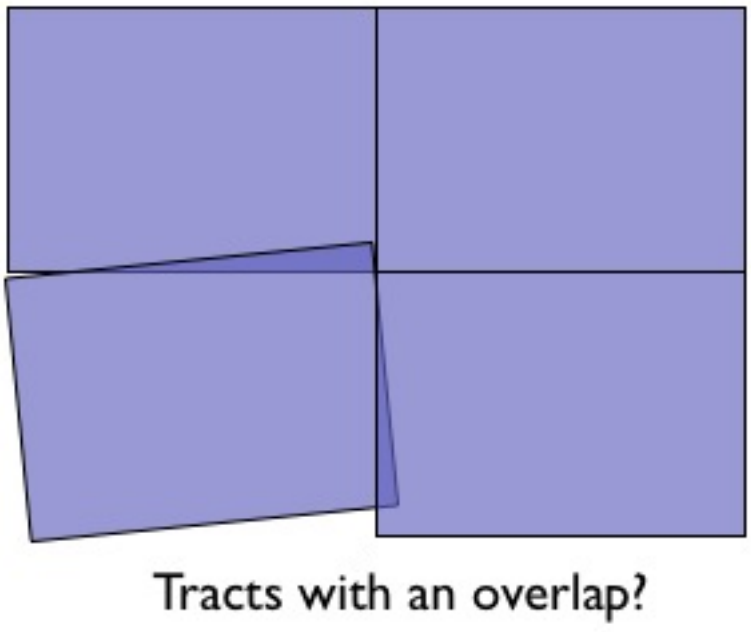
And the SQL looks like this:

```
SELECT docks.*
FROM docks JOIN lakes ON ST_Intersects(docks.geom, lakes.geom)
WHERE ST_Relate(docks.geom, lakes.geom, '1*F00F212');

-- Answer: our (now) three good docks
```

Confirm that the stricter SQL from the previous example does *not* return the new dock.

26.1 Data Quality Testing

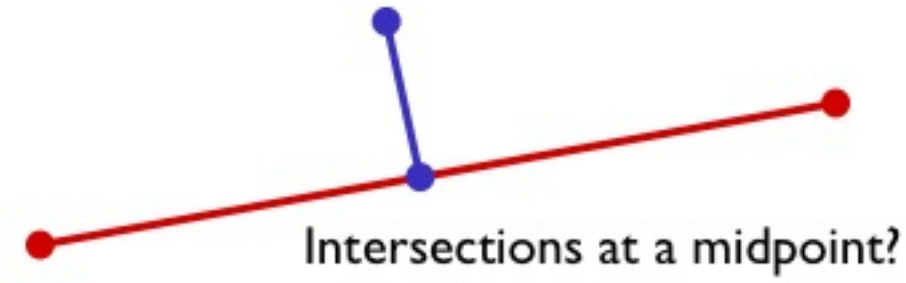
The TIGER data is carefully quality controlled when it is prepared, so we expect our data to meet strict standards. For example: no census block should overlap any other census block. Can we test for that?



Sure!

```
SELECT a.gid, b.gid
FROM nyc_census_blocks a, nyc_census_blocks b
WHERE ST_Intersects(a.geom, b.geom)
  AND ST_Relate(a.geom, b.geom, '2********')
  AND a.gid != b.gid
LIMIT 10;

-- Answer: 10, there's some funny business
```

Similarly, we would expect that the roads data is all end-noded. That is, we expect that intersections only occur at the ends of lines, not at the mid-points.



We can test for that by looking for streets that intersect (so we have a join) but where the intersection between the boundaries is not zero-dimensional (that is, the end points don't touch):

```
SELECT a.gid, b.gid
FROM nyc_streets a, nyc_streets b
WHERE ST_Intersects(a.geom, b.geom)
  AND NOT ST_Relate(a.geom, b.geom, '****0****')
  AND a.gid != b.gid
LIMIT 10;

-- Answer: This happens, so the data is not end-noded.
```

26.1.1 Function List

ST_Relate(geometry A, geometry B): Returns a text string representing the DE9IM relationship between the geometries.

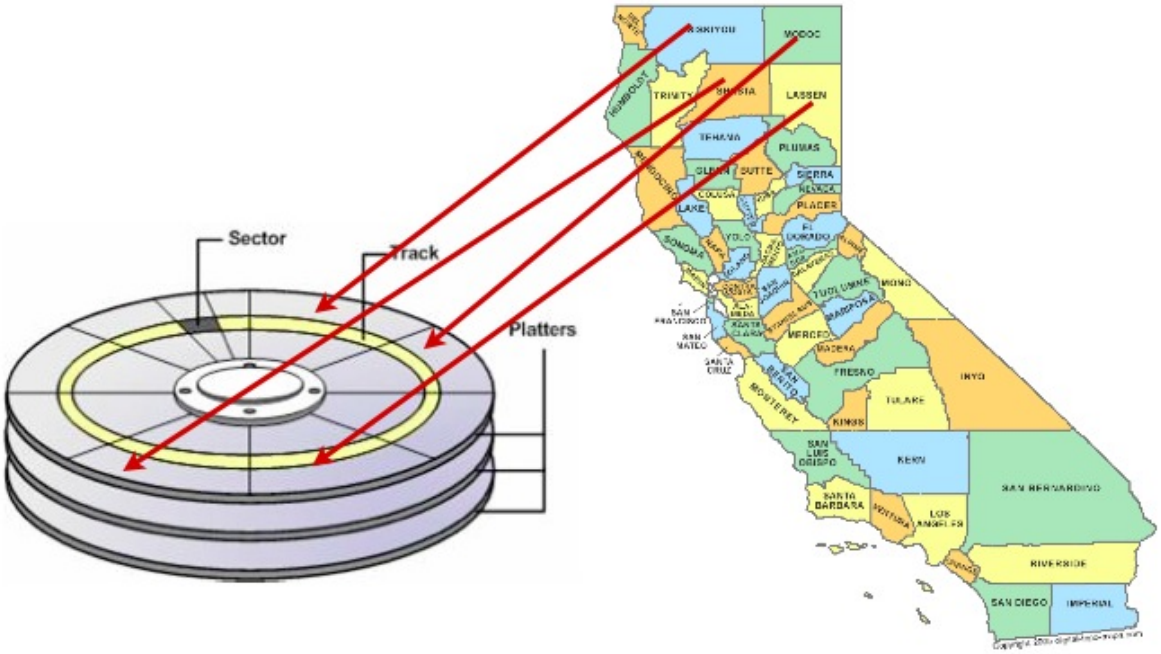
CHAPTER

TWENTYSEVEN

CLUSTERING ON INDICES

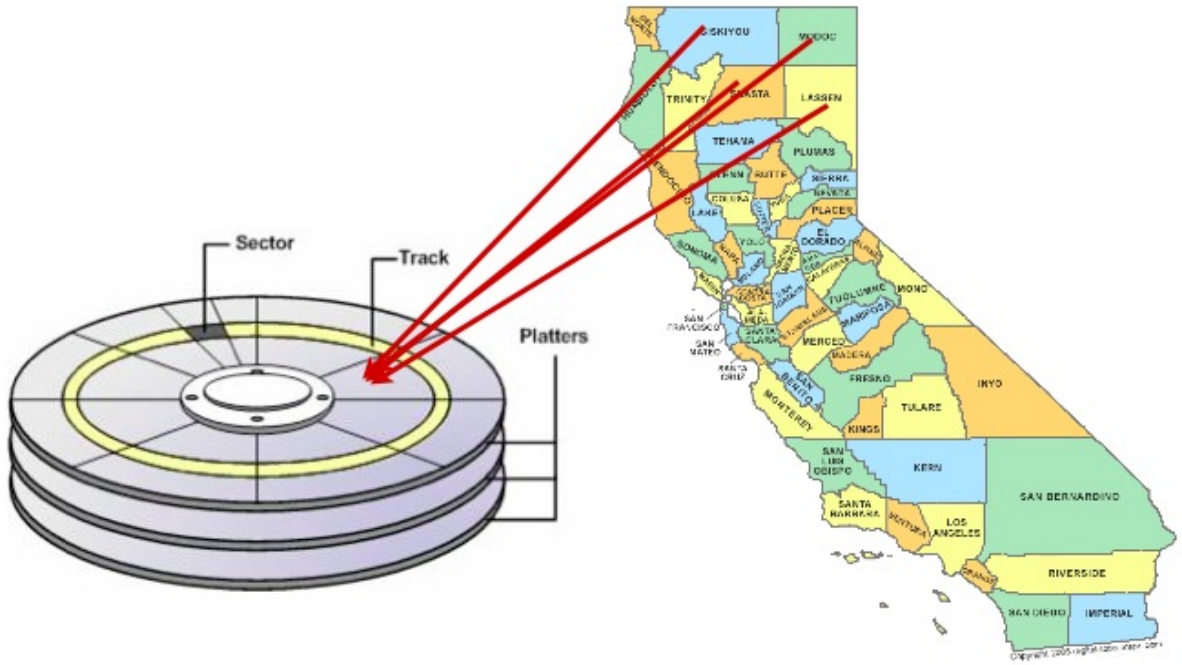
Databases can only retrieve information as fast as they can get it off of disk. Small databases will float up entirely into RAM cache, and get away from physical disk limitations, but for large databases, access to the physical disk will be a limiting stop in disk access speed.

Data is written to disk opportunistically, so there is not necessarily any correlation between the order data is stored on the disk and the way it will be accessed or organized by applications.



One way to speed up access to data is to ensure that records which is likely to be retrieved together in the same result set are located in similar physical locations on the hard disk platters. This is called "clustering".

The right clustering scheme to use can be tricky, but a general rule applies: indexes define a natural ordering scheme for data which is similar to the access pattern that will be used in retrieving the data.



Because of this, ordering the data on the disk in the same order as the index can provide a speed advantage in some cases.

27.1 Clustering on the R-Tree

Spatial data tends to be accessed in spatially correlated windows: think of the map window in a web or desktop application. All the data in the windows has similar location value (or it wouldn't be in the window!)

So, clustering based on a spatial index makes sense for spatial data that is going to be accessed with spatial queries: similar things tend to have similar locations.

Let's cluster our `nyc_census_blocks` based on their spatial index:

```
-- Cluster the blocks based on their spatial index
CLUSTER nyc_census_blocks USING nyc_census_blocks_geom_idx;
```

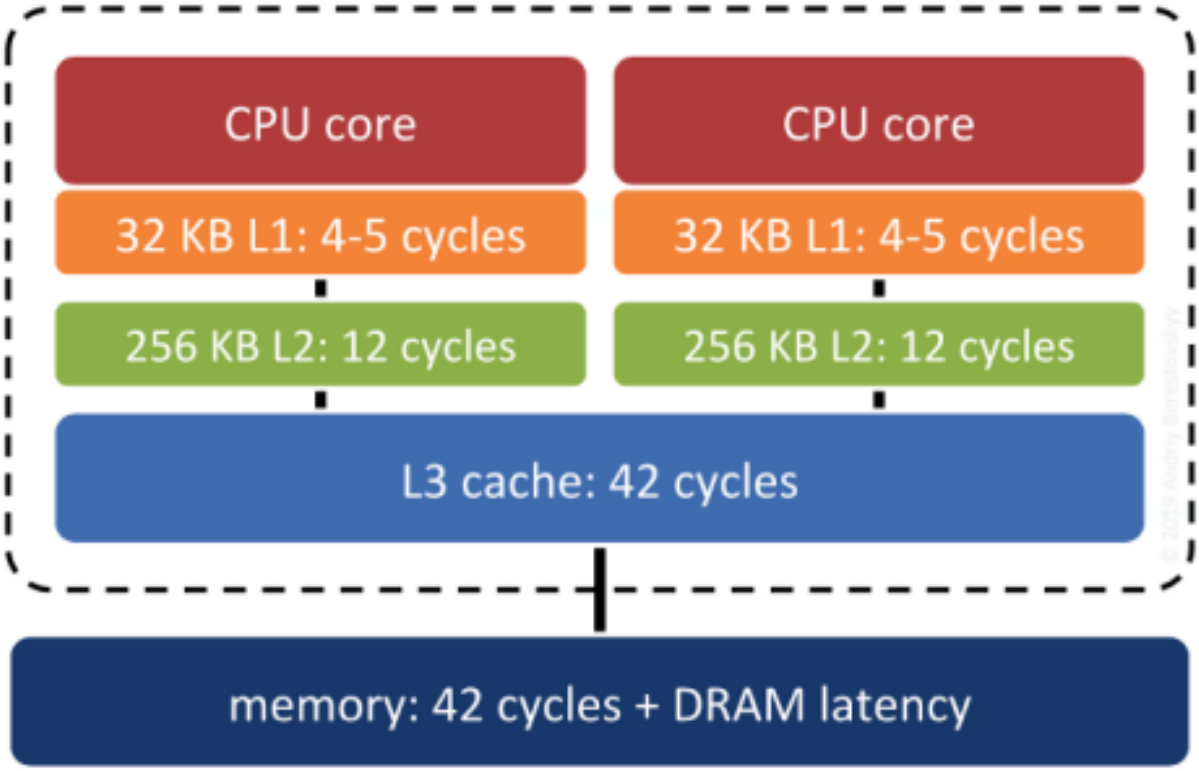
The command re-writes the `nyc_census_blocks` in the order defined by the spatial index `nyc_census_blocks_geom_gist`. Can you perceive a speed difference? Maybe not, because the original data may have already had some pre-existing spatial ordering (this is not uncommon in GIS data sets).

27.2 Disk Versus Memory/SSD

Most modern databases are run using SSD storage, which is much faster at random access than old spinning magnetic media. Also, most modern databases are running on top of data which is small enough to fit into the RAM of the database server, and ends up there as the operating system "virtual filesystem" caches it.

Is clustering still necessary?

Surprisingly, yes. Keeping records that are "near each other" in space "near each other" in memory increases the odds that related records will move up the servers "memory cache heirarchy" together, and thus make memory accesses faster.



System RAM is not the fastest memory on a modern computer. There are several levels of cache between system RAM and the actual CPU, and the underlying operating system and processor will move data up and down the cache heirarchy in blocks. If the block getting moved up happens to include the piece of data the system will need next... that's a big win. Correlating the memory structure with the spatial structure is a way in increase the odds of that win happening.

27.3 Does Index Structure Matter?

In theory, yes. In practice, no really. As long as the index is a "pretty good" spatial decomposition of the data, the main determinant of performance will be the order of the actual table tuples.

The difference between "no index" and "index" is generally huge and highly measurable. The difference between "mediocre index" and "great index" usually takes quite careful measurement to discern, and can be very sensitive to the workload being tested.

27.4 Function List

CLUSTER: Re-orders the data in a table to match the ordering in the index.

CHAPTER

TWENTYEIGHT

3-D

28.1 3-D Geometries

So far, we have been working with 2-D geometries, with only X and Y coordinates. But PostGIS supports additional dimensions on all geometry types, a "Z" dimension to add height information and a "M" dimension for additional dimensional information (commonly time, or road-mile, or upstream-distance information) for each coordinate.

For 3-D and 4-D geometries, the extra dimensions are added as extra coordinates for each vertex in the geometry, and the geometry type is enhanced to indicate how to interpret the extra dimensions. Adding the extra dimensions results in three extra possible geometry types for each geometry primitive:

- Point (a 2-D type) is joined by PointZ, PointM and PointZM types.
- Linestring (a 2-D type) is joined by LinestringZ, LinestringM and LinestringZM types.
- Polygon (a 2-D type) is joined by PolygonZ, PolygonM and PolygonZM types.
- And so on.

For well-known text (*WKT*) representation, the format for higher dimensional geometries is given by the ISO SQL/MM specification. The extra dimensionality information is simply added to the text string after the type name, and the extra coordinates added after the X/Y information. For example:

- POINT ZM (1 2 3 4)
- LINESTRING M (1 1 0, 1 2 0, 1 3 1, 2 2 0)
- POLYGON Z ((0 0 0, 0 1 0, 1 1 0, 1 0 0, 0 0 0))

The ST_AsText() function will return the above representations when dealing with 3-D and 4-D geometries.

For well-known binary (*WKB*) representation, the format for higher dimensional geometries is given by the ISO SQL/MM specification. The BNF form of the format is available from https://git.osgeo.org/gitea/postgis/postgis/src/branch/master/doc/bnf-wkb.txt.

In addition to higher-dimensional forms of the standard types, PostGIS includes a few new types that make sense in a 3-D space:

- The TIN type allows you to model triangular meshes as rows in your database.
- The POLYHEDRALSURFACE allows you to model volumetric objects in your database.

Since both these types are for modelling 3-D objects, it only really makes sense to use the Z variants. An example of a POLYHEDRALSURFACE Z would be the 1 unit cube:

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

28.2 3-D Functions

There are a number of functions built to calculate relationships between 3-D objects:

- ST_3DClosestPoint — Returns the 3-dimensional point on g1 that is closest to g2. This is the first point of the 3D shortest line.
- ST_3DDistance — For geometry type Returns the 3-dimensional cartesian minimum distance (based on spatial ref) between two geometries in projected units.
- ST_3DDWithin — For 3d (z) geometry type Returns true if two geometries 3d distance is within number of units.
- ST_3DDFullyWithin — Returns true if all of the 3D geometries are within the specified distance of one another.
- ST_3DIntersects — Returns TRUE if the Geometries "spatially intersect" in 3d - only for points and linestrings
- ST_3DLongestLine — Returns the 3-dimensional longest line between two geometries
- ST_3DMaxDistance — For geometry type Returns the 3-dimensional cartesian maximum distance (based on spatial ref) between two geometries in projected units.
- ST_3DShortestLine — Returns the 3-dimensional shortest line between two geometries

For example, we can calculate the distance between our unit cube and a point using the ST_3DDistance function:

```
-- This is really the distance between the top corner
-- and the point.
SELECT ST_3DDistance(
  'POLYHEDRALSURFACE Z (
    ((0 0 0, 0 1 0, 1 1 0, 1 0 0, 0 0 0)),
    ((0 0 0, 0 1 0, 0 1 1, 0 0 1, 0 0 0)),
    ((0 0 0, 1 0 0, 1 0 1, 0 0 1, 0 0 0)),
    ((1 1 1, 1 0 1, 0 0 1, 0 1 1, 1 1 1)),
    ((1 1 1, 1 0 1, 1 0 0, 1 1 0, 1 1 1)),
    ((1 1 1, 1 1 0, 0 1 0, 0 1 1, 1 1 1))
  )'::geometry,
  'POINT Z (2 2 2)'::geometry
);

-- So here's a shorter form.
SELECT ST_3DDistance(
  'POINT Z (1 1 1)'::geometry,
  'POINT Z (2 2 2)'::geometry
```

(continues on next page)

(continued from previous page)

```
);

-- Both return 1.73205080756888 == sqrt(3) as expected
```

28.3 N-D Indexes

Once you have data in higher dimensions it may make sense to index it. However, you should think carefully about the distribution of your data in all dimensions before applying a multi-dimensional index.

Indexes are only useful when they allow the database to drastically reduce the number of return rows as a result of a WHERE condition. For a higher dimension index to be useful, the data must cover a wide range of that dimension, relative to the kinds of queries you are constructing.

- A set of DEM points would probably be a *poor* candidate for a 3-D index, since the queries would usually be extracting a 2-D box of points, and rarely attempting to select a Z-slice of points.
- A set of GPS traces in X/Y/T space might be a *good* candidate for a 3-D index, if the GPS tracks overlapped each other frequently in all dimensions (for example, driving the same route over and over at different times), since there would be large variability in all dimensions of the data set.

You can create a multi-dimensional index on data of any dimensionality (even mixed dimensionality). For example, to create a multi-dimensional index on the `nyc_streets` table,

```
CREATE INDEX nyc_streets_gix_nd ON nyc_streets
USING GIST (geom gist_geometry_ops_nd);
```

The `gist_geometry_ops_nd` parameter tells PostGIS to use the N-D index instead of the standard 2-D index.

Once you have the index built, you can use it in queries with the `&&&` index operator. `&&&` has the same semantics as `&&`, "bounding boxes interact", but applies those semantics using all the dimensions of the input geometries. Geometries with mis-matching dimensionality do not interact.

```
-- Returns true (both 3-D on the zero plane)
SELECT 'POINT Z (1 1 0)'::geometry &&&
       'POLYGON ((0 0 0, 0 2 0, 2 2 0, 2 0 0, 0 0 0))'::geometry;

-- Returns false (one 2-D one 3-D)
SELECT 'POINT Z (3 3 3)'::geometry &&&
       'POLYGON ((0 0, 0 2, 2 2, 2 0, 0 0))'::geometry;

-- Returns true (the volume around the linestring interacts with the point)
SELECT 'LINESTRING Z(0 0 0, 1 1 1)'::geometry &&&
       'POINT(0 1 1)'::geometry;
```

To search the `nyc_streets` table using the N-D index, just replace the usual `&&` 2-D index operator with the `&&&` operator.

```
-- N-D index operator
SELECT gid, name
FROM nyc_streets
WHERE geom &&&
      ST_SetSRID('LINESTRING(586785 4492901,587561 4493037)'::geometry,
↪26918);
```

(continues on next page)

(continued from previous page)

```
-- 2-D index operator
SELECT gid, name
FROM nyc_streets
WHERE geom &&
      ST_SetSRID('LINESTRING(586785 4492901,587561 4493037)'::geometry,
↪26918);
```

The results should be the same. In general the N-D index is very slightly slower than the 2-D index, so only use the N-D index where you are certain that N-D queries will improve the selectivity of your queries.

CHAPTER

TWENTYNINE

NEAREST-NEIGHBOUR SEARCHING

29.1 What is a Nearest Neighbour Search?

A frequently posed spatial query is: "what is the nearest <candidate feature> to <query feature>?"

Unlike a distance search, the "nearest neighbour" search doesn't include any measurement restricting how far away candidate geometries might be, features of any distance away will be accepted, as long as they are the *nearest*.

PostgreSQL solves the nearest neighbor problem by introducing an "order by distance" (<->) operator that induces the database to use an index to speed up a sorted return set. With an "order by distance" operator in place, a nearest neighbor query can return the "N nearest features" just by adding an ordering and limiting the result set to N entries.

The "order by distance" operator works for both geometry and geography types. The only difference between how they work between the two types is the distance value returned. For geometry <-> returns the same answer as *ST_Distance* which is dependent on the units of the spatial reference system in use. For geography the distance value returned is the sphere distance, instead of the more accurate spheroidal distance that `ST_Distance(geography,geography)` returns.

Here's the 3 nearest streets to 'Broad St' subway station:

```
-- Get the geometry of Broad St
SELECT ST_AsEWKT(geom, 1)
FROM nyc_subway_stations
WHERE name = 'Broad St';
```

```
SRID=26918;POINT(583571.9 4506714.3)
```

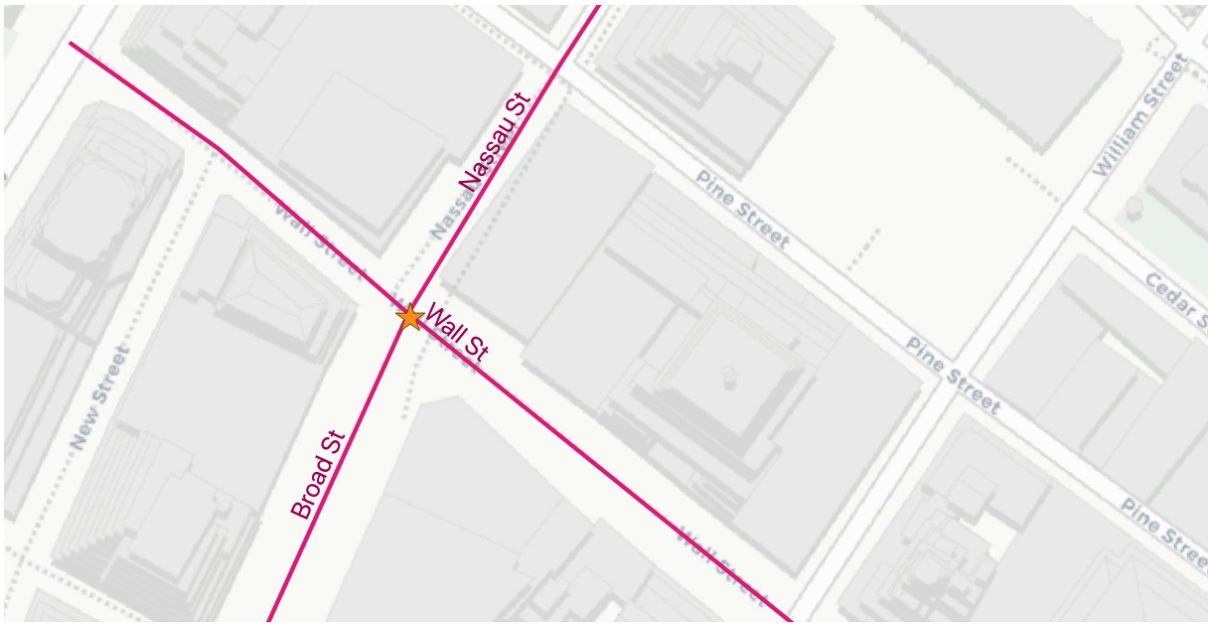
```
-- Plug the geometry into a nearest-neighbor query
SELECT streets.gid, streets.name,
  ST_Transform(streets.geom, 4326),
  streets.geom <-> 'SRID=26918;POINT(583571.9 4506714.3)'::geometry AS dist
FROM
  nyc_streets streets
ORDER BY
  dist
LIMIT 3;
```

```
  gid  |   name    |        dist
-------+-----------+--------------------
 17385 | Wall St   |  0.749987508809928
```

(continues on next page)

(continued from previous page)

```
 17390 | Broad St   | 0.8836306235191059
 17436 | Nassau St  | 1.3368280241070414
```



How can we be sure we are getting an index-assisted query? It's a good idea to check the `EXPLAIN` output for a nearest-neighbor query, because it's possible to get correct answers from non-indexed SQL and the lack of an index might not be obvious until the size of the tables scales up.

This is the output from `EXPLAIN`, note the index scan over the order by:

```
                                QUERY PLAN
-------------------------------------------------------------------------------
↪------
Limit  (cost=0.28..79.58 rows=3 width=31)
   ->  Index Scan using nyc_streets_geom_idx on nyc_streets streets
       (cost=0.28..504685.12 rows=19091 width=31)
         Order By:
         (geom <-> '0101000020266900000EEBD4CF27CF2141BC17D69516315141
↪'::geometry)
```

29.2 Nearest Neighbor Join

The index assisted order by operator has one major draw back: it only works with a **single geometry literal** on one side of the operator. This is fine for finding the objects nearest to one query object, but does not help for a spatial join, where the goal is to find the nearest neighbor for each of a full set of candidates.

Fortunately, there's a SQL language feature that allows us to run a query repeatedly driven in a loop: the LATERAL join.

Here we will find the nearest street to each subway station:

```
SELECT subways.gid AS subway_gid,
       subways.name AS subway,
```

(continues on next page)

(continued from previous page)

```
       streets.name AS street,
       streets.gid AS street_gid,
       streets.geom::geometry(MultiLinestring, 26918) AS street_geom,
       streets.dist
FROM nyc_subway_stations subways
CROSS JOIN LATERAL (
  SELECT streets.name, streets.geom, streets.gid, streets.geom <-> subways.
↪geom AS dist
  FROM nyc_streets AS streets
  ORDER BY dist
  LIMIT 1
) streets;
```

Note the way the `CROSS JOIN LATERAL` acts as the inner part of a loop driven by the subways table. Each record in the subways table gets fed into the lateral subquery, one at a time, so you get a nearest result for each subway record.

The explain shows the loop on the subway stations, and the index-assisted order by inside the loop where we want it:

```
                              QUERY PLAN
-----------------------------------------------------------------------------
 Nested Loop  (cost=0.28..13140.71 rows=491 width=37)
   ->  Seq Scan on nyc_subway_stations subways
       (cost=0.00..15.91 rows=491 width=46)
   ->  Limit
       (cost=0.28..1.71 rows=1 width=170)
         ->  Index Scan using nyc_streets_geom_idx on nyc_streets streets
             (cost=0.28..27410.12 rows=19091 width=170)
               Order By: (geom <-> subways.geom)
```

CHAPTER

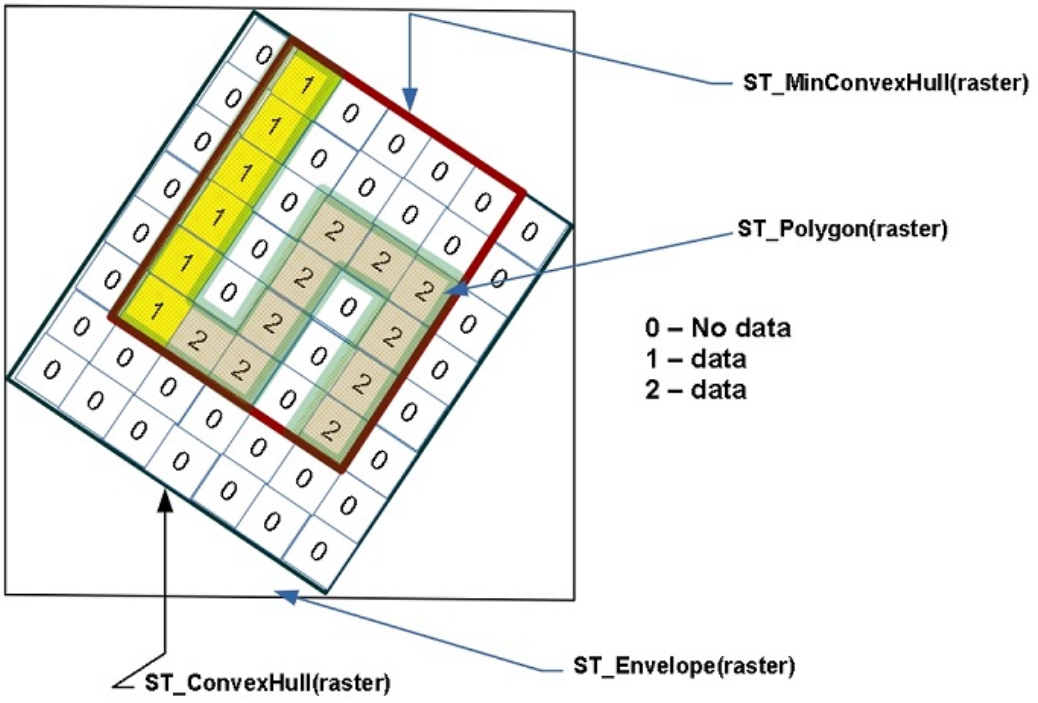
THIRTY

RASTERS

PostGIS supports another kind of spatial data type called a *raster*. Raster data, much like geometry data, uses **Cartesian coordinates** and a spatial reference system. However instead of vector data, raster data is represented as an n-dimensional matrix consisting of pixels and bands. The bands defines the number of matrices you have. Each pixel stores a value corresponding to each band. So a 3-banded raster such as an RGB image, would have 3 values for each pixel corresponding to the Red-Green-Blue bands.

Although pretty pictures such as those you see on your TV screen are rasters, rasters may not be that exciting to look at. In a nutshell, a raster is a matrix, pinned on a coordinate system, that has values that can represent anything you want them to represent.

Since rasters live in cartesian space, rasters can interact with geometries. PostGIS offers many functions that take as input both rasters and geometries. Many operations applied to rasters will result in geometries. Common ones are the *ST_Polygon*, *ST_Envelope*, *ST_ConvexHull*, and *ST_MinConvexHull* as shown below. When you cast a raster to a geometry, what is output is the *ST_ConvexHull* of the raster.



The raster format is commonly used to store elevation data, temperature data, satellite data, and thematic data representing things like environmental contamination, population density, and environmental hazard occurrences. You can use rasters to store any numeric data that has a meaningful coordinate location. The only restriction is that for all data in a specific band the numeric data types have to be the same.

Although raster data can be created from scratch in PostGIS, a more common approach is to load raster data from various formats using the **raster2pgsql** command line tool packaged with PostGIS. Before all of that, you must enable raster support in your database by running the command:

```
CREATE EXTENSION postgis_raster;
```

30.1 Creating Rasters From Geometries

We'll start off by first creating raster data from vector data, and then move on to the more exciting approach of loading data from a raster source. You will find that raster data is available in abundance and often free from various government sites.

We'll start by converting some geometries into rasters using ST_AsRaster function as follows.

```
CREATE TABLE rasters (name varchar, rast raster);

INSERT INTO rasters(name, rast)
SELECT f.word, ST_AsRaster(geom, width=>150, height=>150)
FROM (VALUES ('Hello'), ('Raster') ) AS f(word)
  , ST_Letters(word) AS geom;

CREATE INDEX ix_rasters_rast
  ON rasters USING gist(ST_ConvexHull(rast));
```

The above example CREATEs a table (**rasters**) from geometries formed from letters using the PostGIS 3.2+ ST_Letters function. Rasters similar to geometries, can take advantage of spatial indexes. The spatial index used for raster is a functional index that indexes the geometry convexhull of the raster.

You can see some useful metadata of your rasters with the following query which utilizes the postgis raster ST_Count function to count the number of pixels that have data and the ST_MetaData function to provide all sorts of useful background info for our rasters.

```
SELECT name, ST_Count(rast) As num_pixels, md.*
   FROM rasters, ST_MetaData(rast) AS md;
```

```
name   | num_pixels | upperleftx |     upperlefty      | width | height |
↪  scalex       |       scaley        | skewx | skewy | srid | numbands
--------+------------+------------+---------------------+-------+--------+---
↪----------------+--------------------+-------+-------+------+----------
Hello  |      13926 |          0 | 77.10000000000001 |   150 |    150 |  1.
↪226888888888889 | -0.5173333333333334 |     0 |     0 |    0 |        1
Raster |      11967 |          0 |              75.4 |   150 |    150 | 1.
↪7226319023207244 | -0.5086666666666667 |     0 |     0 |    0 |        1
(2 rows)
```

Note: There are two levels of raster functions. There are functions such as ST_MetaData that work at the raster level and there are functions such as **ST_Count** function and ST_BandMetaData function

that work at the band level. Most functions in postgis raster that work at the band level, work with only one band at a time, and assume the band you want is *1*.

If you have a multi-band raster, and you need to count the pixel not no-data values in a band other than 1, you would explicitly specify the band number as follows *ST_Count(rast,2)*.

Note how all the rasters have a 150x150 dimension. This is not ideal. This means that in order to force that, our rasters, are squished in all sorts of ways. If only we could see the ugliness of the rasters before us.

30.2 Loading Rasters using raster2pgsql

raster2pgsql is a command-line tool often packaged with PostGIS. If you are on windows and used application stackbuilder PostGIS Bundle, you'll find **`raster2pgsql.exe`** in the folder `C:\Program Files\PostgreSQL\15\bin` where the *15* should be replaced with the version of PostgreSQL you are running.

If you are using Postgres.App, you'll find raster2pgsql among the other Postgres.app CLI Tools.

On Ubuntu and Debian, you will need

```
apt install postgis
```

to have the PostGIS commandline tools installed. This may install an additional version of PostgreSQL as well. You can see a list of clusters in Debian/Ubuntu using the **`pg_lsclusters`** command and drop them using the **`pg_dropcluster`** command.

For this and later exercises, we'll be using *nyc_dem.tif* found in the file PG Raster Workshop Dataset https://postgis.net/stuff/workshop-data/postgis_raster_workshop.zip. For some geometry/raster examples, we will also be using NYC data loaded from prior chapters. In-lieu of loading the tif, you can restore the *nyc_dem.backup* included in the zip file in your database using the **`pg_restore`** commandline tool or the pgAdmin **Restore** menu.

Note: This raster data was sourced from NYC DEM 1-foot Integer which is a 3GB DEM tif representing elevation relative to sea level with buildings and overwater removed. We then created a lower res version of it.

The **`rasterpgsql`** tool is similar to the **`shp2gpsql`** except instead of loading ESRI shapefiles into PostGIS geometry/geography tables, it loads any GDAL supported raster format into raster tables. Just like **`shp2pgsql`** you can pass it a spatial reference id (SRID) of the source. Unlike **`shp2pgsql`** it can infer the spatial references system of the source data if your source data has suitable metadata.

For a full exposure of all the possible switches offered refer to raster2pgsql options.

Some other notable options **`raster2pgsql`** offers which we will not cover are:

- Ability to denote the SRID of the source. Instead, we'll rely on raster2pgsql guessing skills.
- Ability to set the *nodata* value, when not specified, raster2pgsql tries to infer from the file.
- Abiliity to load out-of-database rasters.

To load all the tif files in our folder and also create overviews, we would run the below.

```
raster2pgsql -d -e -l 2,3 -I -C -M -F -Y -t 256x256 *.tif nyc_dem | psql -
↪d nyc
```

- -d to drop the tables if they already exist
- The above command uses *-e* to do load immediately instead of committing in a transaction
- *-C* set raster constraints, this is useful for *raster_columns* to show info. You may want to combine with *-x* to exclude the extent constraint, which is a slow constraint to check and also hampers future loads in the table.
- *-M* to vacuum and analyze after load, to improve query planner statistics
- *-Y* to use copy in batches of 50. If you are running PostGIS 3.3 or higher, you can use *-Y 1000* to have copy be in batches of 1000, or even higher number. This will run faster, but will use more memory.
- *-l 2,3* to create over view tables: *o_2_ncy_dem* and *o_3_nyc_dem*. This is useful for viewing data.
- -I to create a spatial index
- *-F* to add file name, if you have only one tif file, this is kinda pointless. If you had multiple, this would be useful to tell you what file each row came from.
- *-t* to set the block size. Note if you are not sure the best size use, use *-t auto* instead and raster2pgql will use the same tiling as what was in the tif. The output will tell you what the blocksize is it chose. Cancel if it looks huge or weird. The original file had a size of 300x7 which is not ideal.
- Use **psql** to run the generated sql against the database. If you want to dump to a file instead, use *> nyc_dem.sql*

For this example, we have only one tif file, so we could instead specify the full file name, instead of **.tif*. If the files are not in your current directory, you can also specify a folder path with **.tif*.

Note: If you are on windows and need to reference the folder, make sure to include the drive letter such as *C:/workshop/*.tif*

You'll often hear in PostGIS lingo, the term **raster tile** and **raster** used somewhat interchangeably. A raster tile really corresponds to a particular raster in a raster column which is a subset of a bigger raster, such as this NYC dem data we just loaded. This is because when rasters are loaded into PostGIS from big raster files, they chopped into many rows to make them manageable. Each raster in each row then is a part of a bigger raster. Each tile covers same size area denoted by the blocksize you specified. Rasters are sadly limited by the 1GB PostgreSQL TOAST limit and also the slow process of detoasting and so we need to chop up in order to achieve decent performance or to even store them.

30.3 Viewing Rasters in Browser

Although pgAdmin and psql have no mechanism yet to view postgis rasters, we have a couple of options. For smallish rasters the easiest is to output to a web-friendly raster format such as PNG using batteries included postgis raster functions like *ST_AsPNG* or *ST_AsGDALRaster* listed in PostGIS Raster output functions. As your rasters get larger, you'll want to graduate to a tool such as QGIS to view them in all their glory or the GDAL family of commandline tools such as **gdal_translate** to export them to other raster formats. Remember though, postgis rasters are built for analysis, not for generating pretty pictures for you to look at.

One caveat, by default all different raster types outputs are disabled. In order to utilize these, you'll need to enable drivers, all or a subset as detailed in Enable GDAL Raster drivers

```
SET postgis.gdal_enabled_drivers = 'ENABLE_ALL';
```

If you don't want to have to do this for each connection, you can set at the database level using:

```
ALTER DATABASE nyc SET postgis.gdal_enabled_drivers = 'ENABLE_ALL';
```

Each new connection to the database will use that setting.

Run the below query and copy and paste the output into the address bar of your web browser.

```
SELECT '' ||
   encode(ST_AsPNG(rast),'base64')
   FROM rasters
   WHERE name = 'Hello';
```

For the rasters created thus far, we didn't specify the number of bands nor did we even define their relation to earth. As such our rasters have an unknown spatial reference system (0).

You can think of a rasters exoskeletal as a geometry. A matrix encased in a geometric envelop. In order to do useful analysis, we need to georeference our rasters, meaning we want each pixel (rectangle) to represent some meaningful plot of space.

The *ST_AsRaster* has many overloaded representations. The earlier example used the simplest such implementation and accepted the default arguments which are 8BUI and 1 band, with no data being 0. If you need to use the other variants, you should use the named arguments call syntax so that you don't accidentally fall into the wrong variant of the function or get **function is not unique** errors.

If you start with a geometry that has a spatial reference system, you'll end up with a raster with same spatial reference system. In this next example, we'll plop our words in New York in bright cheery colors.

We will also use pixel scale instead of width and height so that our raster pixel sizes represent 1 meter x 1 meter of space.

```
INSERT INTO rasters(name, rast)
SELECT f.word || ' in New York' ,
  ST_AsRaster(geom,
    scalex => 1.0, scaley => -1.0,
    pixeltype => ARRAY['8BUI', '8BUI', '8BUI'],
    value => CASE WHEN word = 'Hello' THEN
      ARRAY[10,10,100] ELSE ARRAY[10,100,10] END,
    nodataval => ARRAY[0,0,0], gridx => NULL, gridy => NULL
    ) AS rast
FROM (
    VALUES ('Hello'), ('Raster') ) AS f(word)
  , ST_SetSRID(
      ST_Translate(ST_Letters(word),586467,4504725), 26918
    ) AS geom;
```

If we then look at this, we'll see a non-squashed colored geometry.

```
SELECT '' ||
   encode(ST_AsPNG(rast),'base64')
   FROM rasters
   WHERE name = 'Hello in New York';
```

Repeat for Raster:

```
SELECT '' ||
   encode(ST_AsPNG(rast),'base64')
   FROM rasters
   WHERE name = 'Raster in New York';
```

What is more telling, if we rerun the

```
SELECT name, ST_Count(rast) As num_pixels, md.*
  FROM rasters, ST_MetaData(rast) AS md;
```

Observe the metadata of the New York entries. They have the New York state plane meter spatial reference system. They also have the same scale. Since each unit is 1x1 meter, the width of the word **Raster** is now wider than **Hello**.

```
      name          | num_pixels | upperleftx |    upperlefty     | width |
↪height |       scalex       |       scaley       | skewx | skewy | srid
↪ | numbands
------------------+------------+------------+-------------------+-------+-
↪-------+--------------------+--------------------+-------+-------+------
↪-+----------
Hello               |      13926 |          0 | 77.10000000000001 |   150 |
↪   150 |  1.226888888888889 | -0.5173333333333334 |     0 |     0 |
↪0 |         1
Raster              |      11967 |          0 |              75.4 |   150 |
↪   150 | 1.7226319023207244 | -0.5086666666666667 |     0 |     0 |
↪0 |         1
Hello in New York   |       8786 |     586467 |         4504802.1 |   184 |
↪    78 |                  1 |                  -1 |     0 |     0 |
↪26918 |         3
Raster in New York  |      10544 |     586467 |         4504800.4 |   258 |
↪    76 |                  1 |                  -1 |     0 |     0 |
↪26918 |         3
(4 rows)
```

30.4 Raster Spatial Catalog tables

Similar to the geometry and geography types, raster has a set of catalogs that show you all raster columns in your database. These are raster_columns and raster_overviews.

30.4.1 raster_columns

The *raster_columns* view to the sibling to the *geometry_columns* and *geography_columns*, providing much the same data and more, but for raster columns.

```
SELECT *
    FROM raster_columns;
```

Explore the table, and you'll find this:

```
r_table_catalog | r_table_schema | r_table_name | r_raster_column | srid |
↪scale_x | scale_y | blocksize_x | blocksize_y | same_alignment | regular_
↪blocking | num_bands | pixel_types | nodata_values | out_db | extent |
↪spatial_index
----------------+----------------+--------------+-----------------+------+-
↪--------+---------+-------------+-------------+----------------+---------
↪---------+-----------+-------------+---------------+--------+--------+---
↪------------
nyc             | public         | rasters      | rast            |    0 |
↪        |         |             |             | f              | f
↪         |           |            |                |        |        | t
nyc             | public         | nyc_dem      | rast            | 2263 |
↪     10 |     -10 |         256 |         256 | t              | f
↪         |         1 | {16BUI}     | {NULL}        | {f}    |        | t
nyc             | public         | o_2_nyc_dem  | rast            | 2263 |
↪     20 |     -20 |         256 |         256 | t              | f
↪         |         1 | {16BUI}     | {NULL}        | {f}    |        | t
nyc             | public         | o_3_nyc_dem  | rast            | 2263 |
↪     30 |     -30 |         256 |         256 | t              | f
↪         |         1 | {16BUI}     | {NULL}        | {f}    |        | t
```

(continues on next page)

(continued from previous page)

```
(4 rows)
```

a disappointing row of largely unfilled information for the *rasters* table.

Unlike geometry and geography, raster does not support type modifiers, because type modifier space is too limited and there are more critical properties than what can fit in a type modifier.

Raster instead relies on constraints, and reads these constraints back as part of the view.

Look at the other rows from the tables we loaded using **raster2pgsql**. Because we used the *-C* switch **raster2pgsql** added constraints for the srid and other info it was able to read from the tif or that we passed in. The overview tables generated with the *-l* switch *o_2_nyc_dem* and *o_3_nyc_dem* show up as well.

Let's try to add some constraints to our table.

```
SELECT AddRasterConstraints('public'::name, 'rasters'::name, 'rast'::name);
```

And you'll be bombarded with a whole bunch of notices about how your raster data is a mess and nothing can be constrained. If you look at raster_columns again, still the same disappointing story of many blank rows for *rasters*.

In order for constraints to be applied, all rasters in your table must be constrainable by at least one rule.

We can perhaps do this, let's just lie and say all our data is in New York State plane.

```
UPDATE rasters SET rast = ST_SetSRID(rast,26918)
  WHERE ST_SRID(rast) <> 26918;

SELECT AddRasterConstraints('public'::name, 'rasters'::name, 'rast'::name);
SELECT r_table_name AS t, r_raster_column AS c, srid,
  blocksize_x AS bx, blocksize_y AS by, scale_x AS sx, scale_y AS sy,
  ST_AsText(extent) AS e
  FROM raster_columns
WHERE r_table_name = 'rasters';
```

Ah progress:

```
t         |  c   | srid  | bx  | by  | sx | sy |  e
----------+------+-------+-----+-----+----+----+---------------------------
 ↪---------------
rasters   | rast | 26918 | 150 | 150 |    |    | POLYGON((0 -0.90000000000.
 ↪.
(1 row)
```

The more you can constrain all your rasters, the more columns you'll see filled in and also the more operations you'll be able to do across all the tiles in your raster. Keep in mind that in some cases, you may not want to apply all constraints.

For example, if you plan to load more data into your raster table, you'll want to skip the extent constraint since that would require that all rasters are within the extent of the extent constraint.

30.4.2 raster_overviews

Raster overview columns appear both in the *raster_columns* meta catalog and another meta catalog called *raster_overviews*. Overviews are used mostly to speed up viewing at higher zoom levels. They can also be used for quick back of the envelop analysis, providing less accurate stats, but at a much faster speed than applying to the raw raster table.

To inspect the overviews, run:

```
SELECT *
    FROM raster_overviews;
```

and you'll see the output:

```
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
----------------+----------------+--------------+-----------------+--------
↪--------+----------------+--------------+-----------------+------------
↪----
nyc             | public         | o_2_nyc_dem  | rast            | nyc    ␣
↪          | public         | nyc_dem      | rast            |        ␣
↪  2
nyc             | public         | o_3_nyc_dem  | rast            | nyc    ␣
↪          | public         | nyc_dem      | rast            |        ␣
↪  3
(2 rows)
```

The *raster_overviews* table only provides you the overview_factor and the name of the parent table. All this information is something you could have figured out yourself by the *raster2pgsql* naming convention for overviews.

The *overview_factor* tells you at what resolution the row is with respect to it's parent. An *overview_factor* of *2* means that 2x2 = 4 tiles can fit into one overview_2 tile. Similarly an overview_factor of *1* meants that 2x2x2 = 8 tiles of the original can be shoved into an overview_3 tile.

30.5 Common Raster Functions

The **postgis_raster** extension has over 100 functions to choose from. PostGIS raster functionality was patterned after the PostGIS geometry support. You'll find an overlap of functions between raster and geometry where it makes sense. Common ones you'll use that have equivalent in geometry world are **ST_Intersects**, **ST_SetSRID**, **ST_SRID**, **ST_Union**, **ST_Intersection**, and **ST_Transform**.

In addition to those overlapping functions, it supports the *&&* overlap operator between rasters and between a raster and geometry. It also offers many functions that work in conjunction with geometry or are very specific to rasters.

You need a function like **ST_Union** to reconstitute a region. Because performance gets slow, the more pixels a function needs to analyse, you need a fast acting function **ST_Clip** to clip the rasters to just the portions of interest for your analysis.

Finally you need **ST_Intersects** or **&&** to zoom in on the raster tiles that contain your areas of interest. The *&&* operator, is a faster process than the *ST_Intersects*. Both can take advantage of raster

spatial indexes. We'll cover these bread and butter functions first before moving on to other sections where we will use them in concert with other raster and geometry functions.

30.5.1 Unioning Rasters with ST_Union

The ST_Union function for raster, just as the geometry equivalent **ST_Union**, aggregates a set of rasters together into a single raster. However, just as with geometry, not all rasters can be combined together, but the rules for raster unioning are more complicated than geometry rules. In the case of geometries, all you need is to have the same spatial reference system, but for rasters that is not sufficient.

If you were to attempt, the following:

```
SELECT ST_Union(rast)
    FROM rasters;
```

You'd be summarily punished with an error:

ERROR: rt_raster_from_two_rasters: The two rasters provided do not have the same alignment SQL state: XX000

What is this same alignment thing, that is preventing you from unioning your precious rasters?

In order for rasters to be combined, they need to be on the same grid so to speak. Meaning they must have same pixel sizes, same orientation (the skew), same spatial reference system, and their pixels must not cut into each other, meaning they share the same worldly pixel grid.

If you try the same query, but just with words we carefully placed in New York.

Again, the same error. These are the same spatial ref system, the same pixel sizes, and yet it's still not good enough. Because their grids are off.

We can fix this by shifting the upper left y coordinates ever so slightly and then trying again. If our grids start at integer level since our pixel sizes are whole integer, then the pixels won't cut into each other.

```
UPDATE rasters SET rast = ST_SetUpperLeft(rast,
  ST_UpperLeftX(rast)::integer,
  ST_UpperLeftY(rast)::integer)
WHERE name LIKE '%New York';

SELECT ST_Union(rast ORDER BY name)
  FROM rasters
  WHERE name LIKE '%New York%';
```

Voila it worked, and if we were to view, we'd see something like this:

Note: If ever you are unclear why your rasters don't have the same alignment, you can use the function ST_SameAlignment, which will compare 2 rasters or a set of rasters and tell you if they have the same alignment. If you have notices enabled, the NOTICE will tell you what is off with the rasters in question.

The ST_NotSameAlignmentReason, instead of just a notice will output the reason. It however only works with two rasters at a time.

One major way in which the ST_Union(raster) raster function deviates from the ST_Union(geometry) geometry function is that it allows for an argument called *uniontype*. This argument by default is set to *LAST* if you don't specify it, which means, take the **LAST** raster pixel values in occasions where the raster pixel values overlap. As a general rule, pixels in a band that are marked as no-data are ignored.

Just as with most aggregates in PostgreSQL, you can put a **ORDER BY** clause as part of the function call as is done in the prior example. Specifying the order, allows you to control which raster takes priority. So in our prior example, *Raster* trumped *Hello* because *Raster* is alphabetically last.

Observe, if you switch the order:

```
SELECT ST_Union(rast ORDER BY name DESC)
  FROM rasters
  WHERE name LIKE '%New York%';
```

Then *Hello* trumps *Raster* because Hello is now the last overlaid.

The **FIRST** union type is the reverse of **LAST**.

But on occassion, **LAST** may not be the right operation. Let's suppose our rasters represented two different sets of observations from two different devices. These devices measure the same thing, and we aren't sure which is right when they cross paths, so we'd instead like to take the *MEAN* of the results. We'd do this:

```
SELECT ST_Union(rast, 'MEAN')
  FROM rasters
  WHERE name LIKE '%New York%';
```

Voila it worked, and if we were to view, we'd see something like this:

So instead of trumping, we have a blending of the two forces. In the case of **MEAN** union type, there is no point in specifying order, because the result would be the average of overlapping pixel values.

Note that for geometries since geometries are vector and thus have no values besides there or not there, there really isn't any ambiguity on how to combine two vectors when they intersect.

Another feature of the raster **ST_Union** we glossed over, is this idea of if you should return all bands or just some bands. When you don't specify what bands to union, **ST_Union** will combine same banded numbers and use the **LAST** unioning strategy. If you have multiple bands, this may not be what you

want to do. Perhaps you only want to union, the second band. In this case, the Green Band and you want the count of pixel values.

```
SELECT ST_BandPixelType(ST_Union(rast, 2, 'COUNT'))
  FROM rasters
  WHERE name LIKE '%New York%';
```

```
st_bandpixeltype
------------------
32BUI
(1 row)
```

Note in the case of the **COUNT** union type, which counts the number of pixels filled in and returns that value, the result is always a **32BUI** similar to how when you do a **COUNT** in sql, the result is always a bigint, to accommodate large counts.

In other cases, the band pixel type does not change and is set to the max value or rounded if the amounts exceed the bounds of the type. Why would anyone ever want to count pixels that intersect at a location. Well suppose each of your rasters represent police squadrons and incidents of arrests in the areas. Each value, might represent a different kind of arrest reason. You are doing stats on how many arrests in each region, therefore you only care about the count of arrests.

Or perhaps, you want to do all bands, but you want different strategies.

```
SELECT ST_Union(rast, ARRAY[(1, 'MAX'),
  (2, 'MEAN'),
  (3, 'RANGE')]::unionarg[])
  FROM rasters
  WHERE name LIKE '%New York%';
```

Using the *unionarg[]* variant of the **ST_Union** function, also allows you to shuffle the order of the bands.

30.5.2 Clipping Rasters with help of ST_Intersects

The ST_Clip function is one of the most widely used functions for PostGIS rasters. The main reason is the more pixels you need to inspect or do operations on, the slower your processing. **ST_Clip** clips your raster to just the area of interest, so you can isolate your operations to just that area.

This function is also special in that it utilizes the power of geometry to help raster analysis. To reduce the number of pixels, **ST_Union** has to handle, each raster is clipped first to the area we are interested in.

```
SELECT ST_Union( ST_Clip(r.rast, g.geom) )
  FROM rasters AS r
      INNER JOIN
        ST_Buffer(ST_Point(586598, 4504816, 26918), 100 ) AS g(geom)
          ON ST_Intersects(r.rast, g.geom)
  WHERE r.name LIKE '%New York%';
```

This example showcases several functions working in unison. The **ST_Intersects** function employed is the one packaged with **postgis_raster** and can intersect 2 rasters or a raster and a geometry. Similar to the geometry **ST_Intersects** the raster ST_Intersects can take advantage of spatial indexes on the raster or geometry tables.

30.6 Converting Rasters to Geometries

Rasters can just as easily be morphed into geometries.

30.6.1 The polygon of a raster with ST_Polygon

Lets start with our prior example, but convert it to a polygon using ST_Polygon function.

```
SELECT ST_Polygon(ST_Union( ST_Clip(r.rast, g.geom) ))
  FROM rasters AS r
      INNER JOIN
        ST_Buffer(ST_Point(586598, 4504816, 26918), 100 ) AS g(geom)
          ON ST_Intersects(r.rast, g.geom)
  WHERE r.name LIKE '%New York%';
```

If you click on the geometry viewer in pgAdmin, you can see this in all it's glory without any hacks.

```
SELECT ST_Polygon(ST_Union( ST_Clip(r.rast, g.geom) ))
    FROM rasters AS r
        INNER JOIN
          ST_Buffer(ST_Point(586598, 4504816, 26918), 100 ) AS g(geom)
            ON ST_Intersects(r.rast, g.geom)
    WHERE r.name LIKE '%New York%';
```

Output Messages Geometry Viewer × Notifications

ST_Polygon considers all the pixels that have values (not no-data) in a particular band, and converts them to geometry. Like many other functions in raster, **ST_Polygon** only considers 1 band. If no band is specified, it will consider only the first band.

30.6.2 The pixel rectangles of a raster with ST_PixelAsPolygons

Another popularly used function is the ST_PixelAsPolygons function. You should rarely use **ST_PixelAsPolygons** on a large raster without first clipping because you will end up with millions of rows, one for each pixel.

ST_PixelAsPolygons returns a table consisting of geom, val, x, and y. Where x is the column number, and y is the row number in the raster.

ST_PixelAsPolygons similar to other raster functions works on one band at a time and works on band 1 if no band is specified. It also by default returns only pixels that have values.

```
SELECT gv.*
  FROM rasters AS r
    CROSS JOIN LATERAL ST_PixelAsPolygons(rast) AS gv
  WHERE r.name LIKE '%New York%'
  LIMIT 10;
```

Which outputs:

	geom geometry	val double precision	x integer	y integer
1	01030000202669000001...	10	97	1
2	01030000202669000001...	10	98	1
3	01030000202669000001...	10	99	1
4	01030000202669000001...	10	100	1
5	01030000202669000001...	10	101	1
6	01030000202669000001...	10	102	1
7	01030000202669000001...	10	103	1
8	01030000202669000001...	10	104	1
9	01030000202669000001...	10	105	1
10	01030000202669000001...	10	106	1

and if we inspect using the geometry viewer, we'd see:

If we want all pixels of all our bands, we'd need to do something like below. Note the differences in this example from previous.

1. Setting **exclude_nodata_value** to make sure all pixels are returned so that our sets of calls return the same number of rows. The rows out of the function will be naturally in

the same order.

2. Using the PostgreSQL ROWS FROM constructor , and aliasing each set of columns from our function output with names. So for example the band 1 columns (geom, val, x, y) are renamed to g1, v1, x1, x2

```
SELECT pp.g1, pp.v1, pp.v2, pp.v3
  FROM rasters AS r
    CROSS JOIN LATERAL
    ROWS FROM (
      ST_PixelAsPolygons(rast, 1, exclude_nodata_value => false ),
      ST_PixelAsPolygons(rast, 2, exclude_nodata_value => false),
      ST_PixelAsPolygons(rast, 3, exclude_nodata_value => false )
      ) AS pp(g1, v1, x1, y1,
        g2, v2, x2, y2,
        g3, v3, x3, y3 )
  WHERE r.name LIKE '%New York%'
   AND ( pp.v1 = 0 OR  pp.v2 > 0 OR pp.v3 > 0) ;
```

Note: We used CROSS JOIN LATERAL in these examples because we wanted to be explicit what we are doing. Since these are all set returning functions, you can replace CROSS JOIN LATERAL with , for short-hand. We'll use a , in the next set of examples

30.6.3 Dumping polygons with ST_DumpAsPolygons

Raster also introduces an additional composite type called a **geomval**. Consider a **geomval** as the offspring of a geometry and raster. It contains a geometry and it contains a pixel value.

You will find several raster functions that return geomvals.

A commonly used function that outputs geomvals is ST_DumpAsPolygons, which returns a set of contiguous pixels with the same value as a polygon. Again this by default will only check band 1 and exclude no data values unless you override. This example selects only polygons from band 2. You can also apply filters to the values. For most use cases, **ST_DumpAsPolygons** is a better option than **ST_PixelAsPolygons** as it will return far fewer rows.

This will output 6 rows, and return polygons corresponding to the letters in "Raster".

```
SELECT gv.geom , gv.val
  FROM rasters AS r,
    ST_DumpAsPolygons(rast, 2) AS gv
  WHERE r.name LIKE '%New York%'
      AND gv.val = 100;
```

Note that it doesn't return a single geometry, because it finds continguous set of pixels with the same value that form a polygon. Even though all these values are the same, they are not continguous.

A common approach to produce more complex geometries is to group by the values and union.

```
SELECT ST_Union(gv.geom) AS geom , gv.val
  FROM rasters AS r,
    ST_DumpAsPolygons(rast, 2) AS gv
  WHERE r.name LIKE '%New York%'
  GROUP BY gv.val;
```

This will give you 2 rows back corresponding to the words “Raster” and “Hello”.

30.7 Statistics

The most important thing to understand about rasters is that they are statistical tools for storing data in arrays, that you may happen to be able to make look pretty on a screen.

You can find a menu of these statistical functions in Raster Band Statistics.

30.7.1 ST_SummaryStatsAgg and ST_SummaryStats

Want all stats for a set or rasters, reach for the function ST_SummaryStatsAgg.

This query takes about 10 seconds and gives you a summary of the whole table:

```
SELECT (ST_SummaryStatsAgg(rast, 1, true, 1)).* AS sa
    FROM o_3_nyc_dem;
```

Outputs:

```
count      |     sum     |       mean        |      stddev       | min | max
-----------+-------------+-------------------+-------------------+-----+-----
246794100  | 4555256024 | 18.4577184948911 | 39.4416860598687 |   0 | 411
(1 row)
```

Which tells we have a lot of pixels and our max elevation is 411 ft.

If you have built overviews, and just need a rough estimate of your mins, maxs, and means use one of your overviews. This next query returns roughly the same values for mins, maxs, and means as the prior but in about 1 second instead of 10.

```
SELECT (ST_SummaryStatsAgg(rast, 1, true, 1)).* AS sa
    FROM o_3_nyc_dem ;
```

Now armed with this bit of information, we can ask more questions.

30.7.2 ST_Histogram

Generally you won't want stats for your whole table, but instead just stats for a particular area, in that case, you'll want to also employ our old friends **ST_Intersects** and **ST_Clip**. If you are also in need of a raster statistics function that doesn't have an aggregate version, you'll want to carry **ST_Union** along for the ride.

For this next example we'll use a different stats function ST_Histogram which has no aggregate equivalent, and for this particular variant, is a set returning function. We are using the same area of interest as some prior examples, but we also need to employ geometry **ST_Transform** to transform our NY state plane meters geometry to our NYC State Plane feet rasters. It is almost always more performant to transform the geometry instead of raster and definitely if your geometry is just a single one.

```
SELECT (ST_Quantile( ST_Union( ST_Clip(r.rast, g.geom) ), ARRAY[0.25,0.50,
↪0.75, 1.0] )).*
    FROM nyc_dem AS r
        INNER JOIN
         ST_Transform(
           ST_Buffer(ST_Point(586598, 4504816, 26918), 100 ),
             2263) AS g(geom)
         ON  ST_Intersects(r.rast, g.geom);
```

the above query completes in under 60ms and outputs:

```
quantile  | value
----------+-------
    0.25  |    52
    0.5   |    57
    0.75  |    68
    1     |    78
(4 rows)
```

30.8 Creating Derivative Rasters

PostGIS raster comes packaged with a number of functions for editing rasters. These functions are both used for editing as well as creating derivative raster data sets. You will find these listed in Raster Editors and Raster Management.

30.8.1 Transforming rasters with ST_Transform

Most of our data is in NY State Plane meters (SRID: 26918), however our DEM raster dataset is in NY State Plane feet (SRID: 2263). For the least cumbersome workflow, we need our core datasets to be in the same spatial reference system.

The raster ST_Transform is the function most suited for this job.

In order to create a new nyc dem dataset in NY State Plane meters, we'll do the following:

```
CREATE TABLE nyc_dem_26918 AS
WITH ref AS (SELECT ST_Transform(rast,26918) AS rast
             FROM nyc_dem LIMIT 1)
SELECT r.rid, ST_Transform(r.rast, ref.rast) AS rast, r.filename
FROM nyc_dem AS r, ref;
```

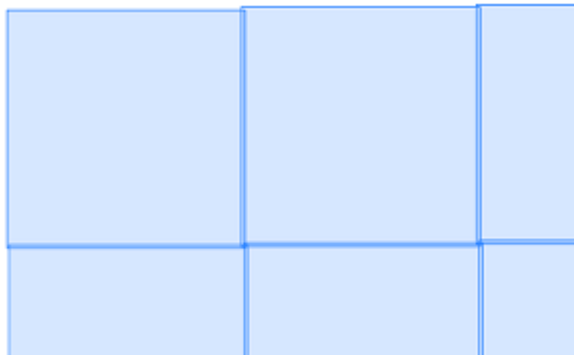
The above on my system took about 1.5 minutes. For a larger data set it would take much longer.

The aforementioned examples used two variants of the **ST_Transform** raster function. The first was to get a reference raster that will be used to transform the other raster tiles to guarantee that all tiles have the same alignment. Note the second variant of **ST_Transform** used doesn't even take an input SRID. This is because the SRID and all the pixel scale and block sizes are read from the reference raster. If you used *ST_Transform(rast, srid)* form, then all your rasters might come out with different alignment making it impossible to apply an operation such as **ST_Union** on them.

The only problem with the aforementioned **ST_Transform** approach is that when you transform, the transformed often exists in other tiles. If you looked at the above output closely enough by outputting the convex hull of the rasters, in the next example you'd see annoying overlaps around the borders.

```
SELECT rast::geometry
  FROM nyc_dem_26918
  ORDER BY rid
LIMIT 100;
```

viewed in pgAdmin would look something like:



30.8.2 Using ST_MakeEmptyCoverage to create even tiled rasters

A better approach, albeit a bit slower, is to define your own coverage tile structure from scratch using ST_MakeEmptyCoverage and then find the intersecting tiles for each new tile, and ST_Union these and then use *ST_Transform(ref, ST_Union...)* to create each tile.

For this we'll be using quite a few functions, we learned about earlier.

```
DROP TABLE IF EXISTS nyc_dem_26918;
CREATE TABLE nyc_dem_26918 AS
SELECT ROW_NUMBER() OVER(ORDER BY t.rast::geometry) AS rid,
  ST_Union(ST_Clip( ST_Transform( r.rast, t.rast, 'Bilinear' ), t.
↪rast::geometry ), 'MAX') AS rast
FROM (SELECT ST_Transform(
    ST_SetSRID(ST_Extent(rast::geometry),2263)
        , 26918) AS geom
      FROM nyc_dem
    ) AS g, ST_MakeEmptyCoverage(tilewidth => 256, tileheight => 256,
                  width => (ST_XMax(g.geom) - ST_XMin(g.geom))::integer,
                  height => (ST_YMax(g.geom) - ST_YMin(g.geom))::integer,
                  upperleftx => ST_XMin(g.geom),
                  upperlefty => ST_YMax(g.geom),
                  scalex =>  3.048,
                  scaley => -3.048,
```

(continues on next page)

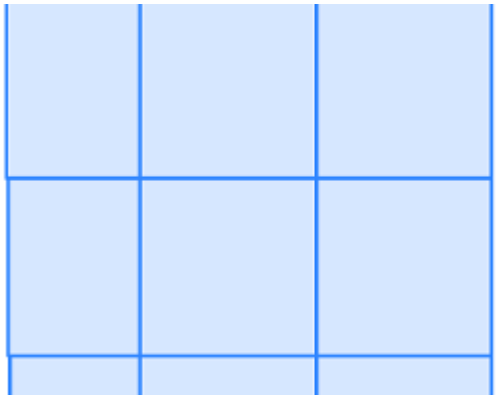
(continued from previous page)

```
                    skewx => 0., skewy => 0., srid => 26918) AS t(rast)
            INNER JOIN nyc_dem AS r
              ON ST_Transform(t.rast::geometry, 2263) && r.rast
GROUP BY t.rast;
```

Repeating the same exercise as earlier:

```
SELECT rast::geometry
  FROM nyc_dem_26918
  ORDER BY rid
LIMIT 100;
```

viewed in pgAdmin we no longer have overlaps:



On my system this took ~10 minutes and returned 3879 rows. After the creation of the table, we'll want to do the usual of adding a spatial index, primary key, and constraints as follows:

```
ALTER TABLE nyc_dem_26918
  ADD CONSTRAINT pk_nyc_dem_26918 PRIMARY KEY(rid);

CREATE INDEX ix_nyc_dem_26918_st_convexhull_gist
    ON nyc_dem_26918 USING gist( ST_ConvexHull(rast) );

SELECT AddRasterConstraints('nyc_dem_26918'::name, 'rast'::name);
ANALYZE nyc_dem_26918;
```

Which should take under 2 minutes for this dataset.

30.8.3 Creating overview tables with ST_CreateOverview

Just as with our original dataset, it would be useful to have overview tables to speed up performance of some operations. ST_CreateOverview is a function fit for that purpose. You can use **ST_CreateOverview** to create overviews also if you neglected to create one during the raster2pgsql load or you decided, you need more overviews.

We'll create level 2 and 3 overviews as we had done with our original using this code.

```
SELECT ST_CreateOverview('nyc_dem_26918'::regclass, 'rast', 2);
SELECT ST_CreateOverview('nyc_dem_26918'::regclass, 'rast', 3);
```

This process sadly takes a while, and a longer while the more rows you have so be patient. For this dataset it took about 3-5 minutes for the overview factor *2* and 1 minute for the overview factor *3*.

The **ST_CreateOverView** function also adds in the needed constraints so the columns appear with full detail in the *raster_columns* and *raster_overviews* catalogs. It does not add indexes to them though and also does not add an rid column. The rid column is probably not necessary unless you need a primary key to edit with. You would probably want an index which you can create with the following:

```
CREATE INDEX ix_o_2_nyc_dem_26918_st_convexhull_gist
    ON o_2_nyc_dem_26918 USING gist( ST_ConvexHull(rast) );

CREATE INDEX ix_o_3_nyc_dem_26918_st_convexhull_gist
    ON o_3_nyc_dem_26918 USING gist( ST_ConvexHull(rast) );
```

Note: ST_CreateOverview has an optional argument for denoting the sampling method. If not specified it uses the default *NearestNeighbor* which is generally the fastest to compute but may not be ideal. Resampling methods is beyond the scope of this workshop.

30.9 The intersection of rasters and geometries

There are a couple of functions commonly used to compute intersections of rasters and geometries. We've already seen **ST_Clip** in action which returns the intersection of a raster and geometry as a raster, but there are others. For point data, the most commonly used is **ST_Value** and then there is the **ST_Intersection** which has several overloads some returning rasters and some returning a set of *geomval*.

30.9.1 Pixel values at a geometric Point

If you need to return values from rasters based on intersection of several ad-hoc geometry points, you'll use ST_Value or it's nearest relative ST_NearestValue.

```
SELECT g.geom, ST_Value(r.rast, g.geom) AS elev
  FROM nyc_dem_26918 AS r
    INNER JOIN
    (SELECT id, geom
      FROM nyc_homicides
      WHERE weapon = 'gun') AS g
      ON r.rast && g.geom;
```

This example takes about 1 second to return 2444 rows. If you used **ST_Intersects** instead of **&&**, the process would take about 3 seconds. The reason *ST_Intersects* is slower is that it performs an additional recheck in some cases a pixel by pixel check. If you expect all your points to be represented with data in your raster set and your rasters represent a coverage (a continguous set non-overlapping raster tiles), then *&&* is generally a speedier option.

If your raster data is not densely populated or you have overlapping rasters (e.g. they represent different observations in time), or they are skewed (not axis aligned) then there is an advantage to having ST_Intersects weed out the false positives.

30.9.2 ST_Intersection raster style

Just as you can compute the intersection of two geometries using **ST_Intersection**, you can compute intersection of two rasters or a raster and a geometry using raster ST_Intersection.

What you get out of this beast, are two different kinds of things:

- Intersect a geometry with a raster, and you get a set of *geomval* offspring. Perhaps one, but most often many.
- Intersect 2 rasters and you get a single *raster* back.

The golden rule for both raster intersection and geometry intersection is that both parties involved must have the same spatial reference system. For raster/raster, they also have to have same alignment.

Here is an example which answers a question you may have been curious about. If we bucket our elevations into 5 buckets of elevation values, which elevation range results in the most gun fatalities? We know based on our earlier summary statistics that *0* is the lowest value and *411* is the highest value for elevation in our nyc dem dataset, so we use that as min and max value for our width_bucket call.

```
SELECT ST_Transform(ST_Union(gv.geom),4326) AS geom ,
  MIN(gv.val) AS min_elev, MAX(gv.val) AS max_elev,
    count(g.id) AS count_guns
  FROM nyc_dem_26918 AS r
    INNER JOIN nyc_homicides AS g
      ON ST_Intersects(r.rast, g.geom)
    CROSS JOIN
     ST_Intersection( g.geom,
      ST_Clip(r.rast,ST_Expand(g.geom, 4) )
      ) AS gv
  WHERE g.weapon = 'gun'
  GROUP BY width_bucket(gv.val, 0, 411, 5)
  ORDER BY width_bucket(gv.val, 0, 411, 5);
```

Is there an important correlation between gun homicides and elevation? Probably not.

Let's take a look at raster / raster intersection:

```
SELECT ST_Intersection(r1.rast, 1, r2.rast, 1, 'BAND1')
  FROM nyc_dem_26918 AS r1
    INNER JOIN
        rasters AS r2 ON ST_Intersects(r1.rast,1, r2.rast, 1);
```

What we get are two rows with NULLLs, and if you have your PostgreSQL set to show notices, you'll see:

NOTICE: The two rasters provided do not have the same alignment. Returning NULL

In order to fix this, we can align one to the other as it's coming out of the gate using ST_Resample.

```
SELECT ST_Intersection(r1.rast, 1,
  ST_Resample( r2.rast, r1.rast ), 1,
    'BAND1')
  FROM nyc_dem_26918 AS r1
    INNER JOIN
        rasters AS r2 ON ST_Intersects(r1.rast,1, r2.rast, 1);
```

Let's also roll it up into a single stats record

```
SELECT (
  ST_SummaryStatsAgg(
    ST_Intersection(r1.rast, 1,
      ST_Resample( r2.rast, r1.rast ), 1, 'BAND1'),
        1, true)
    ).*
  FROM nyc_dem_26918 AS r1
    INNER JOIN
        rasters AS r2 ON ST_Intersects(r1.rast,1, r2.rast, 1);
```

which outputs:

```
count  |  sum   |       mean        |      stddev       | min | max
-------+--------+-------------------+-------------------+-----+-----
  2075 | 99536 | 47.969156626506 | 9.57974836865737 |  33 |  62
(1 row)
```

30.10 Map Algebra Functions

Map algebra is the idea that you can do math on your pixel values. The **ST_Union** and **ST_Intersection** functions covered earlier are a special fast case of map algebra. Then there are the ST_MapAlgebra family of functions which allow you to define your own crazy math, but at cost of performance.

People have the habit of jumping to **ST_MapAlgebra**, probably cause the name sounds so cool and sophisticated. Who wouldn't want to tell their friends, "I'm using a function called ST_MapAlgebra." My advice, explore other functions before you jump for that shot-gun. Your life will be simpler and your performance will be 100 times better, and your code will be shorter.

Before we showcase *ST_MapAlgebra*, we'll explore other functions that fit under the *Map Algebra* family of functions and generally have better performance than *ST_MapAlgebra*.

30.10.1 Reclassify your raster using ST_Reclass

An often overlooked map-algebraish function is the ST_Reclass function, who sits in the background waiting for someone to discover the power and speed it can offer.

What does **ST_Reclass** do? It as the name implies, reclassifies your pixel values based on minimalist range algebra.

Lets revisit our NYC Dems. Perhaps we only care about classifying our elevations as 1) low, 2) medium, 3) high , and 4) really high. We don't need 411 values, we just need 4. With that said lets do some reclassifying.

The classification scheme is governed by the reclass expression.

```
WITH r AS ( SELECT ST_Union(newrast) As rast
  FROM nyc_dem_26918 AS r
        INNER JOIN ST_Buffer(ST_Point(586598, 4504816, 26918), 1000 ) AS␣
↪g(geom)
          ON ST_Intersects( r.rast, g.geom )
        CROSS JOIN ST_Reclass( ST_Clip(r.rast,g.geom), 1,
          '[0-10):1, [10-50):2, [50-100):3,[100-:4','4BUI',0) AS newrast
```

(continues on next page)

(continued from previous page)

```
        )
SELECT SUM(ST_Area(gv.geom)::numeric(10,2)) AS area, gv.val
    FROM r, ST_DumpAsPolygons(rast) AS gv
    GROUP BY gv.val
    ORDER BY gv.val;
```

Which would output:

```
  area      | val
------------+-----
    6754.04 |   1
1753117.51  |   2
1355232.37  |   3
    1848.75 |   4
(4 rows)
```

If this were a classification scheme we preferred, we could create a new table using the ST_Reclass to recompute each tile.

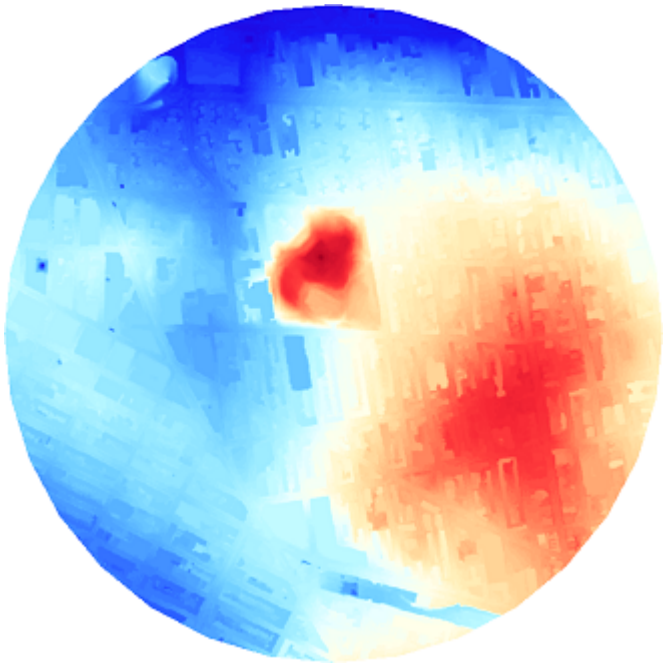
30.10.2 Coloring your rasters with ST_ColorMap

The ST_ColorMap function is another mapalgebraish function that reclassifies your pixel values. Except it is band creating. It converts a single band raster such as our Dems into a visually presentable 3 or 4 banded raster.

You could use one of the built-in colormaps as below if you don't want to fuss with creating one.

```
SELECT ST_ColorMap( ST_Union(newrast), 'bluered') As rast
   FROM nyc_dem_26918 AS r
        INNER JOIN
          ST_Buffer(
            ST_Point(586598, 4504816, 26918), 1000
            ) AS g(geom)
        ON ST_Intersects( r.rast, g.geom)
         CROSS JOIN ST_Clip(rast, g.geom) AS newrast;
```

Which looks like:



The bluer the color the lower the elevation and the redder the color the higher the elevation.

CHAPTER

THIRTYONE

TOPOLOGY

PostGIS supports the SQL/MM SQL-MM 3 Topo-Geo and Topo-Net 3 specifications via an extension called **postgis_topology**. You can learn about all the functions and types provided by this extension in Manual: PostGIS Topology. The *postgis_topology* extension includes another kind of core spatial type, called a **topogeometry**. In addition to the *topogeometry* spatial type, you will find functions for building *topologies* and populating topologies.

Before you can start using topologies, you must install postgis_topology extension as follows:

```
CREATE EXTENSION postgis_topology;
```

After you install the extension, you will see a new schema in your database called *topology*. The *topology* schema catalogs all the topologies in your database.

The *topology* schema contains two tables and all the helper functions for topology.

- topology - lists all the topologies in your database, and what schema they are stored in
- layer - lists all the table columns in your database that hold topogeometries

The *layer* table is very similar to the *raster_columns*, *geometry_columns*, and *geography_columns* catalogs we learned about earlier, but specifically for topogeometries.

31.1 Creating topologies

What exactly is a topology and a topogeometry, and how are they related? Before we explain, let's start by creating a topology to house our NYC topologically perfect data using the CreateTopology function and set the tolerance to be 0.5 meters. Note the 0.5 is in meters since our spatial reference system is State Plany NY meters.

```
SELECT topology.CreateTopology('nyc_topo', 26918, 0.5);
```

Which outputs:

```
1
```

Which is the id it assigned the new topology. Once you run the above command, you will see a new schema in your database called *nyc_topo*. You can name the topology anything you want. My convention is to add *_topo* at the end to distinguish it from other schemas I have in my database.

If you explore the *topology.topology* table,

```
SELECT * FROM topology.topology;
```

You will see:

```
id |   name    | srid  | precision | hasz
---+-----------+-------+-----------+------
  1 | nyc_topo | 26918 |         0 | f
(1 row)
```

31.2 Storage of topologies and topogeometries

A topology is implemented as a schema in a PostgreSQL database. If you explore the *nyc_topo* schema, you will see these tables and views:

- **edge - This is a view built against edge_data, mostly for SQL/MM compliance.** It has a subset of the columns in the *edge_data* table.
- edge_data - Contains all the linestrings that make up the topology
- **face - Contains a list of all closed surfaces that can be formed from the edge_data.** It does not contain the actual geometry, but instead just the bounding box of the geometry.
- node - Contains all the start and end points of all edges as well as points not connected to anything (isolated nodes)
- relation - this defines what elements in a topology make up a topogeometry.

So what is a topogeometry? A topogeometry is a representation of a geometry formed from the edges, faces, nodes, and other topogeometries in a topology.

Where does a topogeometry reside? It resides somewhere else which references elements of a topology via the *relation* table. Although we could throw the topogeometries in our *nyc_topo* schema, the general convention, is to define other tables in other schemas that have a topogeometry, and also have any other kind of data you might be interested in tracking.

31.3 Why use topogeometries?

Using topogeometries keeps your data tidy and connected. Topogeometries are very useful for Cadastral work, where you want to make sure two parcels of land don't overlap each other even if you change the boundaries of one or you want to make sure roads stay connected as you change the geometries that form them. Geometries live in an island of their own, you can duplicate them, morph them. Geometries are carefree, not caring about other geometries that share space with them. Topogeometries, in contrast, follow the rules of their topology; they can't exist unless there is an edge, node, face, or other topogeometry that defines them. A topogeometry belongs to one and only one topology. A topogeometry is a relational model of a geometry and as such as each component (edges/faces/nodes) are moved, added etc, they change not one topogeometry shape, but all topogeometries that have components in common.

We have an *nyc_topo* topology devoid of any data. Let's populate it with our NYC data. Topology edges, faces, and nodes can be created in 2 keys ways.

- Edges, Faces, and Nodes can be created directly using topology primitive functions.

- Edges, Faces, and Nodes can be formed by creating topogeometries. When a topogeometry is created from a geometry and their are missing edges, faces, or nodes that match its coordinates, then new edges, faces, and nodes are created as part of the process.

31.4 Defining topogeometry columns and creating topogeometries

The most common way to populate topologies is to create topogeometries. Lets start by creating a table to hold neighborhoods and then add a topogeometry column using the AddTopoGeometryColumn function.

```
CREATE TABLE nyc_neighborhoods_t(boroname varchar(43), name varchar(67),
  CONSTRAINT pk_nyc_neighborhoods_t PRIMARY KEY(boroname,name) );
SELECT topology.AddTopoGeometryColumn('nyc_topo', 'public', 'nyc_
↪neighborhoods_t',
  'topo', 'POLYGON') As  layer_id;
```

The output of the above is:

```
layer_id
--------
1
```

Now we are ready to populate our table. It's best to ensure your geometries are valid before adding, otherwise you'll get errors such as SQLMM geometry is not simple.

So lets start by adding valid ones. The 1 used here refers to the layer_id generated from the previous query. If you don't know the layer id, you would look it up using the FindLayer function which we'll use in later examples.

For these examples you'll use the function toTopoGeom function to convert a geometry to it's topogeometry equivalent. toTopoGeom function handles a lot of book-keeping for you.

The *toTopoGeom* function inspects the geometry passed in and injects nodes, edges, and faces as needed into your topology to form the shape of the geometry. It will then add relationships to the *relation* table that defines how this new topogeometry is related to these new and existing topology elements. In some cases pieces of the geometry may exist or existing pieces need to be split to form the new geometry.

```
INSERT INTO nyc_neighborhoods_t(boroname,name, topo)
SELECT boroname, name,  topology.toTopoGeom(geom, 'nyc_topo', 1)
  FROM nyc_neighborhoods
  WHERE ST_ISvalid(geom);
```

The above step should take 3-4 secs. Now lets add the invalid ones:

```
INSERT INTO nyc_neighborhoods_t(boroname,name, topo)
SELECT boroname, name,  topology.toTopoGeom(
  ST_UnaryUnion(
    ST_CollectionExtract(
      ST_MakeValid(geom), 3)
      ), 'nyc_topo', 1)
  FROM nyc_neighborhoods
  WHERE NOT ST_ISvalid(geom);
```

The above should take about 300-400ms.

Now we have data in our topology. A quick check will show, nyc_topo.edge, nyc_topo.node, and nyc_topo.face have data:

```
SELECT 'edge' AS name, count(*)
  FROM nyc_topo.edge
UNION ALL
SELECT 'node' AS name, count(*)
  FROM nyc_topo.node
UNION ALL
SELECT 'face' AS name, count(*)
  FROM nyc_topo.face;
```

outputs:

```
name | count
------+-------
edge |   580
node |   396
face |   218
(3 rows)
```

Now we can express declaritively that boros are formed from a collection of neighborhoods by defining a column called *topo* in *nyc_boros_t* table that is of type POLYGON and is a collection of other topogeometries from *nyc_neighborhoods_t.topo* column.

```
CREATE TABLE nyc_boros_t(boroname varchar(43),
  CONSTRAINT pk_nyc_boros_t PRIMARY KEY(boroname) );
SELECT topology.AddTopoGeometryColumn('nyc_topo', 'public', 'nyc_boros_t',
  'topo', 'POLYGON',
    (topology.FindLayer('public', 'nyc_neighborhoods_t', 'topo')).layer_id
        ) AS  layer_id;
```

Which outputs:

In order to populate this new table, we'll use the CreateTopoGeom function. Instead of starting with geometries to form a new topogeometry, the CreateTopoGeom starts with existing topology elements which may be primitives or other topogeometries to define a new topogeometry.

```
INSERT INTO nyc_boros_t(boroname, topo)
SELECT n.boroname,
  topology.CreateTopoGeom('nyc_topo',
  3,  (topology.FindLayer('public', 'nyc_boros_t', 'topo')).layer_id ,
    topology.TopoElementArray_Agg( ARRAY[ (n.topo).id, (n.topo).layer_id␣
↪]::topoelement ) )
  FROM nyc_neighborhoods_t AS n
GROUP BY n.boroname;
```

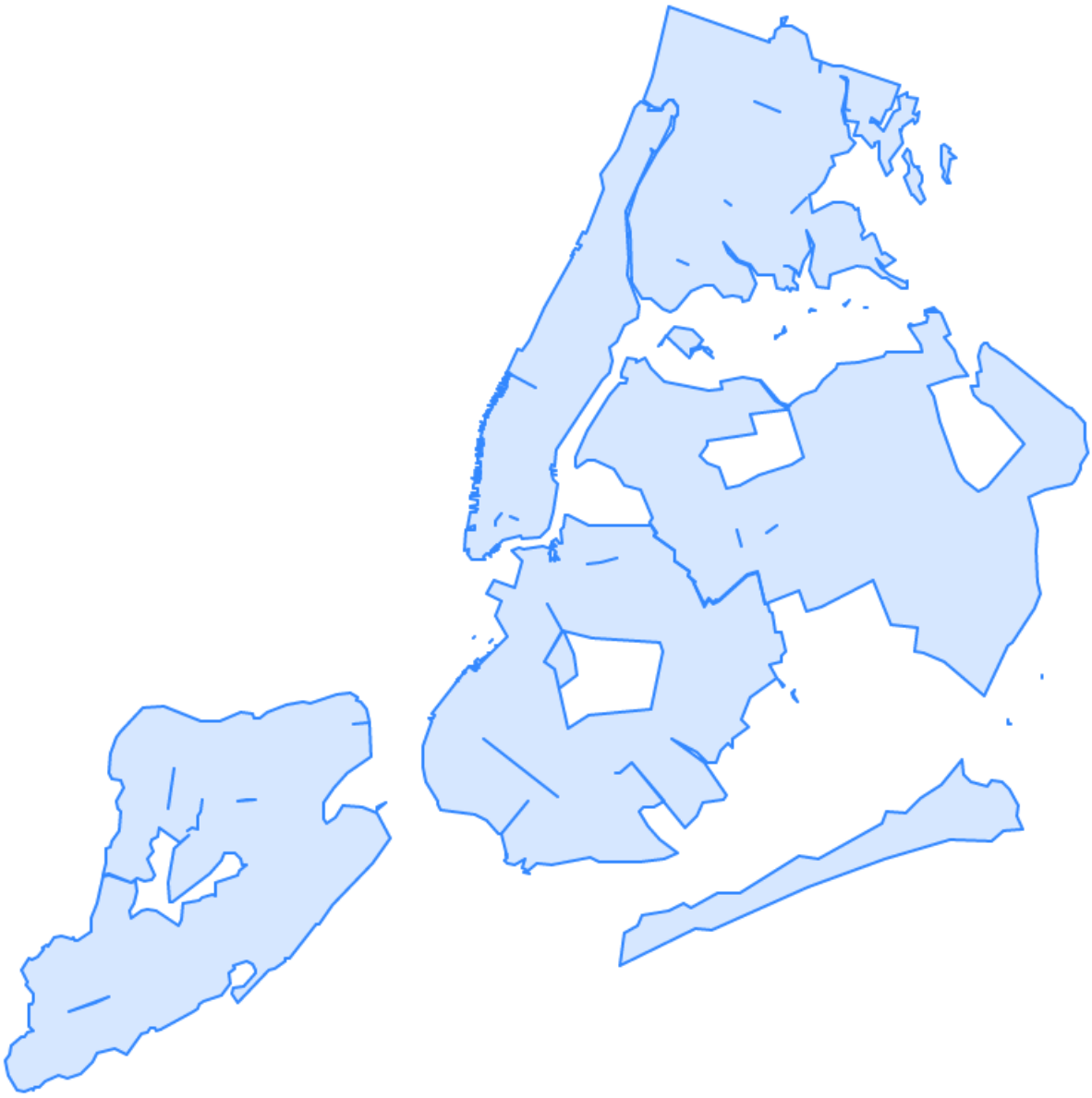
Which will insert 5 records corresponding to the boroughs of New York.

Note: If you are using PostGIS 3.4 or higher, you can use the new cast to cast a topogeometry to a topoelement, and replace *topology.TopoElementArray_Agg(ARRAY[(n.topo).id, (n.topo).layer_id]::topoelement))* in the above example with the shorter *topology.TopoElementArray_Agg(n.topo::topoelement)*

To view these in pgAdmin, you can cast the topogeometry to a geometry as follows:

```
SELECT boroname, topo::geometry AS geom
 FROM nyc_boros_t;
```

The output will look like below:



If you are thinking, what a total mess, yes it is a total mess. This is what happens after numerous cycles of simplification and other geometry processes where each geometry is treated as a separate unit. You get gaps, you get dangling islands, and neighborhoods encroaching on each other's territory.

Luckily we can use topology to clean up this mess and to help us maintain good clean connected data.

Let's put our land surveyor hat on and ask the question, if we are dividing our plots of land into districts (boros or neighborhoods) such that each district may border other districts but should not share any area in common, does it make sense for districts to have areas in common? No it does not make sense. And here we are with our data pointing out some areas belong to more than one neighborhood or more than one borough.

Lets first look at boros and look for neighborhoods that share elements in common:

```
SELECT te, array_agg(DISTINCT b.boroname)
 FROM nyc_boros_t AS b, topology.GetTopoGeomelements(topo) AS te
 GROUP BY te
 HAVING count(DISTINCT b.boroname) > 1;
```

The output is:

```
  te    |     array_agg
--------+-------------------
{44,3}  | {Brooklyn,Queens}
{51,3}  | {Brooklyn,Queens}
{76,3}  | {Brooklyn,Queens}
{114,3} | {Brooklyn,Queens}
{117,3} | {Brooklyn,Queens}
(5 rows)
```

Which tells us that Queens and Brooklyn are in the middle of border wars. In this query we use the GetTopoGeomElements function to declaritively inspect what components are shared across boroughs.

What is returned are a set of topolements. A topoelement is represented as an array of 2 integers with the first number being the id of the element, and the second, being the layer (or primitive type) of the element. PostGIS GetTopoElements returns the primitives of a topogeometry with types number 1-3 corresponding to (1: nodes, 2: edges, and 3: faces). All the topoelements for neighborhoods and boroughs are type 3, which corresponds to a topological face. We can use the ST_GetFaceGeometry to get a visual representation of these shared faces as folows:

```
SELECT te, t.geom, ST_Area(t.geom) AS area, array_agg(DISTINCT d.boroname)␣
↪AS shared_boros
FROM nyc_boros_t AS d, topology.GetTopoGeomelements(topo) AS te
  , topology.ST_GetFaceGeometry('nyc_topo',te[1]) AS t(geom)
GROUP BY te, t.geom
HAVING count(DISTINCT d.boroname) > 1
ORDER BY area;
```

The result will be 5 rows corresponding to border disputes between Queens and Brooklyn.

If we look at our neighborhoods, we'll see a similar story but with 44 border disputes:

```
SELECT te, t.geom, ST_Area(t.geom) AS area, array_agg(DISTINCT d.name) AS␣
↪shared_d
FROM nyc_neighborhoods_t AS d, topology.GetTopoGeomelements(d.topo) AS te
  , topology.ST_GetFaceGeometry('nyc_topo',te[1]) AS t(geom)
GROUP BY te, t.geom
HAVING count(DISTINCT d.name) > 1
ORDER BY area;
```

Because boroughs are an aggregation of neighborhoods, we can fix the borough issue by fixing the neighborhood border disputes.

There are a number of ways we could fix this. We could go out surveying asking people what neighbhood do they think they are standing in. Alternatively we could just assign slivers of land to the neighborhood with the least amount of area or to the highest bidder.

Removing elements from Topogeometries is handled using the TopoGeom_remElement function. So lets get on with it, removing duplicaed elements from neighborhoods with the most amount of area as follows:

```
WITH to_remove AS (SELECT te, MAX( ST_Area(d.topo::geometry) ) AS max_area,
↪ array_agg(DISTINCT d.name) AS shared_d
  FROM nyc_neighborhoods_t AS d, topology.GetTopoGeomelements(d.topo) AS te
    , topology.ST_GetFaceGeometry('nyc_topo',te[1]) AS t(geom)
  GROUP BY te
  HAVING count(DISTINCT d.name) > 1)
  UPDATE nyc_neighborhoods_t AS d SET topo = TopoGeom_remElement(topo, te)
  FROM to_remove
  WHERE d.name = ANY(to_remove.shared_d)
    AND ST_Area(d.topo::geometry) = to_remove.max_area;
```

The result of the above is 29 neighborhoods were updated. If you rerun the border dispute queries for neighborhoods and boroughs, you'll find you have no more border disputes.

We do still have gaps of empty space between neighborhoods caused by intensive simplication. Such issues can be fixed by directly editing the topology using the Topology Editor family of functions and/or filling in the holes and assigning those to neighborhoods.

CHAPTER

THIRTYTWO

TRACKING EDIT HISTORY USING TRIGGERS

A common requirement for production databases is the ability to track history: how has the data changed between two dates, who made the changes, and where did they occur? Some GIS systems track changes by including change management in the client interface, but that adds a lot of **complexity** to editing tools.

Using the database and the trigger system, it's possible to add history tracking to any table, while maintaining simple "direct edit" access to the primary table.

History tracking works by keeping a history table that records, for every edit:

- If a record was created, when it was added and by whom.
- If a record was deleted, when it was deleted and by whom.
- If a record was updated, adding a deletion record (for the old state) and a creation record (for the new state).

32.1 Using TSTZRANGE

The history table uses a PostgreSQL-specific feature–the "timestamp range" type–to store the time range that a history record was the "live" record. All the timestamp ranges in the history table for a particular feature can be expected to be non-overlapping but adjacent.

The range for a new record will start at `now()` and have an open end point, so that the range covers all time from the current time into the future.

```
SELECT tstzrange(current_timestamp, NULL);
```

```
               tstzrange
-------------------------------------
 ["2021-06-01 14:49:40.910074-07",)
```

Similarly, the time range for a deleted record will be updated to include the current time as the end point of the time range.

Searching time ranges is much simpler than searching a pair of timestamps, because of the way an open time range encompasses all time from the start point to infinity. The "contains" operator `@>` for ranges is the one we will use.

```
-- Does the range of "ten minutes ago to the future" include now?
-- It should! :)
```

(continues on next page)

(continued from previous page)

```
--
SELECT tstzrange(current_timestamp - '10m'::interval, NULL) @> current_
↪timestamp;
```

Ranges can be very efficiently indexed using a GIST index, just like spatial data, as we will show below. This makes history queries very efficient.

32.2 Building the History Table

Using this information it is possible to reconstruct the state of the edit table at any point in time. In this example, we will add history tracking to our **nyc_streets** table.

- First, add a new **nyc_streets_history** table. This is the table we will use to store all the historical edit information. In addition to all the fields from **nyc_streets**, we add five more fields.
  - **hid** the primary key for the history table
  - **created_by** the database user that caused the record to be created
  - **deleted_by** the database user that caused the record to be marked as deleted
  - **valid_range** the time range within which the record was "live"

  Note that we don't actually delete any records in the history table, we just mark the time they ceased to be part of the current state of the edit table.

```
DROP TABLE IF EXISTS nyc_streets_history;
CREATE TABLE nyc_streets_history (
  hid SERIAL PRIMARY KEY,
  gid INTEGER,
  id FLOAT8,
  name VARCHAR(200),
  oneway VARCHAR(10),
  type VARCHAR(50),
  geom GEOMETRY(MultiLinestring,26918),
  valid_range TSTZRANGE,
  created_by VARCHAR(32),
  deleted_by VARCHAR(32)
);

CREATE INDEX nyc_streets_history_geom_x
  ON nyc_streets_history USING GIST (geom);

CREATE INDEX nyc_streets_history_tstz_x
  ON nyc_streets_history USING GIST (valid_range);
```

- Next, we import the current state of the active table, **nyc_streets** into the history table, so we have a starting point to trace history from. Note that we fill in the creation time and creation user, but leave the end of the time range and the deleted by information NULL.

```
INSERT INTO nyc_streets_history
  (gid, id, name, oneway, type, geom, valid_range, created_by)
   SELECT gid, id, name, oneway, type, geom,
     tstzrange(now(), NULL),
```

(continues on next page)

(continued from previous page)

```
    current_user
  FROM nyc_streets;
```

- Now we need three triggers on the active table, for INSERT, DELETE and UPDATE actions. First we create the trigger functions, then bind them to the table as triggers.

  For an insert, we just add a new record into the history table with the creation time/user.

```
CREATE OR REPLACE FUNCTION nyc_streets_insert() RETURNS trigger AS
  $$
    BEGIN
      INSERT INTO nyc_streets_history
        (gid, id, name, oneway, type, geom, valid_range, created_by)
      VALUES
        (NEW.gid, NEW.id, NEW.name, NEW.oneway, NEW.type, NEW.geom,
         tstzrange(current_timestamp, NULL), current_user);
      RETURN NEW;
    END;
  $$
  LANGUAGE plpgsql;

CREATE TRIGGER nyc_streets_insert_trigger
AFTER INSERT ON nyc_streets
  FOR EACH ROW EXECUTE PROCEDURE nyc_streets_insert();
```

  For a deletion, we just mark the currently active history record (the one with a NULL deletion time) as deleted.

```
CREATE OR REPLACE FUNCTION nyc_streets_delete() RETURNS trigger AS
  $$
    BEGIN
      UPDATE nyc_streets_history
        SET valid_range = tstzrange(lower(valid_range), current_
↪timestamp),
            deleted_by = current_user
        WHERE valid_range @> current_timestamp AND gid = OLD.gid;
      RETURN NULL;
    END;
  $$
  LANGUAGE plpgsql;


CREATE TRIGGER nyc_streets_delete_trigger
AFTER DELETE ON nyc_streets
  FOR EACH ROW EXECUTE PROCEDURE nyc_streets_delete();
```

  For an update, we first mark the active history record as deleted, then insert a new record for the updated state.

```
CREATE OR REPLACE FUNCTION nyc_streets_update() RETURNS trigger AS
$$
  BEGIN

    UPDATE nyc_streets_history
      SET valid_range = tstzrange(lower(valid_range), current_
↪timestamp),
```

(continues on next page)

(continued from previous page)

```
          deleted_by = current_user
      WHERE valid_range @> current_timestamp AND gid = OLD.gid;

    INSERT INTO nyc_streets_history
        (gid, id, name, oneway, type, geom, valid_range, created_by)
      VALUES
        (NEW.gid, NEW.id, NEW.name, NEW.oneway, NEW.type, NEW.geom,
         tstzrange(current_timestamp, NULL), current_user);

    RETURN NEW;

  END;
$$
LANGUAGE plpgsql;

CREATE TRIGGER nyc_streets_update_trigger
AFTER UPDATE ON nyc_streets
  FOR EACH ROW EXECUTE PROCEDURE nyc_streets_update();
```

32.3 Editing the Table

Now that the history table is enabled, we can make edits on the main table and watch the log entries appear in the history table.

Note the power of this database-backed approach to history: **no matter what tool is used to make the edits, whether the SQL command line, a web-based JDBC tool, or a desktop tool like QGIS, the history is consistently tracked.**

32.3.1 SQL Edits

Let's turn the two streets named "Cumberland Walk" to the more stylish "Cumberland Wynde":

Updating the two streets will cause the original streets to be marked as deleted in the history table, with a deletion time of now, and two new streets with the new name added, with an addition time of now. You can inspect the historical records:

32.4 Querying the History Table

Now that we have a history table, what use is it? It's useful for time travel! To travel to a particular time **T**, you need to construct a query that includes:

- All records created before T, and not yet deleted; and also
- All records created before T, but deleted **after** T.

We can use this logic to create a query, or a view, of the state of the data in the past. Since presumably all your test edits have happened in the past couple minutes, let's create a view of the history table that shows the state of the table 10 minutes ago, **before you started editing** (so, the original data).

```
-- Records with a valid range that includes 10 minutes ago
-- are the ones valid at that moment.

CREATE OR REPLACE VIEW nyc_streets_ten_min_ago AS
  SELECT * FROM nyc_streets_history
    WHERE valid_range @> (now() - '10min'::interval)
```

We can also create views that show just what a particular user has added, for example:

```
CREATE OR REPLACE VIEW nyc_streets_postgres AS
  SELECT * FROM nyc_streets_history
    WHERE created_by = 'postgres';
```

32.5 See Also

- QGIS open source GIS
- PostgreSQL Triggers
- PostgreSQL Range Types

CHAPTER

THIRTYTHREE

BASIC POSTGRESQL TUNING

PostgreSQL is a very versatile database system, capable of running efficiently in very low-resource environments and environments shared with a variety of other applications. In order to ensure it will run properly for many different environments, the default configuration is very conservative and not terribly appropriate for a high-performance production database. Add the fact that geospatial databases have different usage patterns, and the data tend to consist of fewer, much larger records than non-geospatial databases, and you can see that the default configuration will not be totally appropriate for our purposes.

All of these configuration parameters can edited in the *postgresql.conf* database configuration file. This is a regular text file and can be edited using any text editor. The changes will not take effect until the server is restarted.

This section describes some of the configuration parameters that can be adjusted for a more production-ready geospatial database.

Note: These values are recommendations only; each environment will differ and testing is required to determine the optimal configuration. But this section should get you off to a good start.

33.1 shared_buffers

Sets the amount of memory the database server uses for shared memory buffers. These are shared amongst the back-end processes, as the name suggests. The default values are typically woefully inadequate for production databases.

Default value: typically 32MB

Recommended value: about 75% of database memory up to a max of about 2GB

33.2 effective_cache_size

In addition to the memory PostgreSQL sets aside for `shared_buffers` the query planner also takes into account how many disk blocks the operating system may have cached as part of its virtual file system. For systems with large amount of memory, this can be quite large. The `effective_cache_size` is approximately the amount of memory on the machine, less the `shared_buffers`, less the `work_mem` times the expected number of connections, less any memory required for any other processes running on the machine, less about 1GB for other random operating system needs. The database will not **use** the extra cache directly, but it will compute plans expecting that the operating system has cached filesystem data in about that much memory.

Default value: typically 4GB

Recommended value: any amount of "free" memory expected to be around under ordinary operating conditions

33.3 work_mem

Defines the amount of memory that internal sorting operations, indexing operations and hash tables can consume before the database switches to on-disk files. This value defines the available memory for each operation; complex queries may have several sort or hash operations running in parallel, and each connected session may be executing a query.

As such you must consider how many connections and the complexity of expected queries before increasing this value. The **benefit** to increasing is that the processing of more of these operations, including ORDER BY, and DISTINCT clauses, merge and hash joins, hash-based aggregation and hash-based processing of subqueries, can be accomplished without incurring disk writes. The **cost** of increasing is memory that will be used **per connection**, which can be quite high with production levels of connections.

Default value: 1MB

Recommended value: 32MB

33.4 maintenance_work_mem

Defines the amount of memory used for maintenance operations, including vacuuming, index and foreign key creation. As these operations are not terribly common, a higher value will only exact an occasional cost, and may substantially speed up maintenance activities This parameter can alternately be increased for a single session before the execution of a number of **CREATE INDEX** or **VACUUM** calls as shown below.

```
SET maintenance_work_mem TO '128MB';
VACUUM ANALYZE;
SET maintenance_work_mem TO '16MB';
```

Default value: 16MB

Recommended value: 128MB

33.5 wal_buffers

Sets the amount of memory used for write-ahead log (WAL) data. Write-ahead logs provide a high-performance mechanism for insuring data-integrity. During each change command, the effects of the changes are written first to the WAL files and flushed to disk. Only once the WAL files have been flushed will the changes be written to the data files themselves. This allows the data files to be written to disk in an optimal and asynchronous manner while ensuring that, in the event of a crash, all data changes can be recovered from the WAL.

The size of this buffer only needs to be large enough to hold WAL data for a single typical transaction. While the default value is often sufficient for most data, geospatial data tends to be much larger. Therefore, it is recommended to increase the size of this parameter.

Default value: 64kB

Recommended value: 1MB

33.6 checkpoint_segments

This value sets the maximum number of log file segments (typically 16MB) that can be filled between automatic WAL checkpoints. A WAL checkpoint is a point in the sequence of WAL transactions at which it is guaranteed that the data files have been updated with all information before the checkpoint. At this time all dirty data pages are flushed to disk and a checkpoint record is written to the log file. This allows the crash recovery process to find the latest checkpoint record and apply all following log segments to complete the data recovery.

Because the checkpoint process requires the flushing of all dirty data pages to disk, it creates a significant I/O load. The same argument from above applies; geospatial data is large enough to unbalance non-geospatial optimizations. Increasing this value will prevent excessive checkpoints, though it may cause the server to restart more slowly in the event of a crash.

Default value: 3

Recommended value: 6

33.7 random_page_cost

This is a unit-less value that represents the cost of a random page access from disk. This value is relative to a number of other cost parameters including sequential page access, and CPU operation costs. While there is no magic bullet for this value, the default is generally conservative and for databases running on spinning media. The random access cost for SSD should be set even lower.

This value can be set on a per-session basis using the `SET random_page_cost TO 2.0` command, which can be useful for testing how it effects query plans.

Default value: 4.0

Recommended value: 2.0 for spinning media, 1.0 for SSD

33.8 seq_page_cost

This is the parameter that controls the cost of a sequential page access. This value does not generally require adjustment but the difference between this value and `random_page_cost` greatly affects the choices made by the query planner. This value can also be set on a per-session basis.

Default value: 1.0

Recommended value: 1.0

33.9 Reload configuration

After these changes are made, save changes and reload the configuration. The easiest way to do this is to restart the PostgreSQL service.

- In pgAdmin, right-click the server **PostGIS (localhost:5432)** and select *Disconnect*.
- In Windows Services (`services.msc`) right-click **PostgreSQL** and select *Restart*.
- Back in pgAdmin, click the server again select *Disconnect*.

CHAPTER

THIRTYFOUR

POSTGRESQL SECURITY

PostgreSQL has a rich and flexible permissions system, with the ability to parcel out particular privileges to particular roles, and provide users with the powers of one or more of those roles.

In addition, the PostgreSQL server can use multiple different systems to authenticate users. This means that the database can use the same authentication infrastructure as other architecture components, simplifying password management.

34.1 Users and Roles

In this chapter we will create two useful production users:

- A read-only user for use in a publishing application.
- A read/write user for use by a developer in building a software or analyzing data.

Rather than creating users and granting them the necessary powers, we will create two roles with the right powers and then create two users and add them to the appropriate roles. That way we can easily reuse the roles when we create further users.

34.1.1 Creating Roles

A role is a user and a user is a role. The only difference is that a "user" can be said to be a role with the "login" privilege.

So functionally, the two SQL statements below are the same, they both create a "role with the login privilege", which is to say, a "user".

```
CREATE ROLE mrbean LOGIN;
CREATE USER mrbean;
```

34.1.2 Read-only Users

Our read-only user will be for a web application to use to query the `nyc_streets` table.

The application will have specific access to the `nyc_streets` table, but will inherit the necessary system access for PostGIS operations from the `postgis_reader` role.

```
-- A user account for the web app
CREATE USER app1;
-- Web app needs access to specific data tables
GRANT SELECT ON nyc_streets TO app1;

-- A generic role for access to PostGIS functionality
CREATE ROLE postgis_reader INHERIT;
-- Give that role to the web app
GRANT postgis_reader TO app1;
```

Now, when we login as app1, we can select rows from the `nyc_streets` table. However, we cannot run an **`ST_Transform`** call! Why not?

```
-- This works!
SELECT * FROM nyc_streets LIMIT 1;

-- This doesn't work!
SELECT ST_AsText(ST_Transform(geom, 4326))
  FROM nyc_streets LIMIT 1;
```

```
ERROR:  permission denied for relation spatial_ref_sys
CONTEXT:  SQL statement "SELECT proj4text FROM spatial_ref_sys WHERE srid␣
↪= 4326 LIMIT 1"
```

The answer is contained in the error statement. Though our `app1` user can view the contents of the `nyc_streets` table fine, it cannot view the contents of `spatial_ref_sys`, so the call to **`ST_Transform`** fails.

So, we need to also grant the `postgis_reader` role read access to all the PostGIS metadata tables:

```
GRANT SELECT ON geometry_columns TO postgis_reader;
GRANT SELECT ON geography_columns TO postgis_reader;
GRANT SELECT ON spatial_ref_sys TO postgis_reader;
```

Now we have a nice generic `postgis_reader` role we can apply to any user that need to read from PostGIS tables.

```
-- This works now!
SELECT ST_AsText(ST_Transform(geom, 4326))
  FROM nyc_streets LIMIT 1;
```

34.1.3 Read/write Users

There are two kinds of read/write scenarios we need to consider:

- Web applications and others that need to write to existing data tables.
- Developers or analysts that need to create new tables and geometry columns as part of their work.

For web applications that require write access to data tables, we just need to grant extra permissions to the tables themselves, and we can continue to use the `postgis_reader` role.

```
-- Add insert/update/delete abilities to our web application
GRANT INSERT,UPDATE,DELETE ON nyc_streets TO app1;
```

These kinds of permissions would be required for a read/write WFS service, for example.

For developers and analysts, a little more access is needed to the main PostGIS metadata tables. We will need a `postgis_writer` role that can edit the PostGIS metadata tables!

```
-- Make a postgis writer role
CREATE ROLE postgis_writer;

-- Start by giving it the postgis_reader powers
GRANT postgis_reader TO postgis_writer;

-- Add insert/update/delete powers for the PostGIS tables
GRANT INSERT,UPDATE,DELETE ON spatial_ref_sys TO postgis_writer;

-- Make app1 a PostGIS writer to see if it works!
GRANT postgis_writer TO app1;
```

Now try the table creation SQL above as the app1 user and see how it goes!

34.2 Encryption

PostgreSQL provides a lot of encryption facilities, many of them optional, some of them on by default.

- By default, all passwords are MD5 encrypted. The client/server handshake double encrypts the MD5 password to prevent re-use of the hash by anyone who intercepts the password.
- SSL connections are optionally available between the client and server, to encrypt all data and login information. SSL certificate authentication is also available when SSL connections are used.
- Columns inside the database can be encrypted using the pgcrypto module, which includes hashing algorithms, direct ciphers (blowfish, aes) and both public key and symmetric PGP encryption.

34.2.1 SSL Connections

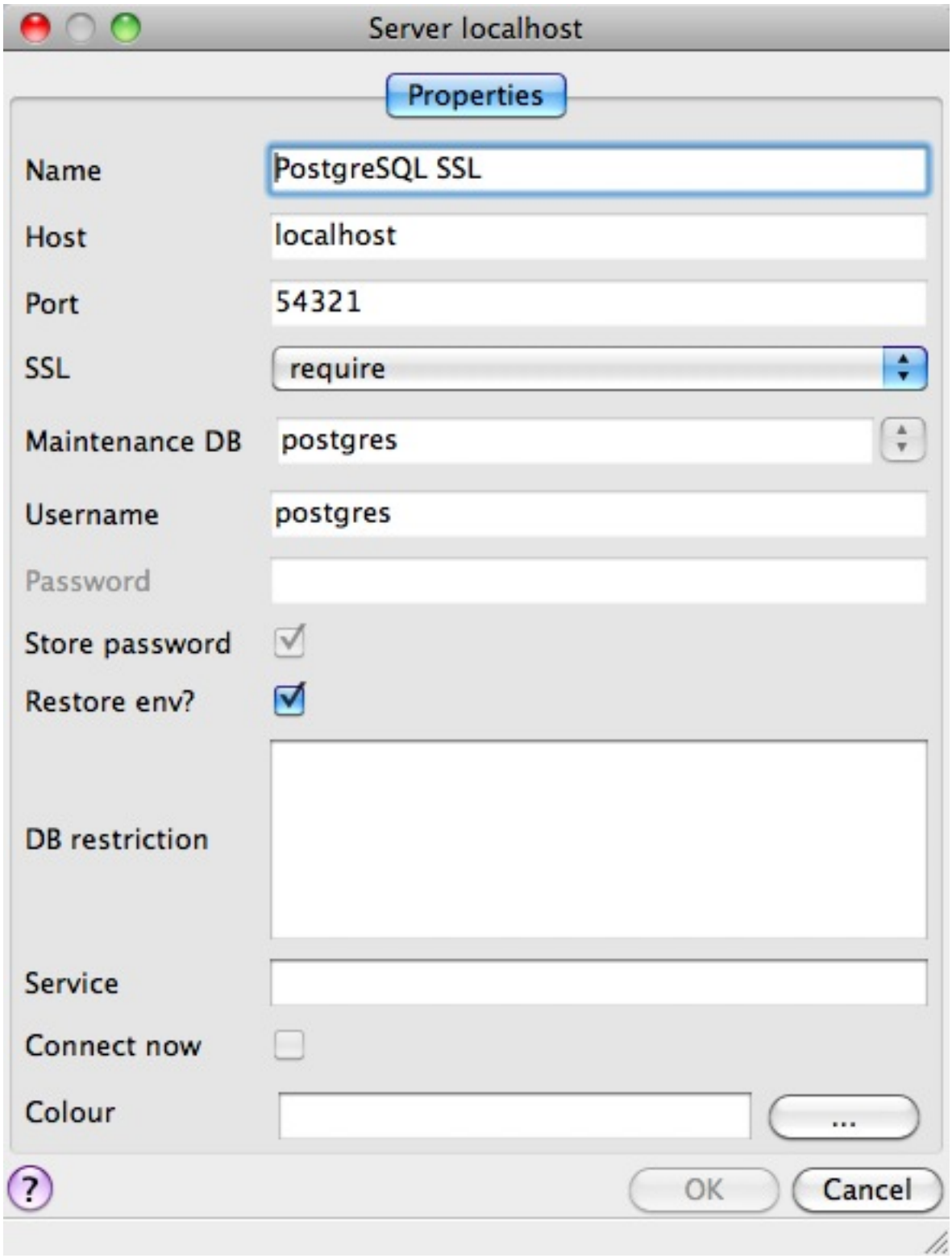
In order to use SSL connections, both your client and server must support SSL.

- First, turn off PostgreSQL, since activating SSL will require a restart.
- Next, we acquire or generate an SSL certificate and key. The certificate will need to have no passphrase on it, or the database server won't be able to start up. You can generate a self-signed key as follows:

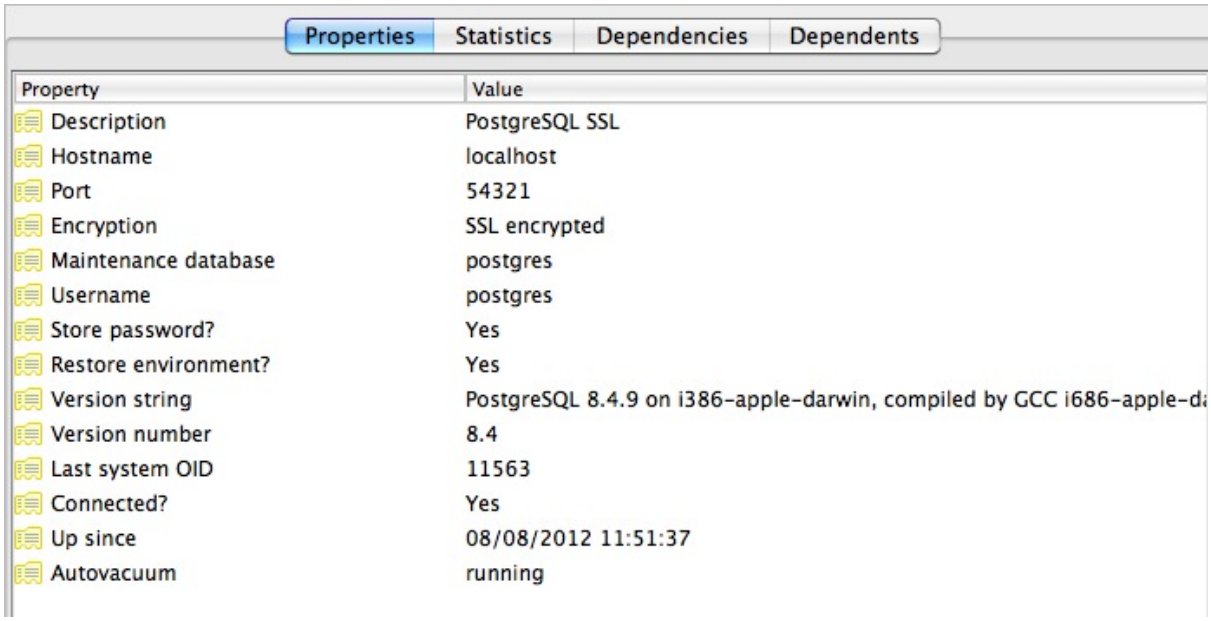
```
# Create a new certificate, filling out the certification info as␣
↪prompted
openssl req -new -text -out server.req

# Strip the passphrase from the certificate
openssl rsa -in privkey.pem -out server.key

# Convert the certificate into a self-signed cert
openssl req -x509 -in server.req -text -key server.key -out server.crt

# Set the permission of the key to private read/write
chmod og-rwx server.key
```

- Copy the `server.crt` and `server.key` into the PostgreSQL data directory.
- Enable SSL support in the `postgresql.conf` file by turning the "ssl" parameter to "on".
- Now re-start PostgreSQL; the server is ready for SSL operation.

With the server enabled for SSL, creating an encrypted connection is easy. In PgAdmin, create a new server connection (File > Add Server. . .), and set the SSL parameter to "require".



Once you connect with the new connection, you can see in its properties that it is using an SSL connection.



Since the default SSL connection mode is "prefer", you don't even need to specify an SSL preference when connecting. A connection with the command line `psql` terminal will pick up the SSL option and use it by default:

```
psql (8.4.9)
SSL connection (cipher: DHE-RSA-AES256-SHA, bits: 256)
Type "help" for help.

postgres=#
```

Note how the terminal reports the SSL status of the connection.

34.2.2 Data Encryption

The pgcrypto module has a huge range of encryption options, so we will only demonstrate the simplest use case: encrypting a column of data using a symmetric cipher.

- First, enable pgcrypto by loading the contrib SQL file, either in PgAdmin or psql.

```
pgsql/8.4/share/postgresql/contrib/pgcrypto.sql
```

- Then, test the encryption function.

```
-- encrypt a string using blowfish (bf)
SELECT encrypt('this is a test phrase', 'mykey', 'bf');
```

- And make sure it's reversible too!

```
-- round-trip a string using blowfish (bf)
SELECT decrypt(encrypt('this is a test phrase', 'mykey', 'bf'), 'mykey
↪', 'bf');
```

34.3 Authentication

PostgreSQL supports many different authentication methods, to allow easy integration into existing enterprise architectures. For production purposes, the following methods are commonly used:

- **Password** is the basic system where the passwords are stored by the database, with MD5 encryption.
- Kerberos is a standard enterprise authentication method, which is used by both the GSSAPI and SSPI schemes in PostgreSQL. Using SSPI, PostgreSQL can authenticate against Windows servers.
- LDAP is another common enterprise authentication method. The OpenLDAP server bundled with most Linux distributions provides an open source implementation of LDAP.
- **Certificate** authentication is an option if you expect all client connections to be via SSL and are able to manage the distribution of keys.
- PAM authentication is an option if you are on Linux or Solaris and use the PAM scheme for transparent authentication provision.

Authentication methods are controlled by the `pg_hba.conf` file. The "HBA" in the file name stands for "host based access", because in addition to allowing you to specify the authentication method to use for each database, it allows you to limit host access using network addresses.

Here is an example `pg_hba.conf` file:

```
# TYPE  DATABASE    USER        CIDR-ADDRESS          METHOD

# "local" is for Unix domain socket connections only
local   all         all                               trust
# IPv4 local connections:
host    all         all         127.0.0.1/32          trust
# IPv6 local connections:
host    all         all         ::1/128               trust
# remote connections for nyc database only
host    nyc         all         192.168.1.0/2         ldap
```

The file consists of five columns

- **TYPE** determines the kind of access, either "local" for connections from the same server or "host" for remote connections.
- **DATABASE** specifies what database the configuration line refers to or "all" for all databases
- **USER** specifies what users the line refers to or "all" for all users
- **CIDR-ADDRESS** specifies the network limitations for remote connections, using network/netmask syntax
- **METHOD** specifies the authentication protocol to use. "trust" skips authentication entirely and simply accepts any valid username without challenge.

It's common for local connections to be trusted, since access to the server itself is usually privileged. Remote connections are disabled by default when PostgreSQL is installed: if you want to connect from remote machines, you'll have to add an entry.

The line for `nyc` in the example above is an example of a remote access entry. The `nyc` example allows LDAP authenticated access only to machines on the local network (in this case the 192.168.1. network)

and only to the nyc database. Depending on the security of your network, you will use more or less strict versions of these rules in your production set-up.

34.4 Links

- PostgreSQL Authentication
- PostgreSQL Encrpyption
- PostgreSQL SSL Support

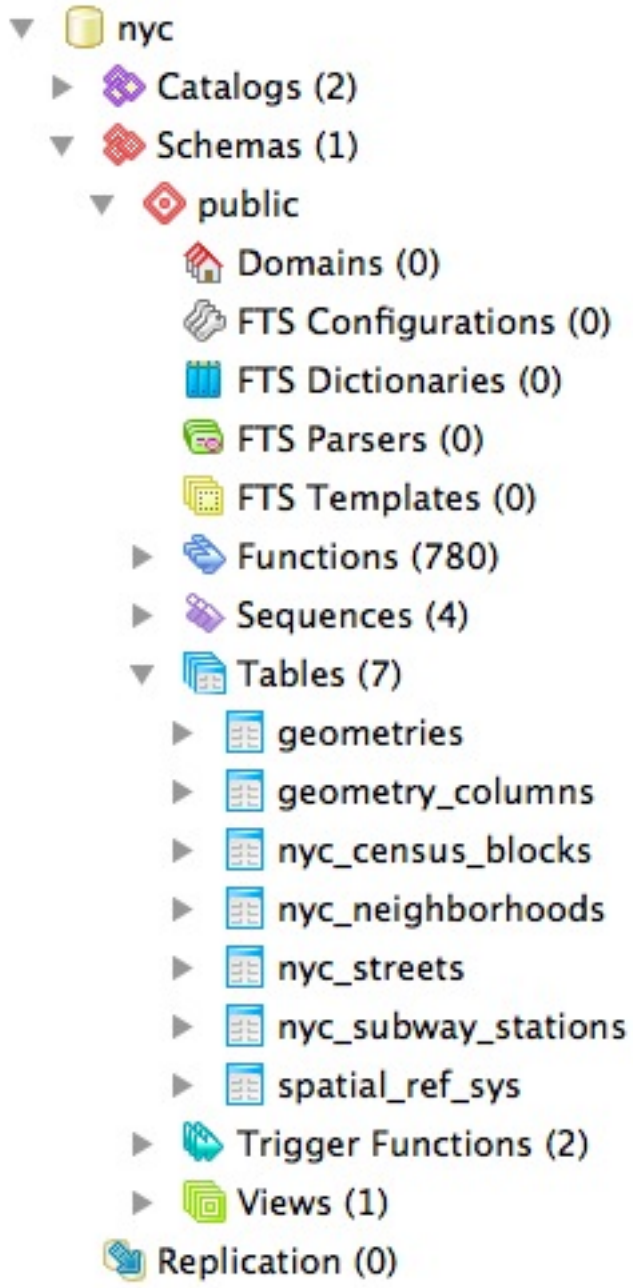
CHAPTER

THIRTYFIVE

POSTGRESQL SCHEMAS

Production databases inevitably have a large number of tables and views, and managing them all in one schema can become unwieldy quickly. Fortunately, PostgreSQL includes the concept of a "_Schema".

Schemas are like folders, and can hold tables, views, functions, sequences and other relations. Every database starts out with one schema, the `public` schema.



Inside that schema, the default install of PostGIS creates the `geometry_columns`, `geography_columns` and `spatial_ref_sys` metadata relations, as well as all the types and functions used by PostGIS. So users of PostGIS always need access to the public schema.

In the public schema you can also see all the tables we have created so far in the workshop.

35.1 Why use Schemas?

There are two very good reasons for using schemas:

- Data that is managed in a schema is easier to apply bulk actions to.
  - It's easier to back-up data that's in a separate schema: so volatile data can have a different back-up schedule from non-volatile data.
  - It's easier to restore data that's in a separate schema: so application-oriented schemas can be separately restored and backed up for time travel and recovery.
  - It's easier to manage application differences when the application data is in a schema: so a new version of software can work off a table structure in a new schema, and cut-over involves a simple change to the schema name.
- Users can be restricted in their work to single schemas to allow isolation of analytical and test tables from production tables.

So for production purposes, keeping your application data separate in schemas improves management; and for user purposes, keeping your users in separate schemas keeps them from treading on each other.

35.2 Creating a Data Schema

Let's create a new schema and move a table into it. First, create a new schema in the database:

```
CREATE SCHEMA census;
```

Next, we will move the `nyc_census_blocks` table to the `census` schema:

```
ALTER TABLE nyc_census_blocks SET SCHEMA census;
```

If you're using the **`psql`** command-line program, you'll notice that `nyc_census_blocks` has disappeared from your table listing now! If you're using PgAdmin, you might have to refresh your view to see the new schema and the table inside it.

You can access tables inside schemas in two ways:

- by referencing them using `schema.table` notation
- by adding the schema to your `search_path`

Explicit referencing is easy, but it gets tiring to type after a while:

```
SELECT * FROM census.nyc_census_blocks LIMIT 1;
```

Manipulating the `search_path` is a nice way to provide access to tables in multiple schemas without lots of extra typing.

You can set the `search_path` at run time using the `SET` command:

```
SET search_path = census, public;
```

This ensures that all references to relations and functions are searched in both the `census` and the `public` schemas. Remember that all the PostGIS functions and types are in `public` so we don't want to drop that from the search path.

Setting the search path every time you connect can get tiring too, but fortunately it's possible to permanently set the search path for a user:

```
ALTER USER postgres SET search_path = census, public;
```

Now the postgres user will always have the `census` schema in their search path.

35.3 Creating a User Schema

Users like to create tables, and PostGIS users do so particularly: analysis operations with SQL demand temporary tables for visualization or interim results, so spatial SQL tends to require that users have CREATE privileges more than ordinary database workloads.

By default, every role in Oracle is given a personal schema. This is a nice practice to use for PostgreSQL users too, and is easy to replicate using PostgreSQL roles, schemas, and search paths.

Create a new user with table creation privileges (see *PostgreSQL Security* for information about the `postgis_writer` role), then create a schema with that user as the authorization:

```
CREATE USER myuser WITH ROLE postgis_writer;
CREATE SCHEMA myuser AUTHORIZATION myuser;
```

If you log in as that user, you'll find the default `search_path` for PostgreSQL is actually this:

```
show search_path;
```

```
  search_path
----------------
 "$user",public
```

The first schema on the search path us the user's named schema! So now the following conditions exist:

- The user exists, with the ability to create spatial tables.
- The user's named schema exists, and the user owns it.
- The user's search path has the user schema first, so new tables are automatically created there, and queries automatically search there first.

That's all there is to it, the user's default work area is now nicely separated from any tables in other schemas.

CHAPTER

THIRTYSIX

POSTGRESQL BACKUP AND RESTORE

There are lots of ways to backup a PostgreSQL database, and the one you choose will depend a great deal on how you are using the database.

- For relatively static databases, the basic pg_dump/pg_restore tools can be used to take periodic snapshots of the data.
- For frequently changing data, using an "online backup" scheme allows continuous archiving of updates to a secure location.

Online backup is the basis for replication and stand-by systems for high availability, particularly for versions of PostgreSQL >= 9.0.

36.1 Laying Out your Data

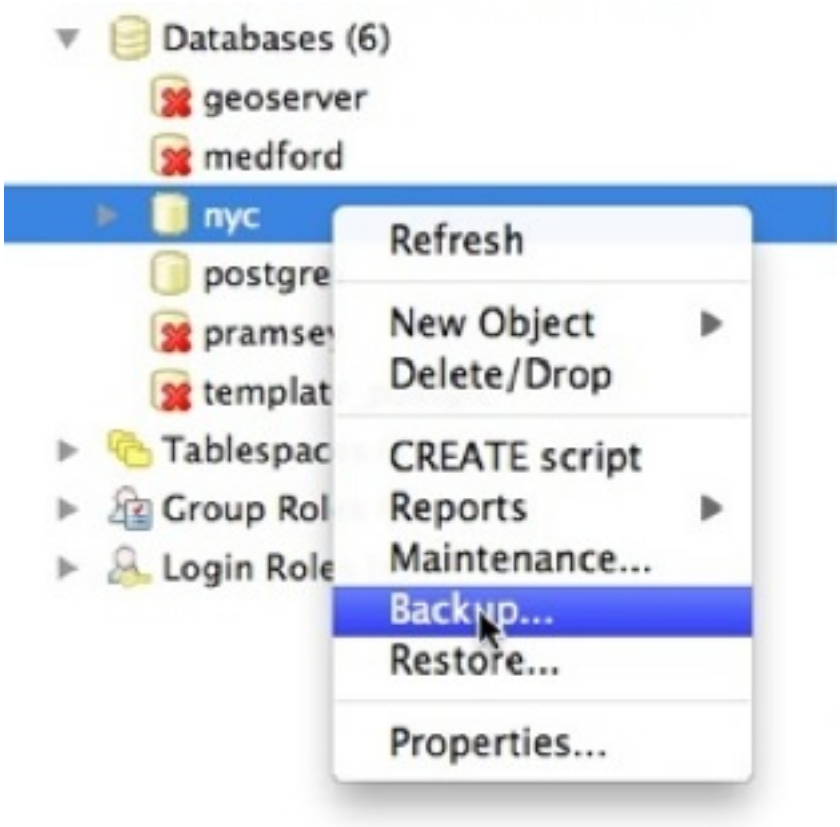
As discussed in *PostgreSQL Schemas*, ensuring that production data is always stored in separate schemas is a very important **best practice** in managing data. There are two reasons:

- Backing up and restoring data in schemas is much simpler than managing lists of tables to be backed up individually.
- Keeping data tables out of the "public" schema allows far easier upgrades, as discussed in *Software Upgrades*.

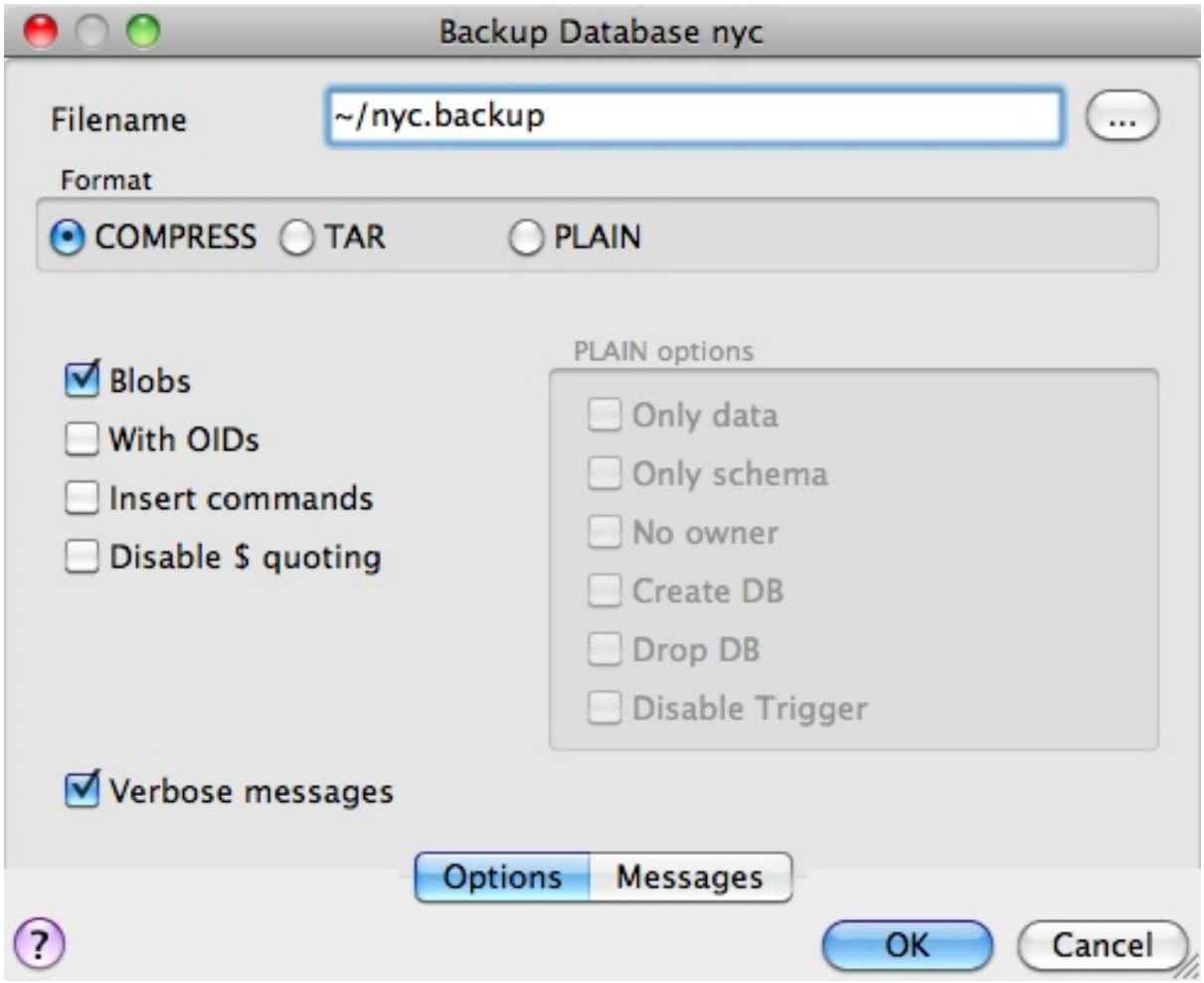
36.2 Basic Backup and Restore

Backing up a full database is easy using the pg_dump utility. The utility is a command-line tool, which makes it easy to automate with scripting, and it can also be invoke via a GUI in the PgAdmin utility.

To backup our `nyc` database, we can use the GUI, just right-click the database you want to backup:



Enter the name of the backup file you want to create.



Note that there are three backup format options: compress, tar and plain.

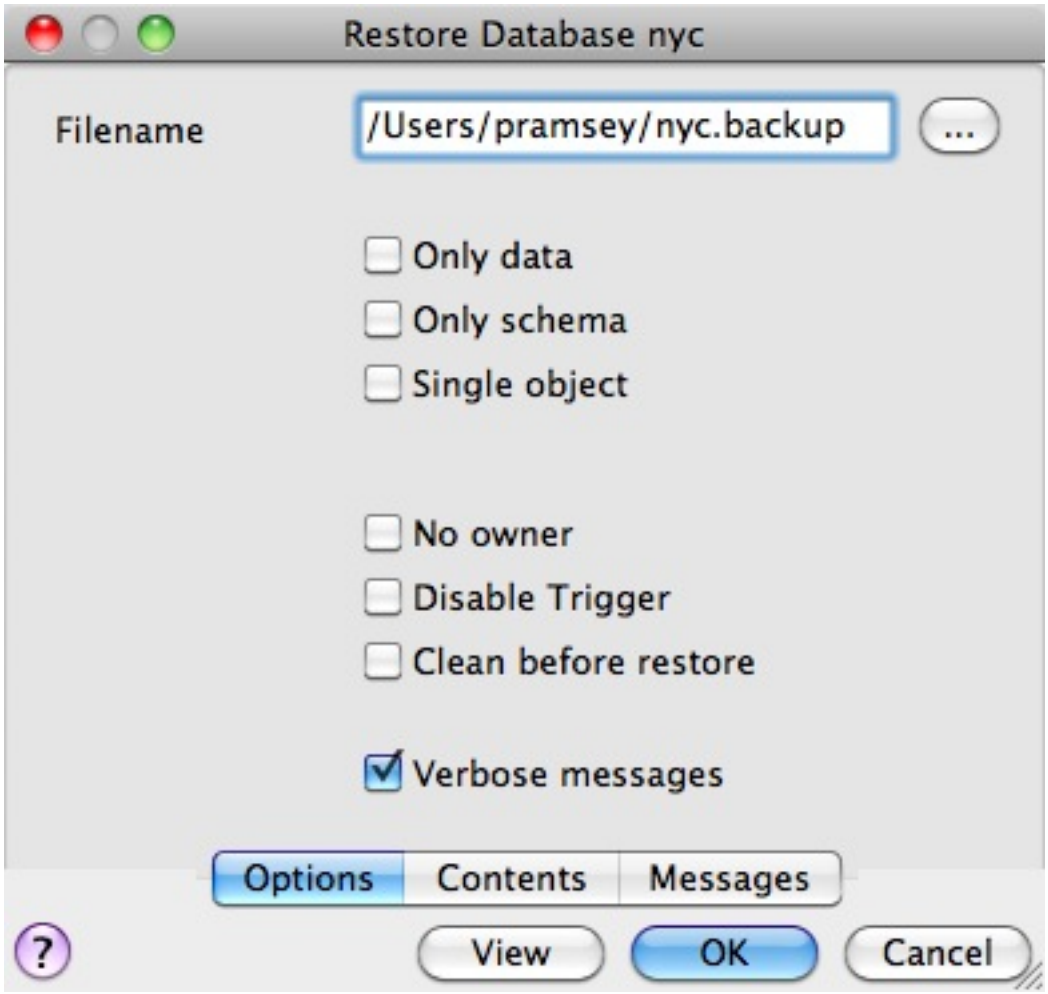
- **Plain** is just a textual SQL file. This is the simplest format and in many ways the most flexible, since it can be editing or altered easily and then loaded back into a database, allowing offline changes to things like ownership or other global information.
- **Tar** using a UNIX archive format to hold components of the dump in separate files. Using the tar format allows the pg_restore utility to selectively restore parts of the dump.
- **Compress** is like the Tar format, but compresses the internal components individually, allowing them to be selectively restored without decompressing the entire archive.

We'll check the Compress option and go, saving out a backup file.

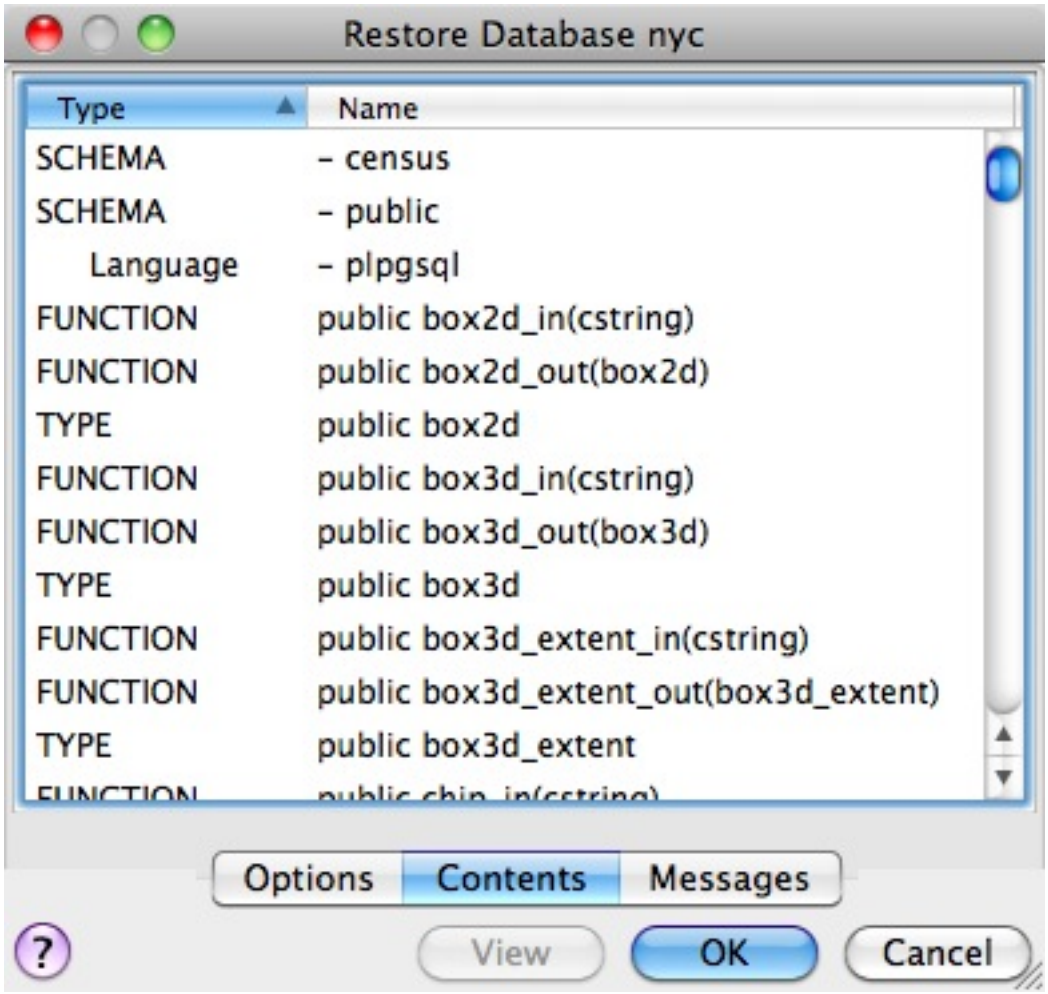
The same operation can be done with the command line like this:

```
pg_dump --file=nyc.backup --format=c --port=54321 --username=postgres nyc
```

Because the backup file is in Compress format, we can view the contents using the pg_restore command to list the manifest. In the PgAdmin GUI, "View" is an option in the panel.



When you look at the manifest, one of the things you might notice is that there are a lot of "FUNCTION" signatures in there.



That's because the pg_dump utility dumps **every** non-system object in the database, and that includes the PostGIS function definitions.

Note: PostgreSQL 9.1+ includes an "EXTENSION" feature that allows add-on packages like PostGIS to be installed as registered system components and therefore excluded from pg_dump output. PostGIS 2.0 and higher support installation using this extension system.

We can see the same manifest from the command-line using pg_restore directly:

```
pg_restore --list nyc.backup
```

The problem with a dump file full of PostGIS function signatures is that we really wanted a dump of our data, not our system functions.

Since every object is in the dump file, we can restore to a blank database and get full functionality. In doing so, we are expecting that system we are restoring to has exactly the same version of PostGIS as the one we dumped from (since the function signature definitions reference a particular version of the PostGIS shared library).

From the command-line the restore looks like this:

```
createdb --port 54321 nyc2
pg_restore --dbname=nyc2 --port 54321 --username=postgres nyc.backup
```

Dumping just data, without function signatures, is where having data in schemas is handy, because there is a command-line flag to only dump a particular schema:

```
pg_dump --port=54321 -format=c --schema=census --file=census.backup
```

Now when we list the contents of the dump, we see just the data tables we wanted:

```
pg_restore --list census.backup

;
; Archive created at Thu Aug  9 11:02:49 2012
;     dbname: nyc
;     TOC Entries: 11
;     Compression: -1
;     Dump Version: 1.11-0
;     Format: CUSTOM
;     Integer: 4 bytes
;     Offset: 8 bytes
;     Dumped from database version: 8.4.9
;     Dumped by pg_dump version: 8.4.9
;
;
; Selected TOC Entries:
;
6; 2615 20091 SCHEMA - census postgres
146; 1259 19845 TABLE census nyc_census_blocks postgres
145; 1259 19843 SEQUENCE census nyc_census_blocks_gid_seq postgres
2691; 0 0 SEQUENCE OWNED BY census nyc_census_blocks_gid_seq postgres
2692; 0 0 SEQUENCE SET census nyc_census_blocks_gid_seq postgres
2681; 2604 19848 DEFAULT census gid postgres
2688; 0 19845 TABLE DATA census nyc_census_blocks postgres
2686; 2606 19853 CONSTRAINT census nyc_census_blocks_pkey postgres
2687; 1259 20078 INDEX census nyc_census_blocks_geom_gist postgres
```

Having just the data tables is handy, because it means we can store to a database with any version of PostGIS installed, as we talk about in *Software Upgrades*.

36.2.1 Backing Up Users

The pg_dump utility operates a database at a time (or a schema or table at a time, if you restrict it). However, information about users is is stored across an entire cluster, it's not stored in any one database!

To backup your user information, use the pg_dumpall utility, with the "–globals-only" flag.

```
pg_dumpall --globals-only --port 54321
```

You can also use pg_dumpall in its default mode to backup an entire cluster, but be aware that, as with pg_dump, you will end up backing up the PostGIS function signatures, so the dump will have to be restored against an identical software installation, it can't be used as part of an upgrade process.

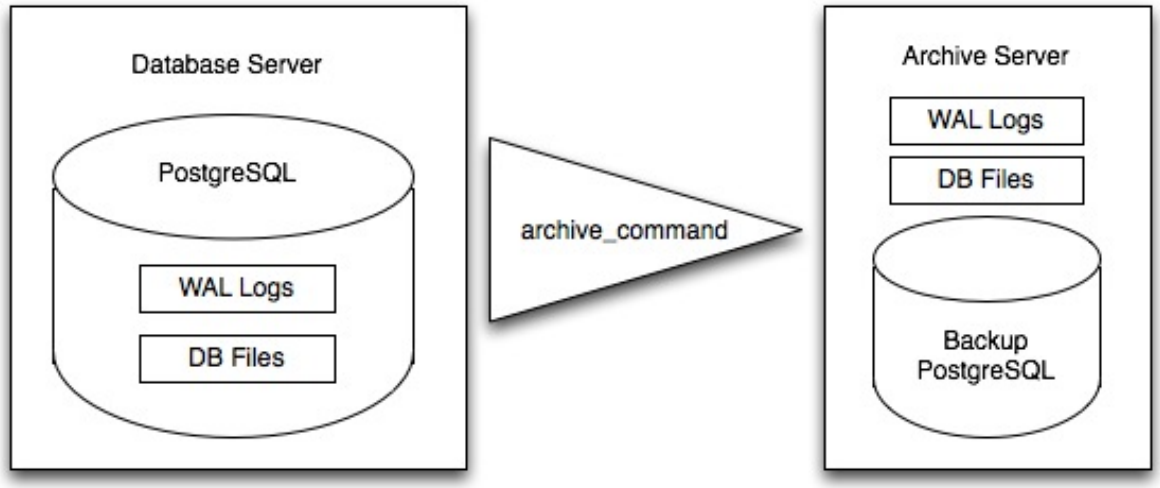
36.3 Online Backup and Restore

Online backup and restore allows an administrator to keep an extremely up-to-date set of backup files without the overhead of repeatedly dumping the entire database. If the database is under frequent insert and update load, then online backup might be preferable to basic backup.

Note: The best way to learn about online backup is to read the relevant sections of the PostgreSQL manual on continuous archiving and point-in-time recovery. This section of the PostGIS workshop will just provide a brief snapshot of online backup set-up.

36.3.1 How it Works

Rather than continually write to the main data tables, PostgreSQL stores changes initially in "write-ahead logs" (WAL). Taken together, these logs are a complete record of all changes made to a database. Online backup consists of taking a copy of the database main data table, then taking a copy of each WAL that is generated from then on.



When it is time to recover to a new database, the system starts on the main data copy, then replays all the WAL files into the database. The end result is a restored database in the same state as the original at the time of the last WAL received.

Because WAL are being written anyways, and transferring copies to an archive server is computationally cheap, online backup is an effective means of keeping a very up-to-date backup of a system without resorting to intensive regular full dumps.

36.3.2 Archiving the WAL Files

The first thing to do in setting up online backup is to create an archiving method. PostgreSQL archiving methods are the ultimate in flexibility: the PostgreSQL backend simply calls a script specified in the `archive_command` configuration parameter.

That means archiving can be as simple as copying the file to a network-mounted drive, and as complex as encrypting and emailing the files to the remote archive. Any process you can script you can use to archive the files.

To turn on archiving we will edit `postgresql.conf`, first turning on WAL archiving:

```
wal_level = archive
archive_mode = on
```

And then setting the `archive_command` to copy our archive files to a safe location (changing the destination paths as appropriate):

```
# Unix
archive_command = 'test ! -f /archivedir/%f && cp %p /archivedir/%f'

# Windows
archive_command = 'copy "%p" "C:\\archivedir\\%f"'
```

It is important that the archive command not over-write existing files, so the unix command includes an initial test to ensure that the files aren't already there. It is also important that the command returns a non-zero status if the copy process fails.

Once the changes are made you can re-start PostgreSQL to make them effective.

36.3.3 Taking the Base Backup

Once the archiving process is in place, you need to take a base back-up.

Put the database into backup mode (this doesn't do anything to alter operation of queries or data updates, it just forces a checkpoint and writes a label file indicating when the backup was taken).

```
SELECT pg_start_backup('/archivedir/basebackup.tgz');
```

For the label, using the path to the backup file is a good practice, as it helps you track down where the backup was stored.

Copy the database to an archival location:

```
# Unix
tar cvfz /archivedir/basebackup.tgz ${PGDATA}
```

Then tell the database the backup process is complete.

```
SELECT pg_stop_backup();
```

All these steps can of course be scripted for regular base backups.

36.3.4 Restoring from the Archive

These steps are taking from the PostgreSQL manual on continuous archiving and point-in-time recovery.

- Stop the server, if it's running.
- If you have the space to do so, copy the whole cluster data directory and any tablespaces to a temporary location in case you need them later. Note that this precaution will require that you have enough free space on your system to hold two copies of your existing database. If you do not have enough space, you should at least save the contents of the cluster's pg_xlog subdirectory, as it might contain logs which were not archived before the system went down.
- Remove all existing files and subdirectories under the cluster data directory and under the root directories of any tablespaces you are using.
- Restore the database files from your file system backup. Be sure that they are restored with the right ownership (the database system user, not root!) and with the right permissions. If you are using tablespaces, you should verify that the symbolic links in pg_tblspc/ were correctly restored.
- Remove any files present in pg_xlog/; these came from the file system backup and are therefore probably obsolete rather than current. If you didn't archive pg_xlog/ at all, then recreate it with proper permissions, being careful to ensure that you re-establish it as a symbolic link if you had it set up that way before.
- If you have unarchived WAL segment files that you saved in step 2, copy them into pg_xlog/. (It is best to copy them, not move them, so you still have the unmodified files if a problem occurs and you have to start over.)
- Create a recovery command file recovery.conf in the cluster data directory (see Chapter 26). You might also want to temporarily modify pg_hba.conf to prevent ordinary users from connecting until you are sure the recovery was successful.
- Start the server. The server will go into recovery mode and proceed to read through the archived WAL files it needs. Should the recovery be terminated because of an external error, the server can simply be restarted and it will continue recovery. Upon completion of the recovery process, the server will rename recovery.conf to recovery.done (to prevent accidentally re-entering recovery mode later) and then commence normal database operations.
- Inspect the contents of the database to ensure you have recovered to the desired state. If not, return to step 1. If all is well, allow your users to connect by restoring pg_hba.conf to normal.

36.4 Links

- pg_dump
- pg_dumpall
- pg_restore
- PostgreSQL High Availability
- PostgreSQL High Availability Continuous Archiving and PITR

CHAPTER

THIRTYSEVEN

SOFTWARE UPGRADES

Because PostGIS resides within PostgreSQL every PostGIS installation actually consists of two versions of software: the PostgreSQL version and the PostGIS version. As a general principle, each version of PostGIS can be theoretically run within a number of versions of PostgreSQL, and vice versa.

In practice, the exact version pair available will be dictated by the packager who has built your PostgreSQL distribution. Most Linux packages includes a couple PostGIS versions for each PostgreSQL version release, allowing the parts to be upgraded either independently or simultaneously, depending on your preferences.

Upgrades can be considered in terms of upgrading each component.

37.1 Upgrading PostgreSQL

There are two kinds of PostgreSQL upgrade scenarios:

- A "minor upgrade" when the software version increases at the "patch" level. For example, from 8.4.3 to 8.4.4, or from 9.0.1 to 9.0.3. Increases of more than one patch version are just fine. Minor upgrades fix bugs but do not add any new features or change behaviour.
- A "major upgrade" when the "major" or "minor" versions increase. For example, from 8.4.5 to 9.0.0, or from 9.0.5 to 9.1.1. Major upgrades add new features and change behavior.

37.1.1 Minor PostgreSQL Upgrades

For "minor upgrades", no special process is necessary. Simply install the new software, and re-start the server.

37.1.2 Major PostgreSQL Upgrades

For "major upgrades" there are two ways to carry out the upgrade.

Dump/Restore

Dumping and restoring involves converting all the data to a platform neutral format (text representations) on dump, and back to native representations on restore, so it can be time consuming and CPU intensive. However, if you are migrating to a new architecture or operating system, it's a required process. It's also a time-tested and well-understood upgrade path, so if your database is not too big, there's no reason not to stick with it.

- Dump your data `pg_dumpall` from the old database.
- Install the new version of PostgreSQL and the same version of PostGIS you are using in your old database. You need to match the PostGIS version so that the dump file function definitions reference an expected version of the PostGIS library.
- Initialize the new data area using the `initdb` program from the new software.
- Start the new server on the new data area.
- Restore the dump file using `pg_restore`.

pg_upgrade

The pg_upgrade utility allows PostgreSQL data directories to be upgraded without the requirement for a dump/restore step. The utility cannot handle changes to the data files themselves, but handles the more common and frequent changes to system tables that occur in PostgreSQL major upgrades.

Note: The full instructions for running the upgrade process are in the pg_upgrade web page at the PostgreSQL site.

The pg_upgrade program expects to have access to both versions of PostgreSQL it is working with, the old and the new version, so you will have to install them both.

- Install the new version of PostgreSQL you will be using.
- Install the same version of PostGIS you are using in the old PostgreSQL into the new PostgreSQL.
- Initialize the new PostgreSQL data area with the new copy of `initdb`.
- Ensure both the old and new PostgreSQL servers are turned off.
- Run pg_upgrade, making sure to use the binary from the new software installation.

```
pg_upgrade
  --old-datadir "/var/lib/postgres/12/data"
  --new-datadir "/var/lib/postgres/13/data"
  --old-bindir "/usr/pgsql/12/bin"
  --new-bindir "/usr/pgsql/13/bin"
```

- If pg_upgrade generated any `.sql` files, run them now.
- Start the new server.

37.2 Upgrading PostGIS

PostGIS deals with minor and upgrades through the `EXTENSION` mechanism. If you spatially-enabled your database using `CREATE EXTENSION postgis`, you can update your database using the same functionality.

First, install the new software so it is available to the database.

Then, run the SQL to upgrade your PostGIS extension.

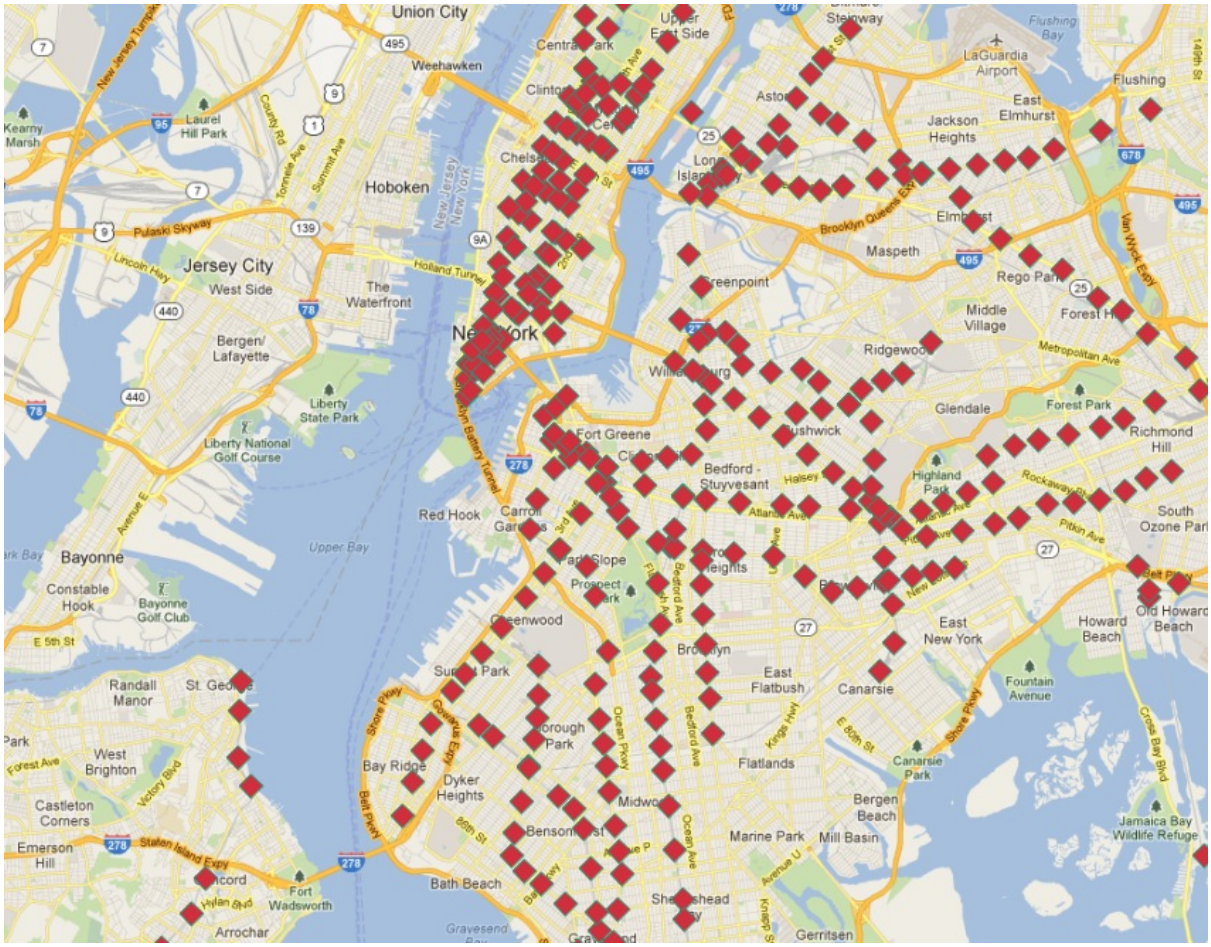
```
-- If you are upgrading from PostGIS 2.5 or later
-- and want the latest installed version
SELECT postgis_extensions_upgrade();

-- If you are upgrading from an earlier version
-- you have to specifically turn on the version you want
ALTER EXTENSION postgis UPDATE TO '2.5.5';
```

CHAPTER

THIRTYEIGHT

ADVANCED GEOMETRY CONSTRUCTIONS

The `nyc_subway_stations` layer has provided us with lots of interesting examples so far, but there is something striking about it:



Although it is a database of all the stations, it doesn't allow easy visualization of routes! In this chapter we will use advanced features of PostgreSQL and PostGIS to build up a new linear routes layer from the point layer of subway stations.

Our task is made especially difficult by two issues:

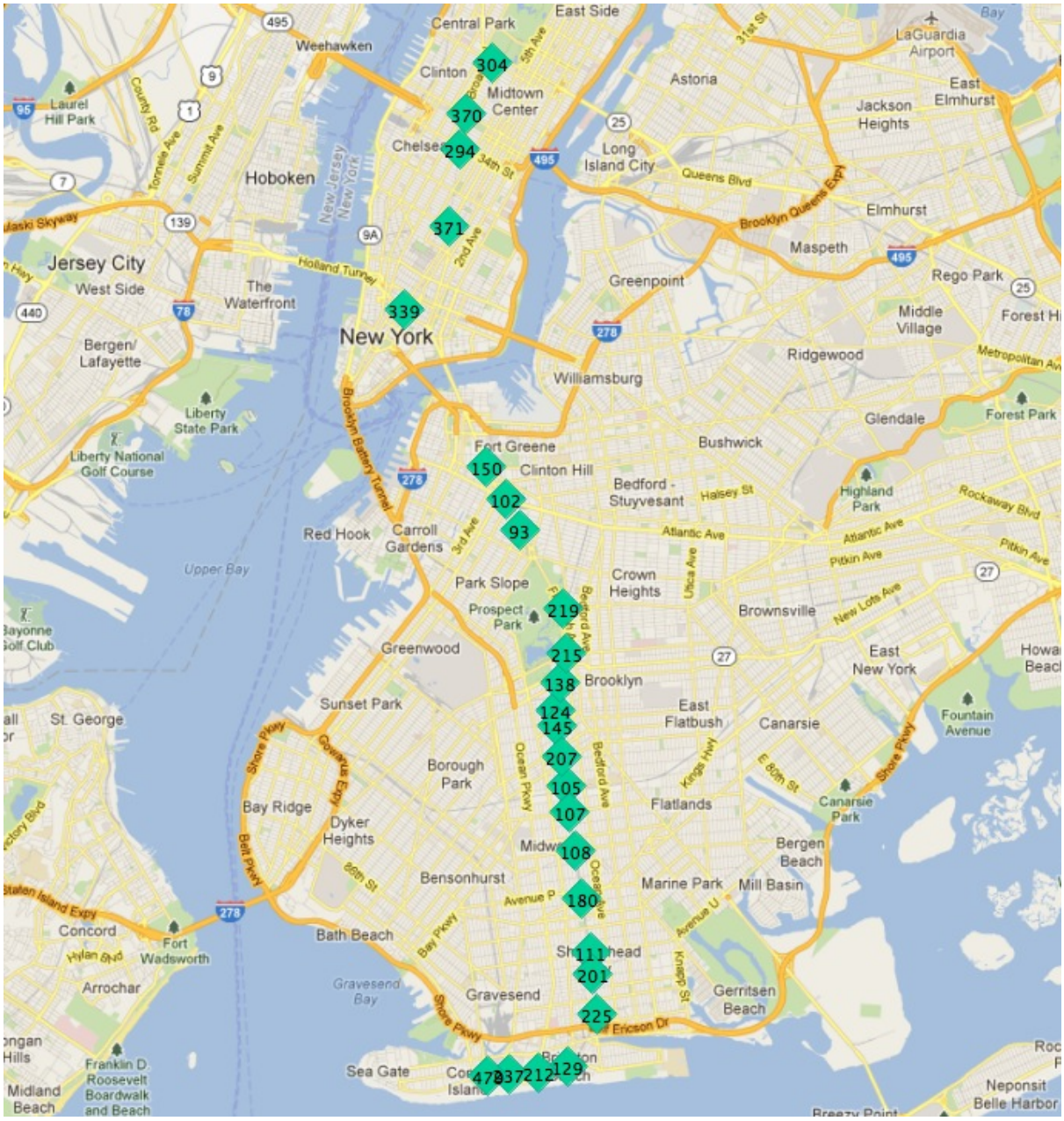
- The `routes` column of `nyc_subway_stations` has multiple route identifiers in each row, so a station that might appear in multiple routes appears only once in the table.
- Related to the previous issue, there is no route ordering information in the stations table, so while it is possible to find all the stations in a particular route, it's not possible using the attributes to determine what the order in which trains travel through the stations.

The second problem is the harder one: given an unordered set of points in a route, how do we order them to match the actual route.

Here are the stops for the 'Q' train:

```
SELECT s.gid, s.geom
FROM nyc_subway_stations s
WHERE (strpos(s.routes, 'Q') <> 0);
```

In this picture, the stops are labelled with their unique `gid` primary key.



If we start at one of the end stations, the next station on the line seems to always be the closest. We can repeat the process each time as long as we exclude all the previously found stations from our search.

There are two ways to run such an iterative routine in a database:

- Using a procedural language, like PL/PgSQL.
- Using recursive common table expressions.

Common table expressions (CTE) have the virtue of not requiring a function definition to run. Here's the CTE to calculate the route line of the 'Q' train, starting from the northernmost stop (where `gid` is 304).

```
WITH RECURSIVE next_stop(geom, idlist) AS (
    (SELECT
      geom,
      ARRAY[gid] AS idlist
    FROM nyc_subway_stations
    WHERE gid = 304)
    UNION ALL
    (SELECT
      s.geom,
      array_append(n.idlist, s.gid) AS idlist
    FROM nyc_subway_stations s, next_stop n
    WHERE strpos(s.routes, 'Q') != 0
    AND NOT n.idlist @> ARRAY[s.gid]
    ORDER BY ST_Distance(n.geom, s.geom) ASC
    LIMIT 1)
)
SELECT geom, idlist FROM next_stop;
```

The CTE consists of two halves, unioned together:

- The first half establishes a start point for the expression. We get the initial geometry and initialize the array of visited identifiers, using the record of "gid" 304 (the end of the line).
- The second half iterates until it finds no further records. At each iteration it takes in the value at the previous iteration via the self-reference to "next_stop". We search every stop on the Q line (**strpos(s.routes,'Q')**) that we have not already added to our visited list (**NOT n.idlist @> ARRAY[s.gid]**) and order them by their distance from the previous point, taking just the first one (the nearest).

Beyond the recursive CTE itself, there are a number of advanced PostgreSQL array features being used here:

- We are using ARRAY! PostgreSQL supports arrays of any type. In this case we have an array of integers, but we could also build an array of geometries, or any other PostgreSQL type.
- We are using **array_append** to build up our array of visited identifiers.
- We are using the **@>** array operator ("array contains") to find which of the Q train stations we have already visited. The **@>** operators requires ARRAY values on both sides, so we have to turn the individual "gid" numbers into single-entry arrays using the ARRAY[] syntax.

When you run the query, you get each geometry in the order it is found (which is the route order), as well as the list of identifiers already visited. Wrapping the geometries into the PostGIS ST_MakeLine aggregate function turns the set of geometries into a single linear output, constructed in the provided order.

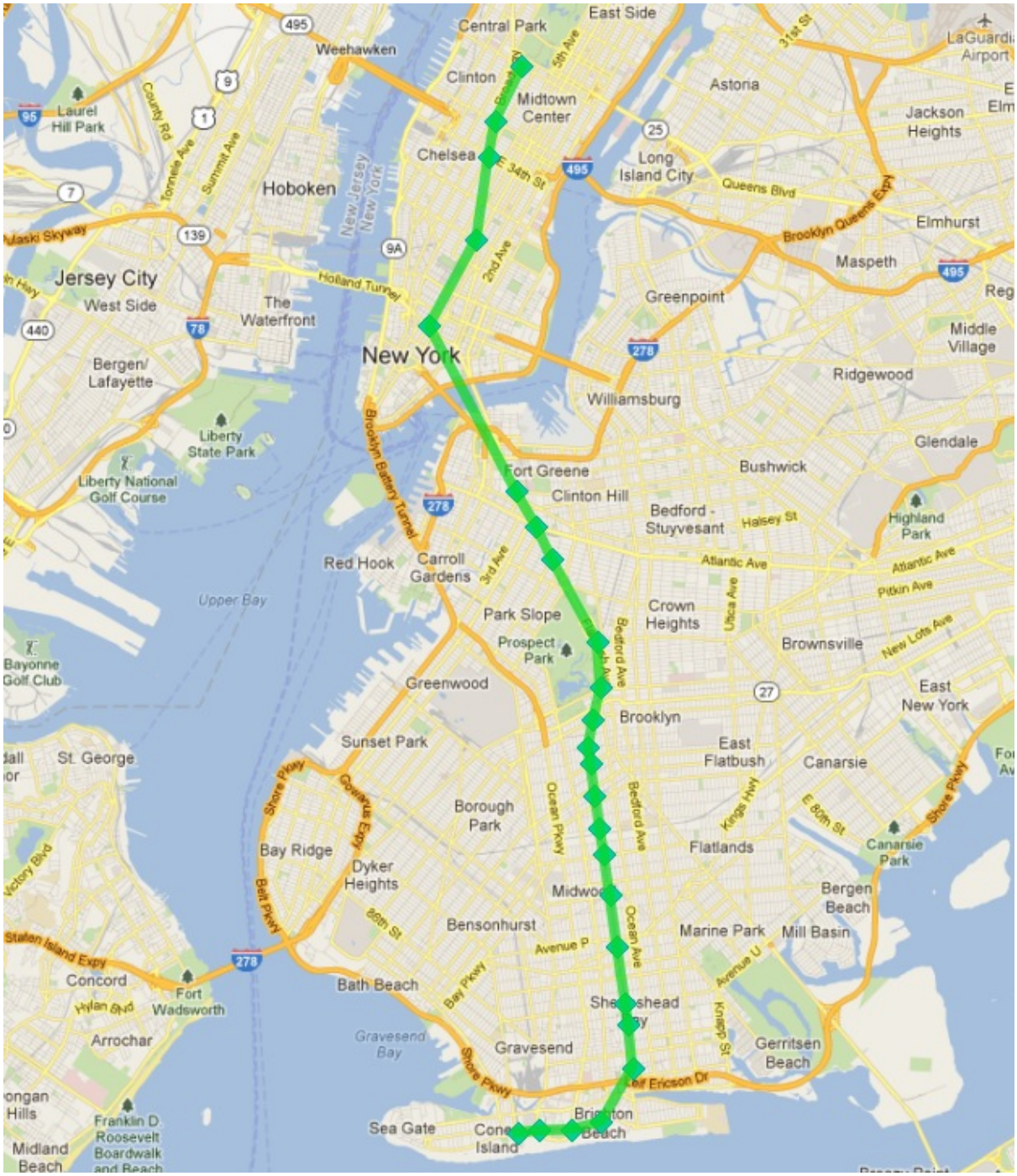
```
WITH RECURSIVE next_stop(geom, idlist) AS (
    (SELECT
      geom,
      ARRAY[gid] AS idlist
    FROM nyc_subway_stations
    WHERE gid = 304)
    UNION ALL
```

(continues on next page)

(continued from previous page)

```
    (SELECT
      s.geom,
      array_append(n.idlist, s.gid) AS idlist
    FROM nyc_subway_stations s, next_stop n
    WHERE strpos(s.routes, 'Q') != 0
    AND NOT n.idlist @> ARRAY[s.gid]
    ORDER BY ST_Distance(n.geom, s.geom) ASC
    LIMIT 1)
)
SELECT ST_MakeLine(geom) AS geom FROM next_stop;
```

Which looks like this:



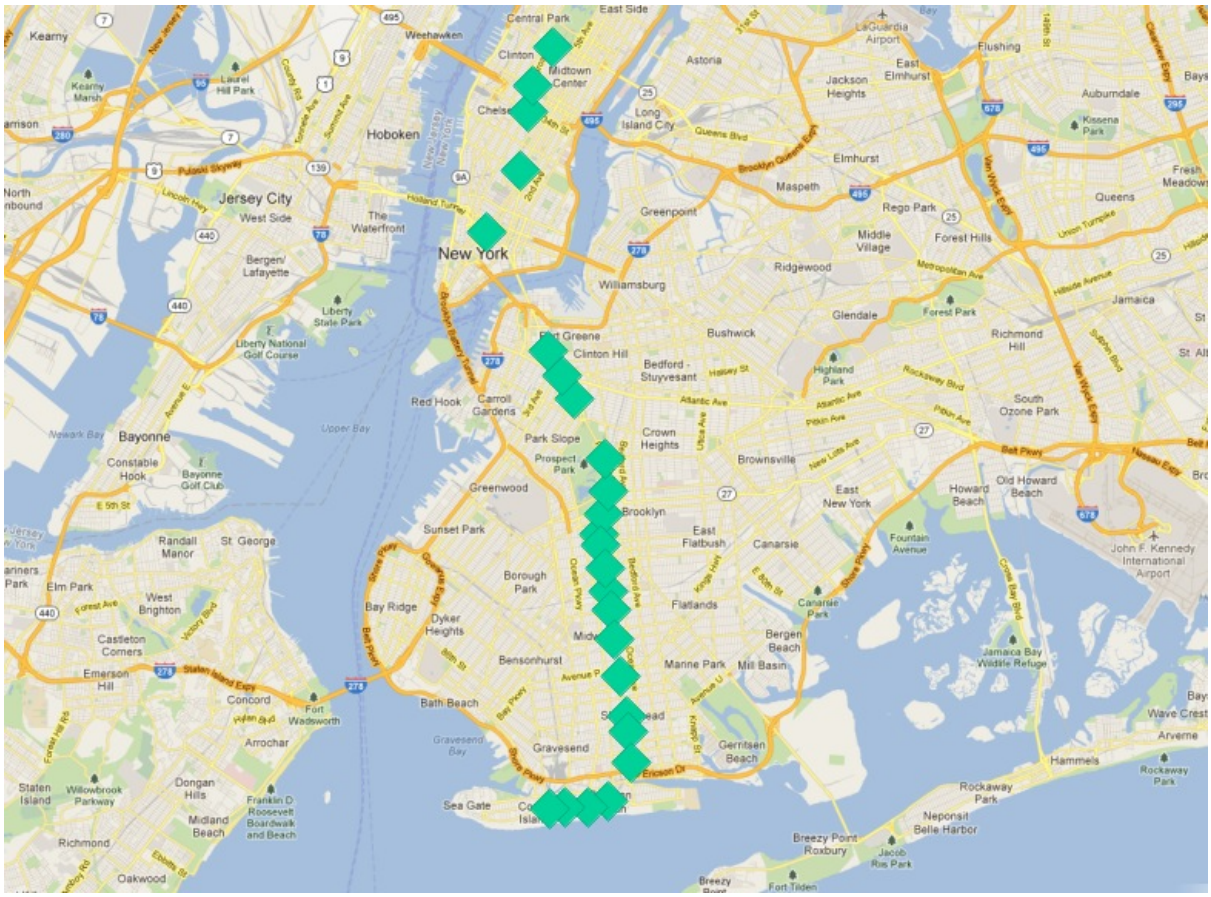
Success!

Except, two problems:

- We are only calculating one subway route here, we want to calculate all the routes.
- Our query includes a piece of *a priori* knowledge, the initial station identifier that serves as the seed for the search algorithm that builds the route.

Let's tackle the hard problem first, figuring out the first station on a route without manually eyeballing the set of stations that make up the route.

Our 'Q' train stops can serve as a starting point. What characterizes the end stations of the route?



One answer is "they are the most northerly and southerly stations". However, imagine if the 'Q' train ran from east to west. Would the condition still hold?

A less directional characterization of the end stations is "they are the furthest stations from the middle of the route". With this characterization it doesn't matter if the route runs north/south or east/west, just that it run in more-or-less one direction, particularly at the ends.

Since there is no 100% heuristic to figure out the end points, let's try this second rule out.

Note: An obvious failure mode of the "furthest from middle" rule is a circular line, like the Circle Line in London, UK. Fortunately, New York doesn't have any such lines!

To work out the end stations of every route, we first have to work out what routes there are! We find the distinct routes.

```
WITH routes AS (
  SELECT DISTINCT unnest(string_to_array(routes,',')) AS route
  FROM nyc_subway_stations ORDER BY route
)
SELECT * FROM routes;
```

Note the use of two advanced PostgreSQL ARRAY functions:

- **string_to_array** takes in a string and splits it into an array using a separator character. PostgreSQL supports arrays of any type, so it's possible to build arrays of strings, as in this case, but also of geometries and geographies as we'll see later in this example.
- **unnest** takes in an array and builds a new row for each entry in the array. The effect is to take a "horizontal" array embedded in a single row and turn it into a "vertical" array with a row for each value.

The result is a list of all the unique subway route identifiers.

```
 route
-------
 1
 2
 3
 4
 5
 6
 7
 A
 B
 C
 D
 E
 F
 G
 J
 L
 M
 N
 Q
 R
 S
 V
 W
 Z
(24 rows)
```

We can build on this result by joining it back to the `nyc_subway_stations` table to create a new table that has, for each route, a row for every station on that route.

```
WITH routes AS (
  SELECT DISTINCT unnest(string_to_array(routes,',')) AS route
  FROM nyc_subway_stations ORDER BY route
),
stops AS (
  SELECT s.gid, s.geom, r.route
  FROM routes r
  JOIN nyc_subway_stations s
```

(continues on next page)

(continued from previous page)

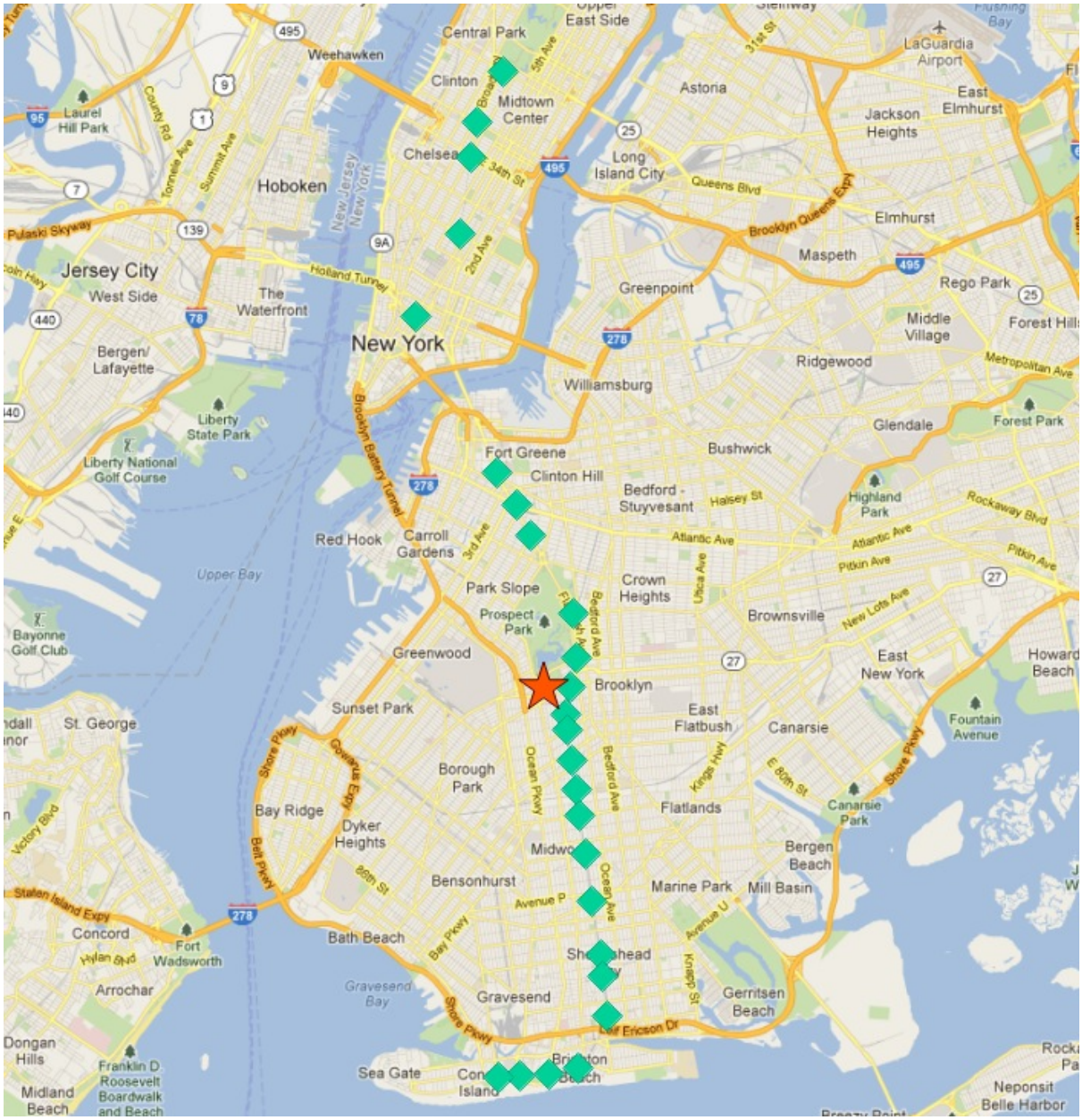
```
  ON (strpos(s.routes, r.route) <> 0)
)
SELECT * FROM stops;
```

```
 gid |                       geom                       | route
-----+--------------------------------------------------+-------
   2 | 01010000020266900000CBE327F938CD21415EDBE1572D315141 | 1
   3 | 01010000020266900000C676635D10CD2141A0ECDB6975305141 | 1
  20 | 01010000020266900000AE59A3F82C132241D835BA14D1435141 | 1
  22 | 0101000002026690000003495A303D615224116DA56527D445141 | 1
                           ...etc...
```

Now we can find the center point by collecting all the stations for each route into a single multi-point, and calculating the centroid of that multi-point.

```
WITH routes AS (
  SELECT DISTINCT unnest(string_to_array(routes,',')) AS route
  FROM nyc_subway_stations ORDER BY route
),
stops AS (
  SELECT s.gid, s.geom, r.route
  FROM routes r
  JOIN nyc_subway_stations s
  ON (strpos(s.routes, r.route) <> 0)
),
centers AS (
  SELECT ST_Centroid(ST_Collect(geom)) AS geom, route
  FROM stops
  GROUP BY route
)
SELECT * FROM centers;
```

The center point of the collection of ‘Q’ train stops looks like this:



So the northern most stop, the end point, appears to also be the stop furthest from the center. Let's calculate the furthest point for every route.

```
WITH routes AS (
  SELECT DISTINCT unnest(string_to_array(routes,',')) AS route
  FROM nyc_subway_stations ORDER BY route
),
stops AS (
  SELECT s.gid, s.geom, r.route
  FROM routes r
  JOIN nyc_subway_stations s
  ON (strpos(s.routes, r.route) <> 0)
),
centers AS (
  SELECT ST_Centroid(ST_Collect(geom)) AS geom, route
  FROM stops
  GROUP BY route
```

(continues on next page)

(continued from previous page)

```
),
stops_distance AS (
  SELECT s.*, ST_Distance(s.geom, c.geom) AS distance
  FROM stops s JOIN centers c
  ON (s.route = c.route)
  ORDER BY route, distance DESC
),
first_stops AS (
  SELECT DISTINCT ON (route) stops_distance.*
  FROM stops_distance
)
SELECT * FROM first_stops;
```

We've added two sub-queries this time:

- **stops_distance** joins the centers points back to the stations table and calculates the distance between the stations and center for each route. The result is ordered such that the records come out in batches for each route, with the furthest station as the first record of the batch.
- **first_stops** filters the **stops_distance** output by only taking the first record for each distinct group. Because of the way we ordered **stops_distance** the first record is the furthest record, which means it's the station we want to use as our starting seed to build each subway route.

Now we know every route, and we know (approximately) what station each route starts from: we're ready to generate the route lines!

But first, we need to turn our recursive CTE expression into a function we can call with parameters.

```
CREATE OR REPLACE function walk_subway(integer, text) returns geometry AS
$$
WITH RECURSIVE next_stop(geom, idlist) AS (
    (SELECT
      geom AS geom,
      ARRAY[gid] AS idlist
    FROM nyc_subway_stations
    WHERE gid = $1)
    UNION ALL
    (SELECT
      s.geom AS geom,
      array_append(n.idlist, s.gid) AS idlist
    FROM nyc_subway_stations s, next_stop n
    WHERE strpos(s.routes, $2) != 0
    AND NOT n.idlist @> ARRAY[s.gid]
    ORDER BY ST_Distance(n.geom, s.geom) ASC
    LIMIT 1)
)
SELECT ST_MakeLine(geom) AS geom
FROM next_stop;
$$
language 'sql';
```

And now we are ready to go!

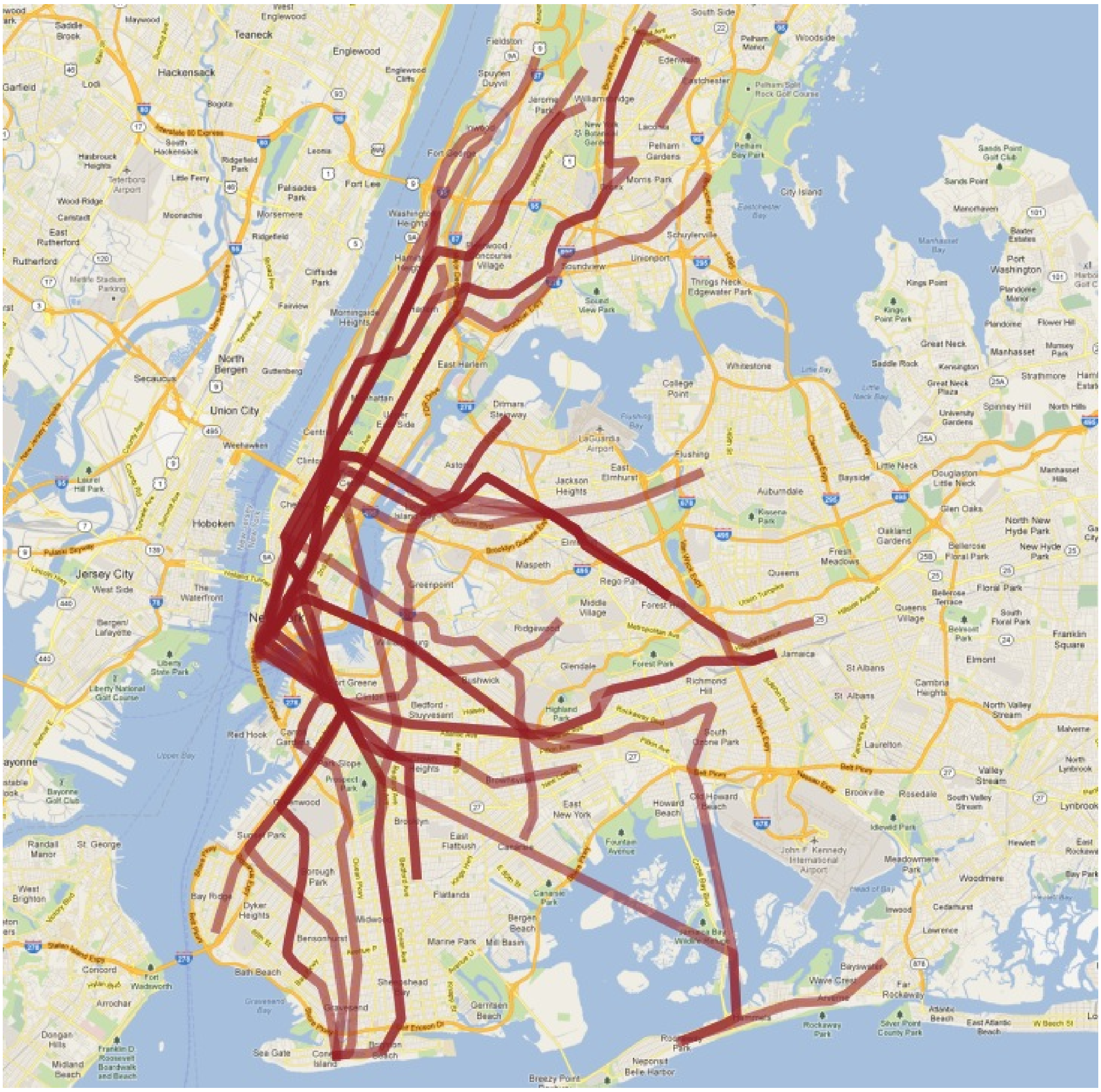
```
CREATE TABLE nyc_subway_lines AS
-- Distinct route identifiers!
WITH routes AS (
```

(continues on next page)

(continued from previous page)

```
  SELECT DISTINCT unnest(string_to_array(routes,',')) AS route
  FROM nyc_subway_stations ORDER BY route
),
-- Joined back to stops! Every route has all its stops!
stops AS (
  SELECT s.gid, s.geom, r.route
  FROM routes r
  JOIN nyc_subway_stations s
  ON (strpos(s.routes, r.route) <> 0)
),
-- Collects stops by routes and calculate centroid!
centers AS (
  SELECT ST_Centroid(ST_Collect(geom)) AS geom, route
  FROM stops
  GROUP BY route
),
-- Calculate stop/center distance for each stop in each route.
stops_distance AS (
  SELECT s.*, ST_Distance(s.geom, c.geom) AS distance
  FROM stops s JOIN centers c
  ON (s.route = c.route)
  ORDER BY route, distance DESC
),
-- Filter out just the furthest stop/center pairs.
first_stops AS (
  SELECT DISTINCT ON (route) stops_distance.*
  FROM stops_distance
)
-- Pass the route/stop information into the linear route generation␣
↪function!
SELECT
  ascii(route) AS gid, -- QGIS likes numeric primary keys
  route,
  walk_subway(gid, route) AS geom
FROM first_stops;

-- Do some housekeeping too
ALTER TABLE nyc_subway_lines ADD PRIMARY KEY (gid);
```

Here's what our final table looks like visualized in QGIS:



As usual, there are some problems with our simple understanding of the data:

- there are actually two 'S' (short distance "shuttle") trains, one in Manhattan and one in the Rockaways, and we join them together because they are both called 'S';
- the '4' train (and a few others) splits at the end of one line into two terminuses, so the "follow one line" assumption breaks and the result has a funny hook on the end.

Hopefully this example has provided a taste of some of the complex data manipulations that are possible combining the advanced features of PostgreSQL and PostGIS.

38.1 See Also

- PostgreSQL Arrays
- PostgreSQL Array Functions
- PostgreSQL Recursive Common TABLE Expressions
- PostGIS ST_MakeLine

CHAPTER

THIRTYNINE

APPENDIX A: POSTGIS FUNCTIONS

39.1 Constructors

`ST_MakePoint(Longitude, Latitude)` Returns a new point. Note the order of the coordinates (longitude then latitude).

`ST_GeomFromText(WellKnownText, srid)` Returns a new geometry from a standard WKT string and srid.

`ST_SetSRID(geometry, srid)` Updates the srid on a geometry. Returns the same geometry. This does not alter the coordinates of the geometry, it just updates the srid. This function is useful for conditioning geometries created without an srid.

`ST_Expand(geometry, Radius)` Returns a new geometry that is an expanded bounding box of the input geometry. This function is useful for creating envelopes for use in indexed searches.

39.2 Outputs

`ST_AsText(geometry)` Returns a geometry in a human-readable text format.

`ST_AsGML(geometry)` Returns a geometry in standard OGC *GML* format.

`ST_AsGeoJSON(geometry)` Returns a geometry to a standard GeoJSON format.

39.3 Measurements

`ST_Area(geometry)` Returns the area of the geometry in the units of the spatial reference system.

`ST_Length(geometry)` Returns the length of the geometry in the units of the spatial reference system.

`ST_Perimeter(geometry)` Returns the perimeter of the geometry in the units of the spatial reference system.

`ST_NumPoints(linestring)` Returns the number of vertices in a linestring.

`ST_NumRings(polygon)` Returns the number of rings in a polygon.

`ST_NumGeometries(geometry)` Returns the number of geometries in a geometry collection.

39.4 Relationships

ST_Distance(geometry, geometry) Returns the distance between two geometries in the units of the spatial reference system.

ST_DWithin(geometry, geometry, radius) Returns true if the geometries are within the radius distance of one another, otherwise false.

ST_Intersects(geometry, geometry) Returns true if the geometries are not disjoint, otherwise false.

ST_Contains(geometry, geometry) Returns true if the first geometry fully contains the second geometry, otherwise false.

ST_Crosses(geometry, geometry) Returns true if a line or polygon boundary crosses another line or polygon boundary, otherwise false.

CHAPTER

FORTY

APPENDIX B: GLOSSARY

CRS A "coordinate reference system". The combination of a geographic coordinate system and a projected coordinate system.

GDAL Geospatial Data Abstraction Library, pronounced "GOO-duhl", an open source raster access library with support for a large number of formats, used widely in both open source and proprietary software.

GeoJSON "Javascript Object Notation", a text format that is very fast to parse in Javascript virtual machines. In spatial, the extended specification for GeoJSON is commonly used.

GIS Geographic information system or geographical information system captures, stores, analyzes, manages, and presents data that is linked to location.

GML Geography Markup Language. GML is the *OGC* standard XML format for representing spatial feature information.

JSON "Javascript Object Notation", a text format that is very fast to parse in Javascript virtual machines. In spatial, the extended specification for GeoJSON is commonly used.

JSTL "JavaServer Page Template Library", a tag library for *JSP* that encapsulates many of the standard functions handled in JSP (database queries, iteration, conditionals) into a terse syntax.

JSP "JavaServer Pages" a scripting system for Java server applications that allows the interleaving of markup and Java procedural code.

KML "Keyhole Markup Language", the spatial XML format used by Google Earth. Google Earth was originally written by a company named "Keyhole", hence the (now obscure) reference in the name.

OGC The Open Geospatial Consortium (OGC) is a standards organization that develops specifications for geospatial services.

OSGeo The Open Source Geospatial Foundation (OSGeo) is a non-profit foundation dedicated to the promotion and support of open source geospatial software.

SFSQL The Simple Features for SQL (SFSQL) specification from the *OGC* defines the types and functions that make up a standard spatial database.

SLD The Styled Layer Descriptor (SLD) specification from the *OGC* defines an format for describing cartographic rendering of vector features.

SRID "Spatial reference ID" a unique number assigned to a particular "coordinate reference system". The PostGIS table **spatial_ref_sys** contains a large collection of well-known srid values and text representations of the coordinate reference systems.

SQL "Structured query language" is the standard means for querying relational databases.

SQL/MM SQL Multimedia; includes several sections on extended types, including a substantial section on spatial types.

SVG "Scalable vector graphics" is a family of specifications of an XML-based file format for describing two-dimensional vector graphics, both static and dynamic (i.e. interactive or animated).

WFS The Web Feature Service (WFS) specification from the *OGC* defines an interface for reading and writing geographic features across the web.

WMS The Web Map Service (WMS) specification from the *OGC* defines an interface for requesting rendered map images across the web.

WKB "Well-known binary". Refers to the binary representation of geometries described in the Simple Features for SQL specification (*SFSQL*).

WKT "Well-known text". Can refer either to the text representation of geometries, with strings starting "POINT", "LINESTRING", "POLYGON", etc. Or can refer to the text representation of a *CRS*, with strings starting "PROJCS", "GEOGCS", etc. Well-known text representations are *OGC* standards, but do not have their own specification documents. The first descriptions of WKT (for geometries and for CRS) appeared in the *SFSQL* 1.0 specification.

CHAPTER

FORTYONE

APPENDIX C: LICENSE

INDEX

C

G

J

K

O

S

W