Mid-Atlantic Chapter of the Urban and Regional Information Systems Association

Workshop

Spatial SQL

Instructor: John Reiser

October 16, 2024

Spatial SQL

A hands-on workshop on using spatial data in database management systems, focusing on PostgreSQL.

Author: John Reiser jreiser@njgeo.org

Getting Started

Online Resources

Links to other resources, the workshop data, and updates to any of the workshop materials will be available at https://learnspatialsql.com/

Downloading a digital version of this workbook from the website above is strongly recommended.

Pre-requisites

- Install PostgreSQL
- Install PostGIS extension
- Install and configure a SQL Client
- Install a Desktop GIS
- Load workshop data into your database

Installing PostgreSQL

PostgreSQL is an open-source, ACID-compliant, transactional relational database management system (RDBMS) that is available on Windows, MacOS, and Linux.

As of September 2024, the current stable release of PostgreSQL is **16**. While version 16 is preferred, you can use **PostgreSQL 13** or greater to complete this workshop. **PostgreSQL 15** is the latest version that is supported by ArcGIS Pro and ArcGIS 11.3. View ESRI's documentation on PostgreSQL requirements for more information.

> You should not use a PostgreSQL version that is end-of-life (EOL) even if it is supported by ArcGIS. It will no longer receive security updates. Check the PostgreSQL versioning policy to see upcoming EOL dates.

Downloading PostgreSQL: https://www.postgresql.org/download/

If you are using **Windows**, you will likely be directed to the EnterpriseDB installer for PostgreSQL. ESRI users through their ArcGIS downloads are also directed to the EnterpriseDB installer. Install the latest version of the PostgreSQL database - or the latest version compatible with ArcGIS, if that is important to you. As of September 2024, you should be installing a version of PostgreSQL 13 or later. The EnterpriseDB installer will also provide a **Stack Builder** application which will allow you to install PostGIS within your instance. More information on installing PostgreSQL on Windows is available on the Workshop web site.

If you are using **macOS** and would like a simple installer, you should instead use Postgres.app http://postgresapp.com/. This installer will include PostGIS. Using Postgres.app in a production environment is strongly discouraged.

Usernames and Passwords

When installing PostgreSQL (either the official distribution or the Postgres.app application on macOS) you will be prompted to set the password for the "postgres" user. For the purpose of this workshop, you can continue to use the "postgres" user, as it is a "superuser" account and will not be limited with what you are able to do to the database. Do not forget your password for the "postgres" account.

Some distributions of the PostgreSQL database, such as Postgres.app, may be configured to allow any user to connect without a password, provided they are logging into the database locally and not over a network connection. In this case, you will not need to specify a password when logging in.

Operating as the "postgres" or any other superuser in a production database without fully understanding what you are doing is very dangerous. As you hopefully move to adopt PostgreSQL in your organization, you should consult with a database administrator to determine the proper configuration of users, roles, and permissions.

Installing PostGIS

PostGIS is an extension to the PostgreSQL database that enables the storage, modification, and analysis of geographic data.

If you are using the EnterpriseDB installer on Windows, you must select PostGIS from the Stack Builder. More details are available on the Workshop web site.

If you are using Postgres.app on macOS, you do not need to install PostGIS. It is already installed.

Downloading PostGIS: http://postgis.net/install/

Install and configure a SQL GUI

If you are new to working with SQL and databases, a graphical user interface (GUI) can be very helpful. There are a variety of SQL GUIs and integrated development environments (IDEs) that you can use to work with a PostgreSQL database.

- **pgAdmin** https://www.pgadmin.org/
- **DBeaver Community Edition** https://dbeaver.io/download/
- **PSequel** http://www.psequel.com/
- and many more...

For this workshop, we will be using the pgAdmin GUI. pgAdmin was designed with PostgreSQL in mind. Unlike some of the other user interfaces, the terms used in the GUI will be Postgres-specific. pgAdmin can also be deployed as a single user, desktop client or as a web application, allowing multiple users access to a database.

We will be using the latest version of **pgAdmin 4**, which as of this writing is pgAdmin 4 v8. You can get pgAdmin from their download page: https://www.pgadmin.org/download/

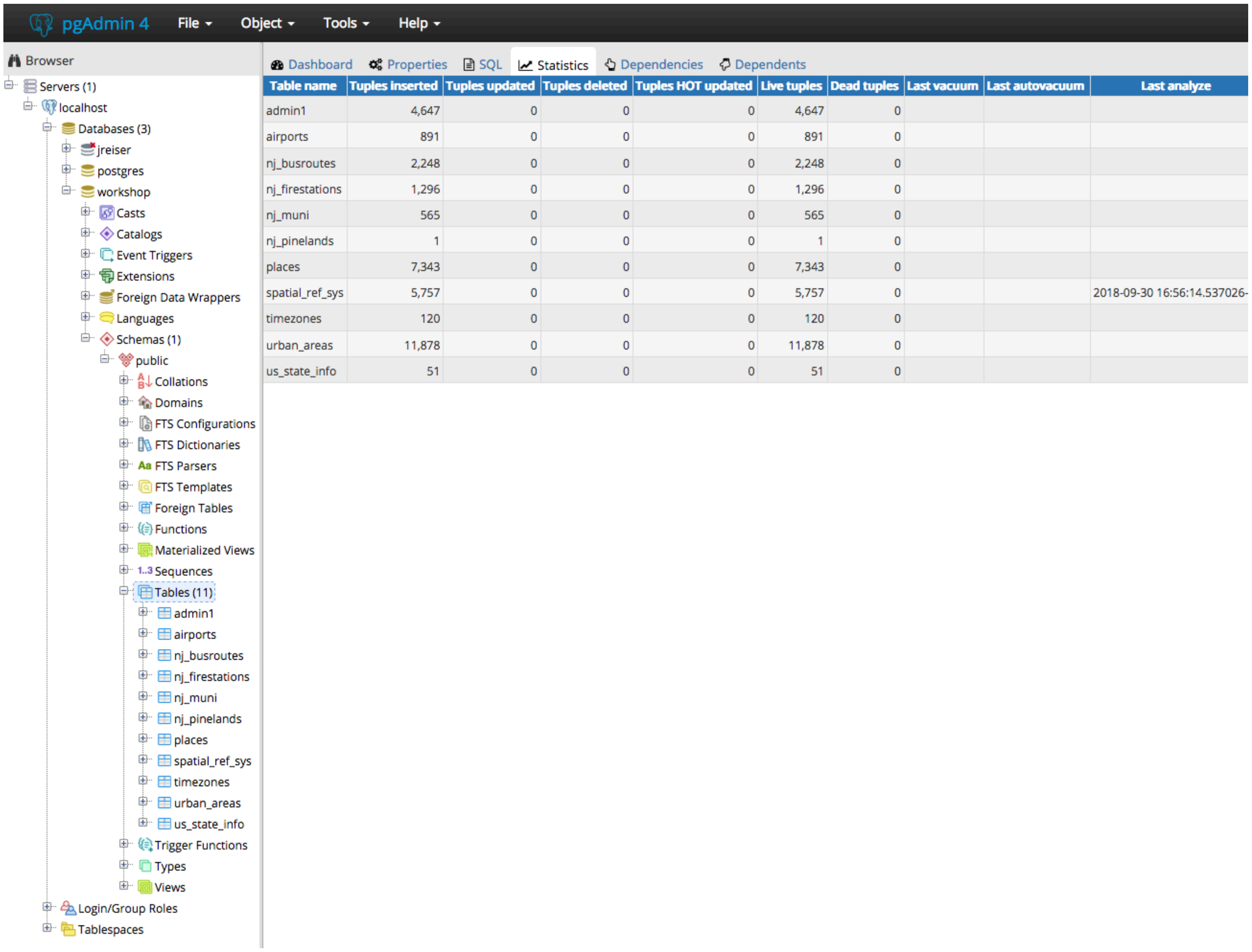


Table name	Tuples inserted	Tuples updated	Tuples deleted	Tuples HOT updated	Live tuples	Dead tuples	Last vacuum	Last autovacuum	Last analyze
admin1	4,647	0	0	0	4,647	0			
airports	891	0	0	0	891	0			
nj_busroutes	2,248	0	0	0	2,248	0			
nj_firestations	1,296	0	0	0	1,296	0			
nj_muni	565	0	0	0	565	0			
nj_pinelands	1	0	0	0	1	0			
places	7,343	0	0	0	7,343	0			
spatial_ref_sys	5,757	0	0	0	5,757	0			2018-09-30 16:56:14.537026-
timezones	120	0	0	0	120	0			
urban_areas	11,878	0	0	0	11,878	0			
us_state_info	51	0	0	0	51	0			

Installing the Workshop Data

To get you started with the workshop, we will be using data from the Natural Earth dataset. The tables used in the workshop examples have already been exported to a PostgreSQL "data dump" file. This is a custom format that can be used to quickly import data into a PostgreSQL database, preserving many data objects, attributes, and permissions.

You can download the Workshop Data File from the LearnSpatialSQL download page: https://www.learnspatialsql.com/downloads/.

Once you have downloaded the file, you can use pgAdmin to import the data into a new database for you. The data dump file contains an entire database snapshot, so when using the import capability of pgAdmin, make sure you also select `✓ Include CREATE DATABASE statement` that will create the `workshop` database for you.

The LearnSpatialSQL download page also contains a version of the workshop data in a plain-text SQL file that can be loaded to a database using psql or SQLite. A pre-configured SQLite version of the workshop materials are also available.

Install Desktop GIS

QGIS

If you do not have access to a desktop GIS, QGIS is an excellent open-source desktop GIS. You will be able to work with PostgreSQL tables and read and manipulate PostGIS spatial columns.

QGIS is available at http://qgis.org/

ArcGIS Pro / ArcGIS for Desktop

If you have ArcGIS Pro or ArcGIS for Desktop, you can connect to PostgreSQL databases. The necessary libraries are included with ArcGIS Pro and in ArcGIS for Desktop starting at version 10.4.

Using Amazon Web Services

An alternative to installing PostgreSQL on your own machine is that you can use a pre-configured instance of PostgreSQL and supporting applications on Amazon Web Services.

Note: using Amazon Web Services will require creating an account with AWS and using a credit card to cover the charges incurred. The estimated cost of running an AWS EC2 (virtual server) instance for 4 hours is likely to be less than 1 dollar. However, even when the server is off, you may be charged for the storage. Nonetheless, this is an easy way for you to get started with PostgreSQL (and cloud computing) if you do not have the ability or desire to install the software on your personal computer.

For more information on the AWS-hosted learning environment, please see https://learnspatialsql.com/cloud/.

Introduction to Spatial SQL

GIS data that you have used in the past has likely fallen into one of two categories:

- Local GIS data, where you must download and maintain it, but are free to alter it as you see fit.
- Web-based GIS services, where you can easily use it through the Internet, however, you typically do not have the ability to make changes to the data and how it is presented.

Connecting to a database is similar to a GIS service (e.g. ArcGIS Services, WMS, WFS, etc.), but with a few distinctions:

- While a database is often hosted on a different computer than the one you are using, it is likely on the same physical network.
- A client-server relationship, however the client and server are more tightly integrated.
- Tighter integration is due to the greater functionality provided to clients.

You will be connecting to a database server hosted on your same computer. You can use the special network name of "localhost" to refer to your own computer when connecting to the database.

Database Object Hierarchy

Think of a relational database handling data this way:

- A **record** is a single item that is composed of a list of properties. For instance, a record for a US State would contain the state's name, population information, date of admission to the Union, and a representation of its shape. Each of these properties would be stored in a **column**.
- A **table** is a set of *records* that have the same list of columns. A definition of the table's list of columns is called a **table schema**. You will define a table schema when you run a `CREATE TABLE` function to make a new table. When you run `CREATE TABLE AS SELECT...` the database infers the *table schema* from the set of records returned.
- In *PostgreSQL*, a group of *tables* that have a need for a logical grouping are placed together in a **schema**. A schema helps keep tables, views, types and functions organized within a database. *SQLite* does not support PostgreSQL-style schemas, as it is a single-user database.
- A **database** is a set of *schemas and tables*. Data can be easily passed between tables in different schema.

Data cannot be easily transferred between databases without making one or more connections to the databases participating in the data transfer.

In PostgreSQL, you can use the pg_dump and pg_restore tools to export and import data from databases. You can also use the powerful Foreign Data Wrappers to connect to other databases or non-Postgres sources of data, such as Oracle, CSV files, or even web services.

In this workshop, we will only be working with one database and we will have exclusive access to that database. However, your PostgreSQL instance can support many databases and provide access to many users over the network.

- A **database server** is software that provides access to the databases maintained on a computer. The database server handles networking and access control, so that only certain computers and users have access to the databases.
- A **database cluster** is a series of computers that collaboratively handle databases. Data is often transferred between database servers within the database cluster using server-level replication. Coupled with load balancers, database clusters are the brains behind many massive web applications today.

Think of it as a hierarchy, where elements of one level of the hierarchy contain all of the elements of the next hierarchical level.

Clusters > Servers > Databases > Schemas > Tables > Records

In practice, your database server may reside on your desktop GIS workstation, and you may only have one database on the server. For instance, you could have a database called "GIS" and keep your data grouped into schemas, such as a "flood" schema containing tables which in turn store map features related to the National Flood Hazard Layer. In that case, you'd have:

localhost Server > GIS database > flood schema > flood_zone table > flood zone record

Your desktop GIS knows how to retrieve, manipulate, and display the data within the records as GIS data, the same way as it would work with a file-based data format. We will find by the end of the workshop that having the database manage some of the work of processing and manipulating the spatial data, many opportunities for more efficient workflows and quality control checks will become apparent.

It may seem like a lot, but all of this technology working in concert enables some amazing things:

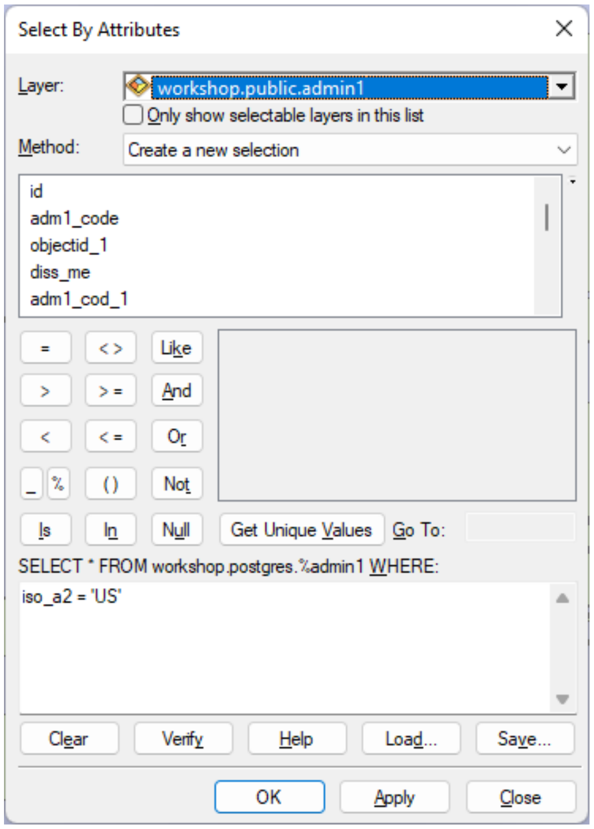
- Tens of thousands of simultaneous reads and writes.
- Multiple users that can share information and remain compartmentalized.
- ACID Compliance.
    - **Atomicity**: Transactions can be performed. Changes made during a transaction will only be stored in

the database once the transaction is committed. Transactions can be rolled back to the previous state.
  - **Consistency**: Transactions must contain valid data. If not, the transaction is rolled back.
  - **Isolation**: Allows for multiple users/transactions to work in concert without impacting other's work in unforeseen ways.
  - **Durability**: Once a transaction has been committed to the database, it will remain in that state until another successful transaction alters it.

To begin harnessing the power of databases, we will need to learn how to speak their language. The language used by most RDBMSs today is Structured Query Language, or SQL. It is a **declarative** programming language, setting it apart from other languages, such as Python and Java.

Speaking SQL

You have likely used some SQL previously when working with GIS. In ArcGIS, the **Select By Attribute** tool is very helpful in selecting a subset of features using a set of user-defined criteria.



SELECT * FROM workshop.postgres.%admin1 WHERE:

iso_a2 = 'US'

You may not have noticed it, but the bottom text area in the Select by Attribute window is labeled with:

```
SELECT * FROM table_name WHERE:
```

When using Select by Attributes, you are writing a `SELECT` statement, the first type of SQL statement we will discuss. Before we can begin speaking SQL, we will need to connect to our database.

Connecting to the Database

We will use both a desktop GIS and pgAdmin simultaneously, so that you can see both the visual representation of the data in QGIS or ArcGIS and the tabular, structural view of the data in pgAdmin.

The first data we will review is from the Natural Earth dataset. Throughout this assignment, we will be using the following data:

- Admin 1 - States, Provinces
- Populated Places
- Urban Areas

All of the Natural Earth data is at the 1:10m scale. Each dataset has been imported to the `workshop` database and stored within the public schema.

To get started with pgAdmin, refer to the documentation on connecting to a database.

> When you initially set up your database, you specified an administrator – typically the "postgres" account. You can use that account for the remainder of the workshop, however you will likely need to set up other users and roles if you are going to use this database in a multi-user, production environment.

If you need assistance with desktop GIS, the following links may be useful: - Connecting to PostgreSQL using QGIS - Connecting to PostgreSQL using ArcGIS

First SQL statements

SQL is a *declarative* language. You *declare* what data you would like to receive back from the database server. You can make simple or very complex statements to the database server that will return what you need.

The most basic request for data follows this form: `SELECT * FROM tablename;` . FROM and SELECT are called SQL clauses and they denote the types of data - whether within the database or provided by the user to alter the query - used in the SQL statement. While FROM and SELECT are almost always present when querying for data, the following clauses are available when constructing a SELECT statement.

- SELECT
- FROM
- JOIN
- WHERE
- GROUP BY
- HAVING
- ORDER BY
- LIMIT

These clauses, however, are executed in a specific order, as such:

1. FROM / JOIN
2. WHERE
3. GROUP BY
4. HAVING
5. SELECT
6. ORDER BY
7. LIMIT

This order of operations is important to consider as we move forward; as your queries become more complex, it will be difficult to troubleshoot queries that are invalid or inefficient. In many cases, violation of the order of operations will lead to invalid SQL or queries that are incredibly slow.

Let's start by exploring the `public.admin1` table that we've already viewed using GIS.

Remember, SQL statements take this basic form: `SELECT * FROM tablename;`, so let's make that request in pgAdmin. Once you are connected to the database, click on the SQL Query button and enter the following:

```
SELECT * FROM public.admin1;
```

In the lower half of the window, you will see all of the columns (the asterisk denotes all columns) and all of the records (we did not limit what was returned with a WHERE statement) for the `public.admin1` table.

Limiting the results

Let's add a WHERE statement to limit the number of records returned. We will use the contents of a column to limit the results to just those dealing with United States records.

Type the following into the SQL Query window and run the query. How many results are returned?

```
SELECT * FROM public.admin1;
```

It would be easier to see what is returned if we limit the columns returned to just the relevant columns. Add the `sov_a3, geonunit` columns to your SELECT query.

```
SELECT sov_a3, geonunit FROM public.admin1;
```

It would also be helpful to order the records by the contents of certain columns. Let's sort on the `geonunit` column.

```
SELECT sov_a3, geonunit
  FROM public.admin1
 ORDER BY geonunit;
```

This can also be written as:

`SELECT sov_a3, geonunit FROM public.admin1 ORDER BY 2;` with the number representing the column to sort. Count from the left-most column, starting with 1.

There's still a large amount of columns returned and many duplicates. You can use the `DISTINCT` keyword to tell the server that you want a unique set of records returned.

```
SELECT DISTINCT sov_a3, geonunit
  FROM public.admin1
 ORDER BY 1, 2;
```

Read through the returned records. What could you use to limit the results to just the US states? Now try the following query. The addition of the `name` column will return the names of the second-level administrative divisions.

```
SELECT DISTINCT sov_a3, geonunit, name
  FROM public.admin1
 ORDER BY 1, 2, 3;
```

From here, write a SQL statement that will return 51 rows (the United States plus the District of Columbia) with only 1 column: `name`. It should look similar to this:

```
SELECT name
  FROM public.admin
 WHERE [...]
 ORDER BY 1;
```

Note that the columns used in the WHERE statement are not required to be included in the list of columns returned.

Using Functions to Alter Output

Run the following query. It should only return one record, because New Jersey is one-of-a-kind.

```
SELECT name, shape
  FROM public.admin1
 WHERE name = 'New Jersey';
```

Taking a look at the shape column, you should see an unintelligible string of characters. This is a binary representation (Well-known Binary) of the Geometry data stored in the shape column.

In our SELECT statements, we can use a variety of functions to alter the values in those columns as they are returned by the database. Two functions, ST_AsEWKT() and ST_AsGeoJSON(), accept geometry data and return text representations of the data. Try out the following two statements.

```
SELECT name, ST_AsEWKT(shape)
  FROM public.admin1
 WHERE name = 'New Jersey';

SELECT name, ST_AsGeoJSON(shape)
  FROM public.admin1
 WHERE name = 'New Jersey';
```

Functions can also be nested, so that the results of one function can then be passed to the outer function in the nested list. For example, we want New Jersey's shape in WKT format, but we want it projected into New Jersey State Plane. Remember, functions can return data in different types and functions expect data of specific types, so the order in which the functions are called is very important.

ST_Transform() is a PostGIS function that changes the projection of the Geometry in a column to a specified projection. Projections are represented using Spatial Reference IDs (SRID). These IDs are stored, along with the projection information, in the `public.spatial_ref_sys` table.

While you could look up the SRID for New Jersey State Plane, I'll save you the trouble and let you know that the SRID is **3424**.

The output of the `ST_Transform()` function is Geometry binary data, which we can then pass into any function expecting Geometry data.

```
SELECT name, ST_AsEWKT(  ST_Transform(shape,3424)  )
  FROM public.admin1
 WHERE name = 'New Jersey';
```

You will see that the coordinates and the SRID are different from before. If you're unsure of the difference in coordinates, you can specify a column more than one time in a SELECT statement.

```
SELECT name,
    ST_AsEWKT(  ST_Transform(shape,3424)  ) as stateplane,
    ST_AsEWKT( shape ) as latlng
  FROM public.admin1
 WHERE name = 'New Jersey';
```

After each column name, you can specify a new column name. In the previous example, we changed the output name of the two `ST_AsEWKT()` functions to `stateplane` for the one with an inner `ST_Transform()` and `latlng` for the one in its native WGS84 coordinate system.

Another PostGIS function is ST_Centroid(). Given a line or polygon geometry, `ST_Centroid()` returns a point representing the center of that object. Can you create a SELECT statement that returns the center point of New Jersey in New Jersey State Plane?

Creating a New Schema

Let's now explore how to make new tables based on the results from a SELECT statement. Using GIS, take a look at the "places" data. You can see that the table has a large amount of columns, with populated places outside of the US. To work on this data, you will want to make a copy of the table in your own schema. However, when making the copy, you want to reduce the number of columns and select only the records that relate to US cities and places.

In your Query Window, type in the following:

```
CREATE SCHEMA workspace;
```

This will create a new schema in which you can store tables, views, and other database objects. When the query planner looks for database objects, it follows a search path, similar to the `PATH` variable in your operating system. To include this new schema in the path, issue the following command to your database:

```
ALTER DATABASE workshop SET search_path = "$user",workspace,public;
```

As an aside, the `"$user"` portion of the command above lets the database know that it should search a schema that shares its name with the current user first. While this might not be applicable for this workshop, it is good practice once you have multiple users in your database.

Going forward, to reference `mytable` table in your `workshop` schema, you will only need to refer to the table as `mytable` instead of `workshop.mytable`. You should keep in mind that if you create a table in your schema called `admin1`, the database will refer to `workspace.admin1` instead of `public.admin1`, because `workspace` is listed in the search path before `public`.

Creating a new table

First, the simple task should be to select just the US places. Use a where statement to limit the records returned to just those in the US.

We also want the following columns in our new table:

- **id** - the *Primary Key* to the table. Primary Keys are unique values that the database can use to keep track of records. Primary Keys are not required. Despite this, you should always have a primary key. Considering we're making a new table as a subset of an existing table, we can use the same column as primary key in the new table. Primary Keys can also be designated from multiple columns; this is called a *Composite Key*.
- **shape** - we want to map these points in the future, so we want to bring them over to the new table.
- **name** - name of the populated place.
- **adm1name** - Admin Level 1 name. In this case, it's state name. Rename this column in your SELECT statement to **state**.
- **pop_max**, **pop_min**, **pop_other** - columns containing different population estimates.
- **max_areami** - generous estimate of the size of the city/place in square miles. Rename this to **areasqmi**.
- **elevation** - estimated elevation of the city/place in meters.
- **timezone** - name of the time zone in which the city is located.

First, build your SELECT statement and confirm that it is valid. Run it to ensure you are happy with the results. Double check that you have the proper columns included and that you've correctly renamed the columns in the statement.

Once your SELECT statement is returning the desired data, it is very easy to make that a table within your own schema. You can use the `CREATE TABLE [tablename] AS SELECT...` statement to put the results of your SELECT into a table. It should look something like the following:

```
CREATE TABLE workspace.usplaces AS
  SELECT [list of columns]
  FROM public.places
  WHERE ISO_A2 = 'US';
```

Using Functions to Calculate New Values and Creating Views

We can also use aggregate functions to calculate statistics on records within a table.

Let's first create a SELECT query that counts the number of city/place points within each state. We will use the `count()` function and the `GROUP BY` clause to count up the number of points within each unique value in a column.

```
SELECT state, count(*)
  FROM workspace.usplaces
```

```
 GROUP BY state
 ORDER BY state;
```

How many points are in New Jersey? California? Alaska?

A count of points alone does not give us a good indication of population by state. Granted, we will be missing most of the population of a state if we only add up the population of the cities represented in this data, but it should give us a better reflection of population. We can use the `SUM()` function to sum up the population of the places in each state.

```
SELECT state, count(*), sum(pop_max)
  FROM workspace.usplaces
 GROUP BY state
 ORDER BY state;
```

What's the sum of the population column for New Jersey? Alaska? This is perhaps a better indication of density, however it's still not perfect. What if we wanted the average value of the population in the place points table? Can you create a SELECT that shows state name, average of the pop_max field, and count of points? Once you get the SELECT query working, create a **table view** of the query. A table view is essentially a query stored in the database that is presented to users as a table. There is no actual table, instead the view references other existing table(s) when called.

The difference in syntax for creating a new table from a SELECT and a new view is simply:

```
CREATE VIEW workspace.placestats AS SELECT [...];
```

PostGIS also has spatial aggregators, that can perform what would be considered a *Dissolve* in ArcGIS. `ST_Union()`, when used with a `GROUP BY clause`, will union the spatial features within the aggregation window and return a new set of spatial features.

Return to GIS and look at the admin1 features. Notice there is a `"region"` column in the admin1 table. Let's make a view that will merge the geometries of the US states together on the fly.

First, we want to limit our set of output records to just the US. We'll again use `iso_a2 = 'US'` to limit our set of records from approximately 4,000 polygons down to the 51 that make up the US States (and DC).

We'll also group on the contents of the `region` column. If you execute the following query, you'll see that there are only four values in the region column for our subset of data.

```
SELECT DISTINCT region FROM public.admin1 WHERE iso_a2 = 'US';
```

We should expect 4 records - and thus 4 multi-part polygons - returned from our SELECT query.

We will also need to provide our view with a unique ID number, so that GIS packages can use it as the primary key. The `public.admin1` table already has a primary key in the id column. However, we can't use the id in our SELECT query as-is, because it will then need to be put into the GROUP BY clause. If we group by a unique value, none of the output records will be merged. What we can do is take either the minimum or maximum of the id column for that set of records that share the same value in the region column. The functions `min()` and `max()` work with a GROUP BY clause and will return a unique value within the set. Then, when loaded into QGIS or ArcGIS, the product of `min()` or `max()` can be used as the ObjectID/Primary Key.

Create your view like the following:

```
CREATE OR REPLACE VIEW workspace.usregions AS
 SELECT min(id) AS objectid, st_union(shape) AS shape, admin1.region
   FROM admin1
  WHERE admin1.iso_a2 = 'US'
  GROUP BY admin1.region;
```

Confirm that your view works by viewing it in your desktop GIS. Note that it may take some time to draw because the database is storing the individual states and their geometries as records in a table. The regions are created dynamically upon retrieval. Later, we will explore **materialized views**, table-like objects that store the view's definition and the data behind it.

Exploring New Data

Return to pgAdmin to review the public schema, as we will begin using some of the additional tables in the schema. Let's first explore the `public.us_state_info` table, which has information about the 50 United States and the District of Columbia. There are columns in this table that match those in the `public.admin1` table, so you will be able to join this additional information to the spatial table.

Open the SQL Editor, run the following and inspect the output rows.

```
SELECT * FROM public.us_state_info;
```

The `us_state_info` table has 7 columns. Two of the columns, `population` and `houseseats`, are integer values, while the rest are text values. While there are some variations (bigint versus integer, character varying versus text) the types of columns are essentially text and integers. One column should stand out, as there is a more appropriate column type for storing date and time values.

Run the following statement, ordering the rows in the `us_state_info` table by date admitted to the Union.

```
SELECT * FROM public.us_state_info ORDER BY statehood;
```

- What's wrong with the sort order?
- Why do you think it is ordering the results in the way that it is?

Working with Different Data Types

You can use the `CAST()` function to dynamically change the type of the data. Rewrite the above statement, but now order on `CAST(statehood AS date)` and see how the results differ. Luckily, PostgreSQL is good at guessing how text should be interpreted as date or time values. You would experience problems, however, if the date was represented in text as "04/06/12", because the SQL interpreter would not know if the date is:

- MM/DD/YY: April 6, 2012 (or 1912)
- DD/MM/YY: June 4, 2012 (or 1912)
- YY/MM/DD: June 12, 2004 (or 1904)

When specifying dates, you should use a simple and unambiguous format for your dates. ISO 8601 outlines standards for representing dates. Absent another well-defined date format already in use, you should always use 8 digits to represent the date. This means that day 4 will be represented as "04". Secondly, you will specify a date using the YYYY-MM-DD format. For example, this workshop falls on October 16, 2024 or, "2024-10-16".

Let's now select only the records for states admitted to the Union after 1875. Simply doing the following will not work:

```
SELECT * FROM public.us_state_info WHERE statehood > 1875;
```

If you were to attempt this, you would receive the following back as an error:

```
ERROR:  operator does not exist: text > integer
```

The greater than and less than operators will work for numeric (integer and decimal) as well as for date and time values. You must explicitly define both operands as date values in order for the comparison to work.

Try the following:

```
SELECT * FROM public.us_state_info
```

```
WHERE CAST(statehood AS date) > CAST('1875-12-31' AS date);
```

You need to put the date value in single quotes to first make it text, otherwise, the SQL interpreter might attempt to subtract the month and day from the year and return an integer. Then you can use CAST to change the text to a date type.

There's also a PostgreSQL-specific shortcut for the CAST function. You can write it like so:

```
value::type
```

The previous SQL statement could then be written like so:

```
SELECT * FROM public.us_state_info
WHERE statehood::date > '1875-12-31'::date;
```

Know Your Types

Making sure the right **types** of values are used is incredibly important. The functions will often perform specific tasks based on the input types and will return values of a type based in the input.

For example, try the following:

```
SELECT 1/4;
```

What value is returned? What is the **type** of the value returned? While we would naturally expect to receive a decimal type when calculating 1/4, all the database knows is that you are working with two integer values and thus you want an integer value returned. You will need to be explicit with your type values so that you receive data back in the proper type.

Here's several ways you can have the database return the correct value of 0.25 for the calculation 1/4:

```
SELECT CAST(1 as numeric)/CAST(4 as numeric);
SELECT 1::numeric/4::numeric;
SELECT 1.0/4.0;
SELECT 1/4.0;   -- just the denominator works, too.
```

And you can always put in **type modifiers** (e.g. `character varying (length)` or `numeric(precision, scale)`) so that the database returns exactly what you want. If you know that you will always receive decimal values between 0 and 1 and you only want two significant digits, then you can put a modifier after the numeric type to return exactly that.

```
SELECT (1.0/4.0)::numeric(3,2);
-- returns 0.25

-- But you need to be mindful of the types inside the parentheses:
SELECT (1/4)::numeric(3,2);
-- returns 0.00
-- calculates 1/4 using integers, then formats numeric
-- the order of operations counts when dealing with functions!
```

Finally, you can name the new column on output, just as you'd name or rename any other column.

For example, if you were to run the following SQL statement:

```
SELECT state, statehood::date AS admission
FROM public.us_state_info;
```

You would cast the "statehood" column from text to date, then rename the column to "admission". This will become important when we make new tables from the results of SELECT statements. SELECT has no problem returning columns without names, however CREATE TABLE AS requires that every column in the SELECT query has a name.

Selecting from Multiple Tables

Let's work on bringing our `us_states_info` table into a new table that will incorporate a subset of the `admin1` data with the columns from `us_states_info`. You may want to revisit the `admin1` table with a `SELECT * FROM public.admin1;` to remind you of the numerous columns in that table.

Here's the columns we want to include in our final table:

- From `admin1`:
    - id (as our primary key)
    - shape
    - name
    - region
    - postal

- From `us_state_info`:
    - statehood (as a date type - and remember to name it 'statehood')
    - capital
    - largestcity
    - population
    - houseseats

While they will not be included in the final table, you will also need to use the following columns:

- **abbr** in `us_state_info` - You will need this field to match up records in admin1, using the postal column.
- **iso_a2** in `admin1` - Without this, you will not have a proper match. You need to limit the rows from `admin1` to just those dealing with the United States. For example, try the following SQL statement to see how many additional matches would occur when attempting to pair up the `us_states_info` record for Minnesota with values from `admin1`.

```
SELECT iso_a2, gn_name, postal, name
FROM admin1
WHERE postal = 'MN';
```

Writing our INNER JOIN

If you recall, INNER JOINs follow this structure:

```
SELECT [columns]
FROM [table 1]
JOIN [table 2] ON
[table 1].[column] = [table 2].[column];
```

Normally, when specifying columns from multiple tables, you include the table name and the column name. For example, `statehood` column would be referred to as `us_state_info.statehood`. That would be painful to write out each table name repeatedly, so you can use short **aliases** to refer to each table in an SQL statement. These aliases can be as short as one character. For example, we will use **"a"** to refer to `public.admin1` and **"i"** to refer to `public.us_state_info`.

Here's a nearly complete SQL statement.

```
SELECT a.id, a.shape, [rest of the columns]
FROM public.admin1 a
```

```
JOIN public.us_state_info i
ON a.postal = i.abbr
WHERE a.[...] ;
```

Using the list above, it's up to you to specify the remaining columns **and** add the correct WHERE statement.

Once you have added the missing components, run the SELECT statement to preview the results and then use CREATE TABLE ... AS SELECT ... to make a new table in your schema with the 51 records for the US. Save the new table in your schema as `usstates`, to go along with your `usplaces` table created in the last assignment. Refer back to the previous assignment if you need a reminder on CREATE TABLE ... AS syntax.

You should also know that is a shorthand for declaring an INNER JOIN. Remember, an INNER JOIN is a subset of the Cartesian Product of two tables, keeping only the records that have matching values in specific columns. The following statements would produce the same results:

```
SELECT x.col1, y.col2
FROM table1 x
JOIN table2 y
ON x.key = y.key;

SELECT x.col1, y.col2
FROM table1 x, table2 y
WHERE x.key = y.key;
```

It is recommended that you use the more explicit form of joining tables until you are very familiar with performing these tasks. If you do happen to make a mistake, the SQL interpreter will give you more helpful feedback when using the explicit form of JOIN.

One last bit of maintenance - we want to use the "id" column as our primary key. While it was the primary key in the admin1 table, the primary key designation does not follow it over to the new table. You will need to explicitly define the primary key for the table. You can do so through the ALTER TABLE statement.

```
ALTER TABLE workspace.usstates ADD PRIMARY KEY (id);
```

Performing Spatial JOINs

Now, we have two tables in our personal schema: `usplaces` (points) and `usstates` (polygons). There are several different ways to spatially relate features to one another. We'll explore Contains and Within to perform some analyses.

Count of Places within each State

Your `usplaces` table should contain a "state" field. We will use both the "state" field and the "shape" field to JOIN the `usplaces` and `usstates` tables together. We can also check to see if any of the values in the `usplaces.state` column are incorrect - assuming that we are confident in the accuracy of each table's spatial values.

A thought exercise to check your work

A couple things to do first. Get the row count for both of your tables and write them down somewhere. Knowing the count of rows on both tables will help you troubleshoot problems if you believe you are not receiving the expected results.

Given that usstates should have **51 rows** and usplaces should have **769 rows**, if you JOIN the two together without a limit or WHERE statement:

- SELECTing from both tables without a WHERE or JOIN will produce **39,219 rows: the Cartesian Product.**
- `usstates LEFT JOIN usplaces [...] GROUP BY usstates.state` should produce **51 rows exactly.**
- A result of **zero rows returned** means that the specified columns in the JOIN do not contain any matching values.
- It's certainly possible that the number of rows returned does not match the above values. You should use the row returned value as another check on your process. Ask yourself if the number of rows returned is what you expect **and** why you think you might have received the number returned.

Performing the Count

Let's perform a spatial join between the `usstates` and the `usplaces` to get a count of place points in each state. First, how do we ultimately want to show this count? A choropleth map would work, so we will need the state boundaries: `usstates.shape` and a count of points in each state: `COUNT(usplaces.shape)`. A state name field would help, too.

```
SELECT s.name, s.shape, COUNT(p.shape)
FROM usstates s
JOIN usplaces p ON ST_Contains(s.shape, p.shape)
GROUP BY s.name, s.shape;
```

The test for the JOIN is if the statement evaluated after "ON" is true. In this case, `ST_Contains()` returns True if `s.shape` *contains* `p.shape`. This is an example of a function that takes two geometry types and returns a boolean (true or false) type as a result.

Speeding Things Up

Indexes are a crucial component when building your database. An index allows the query planner to focus its search on certain sets of rows within a table. While we have been working on smaller datasets, we will naturally work on more complex data as we continue our efforts. Also, recalling the Cartesian Product and how every possible combination must be evaluated (or shortcut using an index scan) when performing a join, spatial functions are much more computationally intensive than comparing two integers.

Revisit your previous query, but this time, add a special command to the beginning:

```
EXPLAIN SELECT s.name, s.shape, COUNT(p.shape)
FROM usstates s
JOIN usplaces p ON ST_Contains(s.shape, p.shape)
GROUP BY s.name, s.shape;
```

`EXPLAIN` tells the Query Planner to return the steps it will take to return the desired result. Running the command above should produce a result similar to the one included below.

```
GroupAggregate  (cost=14444.85..14450.47 rows=51 width=18224)
  ->  Sort  (cost=14444.85..14446.13 rows=511 width=18224)
        Sort Key: s.name, s.shape
        ->  Nested Loop  (cost=0.00..10435.36 rows=511 width=18224)
              Join Filter: ((s.shape && p.shape) AND _st_contains(s.shape, p.shape))
              ->  Seq Scan on usplaces p  (cost=0.00..21.69 rows=769 width=56)
              ->  Materialize  (cost=0.00..20.77 rows=51 width=18168)
                    ->  Seq Scan on usstates s  (cost=0.00..20.51 rows=51 width=18168)
```

While there is valuable information in the output of the `EXPLAIN` command, it can be overwhelming. As you begin to troubleshoot your queries, looking for `Seq Scan` should be a top priority. A Sequential Scan of your table is going to be very expensive, especially as you deal with larger tables and more expensive functions, such as `ST_Contains()`.

We can reduce the time PostgreSQL needs to complete this query by building an index. Indexes can be thought of as specialized lookup tables, where the Query Planner can find certain values quickly. Indexes can be built on most data types, such as text strings, integers, dates, and spatial data.

Let's now build a GiST index on the `shape` spatial column in both tables.

```
CREATE INDEX ON workspace.usstates USING gist(shape);

CREATE INDEX ON workspace.usplaces USING gist(shape);
```

We can now re-run the `EXPLAIN` query and review the results. An example is included below; your results may not exactly match.

```
GroupAggregate  (cost=4137.02..4142.64 rows=51 width=18224)
  ->  Sort  (cost=4137.02..4138.30 rows=511 width=18224)
        Sort Key: s.name, s.shape
        ->  Nested Loop  (cost=0.00..127.53 rows=511 width=18224)
              ->  Seq Scan on usstates s  (cost=0.00..20.51 rows=51 width=18168)
              ->  Index Scan using usplaces_shape_idx on usplaces p
                    (cost=0.00..2.09 rows=1 width=56)
                    Index Cond: (s.shape && shape)
                    Filter: st_contains(s.shape, shape)
```

While there is still a Sequential Scan on `usstates`, the Query Planner is now using an Index Scan on `usplaces_shape_idx`, the index built on the `usplaces` table. The Query Planner decided that it should still scan every record in `usstates`, probably because it is only 51 records. If we performed this same test against `places` and `admin1`, it might rely on both indexes.

Also take note of the "cost" figures in both Query Plan explanations. The cost is a unitless value that denotes the relative cost of performing the steps of the query. The cost values are added up as you move up the hierarchy of steps. In comparing the two plans, you can see that the `Nested Loop` step - there and below are the steps necessary for our join - now has a greatly reduced cost.

Updating Time Zone Field

Let's perform a quick check to see if there are any missing values in the `usplaces.timezone` field. How can we quickly determine if there are any missing values in that field?

First, let's COUNT the number of NULLs in that field.

```
SELECT COUNT(*) FROM usplaces WHERE timezone IS NULL;
```

Returns 77 records.

In the `public` schema, you will find a table called `timezones`. The `timezones` table contains the timezone boundaries, plus a column called `tz_name` that follows the same format of the `usplaces.timezone` field. Additional reading if you're curious about the time zone database.

Let's SELECT all `usplaces` and bring in the `tz_name` field from `public.timezones`.

```
SELECT p.shape, p.name, t.tz_name
FROM usplaces p
JOIN public.timezones t
ON ST_Contains(t.shape, p.shape);
```

You should see that there are 769 rows returned. To check for completeness, run the following:

```
SELECT p.shape, p.name, t.tz_name
FROM usplaces p
JOIN public.timezones t
ON ST_Contains(t.shape, p.shape)
WHERE t.tz_name IS NULL;
```

And you should receive zero rows back. That's a good thing! It means that we have a time zone (supplied by `public.timezones`) for each record in `usplaces`. However, those values have not yet made it back into the `usplaces` table. Run the first query in this section again and you will see that a count of 77 null records will be returned.

We can UPDATE the `usplaces` table to replace all of the values in the `usplaces.timezone` column with the appropriate values from the `public.timezones` table. Now, because you created your `usplaces` table in your schema, you have the permission to update the table. Don't worry about making a mistake, you don't have the permission to change the tables in the `public` schema, so in the absolute worst-case scenario, you will need to DROP the tables in your schema and recreate them.

UPDATE statements take the following form:

```
UPDATE [table to be updated]
SET [column] = [new value]
FROM [other tables] -- note: this line optional
WHERE [condition]
```

We will update the records in `usplaces` using the `public.timezones` table, like so:

```
UPDATE usplaces p
SET timezone = t.tz_name
```

```
FROM public.timezones t
WHERE ST_Contains(t.shape, p.shape)
AND p.timezone IS NULL;
```

Our WHERE statement has two conditions. The first is that the `timezones.shape` value must contain the `usplaces.shape` value. This defines our spatial relationship between the two tables. Like referenced above, this is a shorthand way of defining an INNER JOIN. Second, we only want to update timezone values in `usplaces` if they are currently NULL.

Run the above statement. Then, run the first statement in this section again. You should receive a count of 0 rows, as there are no longer any NULL values in the `usplaces.timezone` column.

Creating a US-only Time Zone Map

While there are functions for testing spatial relationships, there are also functions for acting upon those relationships to create new data. We can use two similarly named functions to produce a new GIS table that will contain the spatial intersection of the Time Zones and the US States.

- ST_Intersects(a.shape, b.shape) tests whether geometries in Table A intersect those in Table B. This function returns a **boolean** type; TRUE if the geometries intersect, FALSE if they do not.
- ST_Intersection(a.shape, b.shape) performs a spatial intersection on two input geometries. This function returns a **geometry** type. If the two input geometries do not intersect, the function returns a NULL geometry.

We will use both of these functions to perform the intersection of these two tables.

- We will want the following columns from `usstates`:
    - name
    - region
    - postal
- We will want the following columns from `public.timezones`:
    - zone
    - tz_name
    - utc_format
- Finally, we will need to generate two new columns:
    - In place of a shape column from either table, we will use the `ST_Intersection()` function within the list of columns, calling this newly generated column "shape". Essentially, we will feed the two shape columns from both tables to `ST_Intersection()` and use the geometries returned

from the function as the new table's "shape" column.
- We cannot use either table's primary key as the primary key for the new table, as there will be duplicates. For example, Indiana is split by two timezones. If we use IDs from `usstates`, there will be at least two duplicate values for each state divided by a timezone. We cannot use the `timezones` IDs, because each timezone intersects several geometries in `usstates`, leading to even more duplicates.

To solve this issue, we need to create a new id for the new table. PostgreSQL has **sequences**, specialized counters for dealing with this type of issue. We can make a new temporary sequence to populate the id column of our new table. We will use a temporary sequence because we will only use it for this one operation. Marking it as temporary saves you the step of explicitly deleting it later using DROP SEQUENCE. It will be removed automatically when you disconnect from the database.

Let's first perform a dry run by using just a SELECT. Once we're happy with the output of the SELECT statement, we will create the sequence, update the SELECT and use it in conjunction with CREATE TABLE ... AS.

```
SELECT 0 AS id, -- placeholder for the sequence
  ST_Intersection(s.shape, tz.shape) as shape,
  s.name, s.region, s.postal,
  tz.zone, tz.tz_name, tz.utc_format
FROM usstates s, public.timezones tz
WHERE ST_Intersects(s.shape, tz.shape);
```

When you run the above, you should receive 80 rows. Inspecting the rows, you'll see that there are some duplicate state names. This is okay, because some states split between timezones will have a record for the portion of the state in one zone and another record for the portion in the different zone.

Let's now make this an actual table.

```
-- creates the sequence
CREATE TEMPORARY SEQUENCE newid START WITH 1;

-- creates the new table
CREATE TABLE ustimezones AS
SELECT nextval('newid') AS id,
  ST_Intersection(s.shape, tz.shape) as shape,
  s.name, s.region, s.postal,
  tz.zone, tz.tz_name, tz.utc_format
FROM usstates s, public.timezones tz
WHERE ST_Intersects(s.shape, tz.shape);
```

```
-- adds the id column as the primary key
ALTER TABLE ustimezones ADD PRIMARY KEY (id);
```

To confirm that you successfully performed the intersection, you can now run:

```
SELECT * FROM ustimezones;
```

And see that you have a new table, with 80 rows and an ID field with unique values. Congratulations!

Materialized Views

Most modern DBMS systems support a **Materialized View**, where the materialized view is defined like your would a conventional table view, but the data is stored along with the view definition. Materialized views can provide you with the benefits of a view, allowing you to reference multiple tables or functions and the performance benefits of single-table access to the data. Indexes can be created on a materialized view in the same manner as a database table. The downsides of a materialized view are that they take up additional space and required additional planning with regard to refreshing the data.

Upon creating a Materialized View, the data referenced in the `CREATE MATERIALIZED VIEW` DDL will be queried and stored. Unlike a regular view, changes to the underlying data will not automatically appear when querying your materialized view. You will need to issue `REFRESH MATERIALIZED VIEW` in order to update the data within the materialized view. If a refresh is started with the "concurrently" flag (e.g. `REFRESH MATERIALIZED VIEW CONCURRENTLY`) then the materialized view will be able to be referenced through other queries and views while the refresh is underway. In order to be able to refresh concurrently, the materialized view *must* have a unique index defined.

When considering using a materialized view, you will need to assess a variety of needs and potential pitfalls. You may need to reduce the time a complicated query needs to run. Alternatively, you might want to snapshot your data, so that you are only analyzing records from before midnight today. In these cases, you will likely use a view for creating the analysis or structure of the data, then creating your materialized view as `CREATE MATERIALIZED VIEW mv_data AS SELECT * FROM v_data_source;`. You may also need to create a view that wraps the materialized view, so that your users access the view only, freeing you to make changes to the materialized view (potentially swapping it out) without your clients knowing.

```
CREATE MATERIALIZED VIEW mv_data_2023 AS SELECT ... ;
CREATE VIEW v_last_year_data AS SELECT * FROM mv_data_2023;

CREATE MATERIALIZED VIEW mv_data_2024 AS SELECT ...
CREATE OR REPLACE VIEW v_last_year_data AS SELECT * FROM mv_data_2024;
```

While a simplified example, you will often find that as you build your database and start to weave a web of interdependencies, it will become harder to unravel as you have more users accessing more views and other objects.

Foreign Data Wrappers

NOTE: Foreign Data Wrappers rely on several external dependencies. It may be difficult to get the necessary libraries working reliably on Windows. The Windows Server instance on Amazon Web Services is properly configured if you encounter difficulty in installing the libraries on your own Windows-based computer.

Many modern DBMS systems allow for the server to access data stored outside of its databases, presenting the data as database tables. PostgreSQL offers this feature as **Foreign Data Wrappers (FDW)** allowing you to reference a variety of external data sources. You can make an FDW that references a table in a remote PostgreSQL database, or MSSQL, Oracle, Access and many other database-like sources. Alternatively, there are FDWs that allow you to reference web-based resources, like an ArcGIS Server or an API endpoint serving up GeoJSON.

We will create a Foreign Data Wrapper around the USGS Earthquakes API so that as earthquakes are observed and provided over the USGS API, they will appear within a table as records with geospatial coordinates.

To set up a Foreign Data Wrapper, you will need to create two new types of objects: a Foreign Server and a Foreign Table. The `CREATE SERVER` statement denotes that you will be using a Foreign Data Wrapper, specifically the `ogr_fdw` wrapper that is included with PostGIS. Documentation on the ogr_fdw extension: https://pgxn.org/dist/ogr_fdw/

```
CREATE EXTENSION ogr_fdw; -- enable the extension.

CREATE SERVER usgs
  FOREIGN DATA WRAPPER ogr_fdw
  OPTIONS (
    datasource 'https://earthquake.usgs.gov/fdsnws/event/1/query.geojson?
    minmagnitude=2.5&orderby=time',
    format 'GeoJSON');
```

You will also write a `CREATE FOREIGN TABLE` statement that defines the fields in the data returned.

```
CREATE FOREIGN TABLE earthquakes (
    fid bigint,
    geom Geometry(PointZ,4979),
    id varchar,
    mag double precision,
    place varchar,
    time bigint,
    updated bigint,
    tz varchar,
    url varchar,
    detail varchar,
    felt integer,
    cdi double precision,
    mmi double precision,
    alert varchar,
    status varchar,
    tsunami integer,
    sig integer,
    net varchar,
    code varchar,
    ids varchar,
    sources varchar,
    types varchar,
    nst integer,
    dmin double precision,
    rms double precision,
    gap double precision,
    magtype varchar,
    type varchar,
    title varchar
) SERVER usgs
OPTIONS (layer 'OGRGeoJSON');
-- "OGRGeoJSON" is the default name provided to a layer loaded using the GeoJSON OGR driver.
```

After your foreign table is created, try a basic select statement to review the results.

```
SELECT * FROM public.earthquakes;
```

You could also open the database in QGIS (or ArcGIS, if available to you) and compare the data in the `earthquakes` table with the USGS Earthquakes web page: https://earthquake.usgs.gov/.

You may notice that this table does take longer than your other tables to query. That is because it is retrieving the data from the public API every time you query the foreign table. To improve the performance, you could create a materialized view on the foreign table in order to maintain a local - but static - copy of the data.

```
CREATE MATERIALIZED VIEW mv_earthquakes AS
SELECT * FROM public.earthquakes;

CREATE INDEX sidx_mv_earthquakes ON mv_earthquakes USING gist(geom);
```

When querying your new `mv_earthquakes` materialized view, you should see that the performance has greatly increased. This is at the expense of having new data at query-time. You will need to update the materialized view through `REFRESH MATERIALIZED VIEW mv_earthquakes;` on a frequency that suits your needs.

The USGS Earthquake example uses GeoJSON, but the OGR FDW extension can access other sources, if your underlying GDAL/OGR libraries were compile to support them. Here are a two examples using different sources provided by different services.

The National Weather Service provides current weather alerts as well as several other weather products through their [OGC Web Services page](https://www.weather.gov/gis/cloudgiswebservices): https://www.weather.gov/gis/cloudgiswebservices. Under "WFS/WCS" you can find the endpoint that will provide vector data related to the current warnings. The link to the WFS service is included in the `CREATE SERVER` statement below.

```
CREATE SERVER nws_weather
  FOREIGN DATA WRAPPER ogr_fdw
  OPTIONS (
    datasource 'WFS:https://mapservices.weather.noaa.gov/eventdriven/services/WWA/
    watch_warn_adv/MapServer/WFSServer?request=GetCapabilities&service=WFS',
    format 'WFS');

CREATE FOREIGN TABLE warnings (
    fid bigint,
    shape Geometry(MultiSurface,3857),
    gml_id varchar,
    objectid integer,
    message_id varchar(254),
    hazard_abbreviation varchar(2),
    alert_type varchar(1),
    forecast_office varchar(4),
    event varchar(4),
    issuance varchar(25),
    expiration varchar(25),
    onset varchar(25),
    ends varchar(25),
    url varchar(254),
    message_type varchar(3),
    hazard_type varchar(40),
    gis_file_date timestamp,
    gis_ingest_date timestamp
) SERVER nws_weather
OPTIONS (layer 'watch_warn_adv:CurrentWarnings');
```

ArcGIS Server REST endpoints can also be queried through the OGR FDW extension. It may take some URL mangling to get the appropriate link to use in the `CREATE SERVER` statement. An easy workflow would be to locate the "Query" link on an ArcGIS Server endpoint, set the `outFields` to only the fields you want to include (or `*` to return all fields) and set the `where` clause for the query as well, as some may require it. You can simply use `1=1` as the where predicate to retrieve all records if it is required.

```
CREATE SERVER njogis_muni
  FOREIGN DATA WRAPPER ogr_fdw
  OPTIONS (
    datasource 'ESRIJSON:https://services2.arcgis.com/XVOqAjTOJ5P6ngMu/ArcGIS/rest/services/
    NJ_Municipal_Boundaries_3424/FeatureServer/0/query?where=1%3D1&returnGeometry=true
    &outFields=*&f=pjson',
    format 'ESRIJSON'
  );

CREATE FOREIGN TABLE fdw_njmunis (
  OBJECTID INT,
  MUN CHARACTER VARYING (28),
  COUNTY CHARACTER VARYING (10),
  MUN_LABEL CHARACTER VARYING (35),
  MUN_TYPE CHARACTER VARYING (12),
  NAME CHARACTER VARYING (40),
  GNIS_NAME CHARACTER VARYING (35),
  GNIS CHARACTER VARYING (8),
  SSN CHARACTER VARYING (4),
  MUN_CODE CHARACTER VARYING (4),
  CENSUS2020 CHARACTER VARYING (10),
  ACRES DECIMAL,
  SQ_MILES DECIMAL,
  POP2020 INT,
  POP2010 INT,
  POP2000 INT,
  POP1990 INT,
  POP1980 INT,
  POPDEN2020 INT,
  POPDEN2010 INT,
  POPDEN2000 INT,
  POPDEN1990 INT,
  POPDEN1980 INT,
  Shape__Area DECIMAL,
  Shape__Length DECIMAL,
  Shape Geometry(MultiPolygon,3424)
) SERVER njogis_muni
OPTIONS (layer 'ESRIJSON')
;
```

Alternatively, you can use the `IMPORT FOREIGN SCHEMA` functionality to bring in all of the available tables from a foreign server. While helpful, it will attempt to create table names based off of the layers in the remote service. GeoJSON and ESRIJSON servers will only have one table, named as such.

If you are not entirely sure of the data within the remote service and would like a quick import, you should make a new schema in which you will import the tables. This way you can review the results and if there are tables you would like to preserve, you can inspect their properties and manually revise the SQL to create the foreign table. A generic example:

```
CREATE SERVER remote_gis_service
  FOREIGN DATA WRAPPER ogr_fdw
  OPTIONS (
    datasource '...'
  );

CREATE SCHEMA import_gis;

IMPORT FOREIGN SCHEMA ogr_all -- special keyword to import all
FROM SERVER remote_gis_service INTO import_gis;
```

Try exploring some additional GIS servers for data you could bring into PostgreSQL.

Additional Assignments

In the next section, we will be using the New Jersey specific data. Before you move on, confirm that you have been able to create a `usstates` table in your schema, that you have updated your `usplaces` table to remove the NULLs from the timezone field and that you have created a `ustimezones` table.

I strongly suggest you reach out for assistance if the previous steps have not been clear.

New Jersey Reports

Using the SQL concepts you learned above, create tables or views to answer the following questions.

Count of Fire Stations within Municipality

How many fire stations are there in each NJ municipality?

Produce a table called `fire_station_count` that contains 564 rows and the following columns:

- Municipal Code (unique 4 character string: for example Atlantic City is '0102')
- County Name
- Municipality Name
- Count of Fire Stations within each Municipality

For extra credit: include the municipality's geometry in the new table. Make a basic choropleth map of the number of fire stations in each municipality using QGIS or ArcGIS.

Municipalities Along Bus Routes

NJ Transit is exploring how changes to its bus service will affect municipalities. To begin this analysis, the research team has asked you to assemble a few basic reports.

Create a table called `camden_routes` that is a simple list of Route Numbers that pass through Camden City.

You will need to make a Spatial Join between `nj_muni` and `nj_busroutes` using `ST_Intersects()`. You will also need to use WHERE to limit the municipalities to just Camden City. You only need to select the `public.nj_busroutes.line` column to get a list of routes. Also, the list should be ordered and unique (using the DISTINCT keyword).

Create another table called `route410_munis` that is a simple list of municipality names where the Route 410 bus passes through the municipality.

You will again make a Spatial Join between `nj_muni` and `nj_busroutes` using `ST_Intersects()`. You will need to use a WHERE to limit the bus routes to just those where `LINE = 410`. The table should have two columns, `"county"` and `"mun"` from the `nj_muni` table and the rows should be ordered by county name, then municipality name.

Municipalities Within the Pinelands

The Pinelands boundary encompasses a very large area in South Jersey. The boundary, in many cases, cuts through a municipality instead of following the municipal boundaries.

Make two tables that contain county and municipal names. The first table, called `pinelands_munis` will contain the county and municipal names for every municipality that *intersects* the `nj_pinelands` boundary. The second table, called `pinelands_within` , will have the same schema as `pinelands_munis` (county and municipal name), but will only contain records for the municipalities that are *completely contained* by the `nj_pinelands` boundary.

Urban Municipalities

Let's return to the Natural Earth data and see how we can incorporate that into our New Jersey data. There is the `urban_areas` data that denotes areas of significant urban development. Let's compare it to the New Jersey municipalities data to see how many NJ municipalities intersect or completely fall within the urban areas.

There is one major catch - the Natural Earth data is in WGS84 and the New Jersey data is in New Jersey State Plane. We can confirm that using the following queries:

```
SELECT DISTINCT ST_SRID(shape)
  FROM nj_muni;
```

```
SELECT DISTINCT ST_SRID(shape)
  FROM urban_areas;
```

The `ST_SRID()` returns the Spatial Reference ID, a numerical value that represents the coordinate system of the geometry in the shape column. The `DISTINCT` clause de-duplicates the result set, that way we only see one numerical value instead of a long list of the same value, as `ST_SRID()` is executed on each value in the shape column.

You will see that the values are different. WGS84 is represented as 4326 while New Jersey State Plane is 3424. More information about these Spatial Reference IDs can be found at http://spatialreference.org/.

PostGIS provides you with a function to transform a geometry to a different coordinate system. `ST_Transform()` takes two arguments, a geometry and an integer SRID, and it returns the passed geometry in the new projection.

Write a query that returns a list of municipality names and their counties that are completely contained by the urban areas. Then, write a query that returns the municipalities that are completely outside of the urban area.

Here are a few hints:

- Consider the types of JOINs you can use. You might want to consider an INNER JOIN for the "contains" query, but that will not work for a query where you want the records that do not spatially intersect.

- `ST_Transform()` can be used in the `JOIN ... ON` clause, nested within `ST_Contains()`, `ST_Intersects()`, or any other function that expects a geometry.
- Either `ST_Transform(nj_muni.shape, 4326)` or `ST_Transform(urban_areas.shape, 3424)` could work, but you might need to experiment with either (but not both!) to get your query to work.

Airports within a Distance from New Jersey

In the `airports` data from Natural Earth, there is only one airport located in New Jersey. You could quickly confirm the airport by writing a query using `airports` and `nj_muni`. How can we write a query that returns a list of airports within a certain distance (10 miles or 52,800 feet) from New Jersey?

One way to write such a query is to use the `ST_Buffer()` function, then use the resulting buffers to see if any of the records in `airports` intersect or are contained by the buffers from New Jersey. `ST_Buffer()` accepts two arguments, a geometry and a distance to create the buffer, in the same linear coordinates as the input geometry's coordinate system. For New Jersey State Plane, the second argument to `ST_Buffer()` would be in US Feet.

You might be inclined to write a query like this:

```
SELECT a.name, a.iata_code
  FROM airports a
  JOIN nj_muni n
    ON ST_Intersects(a.shape,
        ST_Transform(ST_Buffer(n.shape, 5280*10),4326)
       )
```

While the above is perfectly valid, it is very inefficient. If you request the query plan with `EXPLAIN`, you will see that it is performing a sequential scan on both data sets, performing a very expensive function (buffer and transform) on each row of `nj_muni`, over every row in the `airports` table. This query will take a very long time to complete and you may want to consider the how to reduce the number of items in the Cartesian Product of `nj_muni` and `airports`.

First, we're evaluating if something is near New Jersey by comparing it to the 564 municipalities that make up the state. We should evaluate the overall outline of the state, not the individual municipalities, some of which are not near the state border.

We could also create a new table to store our "nj_buffered" data, that represents New Jersey thus:

- All 564 municipalities merged into one polygon.
- Buffered the specified distance

- Transformed to 4326, WGS84

However, we might not want to keep a table on disk with this information. We might also be on a system where we do not have the privileges to create our own tables. How could we "trick" the query planner into performing the expensive tasks only once?

A Common Table Expression, also known as a "WITH" clause, allows us to effectively create a temporary table for the purpose of completing a query. The temporary table or tables created in a WITH clause only exist within the scope of the query. Each temporary table in the WITH clause can be referenced in the main query in the same way any regular table or view can be referenced.

Here's a shell of a query that you can use to write your optimized query to locate airports within 10 miles of New Jersey:

```
WITH nj_buffered AS (
  SELECT [functions to process NJ municipalities] as SHAPE
    FROM nj_muni
)
SELECT a.name, a.iata_code
  FROM airports    a
  JOIN nj_buffered n ON [spatial join type]
```

WITH clauses are very important to understand when optimizing your queries in PostgreSQL. Most database systems have a method for the operator to override how the query planner will plan the execution of a query. Oracle has "hints" - special comments embedded in the query that tell the Query Planner to use a different method to bring the data together. Previously, PostgreSQL used the WITH clause to act as an optimization fence. Starting in version 12, the Query Planner may rewrite the CTE to be an inline subquery if it will perform better. CTEs can still be materialized, as well as allow for recursive queries.

If a precalculation step is needed in other contexts outside of a single query, you can consider making the CTE/WITH clause portion its own materialized view. This allows for precalculation of values and performance increases can be realized through applying indexes to the materialized view.

In an earlier step, we used `ST_Transform()` to reproject the records in a table to a different coordinate system. You could use a **materialized view** to create a view that performs the projection on a base table and then stores the results on disk, for easier retrieval. As the only difference between the two "tables" is the reprojected geometry data, a materialized view makes sense as any updates on the base table will be present in the reprojected data after running `REFRESH MATERIALIZED VIEW`.

Airports and Weather Alerts

Earlier in the section on foreign data, we explored the National Weather Service API that returns active weather advisories. Can you create a list of US airports that are currently experiencing a weather advisory (if any)?

You may need to make a view that can employ `ST_Transform()` to reproject the geometry data received, so that you can join it with the airports data. You may also want to make that same view the source to a materialized view to improve performance.

Next Steps

Hopefully, this workshop has achieved its goals in explaining how databases can be further integrated into your GIS workflow. There still remains much to learn; with any available time, we can discuss and walk through the following:

- Importing Data in to PostgreSQL/PostGIS
- Exporting Data to Different Formats
- Differences Between PostGIS and ArcGIS Enterprise Databases
- Data Warehousing
- More uses of Materialized Views
- Triggers and functions for Quality Control

Closing

Directly manipulating spatial data using SQL can provide for many new opportunities for streamlining your work. Queries to load, extract, analyze, and process data can be run on a schedule or be integrated into other workflows. Getting the most out of the technologies in your toolbox will enable you to do far more than you may have previously thought.