Learn SQL

Learn SQL

Written by: Dave Fowler
Reviewed by: Matt David

Table of Contents

Basic SQL

Quick Introductory SQL Concepts

So you wanna learn SQL huh? You've come to the right place. This interactive SQL tutorial is designed to get you querying as quickly as possible.

I think it's best to just dive right in, but it's going to be incredibly beneficial to go over just a few quick concepts first (trust me, we get to running your first SQL on the very next page).

SQL

SQL might seem intimidating but it's really fairly easy to understand. SQL stands for Structured Query Language and simply put, it's a search language for you to instruct a database about what information you'd like retrieved from it.

Just think of it as an advanced, really structured google search. For example in Google you might ask something like

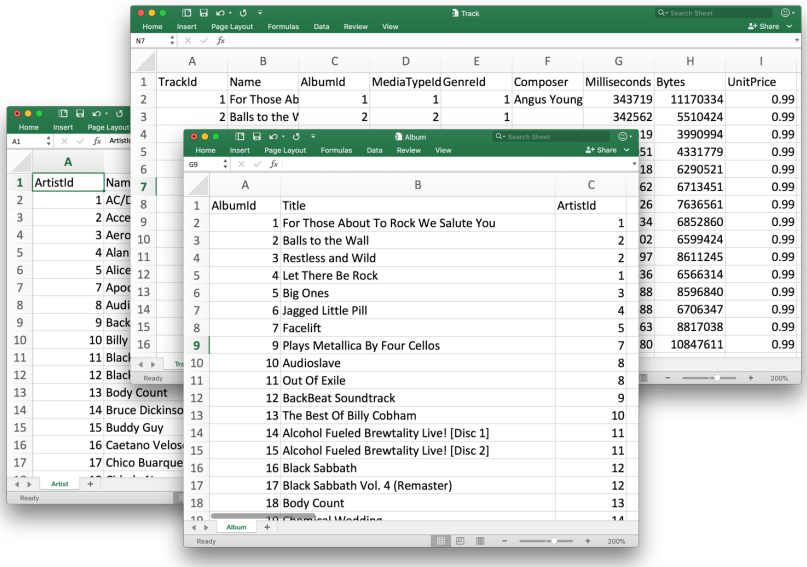
And in SQL, if you had a database with that information in it, the equivalent question might be answered with something like

```
SELECT age FROM presidents WHERE name = 'Barack Obama';
```

Don't worry about understanding the above query yet, you'll get that in no time.

Tables - for those familiar with Excel

Databases organize data in different **tables**. I'm assuming that most people reading this are familiar with Excel or some other Spreadsheet software. This makes it easier to explain what a table is as a Table is basically just a Spreadsheet of data. It has columns with names of fields, and then rows holding the actual data.



In this tutorial we'll be using an example data set that has a bunch of information on *tracks*, *albums* and *artists* in a music collection. Most databases architects will typically split those items into their own tables rather than group them all in one. You'll learn all about linking them together when we get to the section on Table Relationships and JOINs.

SQLBox (Interactive Query Editor)

For this tutorial we've built a tool for you to run SQL commands and see the results along with each lesson. It's a big black box like this so that you can try out each lesson, and play with other things you're curious about along the way.

Having SQLBox means that you don't have to spend a long time setting up your own environment just to get started. Playing with SQL and data is the best way to learn it. Try out running your first SQLBox queries by hitting "Run SQL" below.

After you run a query, a table with the results of your query will show up below it. All of the queries you run on this tutorial are being executed against a real PostgreSQL.

SQLBoxes that have a quiz to them will have a checkbox to their left. Once the answer is correct, the checkbox will be checked! Some of these will have a Hint you can view, and if you ever get really stuck feel free to email us for some help.

Let's get started with our first tutorial on SELECTs.

SELECT

Alright, let's get down to some SQL!! The first command we're going to learn is SELECT as it's the first instruction you need for any SQL statement that's fetching data. There are other starting commands like INSERT and CREATE, but most interactions with databases are SELECTing data.

Let's start with the most simple query and just select a value back. The SQL Box is there for you to try running your SQL in. You can put any SQL you want in there, and don't be afraid as you're not going to break anything. Try whatever you'd like. Experimentation is the best way to learn! Here's an example query we'll start with:

```
SELECT 42;
```

So first, try typing this statement in the SQLBox below and hit "Run SQL" to return the number 42.

Awesome, you just returned a number. There are other things you can fetch besides numbers like "strings" of characters. Try running this one or choosing your own string to return.

```
SELECT 'hello world!'
```

or

```
SELECT 'We are just getting started.';
```

Note that each query needs to end in a semi-colon. That's just how the database knows that you're done giving it instructions. The SQL Box isn't picky about it so you can get away without using it, but other tools you use may be a bit more strict.

Math

While we're playing with numbers we can point out that SQL can also instruct a database to do some math on a result. Try out some queries like these:

```
SELECT 2 + 3;
```

```
SELECT 5 * 12;
```

```
SELECT 164 / 8;
```

It can also concatenate strings

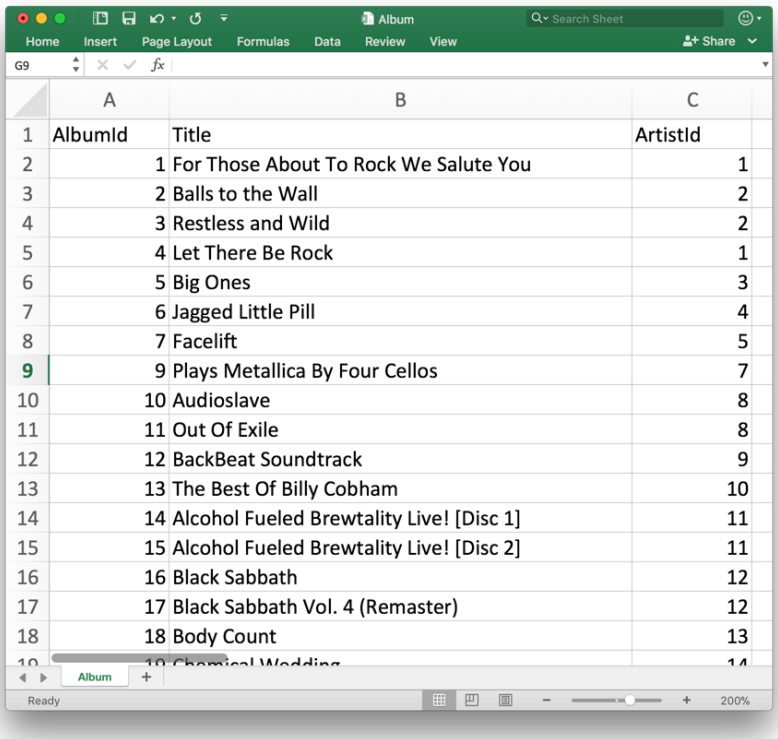
And we'll get more into DATE and TIME queries later but here's a quick example of selecting a date, which besides numbers and strings is another common data type category in PostgreSQL.

Here we just SELECTed data that we typed in ourselves. Obviously SQL would be quite useless if that's all it did, but next we'll cover how you can choose where to SELECT data FROM!

FROM

So now you know how to SELECT data but not yet how to choose where to get that data FROM. Let's get into the real stuff and SELECT data FROM a specific table.

In our example database we have a table called *albums*, which holds info on some music albums. It has three columns, id, title, artist_id. Here's what it looks like in Excel:



To get data that's in this table we need to specify what columns we want to SELECT and FROM where we want to select it. So let's try to get a list of all the album titles we've got stored. We can use the following template to do so:

```
SELECT [stuff you want to select] FROM [the table that it is in];
```

Let's start with a simple one and query for everything (all of it!) from the *albums* table.

Look at all that data! Notice at the bottom of the table we've paginated it for you so it doesn't take up the whole page. All of the columns and rows in the table *albums* have been fetched. You can see the table above looks similar to what the data looks like in the Excel image above.

But of course we don't have to query for all of the columns if we don't want. If we wanted to just get all the album *title*s and *id*s (we didn't care about *artist_id*s) we can query for just those columns.

Notice that the columns will come back in the order that you list them in. Try reversing the column order in the above query by selecting *id* first and then *title*.

* Splat

Sometimes it's annoying to have to list out all the columns that you want to fetch. If you simply want all the columns available SQL has the `*` shortcut. The `*` is called a "splat" and is a handy, frequently used shortcut to get all columns.

There are a lot of other Tables in our example database like *artists* and *tracks*. See if you can use the SELECT * FROM [tablename] structure to explore some of those tables.

Now it's getting interesting right? Right now though we're getting a list of all the results in the table back. We need to learn how to filter, group, manipulate and limit these results.

ORDER BY

By default results are returned in the order that they're stored in the database. But sometimes you'll want to sort them differently. You can do that with the "ORDER BY" command at the end of your queries as shown in the expanded version of our SQL template here

```
SELECT [stuff you want to select] FROM [the table that it is in] ORDER BY [column you want t
```

For example, the following query shows all the tracks ordered by the *album_id*. Try sorting it by other columns. Can you modify it to be sorted by their *name*?

You can list multiple things to ORDER BY, which is useful in the case where there are a lot of duplicate rows. In tracks for instance we can order all of the data by the *composer* and then by how long the song is (*milliseconds*) by listing both of those sorting columns.

Try reversing the order of the colums above (`ORDER BY milliseconds, composer`) and you'll see what happens with the reverse prioritization of first sorting by milliseconds.

ASCending and DESCending Order Direction

By default things are sorted by ascending order. You can choose to reverse that order by specifying DESC, for descending. Similarly if you'd like to specify that you want ASCending you use ASC.

To test your skills, try getting all the *tracks* in order of most expensive to least expensive:

LIMIT and OFFSET

If want to LIMIT the number of results that are returned you can simply use the LIMIT command with a number of rows to LIMIT by.

```
SELECT * FROM artists LIMIT [Number to Limit By];
```

For example

```
SELECT * FROM artists LIMIT 3;
```

This ensures only the first 3 results are returned. Besides returning less results, LIMITing queries can greatly reduce the time they take to run and make your database administrator a lot less angry with you.

Give it a try by fetching yourself the first *6* rows of the *artists* table:

OFFSET

You can also specify an OFFSET from where to start returning data.

```
SELECT * FROM artists LIMIT 5 OFFSET [Number of rows to skip];
```

Say you want to get 5 artists, but not the first five. You want to get rows 3 through 8. You'll want to add an OFFSET of 2 to skip the first two rows:

```
SELECT * FROM artists LIMIT 5 OFFSET 2;
```

Here's a challenge for you. Write a query to fetch the Artists in rows 10 through 20:

Browsing the SCHEMA

The word *SCHEMA* is used to describe a collection of tables and their relationships in your database. A database instance may have several different schemas. When you're working with a set of data, it's useful to be able to browse that schema to get a sense for what data is available to you.

You can browse a schema visually using popular database interfaces like PGAdmin, Postico and Chartio, or in a text-based manner by using SQL itself.

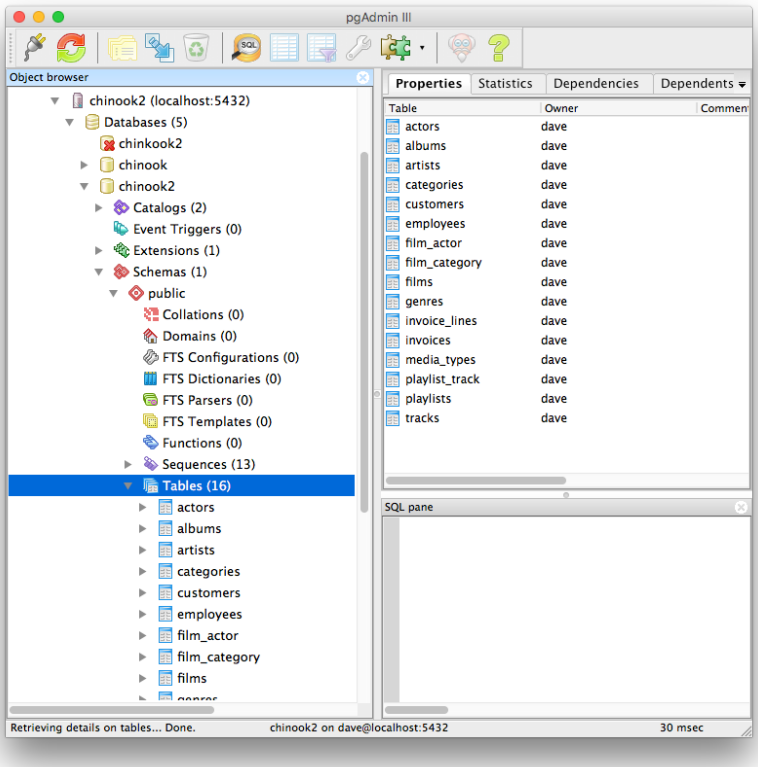
Typically using a visual tool is much easier, but it's totally up to you. Here we're going to quickly cover both, but if you already have a handle on one of the visual editors and are comfortable with finding out what schema is available feel free to skip this part and move on to the Basic SQL Practice Grounds.

Visual Schema Browsing

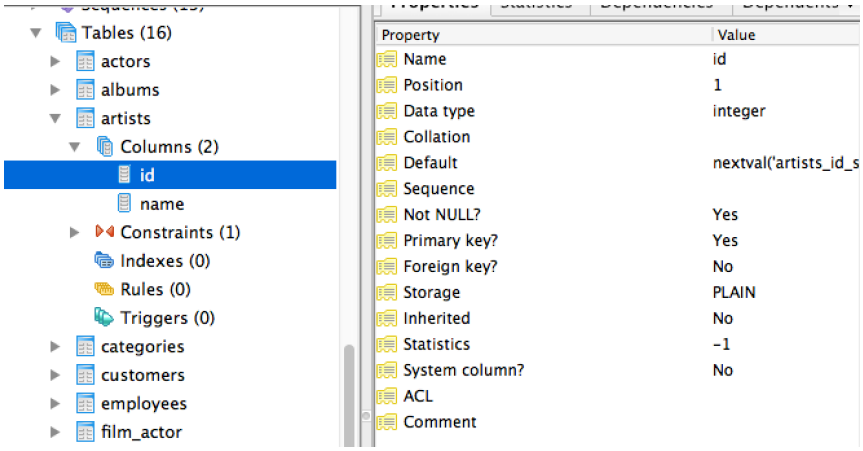
If you're using a good visual interface to PostgreSQL, browsing the schema can be really easy. Below we also show you how to browse the schema in SQL, but unless you just can't use one of these tools I highly suggest you browse the schema visually. It's just easier.

Schemas in PGAdmin

Once connected to a database, you can expand the trees in the left sidebar in PGAdmin to find the database, schema, tables and columns available:

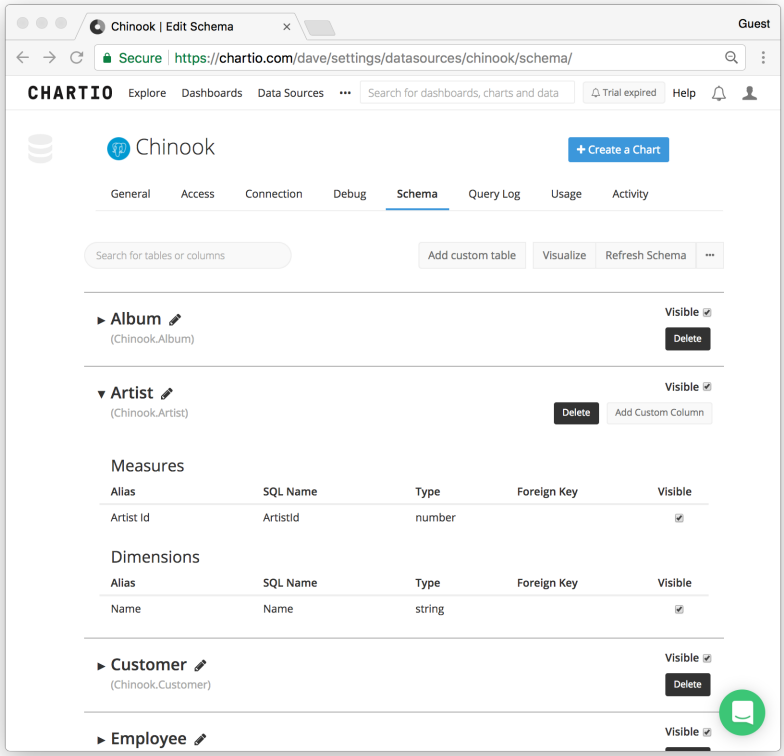


The "Properties" tab in the right top of the interface will display all of the extra properties that the *information_schema* holds on the table or column including default values, data type, and more.



Schemas in Chartio

Chartio's schema viewer simply lists the tables in the Schema tab of any data source connection.



Each table can be expanded to show the columns underneath. In Chartio you can actually change the name/alias, define relationships and create custom tables and columns. This isn't mapped back to database, but used only for the Chartio Visual Data Explorer.

Clicking on *"Visualize"* from the data source Schema page will also create a nice visualization of all of the tables, with their columns listed. In this view, relationships that are defined are also drawn as connections from one table to another.

Schema Browsing using SQL

Using psql or any other SQL interface you can browse your schema information. PostgreSQL is really clever in that all of the schema information is simply stored in a few tables that you can query like any other table. The information is all in a schema called *information_schema*. The table holding all the information on what tables are available is called *tables*. Let's take a look at what's available there:

The above will get all the tables from all schemas. If we want to look at only the tables in our chinook dataset we can query for only things in the `public` schema. The `public`

schema FYI is the default. We do this by just adding a condition on the *table_schema* column.

You'll notice the *artists*, *albums* and *tracks* tables we've been playing with so far in our tutorial, but look at all those others we've been holding out on! There's also *actors*, *employees*, *genres*, etc. We'll dig into looking at the columns in those tables next, but first take a moment to change the query above to look at what tables are available in PostgreSQL's *information_schema* schema.

You'll see that one of the tables available in the *information_schema* is *columns*, and as you might guess, like *tables* held the info on what tables are available *columns* holds the info on the columns.

Let's take a look at the columns in the *tracks* table with the condition `table_name = 'tracks'`.

As shown PostgreSQL stores all kinds of information about each column including the *data_type*, *character_maximum_length*, various precision options and an optional default value.

Sampling a Few Rows

Another great (and probably easier) way of checking out what columns and data is available in a table is to simply grab a few rows of it. Don't grab all the data, it may be a really big table and we don't need it all! Let's take a look at what's available in the *playlists* table with a limited SELECT *.

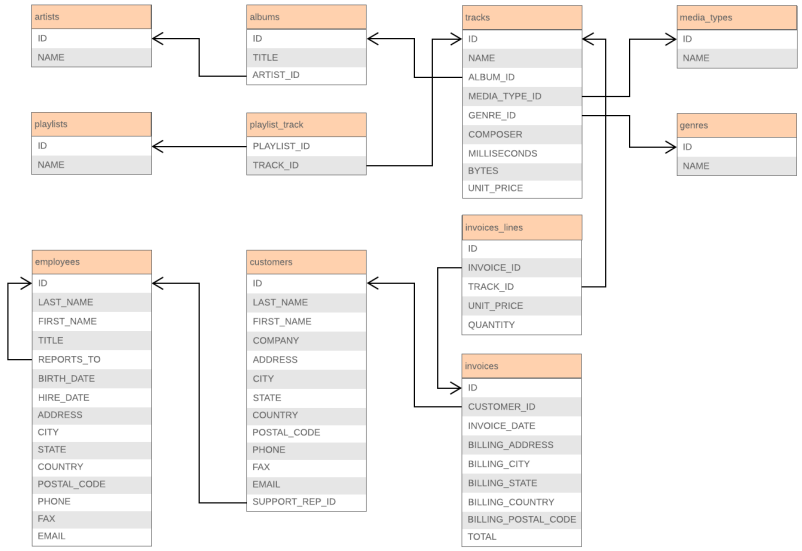
And of course if you want to see how much data there is in there you can run other diagnostic queries like a COUNT(*).

psql Schema Shortcuts

If you're using psql as your connection to your database there are a number of helpful schema browsing shortcuts. The following are to me the most useful for schema browsing and, if interested, you can find the full list of psql commands here.

Shortcut	Description
\d	list of all tables
\d+	list of all relations
\d [table name]	list of the columns, indexes and relations for the [table name]
\dn	list of all schemas (namespaces)
\l	list of all databases
\z	list tables with access privileges

Also if you want to refer to visual representation of the schema we have provided it here:



Basic SQL Practice Grounds

You're through the basics of SQL! This is a great place to stop and get more practice on what you've learned so far. Here we've constructed a list of challenges to give you that practice. Take some time to go through these before moving on to the Mid-Level SQL section.

A few things to keep in mind when going through

- If no specific columns or values are called out to return, assume that it's asking for all the columns (splat `*`).
- If it does ask for specific information like "names", only return that column. If you return other things as well it won't be able to match the correct answer.
- If you're having trouble with a question use the 'Hint' button. If you're really having trouble or think that the answer might be wrong, send us a note at support@chartio.com.
- We check if things are correct not by the query you wrote, but by the results that are returned. This is the best way as there are often a few different ways to get the same result.

Good luck!

Q. Fetch the first 8 rows of the *albums* table.

references: select limit

Q. Fetch the 12th through 32nd (21 total) rows of the *tracks* table.

references: select limit

Q. Fetch the 9th through 49th (41 total) rows of the *artists* table.

references: limit

Q. Get the *name*s of all of the *artists*.

references: from

Q. Get the *name*s of all of the *artists* in reverse alphabetical order. (Z to A)

references: order-by

Q. Get only the last 4 names of the artists sorted alphabetically.

references: order-by limit

Q. Get the rows of the 20 longest tracks.

references: order-by limit

If you've completed all of these CONGRATULATIONS! You're proficient and fluent enough in SQL now to complete a significant portion of analytic and transactional queries.

Mid-Level SQL

WHERE

LIMITing queries is one way to filter down result sets, but we can get a lot more specific with the WHERE clause. The WHERE command is followed by the conditions you'd like to filter by.

```
SELECT * FROM artists WHERE [Filter Conditions];
```

Conditions

Conditions are simply statements that are either true or false. The database takes these statements and evaluates them across all the rows as it scans through your tables and only returns the results that are true.

Let's say for instance that we'd like to see the name of the artist who's id is 85. The condition would be `id = 85`. Try the condition by running the following query:

The query instructed the database to scan the artists table and fetch all the rows where the condition (`id = 85`) was true. As you can see, the only artist with id 85 is Frank Sinatra.

Or if we wanted to lookup all the information on 'Santana' the condition would be `name = 'Santana'`

```
SELECT * FROM artists WHERE name = 'Santana';
```

Try getting the information on 'Kiss':

And another quiz: return the *id* from one of my favorite albums in High School, 'American Idiot'.

Multiple matches

In the above examples we were querying on unique fields so we were only getting one answer in response. That's not always the case however. In the *tracks* table many different tracks belong to the same album, and you can see that in the tracks database there is an *album_id* column. For example if we want to get all of the tracks belonging to an album, who's *id* is 89 we could run:

89 just happens to be the same *album_id* as "American Idiot" had. We've just pulled all the *tracks* from the album American Idiot! We'll get more into how we can JOIN this data based on the common key of *album_id* in a later section.

AND - Requiring Multiple Conditions

In SQL you can filter by any number of conditions. You can add additional conditions by using the AND operator between each new condition. Let's take the query we wrote above and say we only want the tracks from album 89 (American Idiot) AND were also composed only by Green Day themselves.

See if you can modify the query above to also filter on tracks that are longer than `200000` milliseconds.

OR - Requiring Any Condition

You can also use OR to define multiple WHERE conditions when you care whether not both but at least one of the conditions is true. For example, the following query returns tracks that are composed by either Green Day OR AC/DC.

NOT

You can invert a condition by simply putting the *NOT* operator infront of it. For example, the following queries for everything that is NOT composed by Green Day.

Ordering and Parenthesis

You can use any number of OR and AND commands in conjunction to describe your conditions. Just like in math class, SQL has an order of operations. An AND is essentially a

logic multiplication], and an OR is a logic addition, so ANDed conditions are preformed first, and then the ORs.

Also like math, you can use parenthesis to specify the order of operations. As a best practice it's good to use parenthesis wherever it seems like the logic and order might not be too clear. To explore let's attempt to pull all the *tracks* composed Green Day AND any track by AC/DC that is over 240,000 milliseconds.

Notice the use of parenthesis making it clear that we only wanted the longer AC/DC songs. You can see that the Green Day songs under 240,000 milliseconds are still listed. If we change the parenthesis however, the logic applies the millisecond condition to all Green Day songs as well.

Practice

Test your skills out and see if you can query for all *tracks* with price greater than a dollar and a genre (*genre_id*) of 22;

Now see if you can query for all *tracks* with price greater than a dollar and a genre (*genre_id*) of either 22 or 19;

There are a lot more Operators than just the equal sign that enable us do some really complex things. We'll dive into those operators next.

Operators

So far we've only made conditions using the equal (=) or greater than (>) operators. There are many more at our disposal. They are fairly self-explanatory and just need some practice to get down. Here's the table describing the most commonly used operators:

Operator	Description
=	equal
<	less than
>	greater than
<=	less than or equal
>=	greater than or equal
!=	not equal
<>	not equal (yup, there are two ways)

Take a few moments to get familiar with these operators by filtering out some tracks data. Here's a query to get started with:

String operators and Patterns

When you want to match an exact string (like `composer = 'Green Day'`) you can simply use the equal (=) operator. But that condition only does an exact match and therefore only matches songs that were exclusively written by Green Day.

If we want to find all the tracks that were composed by Green Day (either alone, or in conjunction with other artists) we need to be able to match rows where the composer isn't equal to 'Green Day' but contain 'Green Day' somewhere in them.

To condition match part of a string, or identify strings following a pattern we can use either of these string pattern matching operators.

Operator	Description
LIKE	a string matches a pattern
ILIKE	case insensitive version of LIKE
SIMILAR TO	a string matches a regex pattern

They take a bit more explanation than the simple comparators above.

LIKE

Like is the easy/lightweight way to match a string to a pattern. A pattern is a string that can use some special symbols that represent wildcard characters. Besides regular characters, the two wildcard symbols LIKE can use are

Symbol	Description
_	matches any single character
%	matches any number of characters

To make a pattern that will match 'Green Day' inside of any string we put `%` symbols on either side, meaning any number of characters can be before or after Green Day. So with this pattern as our condition, on running the following query the database will scan for matches in each row and return those that are true.

Test your skill: can you create a query to return all of the *artists* with 'Black' in their *name*?

ILIKE

If you want your pattern to not care about whether characters are upper or lower case you can use ILIKE. The I stands for "case (I)nsensitive". So if we wanted to find all composers that had the word "day" in it regardless of case, we could use:

Note that in the above query if we switched ILIKE to LIKE we wouldn't match any Green Day tracks because Day is capitalized.

LIKE and ILIKE Examples

Here are a few more examples of what patterns will and won't match.

```
'Little Richard' LIKE '%Richard'        true
'Little Richard' LIKE '_______Richard'  true
'Little Richard' LIKE '______Richard'   false
'Little Richard' LIKE '%richard'        false
'Little Richard' ILIKE '%richARD'       true
'Little Richard' LIKE '_ittle %'        true
```

You can play around with patterns yourself by switching the LIKE statements out here

SIMILAR TO

SIMILAR is the more advanced way to match a string to a pattern, using a standard pattern format called regular expressions (regexp). These can get really advanced (too advanced for this tutorial) so we won't go over it in detail. If you'd like to dig in further however we have our Full Regular PostgreSQL Expressions page here.

For a quick example of SIMILAR TO, here is a querying with a regex to match all tracks composed by either AC/DC or Green Day.

Dealing with NULLs

Empty cells in a database are called NULL. They're somewhat of a special value and are dealt with a bit differently. You can't use the = or `!=` operators, instead you use the `IS` operator as shown.

```
IS NULL      -- matches NULL values
IS NOT NULL  -- matches all non NULL values
```

The following query will fetch all *tracks* where the *composer* IS NOT NULL. Try running it, and also change it up to return only the rows that do have a NULL *composer*.

Progress Checkin!

The above describes the main toolset of operators you'll need, but if you're interested in learning more checkout the full list of PostgreSQL operators.

You've learned a huge chunk of SQL so far, keep it up! Are you seeing how SQL is almost english like, or at least like an advanced Google search? I hope it's starting to make sense and is getting less intimidating. A few more concepts and a bit of practice and you'll be quite fluent in no time!

Aggregate Functions

Fetching the raw data is nice and all, but now we're going to start actually doing some aggregations and transformations to it! The first and probably most commonly used aggregation function we are going to learn is COUNT. The COUNT function takes whatever you give it and returns the count of how many there are.

The following SQL will count how many albums are in our database. Put another way, we're going to query for a count of the number of rows are in the *ablums* table. Play around yourself and find how many are in the *artists* and *tracks* tables as well.

COUNT of specific columns

As the COUNT goes over the data results it only increments its tally if the data is not NULL (NULL means empty in SQL jargon). Doing a `COUNT(*)` will always return the full count of the number of rows that exist in the table as the splat (`*`) represents each column and there's no way that all the columns in the row would be NULL. If you specify a specific column however, you're returning the COUNT of the number of rows where that column is not NULL. So doing a count of the composer column:

Will be less than the COUNT(*)

because the composer column has some NULL values (aka. it's empty sometimes).

COUNT DISTINCT

A commonly used clause with the count function is DISTINCT. The DISTINCT clause changes the count to only tally the number of unique values in the data. Above we fetched how many *tracks* had composers listed. If we actually wanted to see how many unique composers were in our *tracks* table we could use the COUNT with the DISTINCT clause as shown here:

Can you modify the query above to find how many different *genre_id*s are the *tracks* table?

Aliases

A quick aside here: Notice that the column headers on the above datasets weren't all that clear. SQL does a okay job of finding a name for what you're fetching but often, especially as we start making more complex functions, you'll want to use your own alias for the data. You can do so with the AS key word following your selections:

Be sure to use double quotes (") around your Aliases as double quotes are used for column titles.

Functions

The following is a list of the most commonly used functions in SQL. They work similar to COUNT but perform different calculations on the data.

Function	Description
MAX	returns the largest (maximum) number in a sets
MIN	described
COUNT	returns a count of the # of values in a set
COUNT DISTINCT	returns a count of the # of unique (distinct) values in a set
EVERY	returns true if all data inside is true (same as bool_and)
AVG	returns the average (mean) of the set of numbers
SUM	returns the sum of all the values in the set

The following example gives the range and average prices of the *tracks* using the *MIN*, *MAX* and *AVG* functions.

Can you modify the above query to return how much it would cost to buy one of every track in the database?

We only covered the most commonly used aggregation functions here. If you'd like to see more checkout the full list of PostgreSQL Functions

GROUP BY

So far our aggregation functions have run across all of the data, but it's often useful to split the aggregation into groups.

Let's say for example that we wanted to get not a count of all of the tracks, but how many tracks were in each genre. One way of doing this would be to write a separate query for each genre like this:

```
SELECT COUNT(*) FROM tracks WHERE genre_id = 1;
SELECT COUNT(*) FROM tracks WHERE genre_id = 2;
SELECT COUNT(*) FROM tracks WHERE genre_id = 3;
.
.
.
SELECT COUNT(*) FROM tracks WHERE genre_id = n;
```

But we'd have to know what all the genre_id's were and use some other tool to combine all of the results back together. Not ideal.

Luckily, we have the GROUP BY clause which makes this a whole lot simpler. The GROUP BY clause tells the database how to group a result set, so we can more simply write the queries above as:

How cool is that?! Can you get a count of all tracks by composer?

It's useful here to order the results of this query by the count, so we can see which composers have produced the largest number of tracks (at least in our database).

Above, the NULL composer is being counted as having the most tracks. That's just noise. Using what we just learned abour NULL operators, can you modify the query to filter out the NULL composers?

Multiple GROUP BYs

You can group by more than one thing, and it simply creates a second set of groups inside the first set. Try running the following example which groups first by genre and then by composer.

The priority/order of the groups is the same as how you list them. You can see that switching the order of genre_id and composer in the GROUP BY clause makes quite a different query:

Notice that I also added ORDER BY clauses to make the output a little more clear. ORDER BY's are quite useful and common when using GROUP BY.

GROUP BY Rules

There are a few rules to follow when using GROUP BYs. The largest is that all data that isn't listed as a parameter to GROUP BY needs an aggregation function applied to it. Think of what the following query:

```
SELECT genre_id, unit_price FROM tracks GROUP BY genre_id;
```

It throws an error because the database doesn't know what to do about *unit_price*. While there is only one genre_id per group, there are many unit_prices. They all can't just be output as a value without some aggregation function.

Can you correct the above query to get the average *unit_price* by *genre_id*?

GROUP BY Errors

It's easy to forget this rule and if so you're going to see an error like the following

```
ERROR: column "tracks.composer" must appear in the GROUP BY clause or be used in an aggrega
```

Just remember that that means you have to either add that column to the GROUP BY or apply an aggregation function to it so the database knows what to do.

The following example will throw this error because the database doesn't know what to do with all of the unit prices. Can you modify it to do return the average *unit_price* by *genre_id*?

JOIN Relationships and JOINing Tables

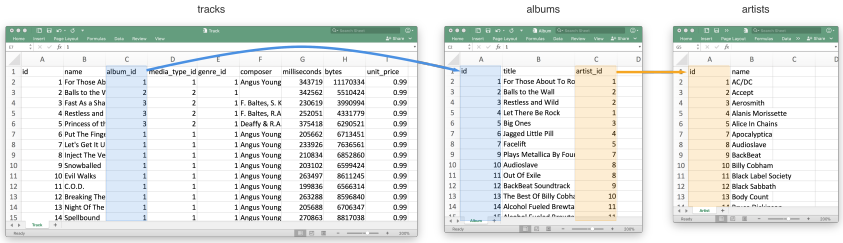
So far we've been working with each table separately, but as you may have guessed by the tables being named *tracks*, *albums*, and *artists* and some of the columns having names like *album_id*, it is possible to JOIN these tables together to fetch results from both!

There are a couple of key concepts to describe before we start JOINing data:

Relationships

PostgreSQL is a Relational Database, which means it stores data in tables that can have relationships (connections) to other tables. Relationships are defined in each tables by connecting Foreign Keys from one table to a Primary Key in another.

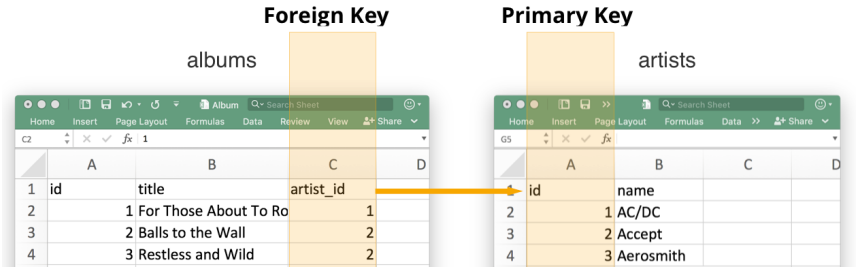
The relationships for the 3 tables we've been using so far are visualized here:



Primary Keys

A primary key is a column (or sometimes set of columns) in a table that is a unique identifier for each row. It is very common for databases to have a column named *id* (short for identification number) as an enumerated Primary Key for each table.

It doesn't have to be *id*. It can be *email*, *username*, or any other column as long as it can be counted on to uniquely identify that row of data in the table.



Foreign Keys

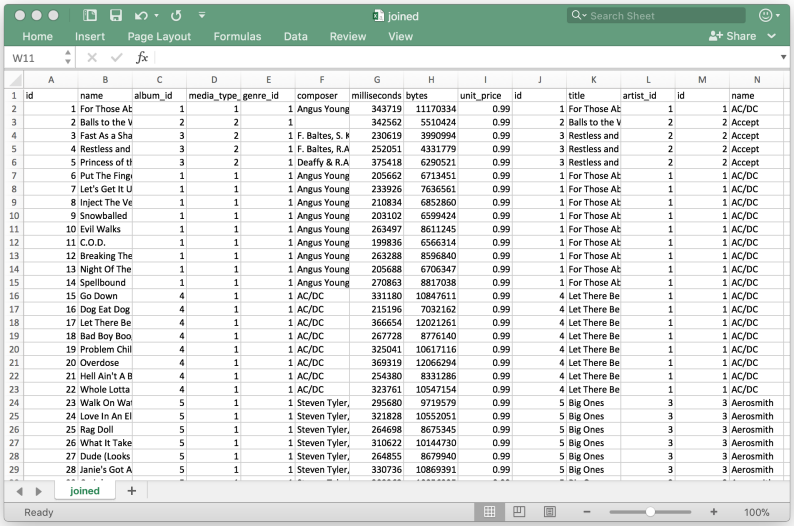
Foreign keys are columns in a table that specify a link to a primary key in another table. A great example of this is the *artist_id* column in the *albums* table. It contains a value of the *id* of the correct artist that produced that album.

Another example is the *album_id* in the *tracks* database. Earlier in this tutorial we looked up all the tracks with an *album_id* of 89. We also looked up which *albums* had an *id* of 89 and found that the tracks referred to the album "American Idiot". TODO: Fix this paragraph/example.

It is very common for foreign key to be named in the format of `[other table name]_id` as *album_id* and *artist_id* are, but again it's not required. The foreign key column could be of any type and link to any column in another table as long as that other column is a Primary Key uniquely identifying a single row.

Why Relationships

If we didn't have relationships we'd have to keep all the data in one giant table like the one in the figure here.



id	name	album_id	media_type_id	genre_id	composer	milliseconds	bytes	unit_price	id	title	artist_id	id	name
1	For Those Ab	1	1	1	Angus Young	343719	11170334	0.99	1	For Those Ab	1	1	AC/DC
2	Balls to the V	2	2	1		342562	5510424	0.99	2	Balls to the V	2	2	Accept
3	Fast As a Sha	3	2	1	F. Baltes, S. H	230619	3990994	0.99	3	Restless and	2	2	Accept
4	Restless and	3	2	1	F. Baltes, R.A	252051	4331779	0.99	3	Restless and	2	2	Accept
5	Princess of th	3	2	1	Deaffy & R.A	375418	6290521	0.99	3	Restless and	2	2	Accept
6	Put The Finge	1	1	1	Angus Young	205662	6713451	0.99	1	For Those Ab	1	1	AC/DC
7	Let's Get It U	1	1	1	Angus Young	233926	7636561	0.99	1	For Those Ab	1	1	AC/DC
8	Inject The Ve	1	1	1	Angus Young	210834	6852860	0.99	1	For Those Ab	1	1	AC/DC
9	Snowballed	1	1	1	Angus Young	203102	6599424	0.99	1	For Those Ab	1	1	AC/DC
10	Evil Walks	1	1	1	Angus Young	263497	8611245	0.99	1	For Those Ab	1	1	AC/DC
11	C.O.D.	1	1	1	Angus Young	199836	6566314	0.99	1	For Those Ab	1	1	AC/DC
12	Breaking The	1	1	1	Angus Young	263288	8596840	0.99	1	For Those Ab	1	1	AC/DC
13	Night Of The	1	1	1	Angus Young	205688	6706347	0.99	1	For Those Ab	1	1	AC/DC
14	Spellbound	1	1	1	Angus Young	270863	8817038	0.99	1	For Those Ab	1	1	AC/DC
15	Go Down	4	1	1	AC/DC	331180	10847611	0.99	4	Let There Be	1	1	AC/DC
16	Dog Eat Dog	4	1	1	AC/DC	215196	7032162	0.99	4	Let There Be	1	1	AC/DC
17	Let There Be	4	1	1	AC/DC	366654	12021261	0.99	4	Let There Be	1	1	AC/DC
18	Bad Boy Boo	4	1	1	AC/DC	267728	8776140	0.99	4	Let There Be	1	1	AC/DC
19	Problem Chil	4	1	1	AC/DC	325041	10617116	0.99	4	Let There Be	1	1	AC/DC
20	Overdose	4	1	1	AC/DC	369319	12066294	0.99	4	Let There Be	1	1	AC/DC
21	Hell Ain't A B	4	1	1	AC/DC	254380	8331286	0.99	4	Let There Be	1	1	AC/DC
22	Whole Lotta	4	1	1	AC/DC	323761	10547154	0.99	4	Let There Be	1	1	AC/DC
23	Walk On Wat	5	1	1	Steven Tyler,	295680	9719579	0.99	5	Big Ones	3	3	Aerosmith
24	Love In An El	5	1	1	Steven Tyler,	321828	10552051	0.99	5	Big Ones	3	3	Aerosmith
25	Rag Doll	5	1	1	Steven Tyler,	264698	8675345	0.99	5	Big Ones	3	3	Aerosmith
26	What It Take	5	1	1	Steven Tyler,	310622	10144730	0.99	5	Big Ones	3	3	Aerosmith
27	Dude (Looks	5	1	1	Steven Tyler,	264855	8679940	0.99	5	Big Ones	3	3	Aerosmith
28	Janie's Got A	5	1	1	Steven Tyler,	330736	10869391	0.99	5	Big Ones	3	3	Aerosmith

Each track for example would have to hold all of the information on it's album and on the artist. That's a lot of duplicate data to store, and if a parameter in any of that changes, you'd have to update it in many different rows.

It gets messy already even for our small example, and just wouldn't be realistic for real company implementation. The world (and data) works better with relationships.

JOINing Tables

So let's get to it! To specify how we join two tables we use the following format

```
SELECT * FROM [table1] JOIN [table2] ON [table1.primary_key] = [table2.foreign_key];
```

Note that the order of table1 and table2 and the keys really doesn't matter.

Let's join the *artists* and *albums* tables. In the above figure we can see that their relationship is defined by the *artist_id* in the *albums* table acting as a foreign key to the *id* column in the *artists* table. We can get the joined data from both tables by running the following query:

Try for yourself to JOIN the tracks and albums tables.

We can even join all 3 tables together if we'd like using multiple JOIN commands

JOIN types

There are a few different types of JOINs, each which specifies a different way for the database to handle data that doesn't match the join condition. These Venn diagrams are a nice way of demonstrating what data is returned in these joins.

JOIN Visual	Type	Description
	INNER	DEFAULT: returns only the rows where matches were found
	LEFT OUTER	returns matches and all rows from the left listed table
	RIGHT OUTER	returns matches and all rows from the right listed table
	FULL OUTER	returns matches and all rows from both tables

We can demonstrate each of these by doing a COUNT(*) and showing how many rows are in each dataset. First, the following query shows us how many columns are in the *artists* and *albums* tables.

And we know that each album does have an artist, but not all artists have an album in our database.

INNER JOIN

The inner join is going to fetch a list of all the albums tied to their artists. So we know that as long as each album does have an artist in the database (and it does) we'll get back 347

rows of data as there are 347 albums in the database. And indeed, that is what we get back from the INNER JOIN:

RIGHT OUTER JOIN

An OUTER JOIN is going to fetch all joined rows, and also any rows from the specified direction (RIGHT or LEFT) that didn't have any connections. In our database, many artists don't have an album stored. So if we do a **RIGHT** OUTER JOIN here which specifies that the right listed *artists* table is the target OUTER table we will get back all matches that we did from the INNER JOIN above *AND* all of the non matched rows from the *artists* table. And here we show we do:

418 OUTER results minus 347 INNER results shows that there are 71 *artists* in the database that aren't associated with one of our *albums*. Can you double check that that's the case with SQL, by adding a WHERE condition to the above query filtering the results for those where there is no *albums.id*?

LEFT OUTER JOIN

If we chose to do a LEFT OUTER JOIN we'd be choosing the *albums* table as the OUTER target. And here we are verifying that there are no extra albums that don't have an artist associated with them.

FULL OUTER JOIN

And finally a FULL OUTER JOIN is going to return the JOINed results and any non-matched rows from **either** of the tables. We know that in the case of this dataset those will only come from the artists table, and the result will be the same as our RIGHT OUTER JOIN above.

Bringing it All Together

We can do more than one JOIN in a query so let's bring *tracks*, *ablums* and *artists* together and see how it looks. Try running the following which is LIMITed to just 5 rows, as it's a large result set and we don't need to see all of it.

Scrolling right you can see that there are a lot of columns as the result has all of the columns of each joined set. You can also see that there's a conflict as there are 2 columns title *name*. One is from the *tracks* table and one is from the *artists* table and the result set isn't handling that properly. It's just using the names from the *artists* table in both columns!

We can fix this by using aliases. In the following we're trying to get the names of 8 tracks along with the name of the artist. Run it and you'll see for yourself. Can you fix the mixup in them both having the same column name using the aliases `AS "Track"` and `AS "Artist"`.

You have now unlocked the knowledge to fully enjoy most of the double entendres in this amazing song about Relationships. Do take a moment to enjoy.

DATE and TIME Functions

DATE and TIME values in PostgreSQL have a whole special set of functions and operators for their proper use. So many queries deal with DATE and TIME information that it's important to get to know the date tools. Below we'll cover and practice the main functions you'll likely need. If you want to get detailed you can checkout the full list of PostgreSQL functions here.

DATE/TIME Datatypes

There are 4 main ways to store date values in a PostgreSQL database:

Data Type	Description	Example	Output
TIMESTAMP	date and time	`TIMESTAMP '2019-12-10 01:14:25'`	2019-12-10T01:14:25
DATE	date (no time)	`DATE '2019-12-10 01:14:25'`	2019-12-10
TIME	time (no day)	`TIME '2019-12-10 01:14:25'`	01:14:25
INTERVAL	interval between two date/times	`INTERVAL '1 day 2 hours 10 seconds'`	1 day, 2:00:10

We'll go over more about each of these.

Date String Formatting

Dates in a database aren't stored as strings, but we input and fetch data from it as if it were a string with the following format for the information:

```
YYYY-MM-DD HH:MM:SS
```

where the letters stand for **Y**ear, **M**onth, **D**ay, **H**our, **M**inutes and **S**econds. Let's say for example that we want to record that we got a new user on December 10, 2019 at exactly 01:14. To represent that exact date and time we would use the format:

```
2019-12-10 01:14:00
```

TODO: this format is also supported: January 8 04:05:06 1999 PST

To get some familiarity try creating and SELECTing a few TIMESTAMPS below. I was born on May 1st, 1983 at exactly 4:00am. Can you fetch that timestamp?

We're just going to jump in here. We need to use a different table as none of the previous ones we've been using have had date fields in them. Another table available to us in chinook is *employees*. Let's get familiar with what columns are in this table by looking at the first few rows. Note that there are several columns so you may have to scroll right to see all of the data:

Each *employee* has two TIMESTAMP columns, one for their *birth_date* and one for their *hire_date*. You can use all of the ORDERing, GROUPing and other functions we learned for other columns on DATE columns as well. Try getting a list of the 4 youngest *employees* in the company.

Formatting Dates to Strings

Often you don't want to show the full raw TIMESTAMP, but rather a nicely formatted, potentially truncated version. For example, let's say we want to get a list of the *employees* names and the year that they were hired. To do so we'll need to parse the *hired_date* to just pull out the year. We can do so with the TO_CHAR function which works as follows

```
TO_CHAR([date type], [pattern])
```

where [date type] is a column or value of any of the above listed date/time data types, and [pattern] is a string indicating how to format the output date. The main symbols you'll want to use to create your format patterns are here

Pattern	Description	Example	Output
HH	**H**our (01-12)	`TO_CHAR(TIME '4:15 pm', 'HH')`	04
HH24	**H**our (01-24)	`TO_CHAR(TIME '4:15 pm', 'HH24')`	16
MI	**M**inute	`TO_CHAR(TIME '4:15 pm', 'MI')`	15
SS	**S**econds	`TO_CHAR(TIME '4:15:23 pm', 'SS')`	23

Pattern	Description	Example	Output
am	displays whether time is **am** or pm	`TO_CHAR(TIME '4:15 pm', 'am')`	am
YY	last 2 digits of the **Y**ear	`TO_CHAR(DATE '2019-12-10', 'YY')`	19
YYYY	4 digits of the **Y**ear	`TO_CHAR(DATE '2019-12-10', 'YY')`	2019
MM	**M**onth # of the year.	`TO_CHAR(DATE '2019-12-10', 'MM')`	12
Month	written **Month** of the year capitalized	`TO_CHAR(DATE '2019-12-10', 'Month')`	December
Mon	abbreviated of **Mon**th of year	`TO_CHAR(DATE '2019-12-10', 'Mon')`	Dec
DD	Day # of the month	`TO_CHAR(DATE '2019-12-10', 'DD')`	10
Day	written **Day** of the week	`TO_CHAR(DATE '2019-12-10', 'Day')`	Tuesday
Dy	abbreviated **D**ay of the week	`TO_CHAR(DATE '2019-12-10', 'Dy')`	Tue
WW	**W**eek # of the year	`TO_CHAR(DATE '2019-12-10', 'WW')`	50
Q	**Q**uarter of the year	`TO_CHAR(DATE '2019-12-10', 'Q')`	4
TZ	**T**ime**Z**one	`TO_CHAR(DATE '2019-12-10', 'TZ')`	UTC

The above patterns can be string together to get the format you eventually want. Some common outputs are:

and

and

You don't have to memorize these (it's hard to!). It's just good to get familiar with how it works and then reference back to it when you need it in the future.

Number formatting

There are a couple of extra tools you can use on patterns that output numbers.

Formatter	Description	Example	Output
FM	**F**ill **M**ode will remove any 0's at the front of a 2 digit number.	`TO_CHAR(DATE '2019-12-05', 'FMDD')`	5
th	adds the ordinal suffixes like **st**, **nd** or **th** to the end of a number	`TO_CHAR(DATE '2019-12-05', 'FMDD')`	05th

And of course you can combine the two to get

String Formatting

For string outputs, most of the patterns above support different casing output based on the case you use for the pattern. Some examples using different casings of "Day":

And you can see the following common date format in UPPERCASE, Capitalized and lowercase formats:

Note that the case for numeric values doesn't change. Still use DD for the day # of the month and YYYY for year.

We're going to move on in the tutorial but if you'd like more details checkout the full list of PostgreSQL date formatting functions .

Current DATE and TIME Functions

PostgreSQL supports a number of special values, or functions to help bet the current DATE, TIMESTAMP or TIME. The most used ones are

```
CURRENT_DATE
CURRENT_TIME
CURRENT_TIMESTAMP
```

and they are used by just putting them in the query

GROUPing BY DATE

In analytic queries, it's very common to group things by dates. For example you may want to see new users by year, month, week or day. To do so, you'll want to use the TO_CHAR function to convert the dates into a truncated string before you GROUP BY it. You don't

want to simply GROUP BY the raw date as those are accurate down to the millisecond so grouping by the unaltered date would be like making GROUPs for each millisecond.

The following examples are using the *hire_date* field from the *employees* table and show a lot of common formats you can use for these groups. These are what we use at Chartio for our date group formatting standards.

Group Period	Example SQL	Example Output
Second	`TO_CHAR(hire_date, 'YYYY-MM-DD"T"HH24":"MI:SS')`	2018-03-04T00:00:00
Minute	`TO_CHAR(hire_date, 'YYYY-MM-DD"T"HH24":"MI')`	2018-08-14T00:00
Hour	`TO_CHAR(hire_date, 'YYYY-MM-DD"T"HH24')`	2018-01-02T00
Day	`TO_CHAR(hire_date, 'YYYY-MM-DD')`	2003-10-17
Week	`TO_CHAR(hire_date, 'IYYY"-W"IW')`	2002-W33
Month	`TO_CHAR(hire_date, 'YYYY-MM')`	2002-05
Quarter	`TO_CHAR(hire_date, 'YYYY"-Q"Q')`	2003-Q2
Year	`TO_CHAR(hire_date, '"Y"YYYY')`	Y2012
Hour of Day	`TO_CHAR(hire_date, 'HH24')`	14
Day of Week	`TO_CHAR(hire_date, 'FMDay')`	Thursday
Day of Month	`TO_CHAR(hire_date, 'DD')`	17
Day of Year	`TO_CHAR(hire_date, 'DDD')`	125
Month of Year	`TO_CHAR(hire_date, 'FMMonth')`	October

Feel free to try out any of the above formats on the query below:

There are only 8 *employees* in our database so we're not dealing with too many groups there. You can get a little more granular with the *invoices* table and it's *invoice_date* column with 250 rows.

The above query returns the number of *invoices* created per year. Can you modify it to get a SUM of the *total* amount invoiced by month?

Mid Level SQL Practice Grounds

You've covered the majority of the main use cases of SQL! You know the stuff, but now you've got some practicing to do to become really fluent and skilled at it. Here we've constructed a large list of challenges to give you that practice. If you forgot the rules of our practice playgrounds you can review them in the Basic SQL Practice page.

Good luck!

Q. Fetch all the tracks that are over 300000 milliseconds long.

references: where

Q. Fetch the *id* for the artist 'Miles Davis'.

references: where

Q. Get all the *tracks* with *gerne_id* of 20 from longest to shortest.

references: where order-by

Q. Get the *artists* who's *id*s are between 55 and 98.

references: operators

Q. Get all the *tracks* except those with *genre_ids* of 15 or 18.

references: operators

Q. Get all the *tracks* that were composed by just Miles Davis

references: operators

Q. Get all the *tracks* that Miles Davis had a part in composing.

references: operators

Q. Fetch the the *name*s of the *tracks* with the word 'wild' in it, regardless of case

references: operators

Q. How many *tracks* did Little Richard helped compose?

references: operators aggregate

Q. How many *tracks* with *genre_id* of 1 have composers listed?

references: aggregate

Q. How many unique composers are there in the *tracks* table with the *genre_id* of 1.

references:

Q. What is the average length for tracks with *genre_ids* of either 5, 7 or 10?

references: operators aggregate

Q. Get a list of *genre_id*s and the number of tracks in each.

references: group-by

Q. A list of the number of tracks grouped by genre_id and then album_id with the column order of *genre_id*, *album_id* and count.

references: group-by

Q. Take the above query, but order the *album_id* in descending order, keeping genre_id odered the same

references: order-by group-by

Q. Take the above query with the same ordering but group by *album_id* and then *group_id* and change the order of the results to reflect that switch.

references: order-by group-by

Q. Get the first 5 *tracks* JOINed with their albums info.

references: joins limit

Q. Get the first 5 albums JOINed with their artists info.

references: joins limit

Q. Return each *customer_id* and the *total* of all of their *invoices*.

references: group-by aggregate

Q. There is a foreign key *customer_id* that relates *invoices* to the *customers* table. Fetch the company name in the *company* column of the *customers* table and the *total* amount that each has been invoiced by joining these tables."

references: group-by aggregate joins

Q. Get the *first_name*s and *birth_date*s of each of the *employees* in the format: January 01, 1976

references: dates

Q. Get the *first_name*s and *birth_date*s of each of the *employees* in the format: Jan 1st, 1976

references: dates

Q. Get the *first_name*s and *birth_date*s of each of the *employees* in the format: 09/23/1987

references: dates

Q. Get the year of the *invoice_date* in the format `Y2012` and total number of *invoices* per year.

references: dates

Q. Get the year and month of *invoices* and the *total* amount that was invoiced for that year and month.

references: dates aggregate

If you've completed all of these CONGRATULATIONS! You're proficient and fluent enough in SQL now to complete a significant portion of analytic and transactional queries.

Extras

What is the difference between UNION and UNION ALL

UNION and UNION ALL are SQL operators used to concatenate 2 or more result sets. This allows us to write multiple SELECT statements, retrieve the desired results, then combine them together into a final, unified set.

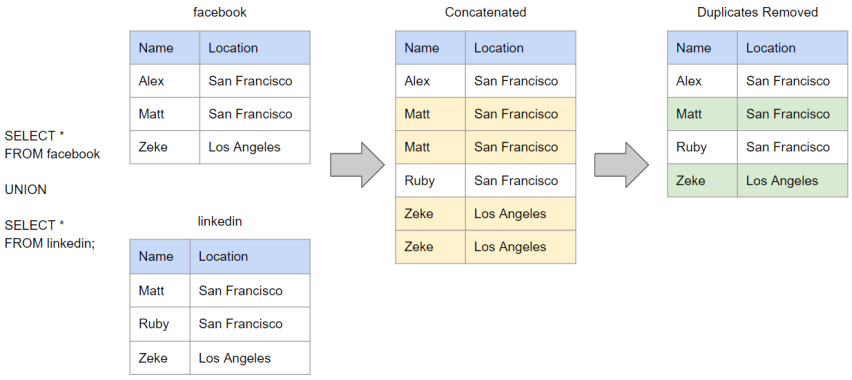
The main difference between UNION and UNION ALL is that:

- **UNION:** only keeps *unique* records
- **UNION ALL:** keeps all records, including *duplicates*

UNION ALL Difference

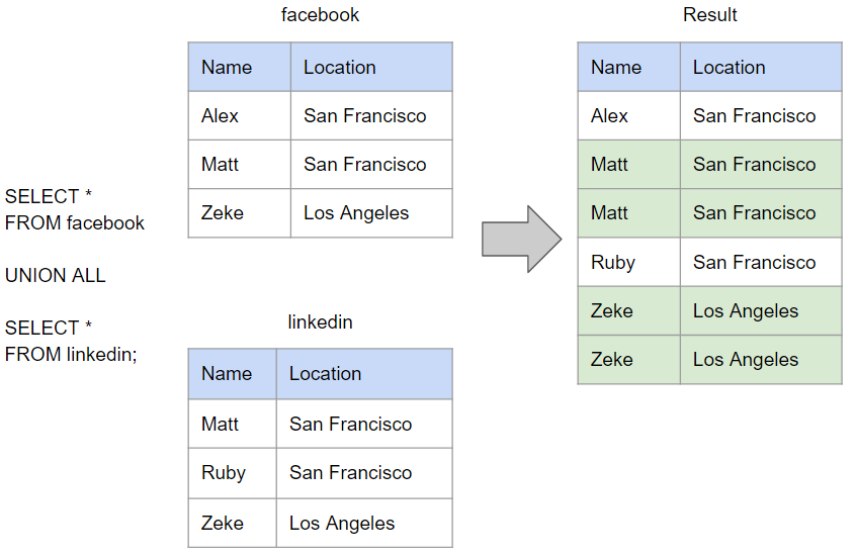
UNION ALL *keeps all of the records* from each of the original data sets, UNION *removes any duplicate records*. UNION first performs a sorting operation and eliminates of the records that are duplicated across all columns before finally returning the combined data set.

UNION



UNION ALL

If we were to now perform the UNION ALL on the same data set, the query would skip the deduplication step and return the results shown.



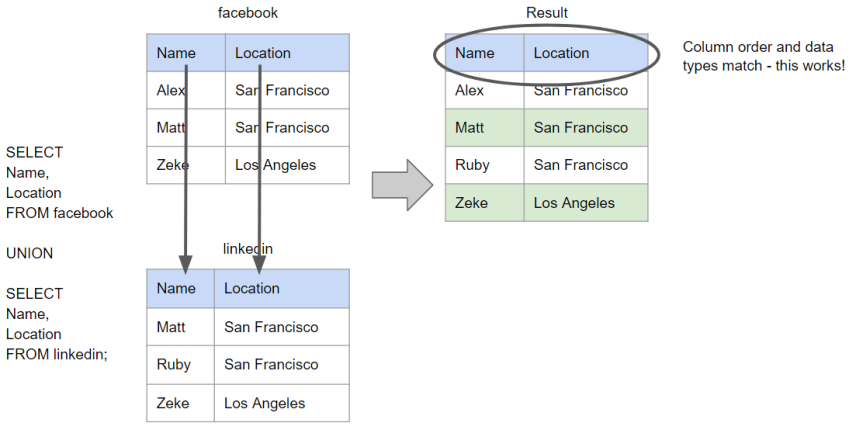
*Note: In both of these examples, the field names from the first SELECT statement are retained and used as the field names in the result set. These can be changed later if desired.

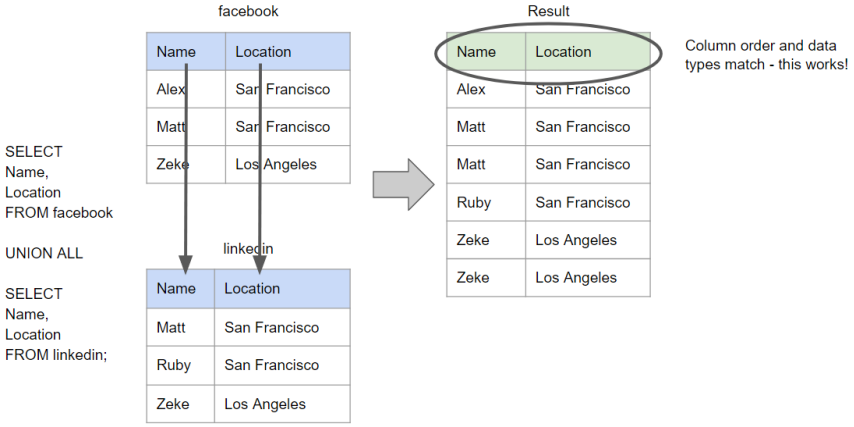
UNION-ing data

UNION or UNION ALL have the same basic requirements of the data being combined:

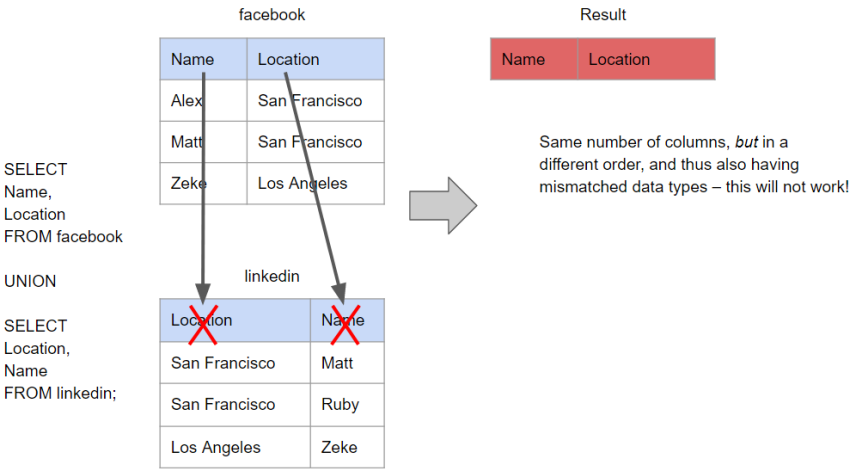
1. There must be the same number of columns retrieved in each SELECT statement to be combined.
2. The columns retrieved must be in the same order in each SELECT statement.
3. The columns retrieved must be of similar data types.

The next 2 examples shows that we would return results whether we used UNION or UNION ALL since all required criteria are met.





This final example would fail. While we have the correct number of columns, they are now queried in the wrong order in the second SELECT statement and thus the data types also do not match up. This will result in an error being returned.



Summary

We have seen that UNION and UNION ALL are useful to concatenate data sets and to manage whether or not we retain duplicates. UNION performs a deduplication step before

returning the final results, UNION ALL retains all duplicates and returns the full, concatenated results. To allow success the number of columns, data types, and data order in each SELECT must be a match.

Exclude a Column

In some cases, you may have a table with many fields and desire to write a query that selects nearly all of them. In a situation like this, it would be nice to be able to write a query that combines a SELECT all with a shorter list of exclusions.

Unfortunately, since SQL is a declarative language, this cannot be done. When we use SQL, we must specify what we want, not what we do not want. The 2 best viable ways to approach this problem are as follows:

Omit Columns

List out all columns in your query, omit the undesired fields by:

- Not including
- Deleting
- Commenting out

Not including columns or deleting columns you don't want in your SELECT statement is straightforward. However if you would want to show that you are leaving out certain columns intentionally you can comment them out by using two dashes –

```
SELECT
        column1,
        --column2,
    column4
FROM
        table_name;
```

Here we deleted column3 and commented out column 2.

The SQL DESCRIBE statement is useful here to obtain the full list of the fields in a table, especially if there are a great number.

DESCRIBE *table_name*;

It will produce a table with all column names from the table being described and some other meta information.

Field	Type	Null	Key
column1	int	NO	PRI
column2	varchar(30)	YES	
column3	datetime	YES	
column4	varchar(40)	YES	

Create View

If you will often be querying a table and retrieving most of its columns, then it may make sense to create a view. The view would persist as a "virtual table," against which SELECT * queries could be run.

Here is how it would look if we wanted to end up with all of the columns except *column2*:

```
CREATE VIEW view_name AS
SELECT
        column1,
        column3,
        column4
FROM table_name;
```

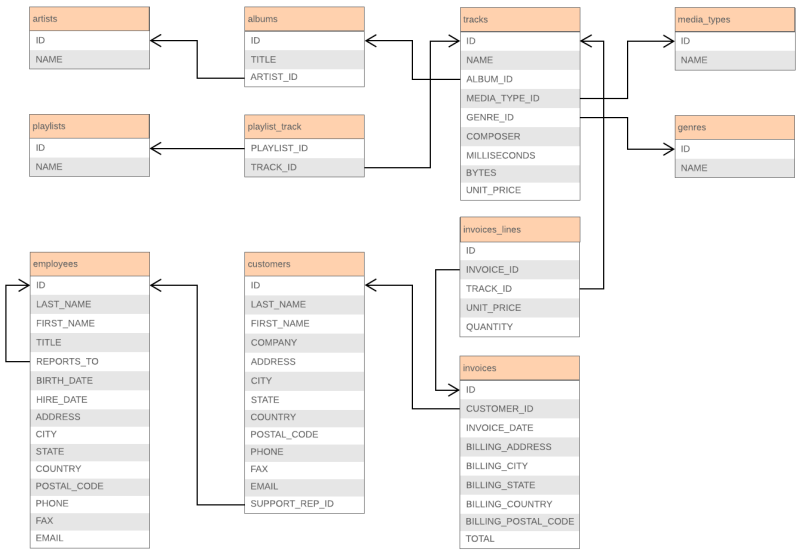
This view could then be queried using SELECT *

```
SELECT *
FROM view_name;
```

In theory this is a very good idea, however as the view definition becomes more extensive and/or complex, there could be a negative performance impact. Since the view is a "virtual table" the data does not reside within the view. Each time a query is made against the view, the view's definition query against the original database tables must also run. The convenience gained by being able to SELECT * should be weighed against the time and resources needed to support the view.

Additional Practice

For practicing we will be using an online music store database. Here is the entity relationship diagram of the schema.



Feel free to explore the data by using SELECT * FROM [table name] in the SQL editor below:

Select Questions

Select all columns and rows from the albums table in the SQL box below:

Select all columns from the albums table where the album title is 'Let There Be Rock' in the SQL box below:

What are all the genres of the music in this database?

Join Questions

Join the Artist and the Album table with an **inner join** in the SQL box below:

Join the Artist and the Album table with an **left join** in the SQL box below:

Join the Artist and the Album table with an **outer join** in the SQL box below:

The INNER JOIN found every instance where the albums.artist_id equalled an artists.id and joined the data together to create a row in the final table.

The LEFT JOIN performed an INNER JOIN and then also added rows to the final table where the left table (albums) did not have matches.

The OUTER JOIN performed both an INNER and LEFT JOIN and then also added rows to the final table where the right table (artists) did not have matches.

How many Artists have more than 1 Album:

Common Errors Questions

Fix the following queries:

AND OR Boolean Logic

AND is used to find where multiple conditions are true

OR is used to find where at least one out of multiple conditions are true

To get more technical, boolean logic is a way of representing how bits in a computer are processed. Let's explore more about these conditional statements (e.g. if-else, where, or case-when statements) with truth tables to understand how precisely boolean logic works.

Truth Tables

For example, let's look at the following conditional:

If: `A and B`

Then: `C`

This returns the value C, when the values A and B are true. We can represent this using something called a truth table. A truth table is a way of representing every possible input and it's corresponding output. The truth table for this AND statement looks like this:

```
A B C
1 1 1
1 0 0
0 1 0
0 0 0
```

In the truth table, a `1` represents true while a `0` represents false. From looking at this table it is evident that the only time C is true, is when both A and B are true.

There is also an OR statement. The OR statement is true when A OR B is true:

If: `A or B`

Then: `C`

Truth table:

```
A B C
1 1 1
1 0 1
0 1 1
0 0 0
```

This truth table might be a little different then you were expecting. This is because an OR statement is only false when both input values (A and B) are False.

Building a Truth Table

Truth tables can be made for combinations of gates as well with more inputs. Look at the following statement for example:

If: `(A or B) and C`

Then: `D`

The first step to building a truth table is to decide how many rows we need. The way to decide this is to check how many inputs we have and raise two to that number. In this case we have 3 inputs so we need `2^3` or 8 rows.

Next we need to decide how many columns to use. In this case we will have one column for each input, one for the output, and one for the value of A and B. The truth table will look like this:

```
A B C A or B D
1 1 1 1      1
1 1 0 1      0
```

1 0 1 1	1
1 0 0 1	0
0 1 1 1	1
0 1 0 1	0
0 0 1 0	0
0 0 0 0	0

As expected, when the table is filled out, the only true output is when all 3 inputs are true.

Short Circuit Logic

Because of the way `AND` and `OR` logic works, programming languages can use something called “short circuit logic”. This is when not all inputs are evaluated, because the computer can guess the answer from the first input that was checked. To see how this works, look at the AND truth table again:

A	B	C
1	1	1
1	0	0
0	1	0
0	0	0

Notice that, when A is False (`0`), C is also always False. This is because C is only true when both inputs are true, therefore a single false means C is false.

If a computer is using an `AND` condition and the first input is false, then the second input, B, will never be checked. OR will evaluate as true without checking the second input when the first input is true. This ability for the computer to invalidate later boolean logic steps can save a lot of unneeded processing power for your query.

Examples in SQL

Example of a `WHERE` condition:

```
SELECT * FROM [table]

WHERE

[column A]=5 AND [column B]=22;
```

Example of a `CASE-WHEN` Statement

```
CASE

WHEN [column A]=21 OR [column B]=7 THEN [Action]

END
```

Summary

- Boolean AND / OR logic can be visualized with a truth table
  - Truth tables two to the number of inputs rows in them
  - 1 - true
  - 0 - false
- Short Circuit Logic
  - If the first input guarantees a specific result, then the second output will not be read
  - AND - first input of false will short circuit to false
  - OR - first input of true will short circuit to true

Copying Data Between Tables

Copy into a new pre-structured table:

```
CREATE TABLE [Table to copy To]
AS [Table to copy From]
WITH NO DATA;
```

- Note: “WITH NO DATA” specifies that the new table should only copy the table structure with no data

Copy into pre-existing table:

```
INSERT INTO [Table to copy To]
SELECT [Columns to Copy]
FROM [Table to copy From]
WHERE [Optional Condition];
```

Copying data between tables is just as easy as querying data however it will take a bit longer to run than a normal query. It can be used to update an inventory, create a table that has different permissions than the original, and much more.

Example:

Take for example a shopkeeper who needs to create a master list of all the items in his store to conduct a store-wide audit. However the data he needs exist in separate tables containing the inventories of each department:

```
shop=# \dt
          List of relations
 Schema |   Name    | Type  | Owner
--------+-----------+-------+-------
 public | clothes   | table | matt
 public | grocery   | table | matt
 public | hardware  | table | matt
(3 rows)
```

In order to create a master list that contains all of the store’s items and prices the shopkeeper needs to create the table for all items and copy the data from each of the departments into the new table.

Creating the table

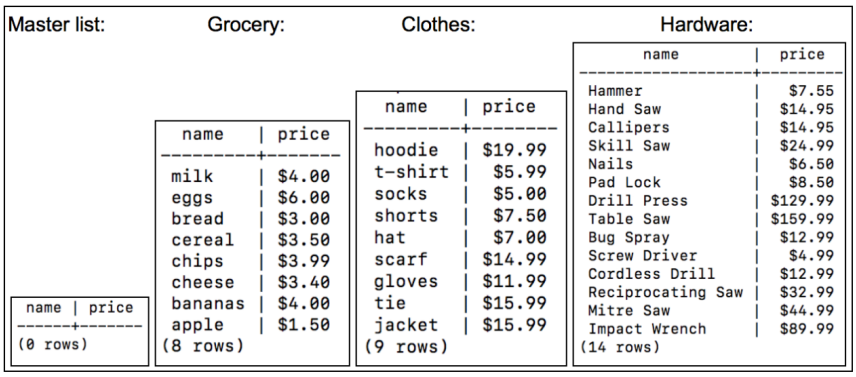
The shopkeeper needs to first make a new table to contain the data. The master list needs to have the same table structure (columns, data-types, etc.). The easiest way to create a table with the same table structure as a different table is to use:

```
CREATE TABLE [New Table] AS TABLE [Old Table] WITH NO DATA;
```

Once filled out, this command will create a new table with the same table structure, but without any data. The shopkeeper can use this to create his master list:

```
shop=# CREATE TABLE masterlist AS TABLE clothes WITH NO DATA;
CREATE TABLE AS
```

With this done, the shopkeeper now has the following tables:



INSERT INTO command

Now that the shopkeeper's master list has been created and structured, the data needs to be inserted into the table. This can be done using the INSERT command. This command inserts specified values into a specified table. It is often used to insert single values into tables by running the command as such:

```
shop=# INSERT INTO masterlist VALUES ('milk', 4);
INSERT 0 1
```

When using INSERT INTO with the VALUES command it is possible to add entries by hand, however a query can also be used in place of the VALUES command. For example to copy all items from the table "hardware" to the table "masterlist" the following query can be run:

```
shop=# INSERT INTO masterlist (SELECT * FROM hardware);
INSERT 0 14
```

This query uses a subquery to find all values in "hardware" and then adds them to the "masterlist". In order to copy data from all the tables, the shopkeeper can use UNION to merge the tables together in the subquery:

```
shop=# INSERT INTO masterlist
shop-# (SELECT * FROM hardware UNION SELECT * FROM grocery UNION SELECT * FROM clothes);
INSERT 0 31
```

This gives the shopkeeper the desired result so that he can begin his audit:

name	price
Bug Spray	$12.99
Callipers	$14.95
Cordless Drill	$12.99
Drill Press	$129.99
Hammer	$7.55
Hand Saw	$14.95
Impact Wrench	$89.99
Mitre Saw	$44.99
Nails	$6.50
Pad Lock	$8.50
Reciprocating Saw	$32.99
Screw Driver	$4.99
Skill Saw	$24.99
Table Saw	$159.99
apple	$1.50
bananas	$4.00
bread	$3.00
cereal	$3.50
cheese	$3.40
chips	$3.99
eggs	$6.00
gloves	$11.99
hat	$7.00
hoodie	$19.99
jacket	$15.99
milk	$4.00
scarf	$14.99
shorts	$7.50
socks	$5.00
t-shirt	$5.99
tie	$15.99

(31 rows)

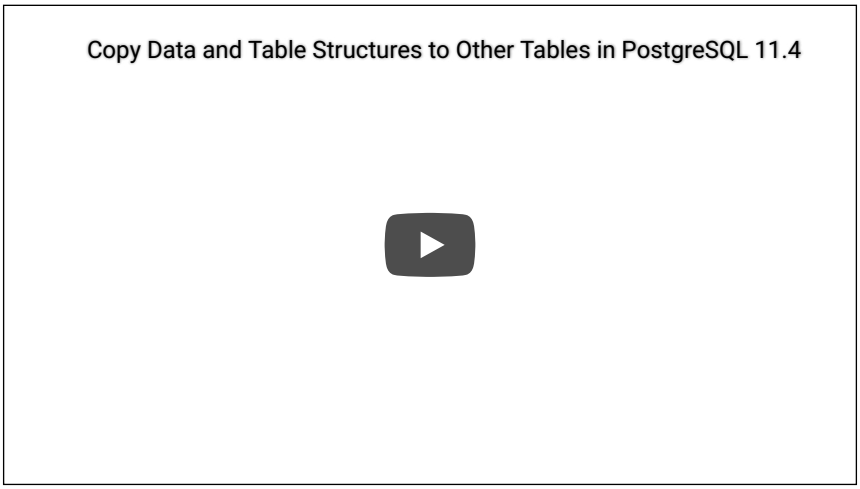
Adding Conditions

Copying data with INSERT INTO can also be done with conditions. For example, if the shopkeeper is only interested in items over $50 these values can be copied by using:

```
INSERT INTO masterlist [SELECT statements] WHERE price>money(50);
```

Each SELECT statement can also have its own where statement for table specific conditions. After the table is created and filled it can be manipulated, added to or removed from without affecting the tables the data was copied from.

Video example



Summary

- To copy create a pre-structured table:
  - CREATE TABLE [Table to copy **To**] AS [Table to copy **From**] WITH NO DATA;
  - Table will be pre structured to handle data from the 'table to copy from'
- Copy into pre-existing table:
  - INSERT INTO [Table to copy **To**] SELECT [Columns to Copy] FROM [Table to copy **From**] WHERE [Optional Condition];
- Will create independent copy in the new table

References

1. https://www.postgresql.org/docs/9.5/sql-insert.html
2. https://stackoverflow.com/questions/25969/insert-into-values-select-from/25971

Export to CSV with \copy

The Commands:

In order to export a table or query to csv use one of the following commands:

For Client-Side Export:

```
\copy [Table/Query] to '[Relative Path/filename.csv]' csv header
```

For Server-Side Export:

```
COPY [Table/Query] to '[Absolute Path/filename.csv]' csv header;
```

Example Absolute Path: '/Users/matt/Desktop/filename.csv'

Example Relative Path: 'Desktop/filename.csv'

Key words:

- csv: this tells the copy command that the file being created should be a CSV file.
- header: this tells copy command to include the headers at the top of the document.

Using the Commands

CSV Files

Comma Separated Value (CSV) files are a useful format for storing data. Many tools support importing data from CSV files because it is an easy to read format that is plain text and not metadata dependent.

In psql there are two commands that can do this, both slightly different.

The first is the *\copy* meta-command which is used to generate a client CSV file. This command takes the specified table or query results and writes them to the client's computer.

The second command, *COPY*, generates a CSV file on the server where the database is running.

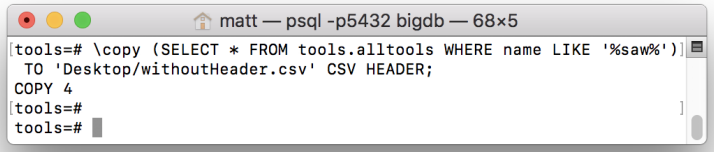
The *\copy* command

The *\copy* meta-command is used for exporting to a client computer. It is useful for copying a database that may have somewhat restricted access and for creating a personal copy of the data. For example, a user may want to generate a csv so that they can analyse financial data in excel. The format of a *\copy* to csv is as follows:

```
\copy [Table/Query] to [Relative Path] csv header
```

The [Table/Query] section can be filled with a table or query. For example to copy all entries from a table, the table name can be put here. To copy all entries that contain "saw"

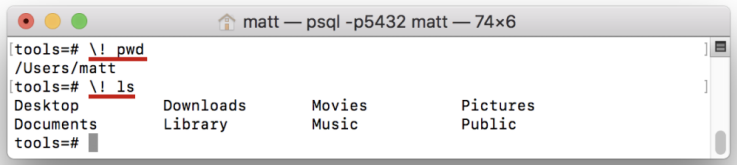
in their names from the table of tools to a csv, the following commands could be run:



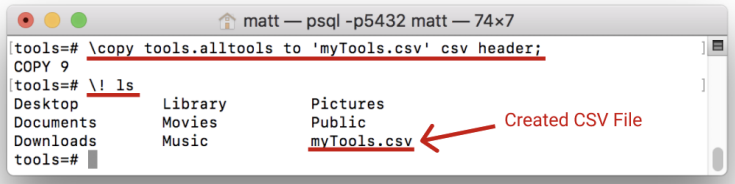
The [Relative Path] is the path from where psql is currently saving files to where you want to save the file. The location that psql is currently saving can be found by using the *\! pwd* command.

Note: The *\!* meta-command takes whatever arguments it is given and runs them as a bash command within psql.

The pwd command prints the current working directory. The meta-command \! pwd and \! ls are shown being used below:



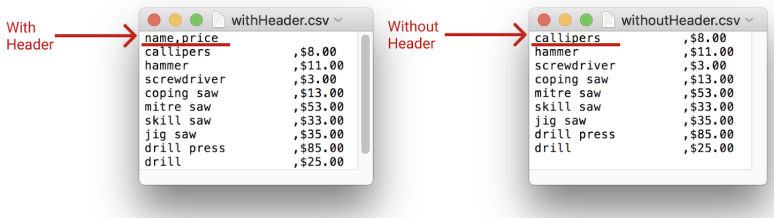
This means that if the file name "myTools.csv" is used as the [Relative Path], it will be saved in */Users/matt/* as can be seen below:



The file can also be saved elsewhere by entering a specific relative path. For example, if *'/Desktop/[Filename].csv'* is entered as the path, the file will be saved to the desktop.

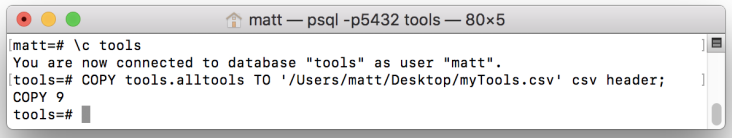
Following the Relative Path in the command is the text *'csv header;'* This text does two things. The *'csv'* specifies that the data should be stored in the CSV format. Other possible formats are 'text' and 'Binary.'

The *'header'* specifies that, when the data is copied to a csv file, the names of each column should be saved on the first line as shown here:



The *COPY* command

The *COPY* command also requires the user to specify a Table/Query to copy. Unlike *\copy* however, the *COPY* command requires an absolute path. This is because *COPY* is for copying a database from a server to another location on the same server; not to a client computer. The *\! pwd* is very useful for finding the absolute path if you do not know where to save the file. In order to save to the desktop using \copy *'Desktop/[Filename].csv'* would be used. In order to do this with *COPY*, *'/Users/[Username]/Desktop/[Filename].csv'* would need to be used as shown below:



Summary

- To copy a table or query to a csv file, use either the *\copy* command or the *COPY* command.
- *\copy* should be used for a copy to local systems
    - *\copy* uses a relative path
- *COPY* should be used to create a csv on the server's side.
    - *COPY* uses an absolute path.

References

1. https://www.postgresql.org/docs/9.2/app-psql.html#APP-PSQL-META-COMMANDS-COPY
2. https://www.postgresql.org/docs/9.2/sql-copy.html
3. https://tableplus.io/blog/2018/04/postgresql-how-to-export-table-to-csv-file-with-header.html

PostgreSQL Generate_Series

Generate a Series in Postgres

`generate_series([start], [stop], [{optional}step/interval]);`

Generate a series of numbers in postgres by using the `generate_series` function.

The function requires either 2 or 3 inputs. The first input, [start], is the starting point for generating your series. [stop] is the value that the series will stop at. The series will stop once the values pass the [stop] value. The third value determines how much the series will increment for each step the default it 1 for number series

For example:

Will output the rows: 1,2,3,4,5,6,7,8,9,10

Let's look at what happens when we start with a number that has a decimal value:

Will output the rows: 0.5,1.5,2.5,3.5,4.5

Note that the value starts at 0.5, but still increments by 1. In order to change the increment, we have to state explicitly how much to increment by as a third option in the function:

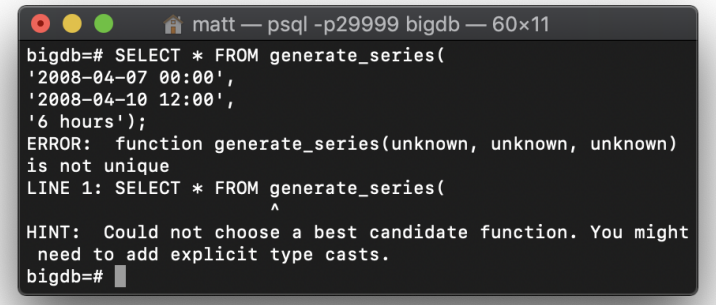
This will output the rows: 1,3,5,7,9

Timestamps

Generate_series() will also work on the timestamp datatype. This may need an explicit cast to work.

For example, to create a list of timestamps from `2018-04-07 00:00` to `2018-04-10 12:00` with one timestamp every 6 hours, the following SQL query can be run:

Note the `::timestamp`. This is an explicit cast to the timestamp data type. The reason for this is because without the cast the data type is too ambiguous. This results in an error being thrown when the query is run:



This error can be avoided by adding the typecast. This will only happen on certain inputs which are ambiguous in terms of data type.

Interval Format

Notice the use of '6 hours' for the third option in the image above. When generating a time series there are additional options for how you define the way the series increments.

The 3rd input, the interval, follows the format of `[quantity] [type] [{optional} direction]`.

`[quantity]` => 6

`[type]` => hours

`[{optional}direction]` => We didn't put anything here because the default is positive.

In the case of `6 hours`, the quantity is 6, the type is hours, and the direction is omitted so it defaults to positive. If you want the same list but opposite order you can change the interval to '6 hours ago'.

Adding `ago` specifies that you want the timestamps to change by 6 hours in the **negative** direction. This will however return 0 rows unless you reorder your start and stop values.

The interval can also be created using a shorthand form. Some of the time types can be abbreviated as shown by this table:

Type	Abbreviations
Millennium	-
Century	-
Decade	-
Year	Y
Month	M
Week	W
Day	D
Hour	H
Minute	M
Second	S
Millisecond	-
Microsecond	-

In order to use the abbreviations we can create the interval using a shorthand notation. This follows the following format:

```
P [Quantity] [date unit] ... T [quantity] [time unit] … ;
```

The P is used to show that the interval is starting and the T indicates that the date (year/month/day) portion of the interval is over and this is now the time (hours/minutes/seconds) portion of the interval

Using this format, an interval of 5 days and 3 hours would be:

```
P5DT3H
```

An interval of 9 years 8 months 7 days 6 hours 5 minutes and 4 seconds would be:

```
P9Y8M7DT6H5M4S
```

To write an interval of just 6 hours use:

```
PT6H
```

While this shorthand is much faster to write, it does sacrifice some of its readability to achieve this.

Summary

- Standard form: `generate_series([start], [stop], [{optional}step/interval]);`
- generate_series() can take several different sets of inputs
  - Can be Numeric or Timestamp data types
  - If start/stop are timestamps:
    - Use an explicit type cast
    - Use an interval (e.g. 6 hours or 1 week ago)
- Step defaults to 1 for numeric unless otherwise specified.
- Time interval can be written in shorthand:
  - Format: P [quantity] [unit] … T [quantity] [unit] …;
  - P5DT6H7M = 5 days 6 hours 7 minutes

Resources

http://www.postgresqltutorial.com/postgresql-interval/

https://www.postgresql.org/docs/current/functions-srf.html

How to Create a Copy of a Database in PostgreSQL

To create a copy of a database, run the following command in psql:

```
CREATE DATABASE [Database to create]
WITH TEMPLATE [Database to copy]
OWNER [Your username];
```

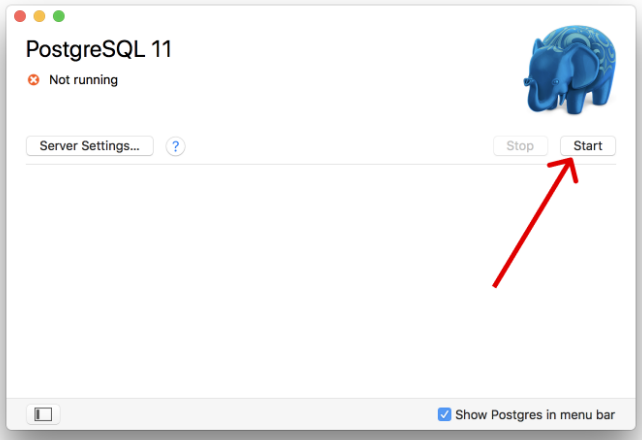
For more information continue reading.

Starting the Server

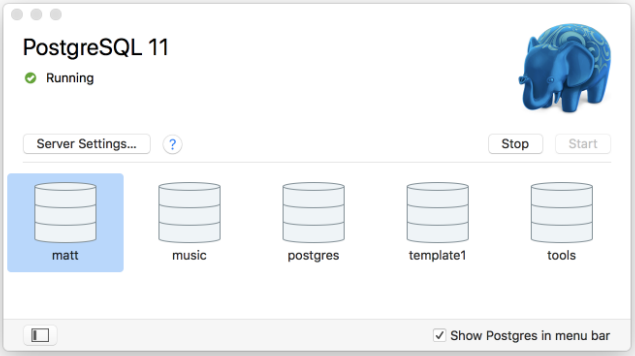
The first step to copying a database is to open psql (the postgreSQL command line). On a macOS this can be done when you start the server.

Open the Postgres app:

In order to start the server, click the start button.

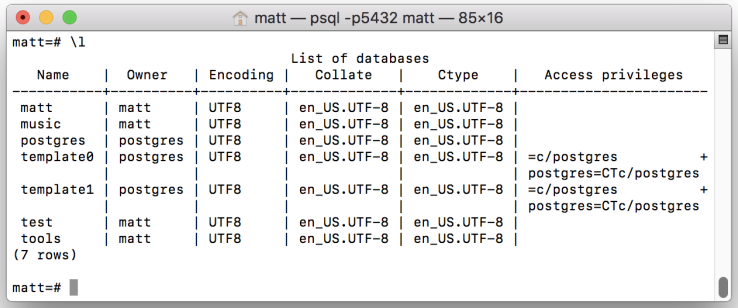


Once this is done, a list will appear showing your databases:



Double-click a database in order to open a psql command line interface. This will open a new window with a connection:

Now that a connection has been established, we can begin writing queries. You can switch to other databases by typing "\c [Database Name]". To look at all the databases, the \list or \l meta-command can be used:



Copying the Database

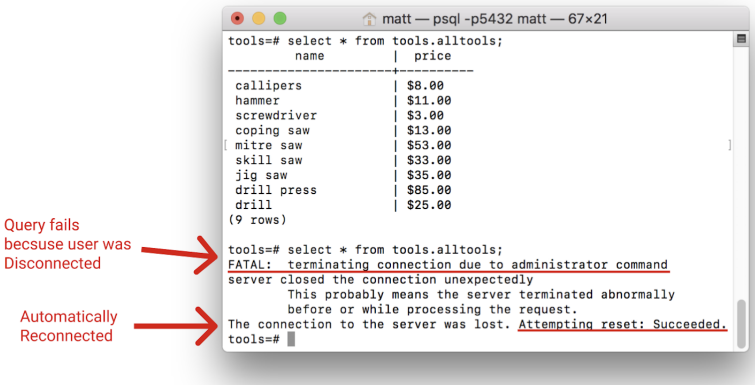
```
CREATE DATABASE [Database to create]
WITH TEMPLATE [Database to copy]
OWNER [Your username];
```

Replace the bracketed portions with your database names and username. This query will generate a copy of the database as long as the "Database to copy" is not currently being accessed. If the "Database to copy" is being accessed by a user, that connection will have to be terminated before copying the database. To do this, run the following command:

```
SELECT pg_terminate_backend(pg_stat_activity.pid)
FROM pg_stat_activity
WHERE pg_stat_activity.datname = '[Database to copy]'
AND pid <> pg_backend_pid();
```

This query will terminate any open connections to the "Database to copy", and will cause brief interruptions to anyone accessing the "Database to copy". It will disconnect users

from the database, however psql will automatically reconnect a user whenever they run their next query as shown below:



Once they reconnect they can then run queries again against the database.

Note: They will not be able to reconnect until the database is completely copied.

Once you terminate the connections, create the copy using the first command to CREATE a copy of the specified database. Due to the fact that people can reconnect between the time you terminate and the time you copy, you may want to structure your commands like so:

```
SELECT pg_terminate_backend(pg_stat_activity.pid)
FROM pg_stat_activity
WHERE pg_stat_activity.datname = '[Database to copy]'
AND pid <> pg_backend_pid();
CREATE DATABASE [Database to create]
WITH TEMPLATE [Database to copy]
OWNER [Your username];
```

When structured and run like this, the CREATE DATABASE command will run immediately after terminating connections. This will help ensure no connections form between terminating connections and copying the database.

How to Export PostgreSQL Data to a CSV or Excel File

PostgreSQL has some nice commands to help you export data to a Comma Separated Values (CSV) format, which can then be opened in Excel or your favorite text editor.

To copy data out first connect to your PostgreSQL via command line or another tool like PGAdmin.

Copying Full Tables

To copy a full table to a file you can simply use the following format, with *[Table Name]* and *[File Name]* being the name of your table and output file respectively.

```
COPY [Table Name] TO '[File Name]' DELIMITER ',' CSV HEADER;
```

For example, copying a table called *albums* to a file named */Users/dave/Downloads/albums.csv* would be done with.

```
COPY albums TO '/Users/dave/Downloads/albums.csv' DELIMITER ',' CSV HEADER;
```

Note, PostgreSQL requires you to use the full path for the file.

Copying a Query Result Set

Besides exporting full tables you can also export the results of a query with the following format where *[Query]* and *[File Name]* are your query and output file name respectively.

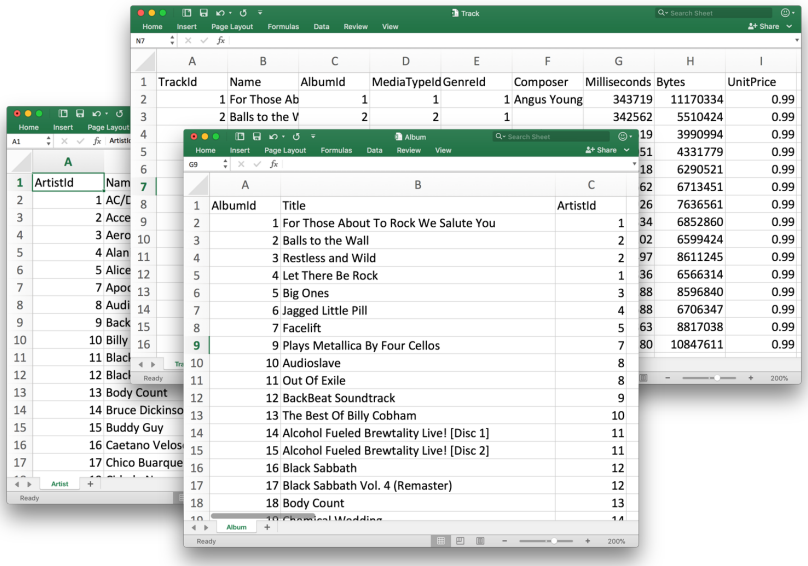
```
COPY ([Query]) TO '[File Name]' DELIMITER ',' CSV HEADER;
```

For example, the following query exports all the blues (genre #6) tracks from a table.

```
COPY (SELECT * FROM tracks WHERE genre_id = 6) TO '/Users/dave/Downloads/blues_tracks.csv' l
```

Opening

After you have run the copy command you can then open the .CSV file(s) with Excel or your favorite text editor.



Did you know, that you can also import data from CSV or Excel files into PostgreSQL?

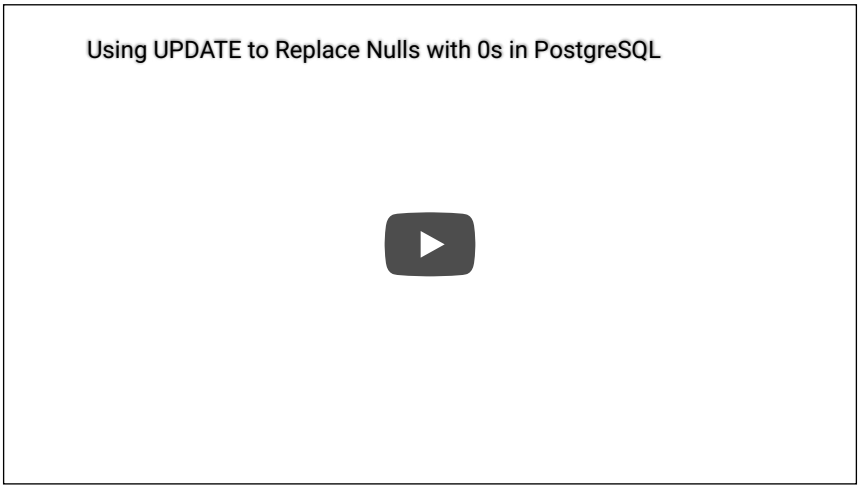
How to Replace Nulls with 0s in SQL

```
UPDATE [table]
SET [column]=0
WHERE [column] IS NULL;
```

Null Values can be replaced in SQL by using UPDATE, SET, and WHERE to search a column in a table for nulls and replace them. In the example above it replaces them with 0.

Cleaning data is important for analytics because messy data can lead to incorrect analysis. Null values can be a common form of messy data. In aggregation functions they are ignored from the calculation so you need to make sure this is the behavior you are expecting, otherwise you need to replace null values with relevant values.

Video



How to use the UPDATE command?

The UPDATE command is a DML command as opposed to a DDL (Data Definition Language), DCL (Data Control Language), or TCL (Transaction Control Language) command. This means that it is used for modifying preexisting data. Other DML commands include: SELECT, INSERT, DELETE, etc.

UPDATE takes a table and uses the SET keyword to control what row to change and what value to set it to. The WHERE keyword checks a condition and, if true, the SET portion is run and that row is set to the new value. If false, it is not set to the new value.

Update can be used for a lot of different problems. For example:

To add 1 to every value in a column you can run:

```
UPDATE [table]
SET [column]=[column]+1;
```

Takes the values in a column and adds 1 to them.

To set every value to a random integer on the interval [1,10]:

```
UPDATE [table]
SET [column]=1+random()*9::int;
```

Generates a random double precision (float8) type number from [0,1), multiplies it by 9, and adds 1 to that value and casts it to an integer type for each row.

To set values to 0 for even 1 for odd:

```
UPDATE [table]
SET [column]=MOD([column],2);
```

Uses MOD to set the column values to the remainder of the column values divided by 2.

Summary

- To replace Nulls with 0s use the UPDATE command.
- Can use filters to only edit certain rows within a column
- Update can also be used for other problems like:
    - Generating random data
    - Adding one to every row in a column (or where a condition is true)
    - Setting Values based on if a column is even or odd
    - Etc.

How to Start a PostgreSQL Server on Mac OS X

There are two main ways to install PostgreSQL on mac OS X.

1. The homebrew package manager
2. Downloading the app file from postgresapp.com.

Using Homebrew

Homebrew can be installed by running the following command in a terminal:

```
/usr/bin/ruby -e "$(curl -fsSL
https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

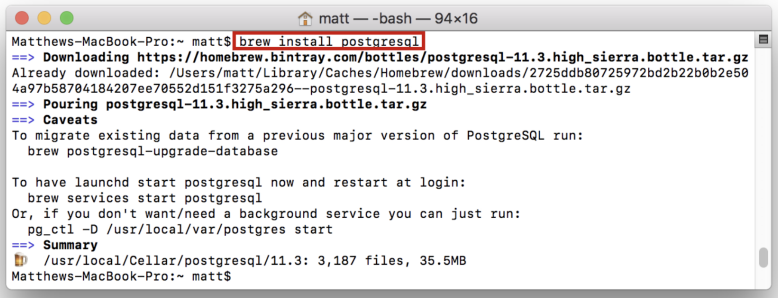
If Homebrew is already installed, make sure that it is up to date by running:

```
brew update
```

Then ensure there are no conflicts or errors using:

```
brew doctor
```

Homebrew is a powerful package manager with many uses, including installing and running postgreSQL. This can be done by typing the following command into a terminal:



Now that postgres is installed the default server can be started by running the command:

```
pg_ctl -D /usr/local/var/postgres start
```

This will start up a postgres server hosted locally on port 5432. The server will be run out of the directory `/usr/local/var/postgres`.

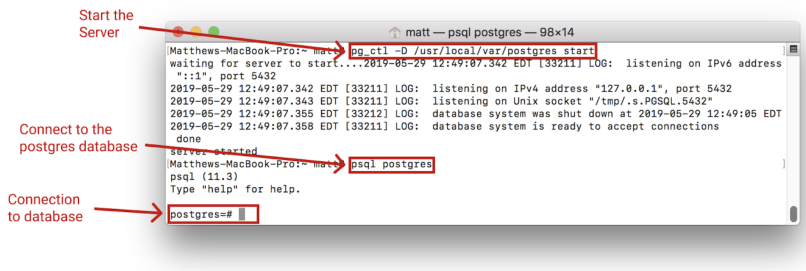
It can now be accessed by typing the following command:

```
psql postgres
```

This will connect to the server and access the postgres database. Once this is done:

- Schemas and tables can be created
- Data can be loaded and deleted from the database
- Queries can be run

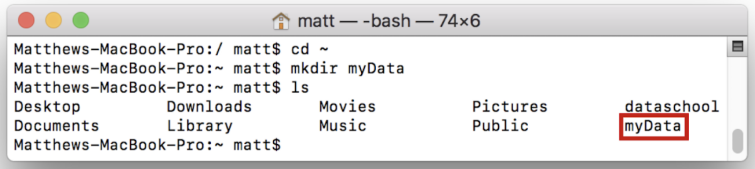
The process should look like this:



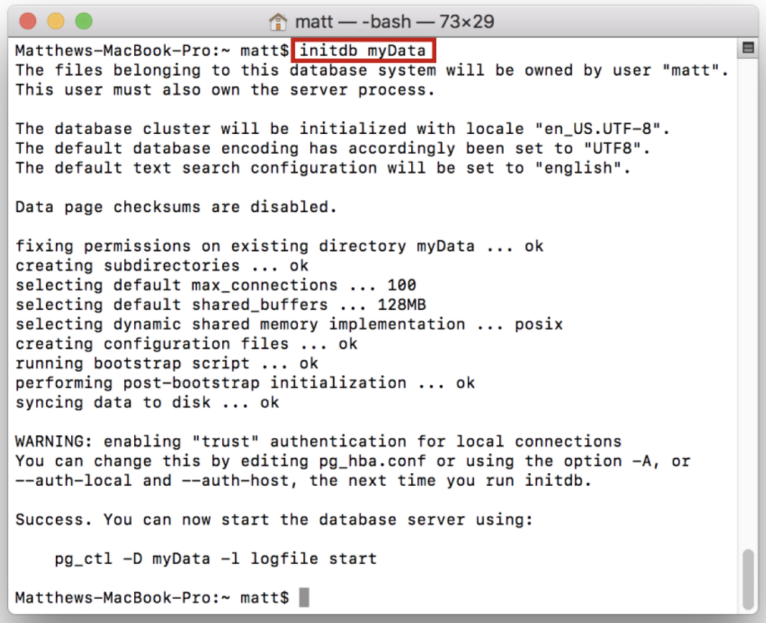
This shows that the server has been started and can be connected to.

(Optional) Creating a Custom Data Directory

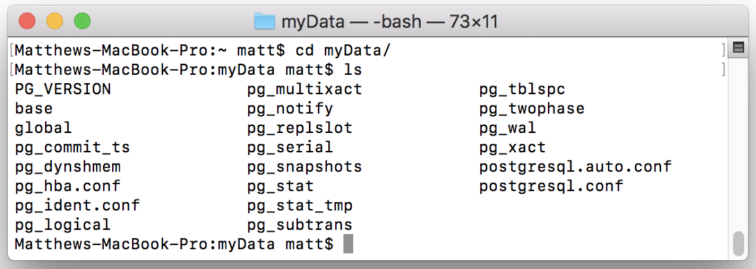
A custom data directory can also be used for a server. To do this, first create a directory to be used as the server location. For example, create a directory called myData in the home directory:



Once the directory is created, the server can be initialized. This means that we configure the directory and add the necessary files to run the server. To do this run the `initdb` command as shown:



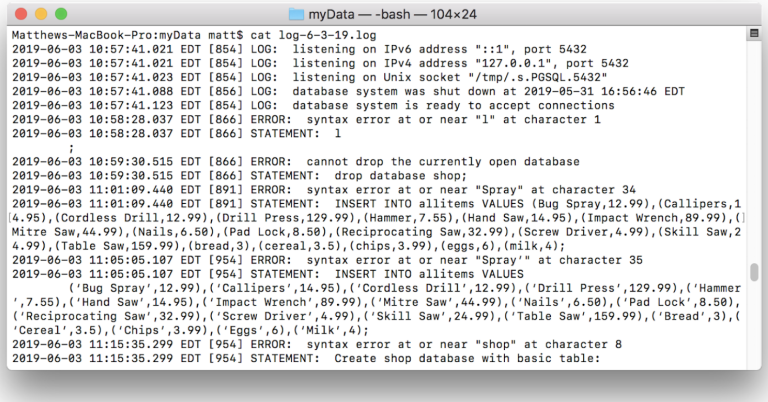
This will fill the myData directory with files necessary to run the server:



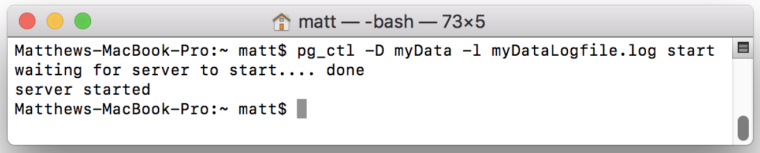
Now that the server is initialized and the log file is created, you can start the server from this directory. To do this use the command and substitute in for the specified values:

```
pg_ctl -D [Data Directory] -l [Log file] start
```

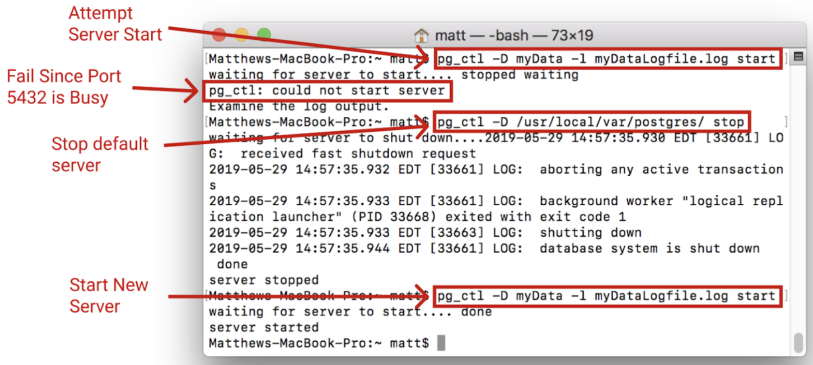
The “Data Directory” refers to the directory that was just initialized (in this case myData). The “Log file” is a file that will record server events for later analysis. Generally log files are formatted to contain the date in the file name (e.g. “2018-05-27.log” or “myData-logfile-2018-05-27.log”) and should be stored outside of the database that they are logging so as to avoid unnecessary risks. Log files can be dense to read but are very useful for security and debugging purposes:



The command above will generate a log file like the one shown, start the server, and tie the log file to the server. If a log file is not specified, events will be logged to the terminal:



The server will only start if the port is free. If the default server is running it must first be stopped using the `pg_ctl -D /usr/local/var/postgres stop` command:



Once started, it can be connected to the same way as before using:

```
psql postgres
```

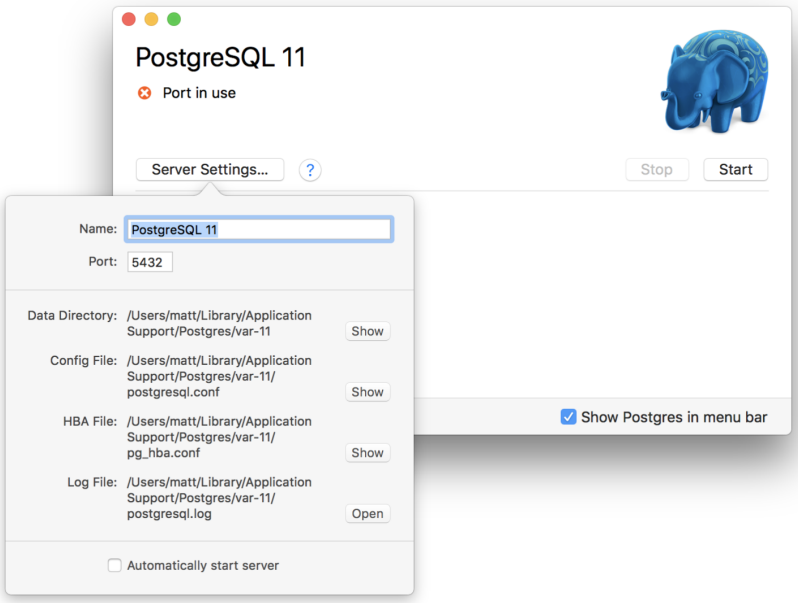
Using PostgreSQL App

To run a server through the postgres app, the first step is to download the program. The app can be downloaded on postgresapp.com. Once the app is downloaded and moved into the applications folder, the app can be opened.

Open the Postgres app:

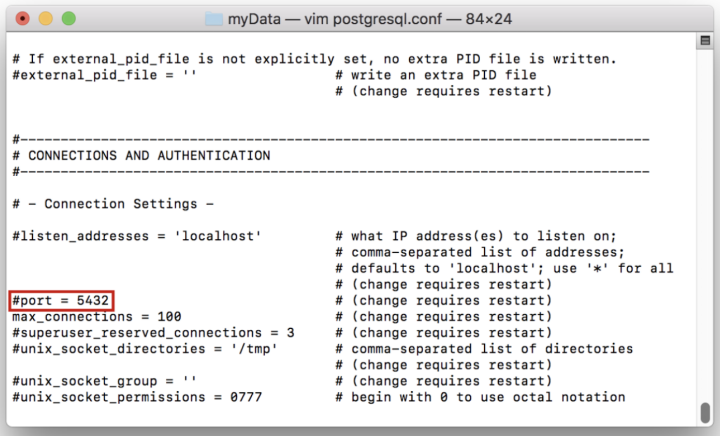
In order to start the server, click the start button.

This will start the server. Details on the server can be found by opening the server settings:



This interface shows all the essential information regarding the server. It also allows the port to be changed very easily. This is useful because multiple PostgreSQL servers can

Note: To change the port in the terminal, the 'postgres.conf' file (which can be found in the data directory) must be edited. This looks like the following:



Using Terminal with the PostgreSQL App

Once the app has been downloaded, command line tools can be used as well. These tools can be accessed by typing:

```
/Applications/Postgres.app/Contents/Versions/latest/bin/[Tool Name][Options and/or Arguments
```

For example, the 'postgres' database on the server can be connected to using the psql tool with postgres as an argument:

```
/Applications/Postgres.app/Contents/Versions/latest/bin/psql postgres
```

Rather than typing out the full path each time however, the path can be added to a file that will allow significantly easier access to the tools, allowing the tools be accessed from any directory on the computer. To do this, the following command can be run in the terminal:

```
sudo mkdir -p /etc/paths.d && echo /Applications/Postgres.app/Contents/Versions/latest/bin
```

Once this is done, the 'postgres' database can be accessed by simply typing:

```
psql postgres
```

Summary

- Homebrew:
  - Download/update Homebrew
  - Use Homebrew to install postgres
  - (Optional) Create New Data Directory
    - `initdb`
  - Start Server
- App:
  - Download app and move to Applications
  - Run App
  - (Optional) Set different port for multiple servers
  - Start Server
  - (Optional) Add path so that command line tools are easy to access

References

1. https://www.postgresql.org/docs/10/app-initdb.html
2. https://postgresapp.com
3. https://www.postgresql.org/docs/10/app-pg-ctl.html

Importing Data from CSV in PostgreSQL

Importing from CSV in PSQL

As mentioned in this article on exporting data to CSV files, CSV files are a useful format for storing data. They are usually human readable and are useful for data storage. As such, it is important to be able to read data from CSV articles and store the data in tables. This can be done in psql with a few commands.

Syntax:

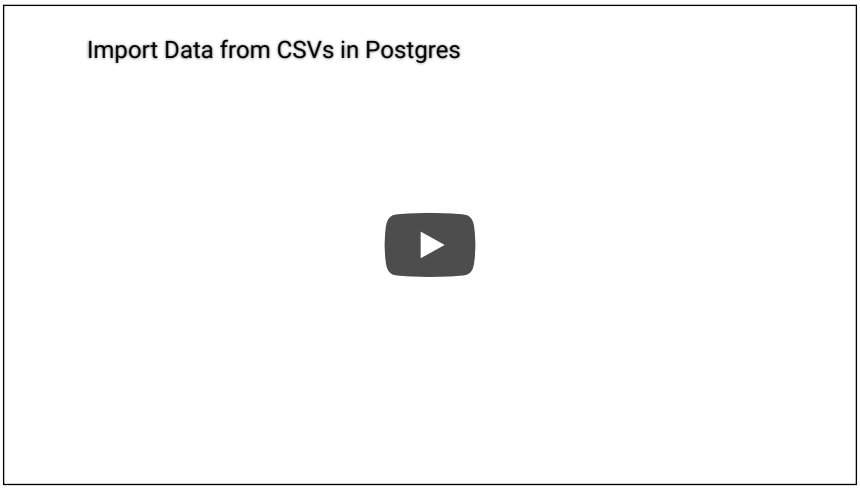
```
COPY [Table Name](Optional Columns)
FROM '[Absolute Path to File]'
DELIMITER '[Delimiter Character]' CSV [HEADER];
```

Key Details:

There are a few things to keep in mind when copying data from a csv file to a table **before importing the data**:

1. **Make a Table:** There *must* be a table to hold the data being imported. In order to copy the data, a table must be created with the proper table structure (number of columns, data types, etc.)
2. **Determine the Delimiter:** While CSV files usually separate values using commas, this is not always the case. Values can be separated using '|'s or tabs (\t) among other characters. (NOTE: for tab delimited CSV files (also known as TSV files however the CSV command is still used for TSV) use: "DELIMITER E'\t' " The 'E' allows for the tab character to be used)
3. **Does the Data Have a Header:** Some CSV files will have Headers while others will not. A Header is a file which contains the column names as the first line of values in the file. If a header is present, include **HEADER** at the end of the query. If there is not a header in the data, do not include **HEADER**.

Video:



Example:

Take this list of items as an example:

```
name,price
Bug Spray,$12.99
Callipers,$14.95
Cordless Drill,$12.99
Drill Press,$129.99
Hammer,$7.55
Hand Saw,$14.95
Impact Wrench,$89.99
Mitre Saw,$44.99
Nails,$6.50
Pad Lock,$8.50
Reciprocating Saw,$32.99
Screw Driver,$4.99
Skill Saw,$24.99
Table Saw,$159.99
apple,$1.50
bananas,$4.00
bread,$3.00
cereal,$3.50
cheese,$3.40
chips,$3.99
eggs,$6.00
gloves,$11.99
hat,$7.00
hoodie,$19.99
jacket,$15.99
milk,$4.00
scarf,$14.99
shorts,$7.50
socks,$5.00
t-shirt,$5.99
tie,$15.99
```

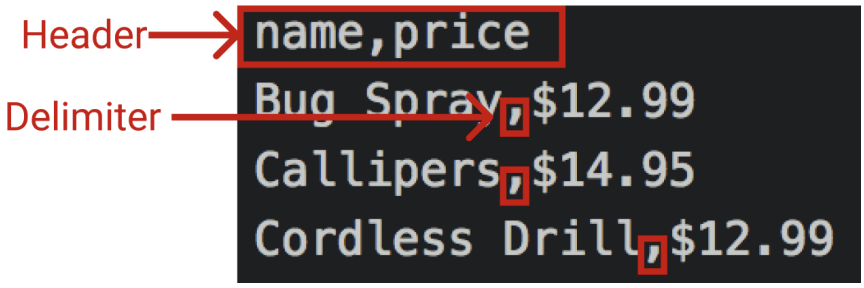
This data contains two columns: ‘name’ and ‘price.’ Name appears to be a VARCHAR due to it’s different lengths. Price appears to be MONEY. This will help in creating the table to load the CSV file into.

The first step, as stated before, is to create the table. It must have at least two columns, one a VARCHAR type and the other a MONEY type:

```
shop=# CREATE TABLE items(item VARCHAR(25), value MONEY);
CREATE TABLE
```

Note: It is also possible to import the csv data to a table with more than 2 columns, however the columns that should be copied to will need to be specified in the query (e.g. **COPY items(item, value) FROM**...).

Now that a table, 'items,' has been created to house the data from the csv file, we can start writing our query. The second step in copying data from CSV is to check the delimiter and the third step is to check for a header. In this case, the delimiter is ',' and there is a header in the file:



Since the header and the delimiter is known, the query can now be written. As before, the syntax is:

```
COPY [Table Name](Optional Columns)
FROM '[Absolute Path to File]'
DELIMITER '[Delimiter Character]' CSV [HEADER];
```

So in order to import the csv we will fill out the necessary parts of the query:

- [Table Name] - items
- [Absolute Path] - this is the location of the csv file. In this example it is on the desktop, so the path is: '/Users/matt/Desktop/items.csv'
- [Delimiter Character] - ','
- [HEADER] - the data does have a header

So the final query will be:

COPY items FROM '/Users/matt/Desktop/items.csv' DELIMITER ',' CSV HEADER;

Running this query will look like:

```
shop=# COPY items FROM '/Users/matt/Desktop/items.csv' DELIMITER ',' CSV HEADER;
COPY 31
```

The message COPY 31 indicates that 31 rows were successfully copied from the CSV file to the specified table.

Summary

- Make a table to store the data
- Determine what delimiter was used
- Verify if a header is exists
- Copy from the csv file

Meta commands in PSQL

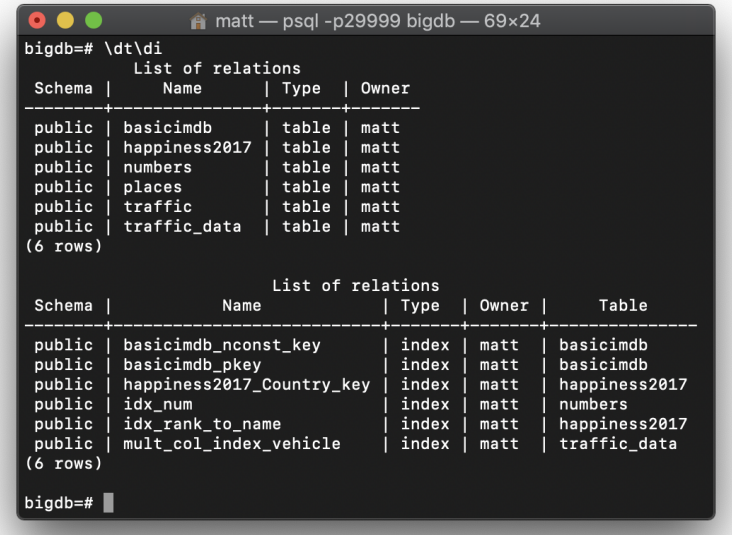
Meta commands are a feature that psql has which allows the user to do powerful operations without querying a database. There are lots of metacommands. Here is a list of some of the more common meta commands along with a very brief description:

- \c [database name] - connect to a specified database
- `\l` - list all databases
- `\d` - display tables, views, and sequences
  - `\dt` - display just tables
  - `\dv` - display views
  - `\dm` - display materialized views
  - `\di` - display indexes
  - `\dn` - display schemas
  - `\dT` - display data types
  - Etc.
- `\sv` [view name] - show a views definition
- `\x` - toggle expanded display. Useful for tables with a lot of columns being accessed
  - Can be toggled on/off or set to auto
- `\set` - list all internal variables
  - `\set` [Name] [Value] - set new internal variable
- `\unset` [Name] - delete internal variable
- `\cd` - change the directory that psql is working in
- `\!` [Command] - execute shell command
  - Ex. `\! ls` or `\! pwd`
- `\timing` - toggles timing on queries
- `\echo` [message] - print the message to the console
- \copy - copies to a file
- `\i [filename]` - execute commands from a file
- \o [file] - writes output to file instead of console
- `\q` - exits psql

Advanced Meta Command Techniques

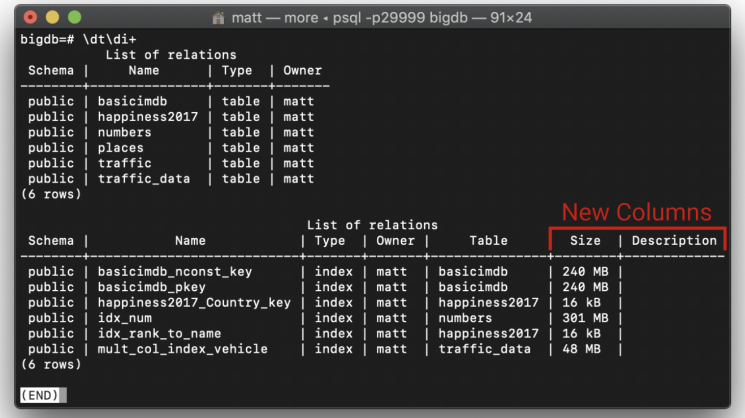
Multiple Meta Commands

Multiple meta commands can be used in one line. For example you could use `\dt\di` to list all tables and then list all indexes with additional technical information on the indexes.



Extra Details

Adding + to the end of the meta command that lists items will provide a small amount of extra technical information. This will work on any `\d` commands as well as some others.



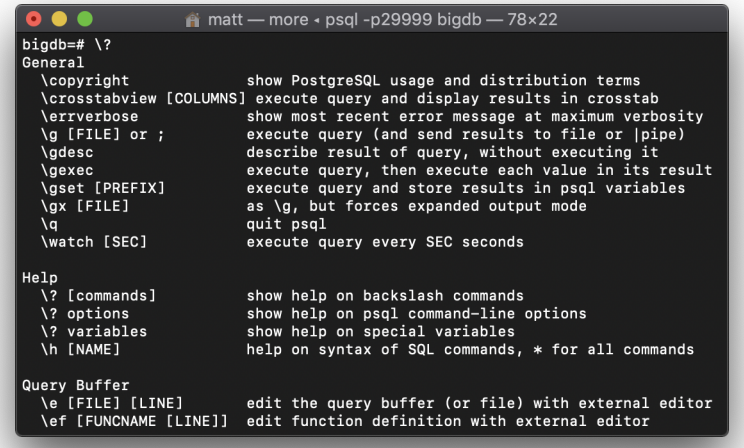
Common Error

Meta commands are delimited by a new line as opposed to a `;`. This means that you would never see a meta command look like this: `\x;` The semicolon at the end is unnecessary and will throw an error:

```
bigdb=# \x;
Invalid command \x;. Try \? for help.
```

More Meta Commands

There are quite a few more meta commands that were not listed as they were for relatively niche usages and not as commonly used. For a full list of meta commands use `\?`. This will bring up the meta command help page with a full list of every meta command and a brief description of its functionality.



This image shows the first commands from `\?` There are 102 commands in total.

Summary

- Meta commands are useful commands that can be run from a psql client.
- All metacommands begin with `\`
- Adding + will provide extra detail on certain metacommands
- For a full list of meta commands use `\?` in psql
- Do not end meta commands with `;`

Outputting Query Results to Files with \o

```
\o [filename].txt
[Query or Queries to write to file];
\o
```

Outputting query results to a file instead of the terminal allows the data to be saved later analysis. The results can be shared easily and provide a snapshot of the data at the time of the query.

In order to output to a file, several methods can be employed. In this article the `\o` method of writing to files will be explored. One other method is using either `\copy` or `COPY` which are discussed in this article.

Video

The \o metacommand

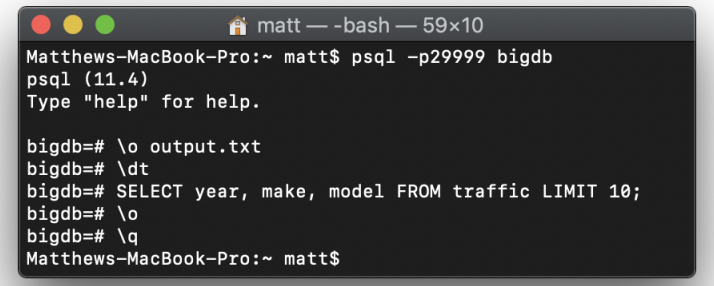
`\o` is a metacommand. This means that it is delimited by a new line in the terminal rather than being part of the query. Simply write the metacommand and then press enter/return to run the command.

`\o` works like this:

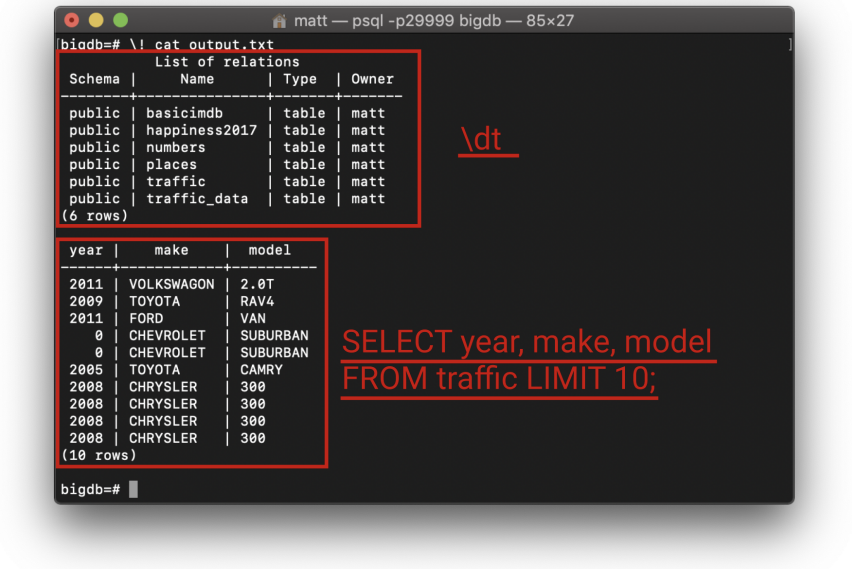
- `\o [filename].txt`
  - This will start writing the results of subsequent queries and certain metacommands to the specified file.
- [Query or Queries to write to file];
  - Since these lines are after `\o [filename]`, these queries will be logged to the file.
    - Depending on version, the results of `\d` as well as `\di`, `\dt`, etc will be printed to the new file. (see example below)
    - `\!` Commands will not be printed to the file.
    - If the output of a command is logged on the console, this means that it was not written to the file. If the result is not shown on the console, then the result was sent to the file
- `\o`
  - Using `\o` again will close the file. This means that after running `\o` the file is done being written and can not be reopened using `\o [filename].txt`. Running `\o [filename].txt` again with the same filename **will overwrite the file.**
  - Can also be terminated with `\q`

Example use

Let's look at an example of `\o` being used:



As you can see, the output of `\dt` and the `SELECT` query are not shown on the console. This indicates that they are being logged in the file. We can confirm this if we check the file. This can be done manually outside of psql or through psql using the `\!` meta command:



`\!` allows the user to use terminal commands and see the results without leaving the psql environment. As such, the file contents can be checked quickly using commands like `cat` which displays the contents of the text file to the screen.

Summary

- `\o` can be used to write query results to a file instead of the console:
  - `\o [filename].txt`
  - `[Query or Queries to log to file];`
  - `\o`
- Can write the results of certain meta commands to the file.
- Can be checked using: `\! cat [filename].txt`

How To Generate Random Data in PostgreSQL

There are occasionally reasons to use random data, or even random sequences of data. PostgreSQL supports this with the `random` SQL function. The following are some nice examples of how to use this.

The random() Function

Click to run the following multiple times and you'll see that each time a different random number between 0 and 1 is returned.

If you'd like to scale it to be between 0 and 20 for example you can simply multiply it by your chosen amplitude:

And if you'd like it to have some different offset you can simply subtract or add that. The following will return values between -10 and 10:

Seeding the Random

Often you want random data but would like it to be the same random data every time it's run. To do so we want to set the starting seed (always between 0 and 1) for the random number generator.

Try running the following query multiple times:

Notice that it returns a random result as expected, but unlike above, it's the same random result every time. Change the seed value (.123) in the `setseed` function above and notice that it will now choose a different random value but maintain that on multiple runs. To get the answer correct to the above SQLBox, set the seed to .42.

To understand what's happening, imagine that there is a long list of random numbers that the computer chooses from. Setting the seed is like telling PostgreSQL to always start at the same spot every time.

A quick tip: some SQL interfaces's (like Chartio's) won't let you run/return multiple queries in a connection, which is necessary to set the seed. This can be worked around by using the WITH function as shown here:

Random Sequences

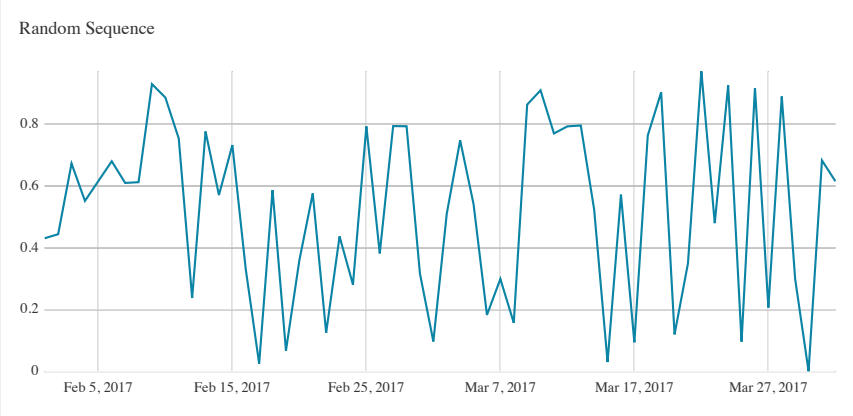
If you'd like full sequences of random data you can use the `generate_series` function to generate a series of dates.

```
... FROM generate_series([start date], [end date], [increment])
```

The following example gets a random value for each day between February 2017 and April 2017.

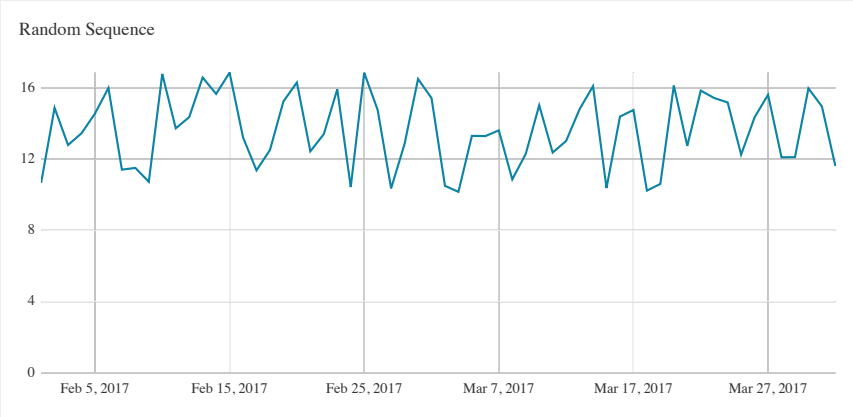
```
SELECT TO_CHAR(day, 'YYYY-MM-DD'), random() FROM generate_series
        ( '2017-02-01'::date
        , '2017-04-01'::date
        , '1 day'::interval) day
```

We've visualized the sequence with Chartio here to make it more clear what's going on with the data.



The above results are all between 0 and 1 as again that is what's returned from random(). As above, to add an amplitude and minimum offset to it we can simply multiple and add to the random value. The following makes a random sequence with values in the range of 10 to 17.

```
SELECT TO_CHAR(day, 'YYYY-MM-DD'), 10 + 7*random() FROM generate_series
        ( '2017-02-01'::date
        , '2017-04-01'::date
        , '1 day'::interval) day
```

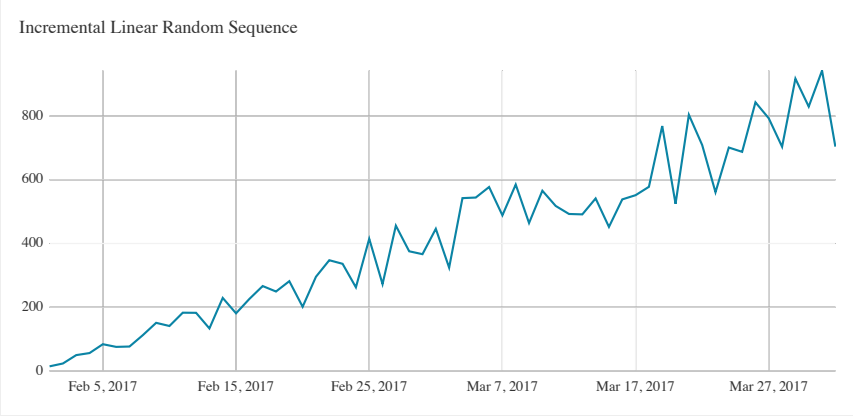


Random Growth Sequence

To make a sequence increase linearly we can use PostgreSQL's `row_number() over()` functions to get an increasing count of what row we're on.

```
SELECT TO_CHAR(day, 'YYYY-MM-DD'), (10 + 7*random())*(row_number() over()) as value FROM ge
        ( '2017-02-01'::date
        , '2017-04-01'::date
        , '1 day'::interval) day
```

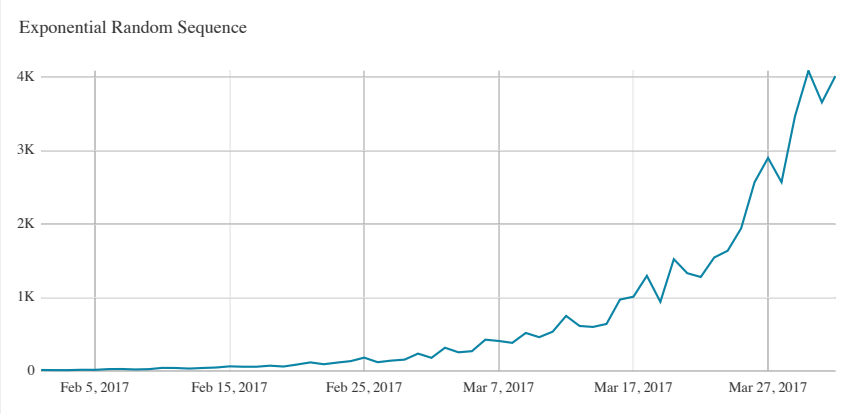
Multiplying the row number by our random makes our data linearly increase as you can see in the chart.



Random Exponential Sequence

If we want to randomly model exponential growth, we can use the `row_number` in the exponent. Here we're having a daily exponential growth of 10% (see the `1.1^(row_number() over())`) in the query:

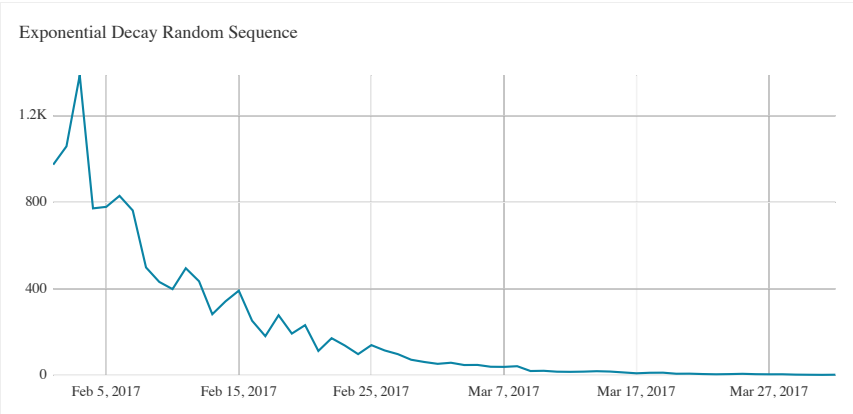
```
SELECT TO_CHAR(day, 'YYYY-MM-DD'), (10 + 7*random())*(1.1^(row_number() over())) as value FR
        ( '2017-02-01'::date
        , '2017-04-01'::date
        , '1 day'::interval) day
```



Random Exponential Decay Sequence

Similarly to get a exponential decay we can take the power of a number less than 1 (see `(.9^(row_number() over()))`).

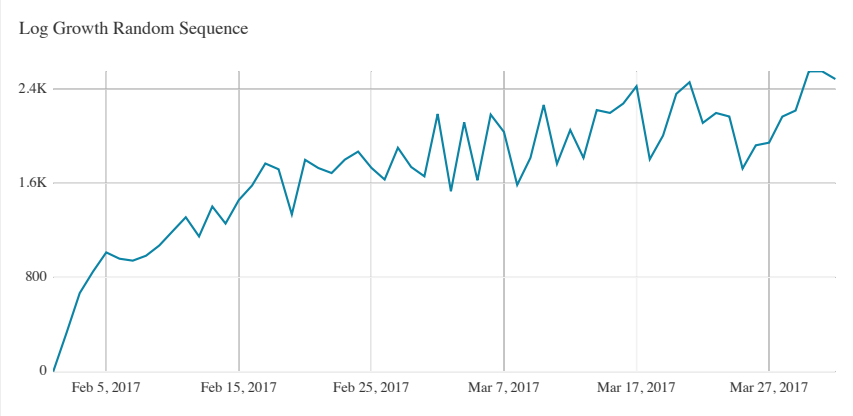
```
SELECT TO_CHAR(day, 'YYYY-MM-DD'), (1000 + 1000*random())*(.9^(row_number() over())) as valu
        ( '2017-02-01'::date
        , '2017-04-01'::date
        , '1 day'::interval) day
```



Random Log Growth Sequence

And PostgreSQL also has a log function we can use to model random logarithmic growth:

```
SELECT TO_CHAR(day, 'YYYY-MM-DD'), (1000 + 500*random())*log(row_number() over()) as value F
        ( '2017-02-01'::date
        , '2017-04-01'::date
        , '1 day'::interval) day
```



There are a lot great things you can do with PostgreSQL's random() function combined with generating series to get sequences. Feel free to play around with a few yourself in the SQLBox below, or using Chartioif you'd like to visualize them as well.

Using ALTER in PostgreSQL

In SQL, tables, databases, schemas, groups, indexes, servers, and more can be modified using the ***ALTER*** command. This command enables the user to modify a specific aspect of the table, database, group, etc. while leaving the rest of the data untouched.

There are many alterable things in postgreSQL heavily discussed in the PostgreSQL Documentation. This article will only focus on a few main uses of ALTER (ALTER TABLE and ALTER DATABASE.) For a comprehensive list, check the documentation here.

Warning: Altering tables and databases alters critical parts of their structure. As such, queries that ran on tables/databases that were altered may no longer work and may need to be rewritten.

Video

ALTER TABLE

Altering tables is a very common use of ALTER. Using ALTER TABLE is very useful for adding, removing, and editing columns:

```
ALTER TABLE traffic
ADD COLUMN nameofdriver VARCHAR;
```

This query will **add a column** called 'nameofdriver'.

This column can be **dropped** by using ALTER as well. To do this:

```
ALTER TABLE traffic
DROP COLUMN nameofdriver;
```

ALTER can also be used to **change the datatype** of a pre-existing column. For example, you can change a boolean to a char:

```
ALTER TABLE traffic
ALTER COLUMN belts
TYPE char USING belts::char;
```

This usage of ALTER takes a column and converts it into a different type using a specified method for this (in this case the cast: belts::char).

Table Constraints

Another usage of ALTER TABLE is to **add table constraints**. For example, if a column should be unique:

```
ALTER TABLE traffic
ADD CONSTRAINT unique_seqid UNIQUE (seqid);
```

This command can also be used to add a constraint to the whole table.

NOTE: An error will be thrown if a constraint is added to a column that already breaks that constraint (e.g. adding the UNIQUE constraint to a non-unique column will throw an error).

Common constraints include: ***NOT NULL***, ***PRIMARY KEY***, and ***UNIQUE*** (full list included in the documation). The constraint can also be dropped using the same command with the ***DROP CONSTRAINT*** command instead:

```
ALTER TABLE traffic
DROP CONSTRAINT unique_seqid;
```

Renaming and Changing Schemas

ALTER TABLE can also be used to rename the table or column that is being accessed. To do this, use the rename command:

```
ALTER TABLE traffic
RENAME TO violations;
```

Or

```
ALTER TABLE traffic
RENAME COLUMN dateofstop TO date;
```

The **schema** that a table is using can be changed by using:

```
ALTER TABLE public.traffic
SET SCHEMA mySchema;
```

ALTER DATABASE

Databases can also be modified using the ALTER command. There are fewer things that can be modified in a Database, however they have very serious effects. As such they often have required permissions to execute them. The things that can be changed using ALTER DATABASE are:

- **Name:** The database can be renamed.

```
ALTER DATABASE [database name]
RENAME TO [new name];
```

- **Allow Connections:** Whether the database allows connections to itself. **NOTE:** this will block all connections when true, even connections from localhost. It will need to be set to false before it can be connected to again.

```
ALTER DATABASE [database name]
WITH ALLOW_CONNECTIONS [true/false];
```

- **Connection Limit:** Limit the number of simultaneous connections. Set to -1 for unlimited connections.

```
ALTER DATABASE [database name]
WITH CONNECTION_LIMIT [number of connections];
```

- **Template:** Can set the database to be or not to be a template. A template is a designation that some tables get which allows the database to be copied by a user with lower privileges so that they can have a pre-structured database and fill it out with their own data.

```
ALTER DATABASE [database name]
WITH IS_TEMPLATE [true/false];
```

- **Owner:** Can set the owner of the database. Only the current database owner and superusers can change the owner.

```
ALTER DATABASE [database name]
OWNER TO [username];
```

- **Tablespace:** Can set which default tablespace is used. A tablespace is a way to logically separate databases on the disk so that they can handle more throughput. See the documentation for more.

```
ALTER DATABASE [database name]
SET TABLESPACE [new tablespace];
```

- **Configuration Parameters:** Can be used to override system preferences on an individual basis. Some possible parameters that can be edited are: enable_indexscan, enable_bitmapscan, statement_timeout, and more.

```
ALTER DATABASE [database name]
SET [configuration parameter] TO [value];

ALTER DATABASE [database name]
RESET [configuration parameter];

ALTER DATABASE [database name]
RESET ALL;
```

- **Example:**

```
ALTER TABLE traffic SET enable_indexscan TO OFF;
```

- This will disable index scans on the specified database

References

- https://www.postgresql.org/docs/11/sql-alterdatabase.html
- https://www.postgresql.org/docs/11/ddl-constraints.html
- https://www.postgresql.org/docs/11/sql-altertable.html