1.SQL as a tool for data science

Relational databases in data science

- [Instructor] Relational databases are the workhorses of business data management. They are used to collect, store, and manage transaction data like sales transactions, as well to analyze large volumes of data like those found in data warehouses. It is safe to say the bulk of data used in businesses and other organizations is stored in relational databases. This makes relational databases an important source for data scientists. Now, relational databases are more than just a storehouse for data. They are data management systems that can be used to perform common transformation and analysis operations like linking data across data stores, filtering and reformatting data depending on how it needs to be used, aggregating data to provide kind of a big picture summary, as well as answering specific questions about business operations. Let's consider what goes into a production data science environment. To start, we have **data sources that can include relational and nonrelational databases. Those nonrelational databases, by the way, are often called NoSQL databases. Here, we'll refer to both of these as data stores**. Application and web logs are another source of data for analysis. These are typically less structured than relational data and they may require significant filtering, formatting, and data extraction operations. Also, it's not uncommon to also need small manually curated data sets such as reference data that may be maintained in a spreadsheet. Now, we also have tools for extracting, transforming, and loading data. Data scientists might need data from multiple sources. In these cases, a common practice is to collect data into a single repository prior to analysis. For example, an analyst might extract data from a transaction processing database, an application log, and a personally managed spreadsheet, and then store that all in a relational database. Now, if it's a really large amount of data, the data scientists may use a big data platform such as Hadoop or Spark. Transformations are operations performed on data to make the data more suitable for analysis. Now, we often need to reformat dates, replace missing values with zeroes, and strip out trailing blanks from strings. Some analysis like building machine learning models can require additional transformations such as changing the scale of numeric values or using even more involved calculations to estimate missing values. Loading an ETL process occurs when data is moved into a data store for analysis. Now, this sounds simple, but large data sets can require specialized techniques like parallel data loads to ensure that the data is loaded in a reasonable period of time. Analysis and modeling tools are used by data scientists for a range of activities like identifying useful information, building predictive models, detecting anomalies, and to make any recommendations. These tools read data from data stores and may write results back to databases as well. The final result, of course, is insight into business problems and relational databases play key roles throughout the data science process, that ultimately lead to those insights.

SQL data manipulation features

- [Instructor] SQL has two types of commands. Data manipulation commands and data definition commands. We'll look at data manipulation commands in this video and discuss data definition commands in the next. Before discussing specific commands, let's understand why data scientist need to know these commands to do their work. A majority of data science work is about collecting, cleaning, and restructuring data. Only after that preparation can we move ahead with analysis work. A common assumption is that about 70% to 80% of time on a data science project is spent on data manipulation, much of that time is spent working with SQL. INSERT is a SQL command for putting data into a table. INSERT statements include the name of the table, the name of the columns with values being inserted, and a list of values to insert. A simple INSERT command puts one row into a table. This set of three INSERT commands adds three rows to the table. The UPDATE command is used to change data that is already in a table. The UPDATE command includes the name of the table to update, the columns that will be updated and the new values that will be assigned, and optionally a WHERE clause that filters which rows will be updated. For example, this command changes the country name from USA to United States. If a WHERE clause is not specified, then all rows in a table are updated. The DELETE command is used to remove rows from a table. The command includes a table name, and an optional WHERE clause to determine which rows should be deleted from the table. For example, this DELETE command removes those rows from the table that have Canada as a country value. If no WHERE clause is specified, all rows in the table are deleted. The SELECT command is used to retrieve data from a relational database. It has many options and features that make it a valuable tool for data scientist who are exploring, collecting, and analyzing data. In this lesson, we will only briefly discuss the SELECT command but later videos will go into more options. The SELECT command includes one or more table names, a list columns to retrieve, optionally one or more joins of tables, a WHERE clause, aggregate functions, and sorting and grouping commands. A SELECT command includes at least one table name indicating which table to retrieve columns from, a list of columns to retrieve from a table, and this may be the special symbol star which means return all columns. This command shows how to retrieve all columns from all rows in the country regions table. Optionally, a WHERE clause can be used to filter the rows returned. As in this example, which returns only rows with the specified row IDs. In the next video, we will quickly review how to create tables, views, and indexes.

SQL data definition features

- [Instructor] SQL has two types of commands: data manipulation commands, and data definition commands. We're going to discuss data definition commands in this video. Data definition commands are used to define structures for organizing data in a relational database. Imagine you need to manage thousands of paper reports for your job. Now, you could just leave this in a pile and search through the pile every time you need to find a particular report. However, it's much easier to find a paper report when they're organized in boxes by type of report and then organized within a box by date. The same idea applies to organizing data in a relational database. We structure data using tables, which are collections of related data records, indexes, which are sets of data about the location of records, views, which are used when we repeatedly want to access the same set of data from one or more tables, and then finally schemas, which are collections of tables, indexes, views, and other data structures. Tables are used to organize related sets of data, like information about employees, products, and events. We use the CREATE TABLE command to define a table. The CREATE TABLE command includes the name of a table. In this case, the name is staff. Next is a list of column names, such as id, followed by a data type, such as INTEGER. Tables typically have multiple columns. Each column is used to store one piece of data. Each column has a data type. Some of the most commonly used data types are integers, varchars, which are for varying length character strings, and dates. There are many more types for representing more specialized kinds of data, from monetary units to JSON structures. The CREATE TABLE command has a clause called PRIMARY KEY. This indicates which column or set of columns uniquely identifies each row. When a primary key is defined on a table, whenever a row is inserted into the table, it must have values in the primary key column or columns. Also, that key can't already exist in the table. It has to be unique. Often, we want to look up one piece of information in a table. This is like finding a single book in a library. Before libraries had databases, they used card catalogs. These were file cabinets filled with cards with information on where to find books. So you could look up a book by name, and that corresponding card would indicate where in the library you could find the book. Using the catalog is a lot faster than walking through the library, checking every book on the shelves. Indexes in relational databases serve the same purpose. The CREATE INDEX command lets us build an index to quickly look up rows and tables. The CREATE INDEX command takes the name of the index. It's a common practice to use a prefix like idx to indicate that this is an index. The ON clause indicates which table will have the index. The keyword ON is followed by the name of the index table. The USING clause indicates the columns that will be used in the index. In this example, the last name of the staff member is used in the index. This command will allow the query engine of the database to quickly find the row of data by last name. Now, views are structures that help us focus on the most important data for a particular use. For example, someone in human resources might be interested in knowing an employee's start date and their department name while someone else in finance might be more interested in their salary. Views help make it easy to query data without including unneeded columns in our results. Views also help retrieve data from multiple tables without having to repeatedly specify joins. The CREATE VIEW command creates a view. The command includes a name for the view. This view name is staff_div, indicating that it includes columns from the staff and company division tables. The next section of the CREATE VIEW command is a SELECT statement, which defines the columns to include in a view. The columns in this SELECT statement use a prefix, such as s or cd. These are called aliases. They're a short-hand way of indicating which table is a source for each column. The FROM clause indicates the tables used in the view. In this case, it also includes an alias. When multiple tables are used, there's often a JOIN between tables. The JOIN clause indicates how to link rows between tables. The LEFT JOIN is used when all rows from the left side of the equal statement should be included, even if a corresponding column is not found on the right side table. In this example, the s or staff table is on the left, and the cd or company division table is on the right. Schemas are like floor plans. They organize groups of related structures such as database, tables, and views. The CREATE SCHEMA command tells the database management system to create an organizational space, and in that space, we will keep related tables, indexes, views, and other structures. The schema command takes a name of the schema. Now, depending on the database management system that you are using, there may be additional clauses in the CREATE SCHEMA command. Data definition commands create data structures for organizing data. The top-level grouping is a schemas. Schemas can have multiple tables and each table can have optional indexes. Views can also be created to streamline querying, allowing developers and analysts to focus on the most important columns and easily retrieve data from multiple tables.

ANSI standard SQL and variants

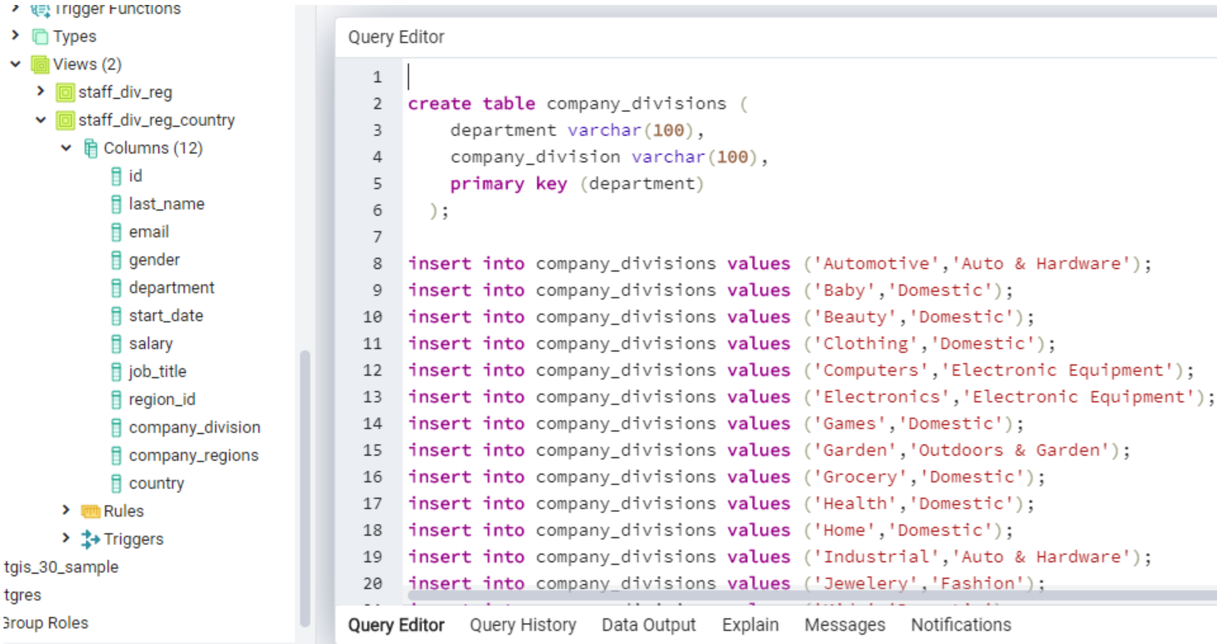
- [Instructor] SQL is a language that has changed over time. Databases offer different versions of SQL, so when we talk about SQL in general, we usually refer to the ANSI standards SQL. The American National Standards Institute, known as ANSI, defined a standard version of SQL in the mid-1980s. Since then, the standard has been updated seven times to add features, many of which are useful to data scientists. With changes over time in the standard and database management systems offering different versions of SQL, what does this mean for data scientists? It means the SQL implementation that you use may be different from the current ANSI standard and it may be different from the SQL used in this course. The foundational SQL commands like insert, delete, and most create commands will be the same across implementations, but you should check your database documentation, especially for missing data types and functions. Some databases like MySQL do not support windowing functions, which are really quite useful for some types of data science analysis. In addition, other databases like PostgreSQL offer geographic data types that are not generally available.

Installing PostgreSQL

- [Instructor] It's time to install our database. We'll be using PostgreSQL, also called Postgres for our work. Let's start at the Postgres download page at www.postgresql.og/download. There are a number of options for different operating systems. I'm working on a Mac, so I'll select the link for that option. Now there are several ways to install Postgres on a Mac and I'm going to use the Enterprise DB interactive installer. It's easy to use and it's available for Mac, Windows and Linux operating systems. So I'll click on the link for that. I'll select the most recent version of Postgres and the Mac OS X operating system. If you're using Windows, you can select the appropriate Windows version for your platform. And I'll select download now. Now that the Postgres package is downloaded, I'll open it and start the installation. Now a dialogue message will appear on a Mac operating system, but you may not see this in Windows. I'm simply going to agree to open, and I'll provide my operating system password. The installation may be slightly different in Windows, but the process is basically the same. So, the wizard has started. I'll select next. I'm going to choose all the defaults and I'll give a password for the database. We'll want to remember that. We'll need that when we access the database. Now while Postgres is installing, I want to mention that one of the advantages of using the Enterprise DB installer, is that it automatically installs pgAdmin, a GUI admin tool that we'll be using. We don't need any additional packages, so I'll **uncheck the stack builder** option, and select finish. Now, I'll start pgAdmin. Now you may see a notice that there's a newer version of pgAdmin available. We're just going to ignore that. This version works just fine for what we're doing. In the **left pane**, you'll see a hierarchical navigation control that includes a **Postgres database. Let's open that** and I'll specify the password I provided during the installation. Now, there is a Postgres database that is the default database. But I want to create a different one for our project, so I'll going to control click on the Mac, or **right click in Windows**, and I'm going to **select create database**. I'm going to **create a database called data_sci**, short for data science. I'll save that. Now, let's open data_sci and **navigate down through schemas**, through the public schema, and let's take a look at the things that are available. We have, for example**, a list of tables**. If we click on that, we'll notice that there are no tables listed. **We'll create tables and insert data next**.

CREATE TABLE and INSERT DATA:02_02 file

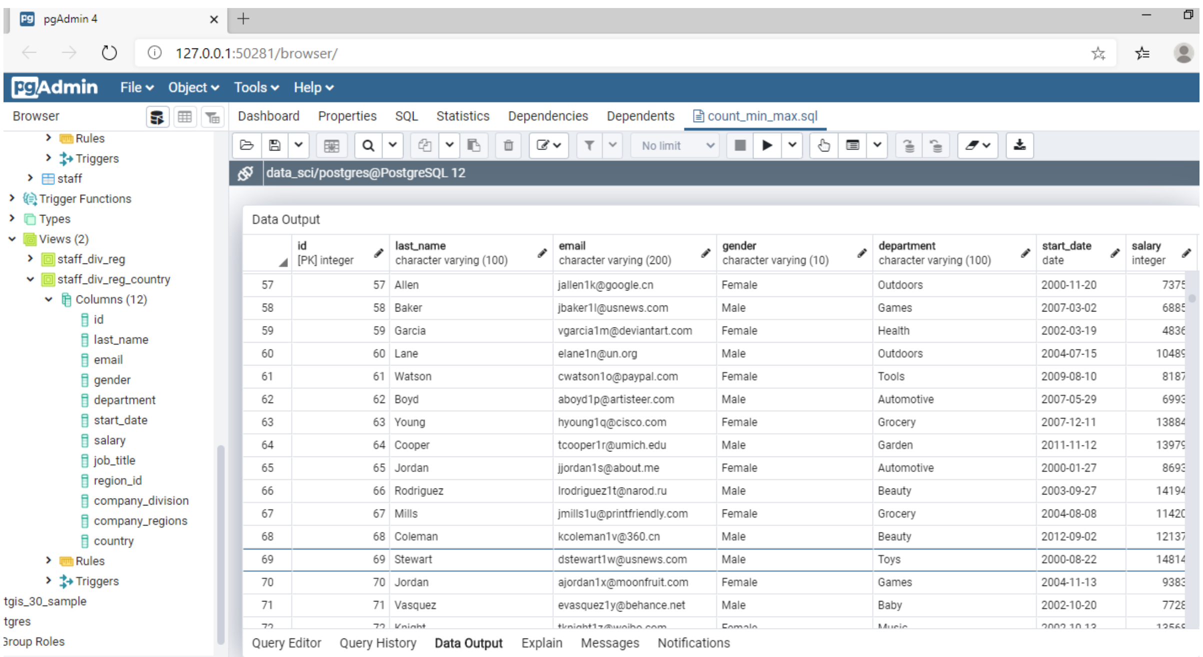
- [Instructor] Now I have **pgAdmin started,** and I'm going to click on the Tools menu, and select Query Tool. This will open a query window. Notice, in the blue bar in the query window, it says data_sci on postgres. That means we're working with the data_sci database, which is what we want. So now, let's open a script file. And if you have access to the exercise files, you'll want to open the mock_staff.sql file, which I have in my user directory, and I will open that, mock_staff, and I will select that. And what you'll notice here, mock_'s staff table **includes a series of create table statements and insert statements, and this will create the schema** and the data that we'll need for our exercises. So I'm going to go up to the Execute button, and I will execute this statement. And that has finished executing. Now let's Control click on the Tables item in the hierarchical navigation, or right-click if you're using Windows. And let's select Refresh. And now let's open the Tables again. What you'll notice is we have three tables, company_divisions, company_regions, and staff. So let's take a look at their contents. I'll open another query window. pgAdmin lets you have multiple query windows open at once. This helps when you're working with multiple queries. So first, we'll query company_divisions. SELECT * FROM company_divisions, and we'll execute that. And you'll notice we receive 21 rows back, and the two columns in this table are a list of departments and then the company divisions to which those departments belong. So, for example, Baby, Beauty, and Clothing are all in Domestic, while Computers and Electronics are in the Electronic Equipment division. Now, let's look at company_regions. And we'll execute that, and we'll notice this is slightly different. We have an integer ID for our key, and we have a series of geographic regions, each grouped into one of two countries. The staff table has information about employees at this fictional company. We'll select * from staff and execute. We'll notice we have a thousand rows returned. The staff table includes columns for the last name, email address, gender, department, start date, salary, job title, and region id. We'll be using these tables to demonstrate some useful ways to use SQL for data science analysis.


Trigger Functions
Types
Views (2)
staff_div_reg
staff_div_reg_country
Columns (12)
id
last_name
email
gender
department
start_date
salary
job_title
region_id
company_division
company_regions
country
Rules
Triggers
tgis_30_sample
tgres
Group Roles
Query Editor
create table company_divisions (
department varchar(100),
company_division varchar(100),
primary key (department)
);
insert into company_divisions values ('Automotive','Auto & Hardware');
insert into company_divisions values ('Baby','Domestic');
insert into company_divisions values ('Beauty','Domestic');
insert into company_divisions values ('Clothing','Domestic');
insert into company_divisions values ('Computers','Electronic Equipment');
insert into company_divisions values ('Electronics','Electronic Equipment');
insert into company_divisions values ('Games','Domestic');
insert into company_divisions values ('Garden','Outdoors & Garden');
insert into company_divisions values ('Grocery','Domestic');
insert into company_divisions values ('Health','Domestic');
insert into company_divisions values ('Home','Domestic');
insert into company_divisions values ('Industrial','Auto & Hardware');
insert into company_divisions values ('Jewelery','Fashion');
Query Editor
Query History
Data Output
Explain
Messages
Notifications

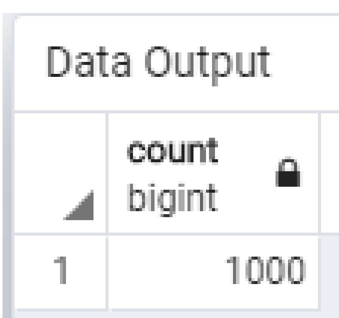
The COUNT, MIN, and MAX functions-02_03 csv file

- [Instructor] Starting with PGM in open select the data sci database and within that the public schema. Next click on the **tools menu** at the top of the screen and select **Query Tool**. This will open a sequel window. We'll do most of our work in query windows like this. The top window has an area where we enter sequel commands. The results of our queries will appear in the bottom panel. Let's type a simple select command in the query window. Select, star, from, staff. I'm just going to move the results pain down slightly. Click on the execute button which has the lightening bolt icon. This will execute the query. Notice the query result rows appear in the lower pane. You can scroll through to see the rows. One thing that I like to do when I'm checking a table is to just return the first several rows instead of retrieving all of them. This is especially useful when working with large tables. So I'll **add a limit clause** and restrict it to the first 10 rows. I'll then click the query execution command and you'll notice that we have rows down here. We have nine showing and 10 actually. So the results are limited to the first 10 rows when we use the limit clause. Okay, now let's work with some basic aggregate functions. Let's start by counting employees across the company and then across different groups. So the first thing I'll do is remove this limit clause, since I want to work with the whole set of employees. The first I want to do is simply count the number of people in the company. I'll use the count function and I'll count everyone in the staff table. I'll execute that command and you'll notice the count returns 1000 rows. And that is correct. There are 1000 employees in the staff table. Now, let's count by gender. So the first thing I want to do, is I want to count the staff, but I want to group by gender. And this will count how many employees are in each gender group. So execute the statement. And notice I have two numbers coming back now. 496 and 504. Problem is, I don't know which is female and which is male. **The problem is I did not include gender in the list of columns to return in the select statement**. So I'll add gender now, and I'll say return the gender and the count for each gender in the table. And I'll execute. And now you'll see I have the indicators of which is male and which is female and their associated counts. Let's do a little different grouping next. Instead of grouping by gender, let's group by department. And I'll group by department here. So I'm grouping by department. I'm including the department name along with the count in the rows that are returned. I'll select the execute statement. And you'll notice here I've had 22 rows returned. That's because there are 22 distinct departments in the database. As you can see I have the count of the number of employees in each department. So now that we've looked at counting let's look at a couple other aggregate functions that are commonly used. Those are the max and the min functions. Let's say I want to know what is the salary of the highest paid employee. Well the first thing I want to do is I want to select the salary, but I also want to select the maximum salary. I don't want all of them. So I'll use the max function. And I'll say select the max from staff. I don't need to group by anything because I want to go across the entire set of employees. And when I click that I'll notice that the highest salary is approximately $150000. Roughly. Now I may also want to know what's the minimum salary that any employee is paid. I can use the min function for that. So again I'll execute the statement with that command, and I'll notice the lowest salary is about $40000. Now sometimes I want to see both the max and the min salary at the same time. And that's no problem. I can include both the max and the min in the same select statement. And I'll execute. And you'll notice I have the maximum and the minimum both returned. Now, just as we can count rows in groups we can also apply the max and min functions to groups of subsets of rows. So let's show the minimum and maximum salary in each department. Well I want to group by department, but I also want to remember to include the department name in the results. So now what I'm doing is I'm selecting for each department, find the maximum and minimum salaries and then present them all back to me. So I'll select the execute statement and now you'll notice here once again we have the list of the 22 different departments, but we also have the minimum salary paid to anyone in that department as well as the maximum paid to anyone in that department. Now, just as we had used gender before in our counts, we can also use it here. So we can again group by gender. Find the min and max salary. And we can group by gender. And we click the execute button. And now here we have the min and max salaries for male and female, and we have that across all of the staff.

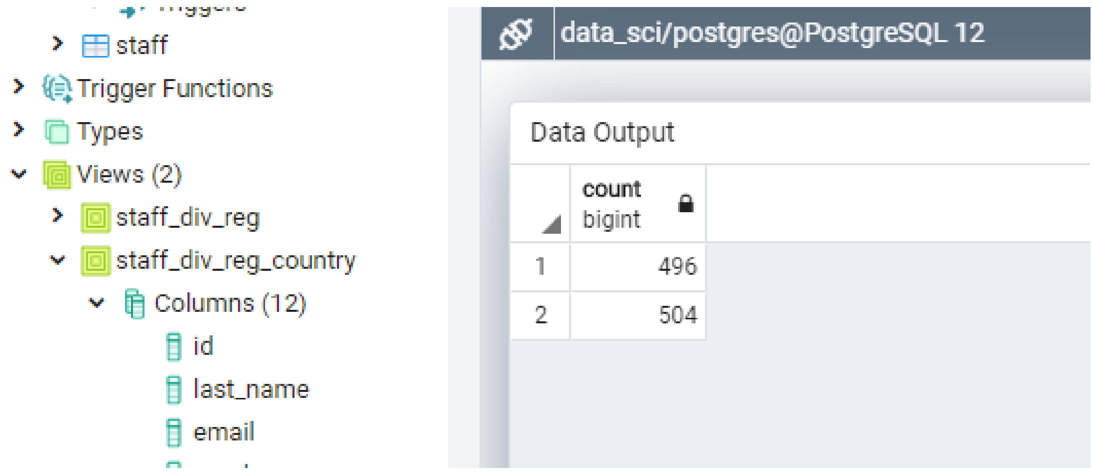
/* Start with a simple select statement */
select * from staff;



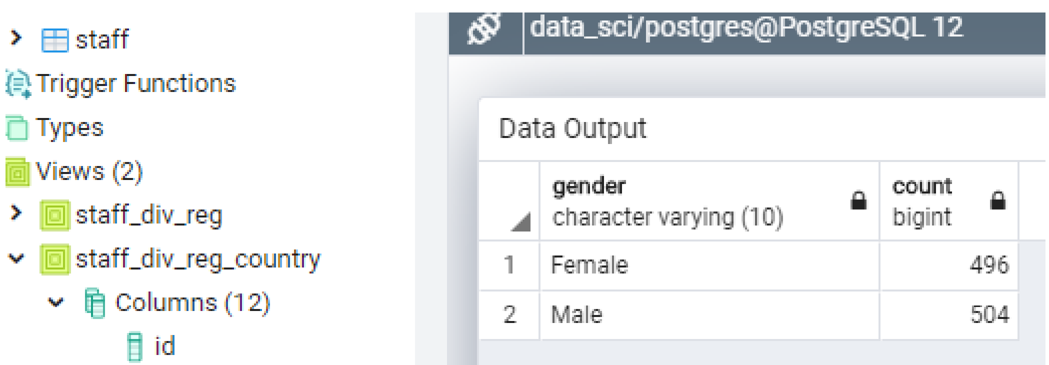
/* Show the number of rows in the table staff */
select count(*) from staff;



```
/* Show the number of staff members by gender */
select
   count(*)
from
   staff
group by
   gender;
```

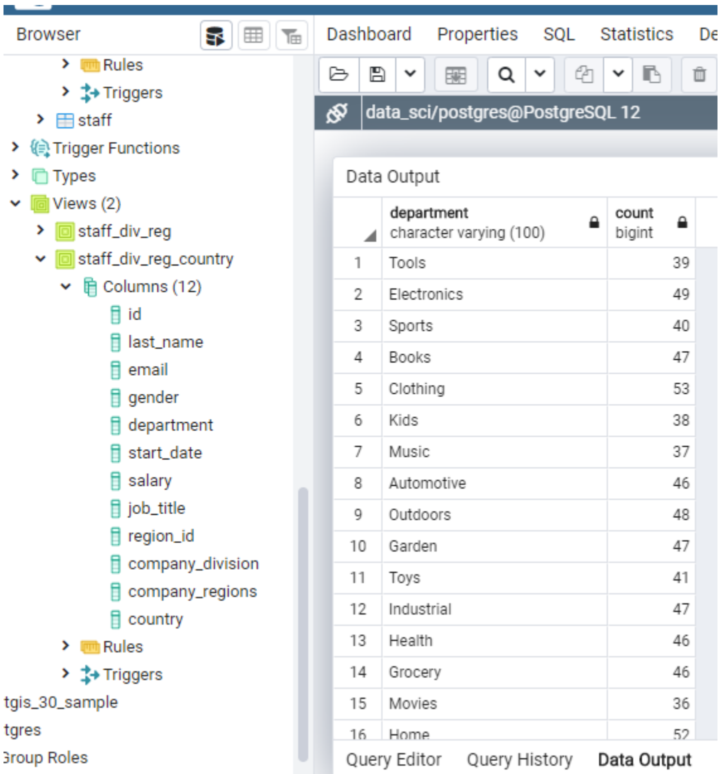



```
/* Now, show the number of staff member by gender and include the name of the gender */
select
  gender, count(*)
from
   staff
group by
   gender;
```



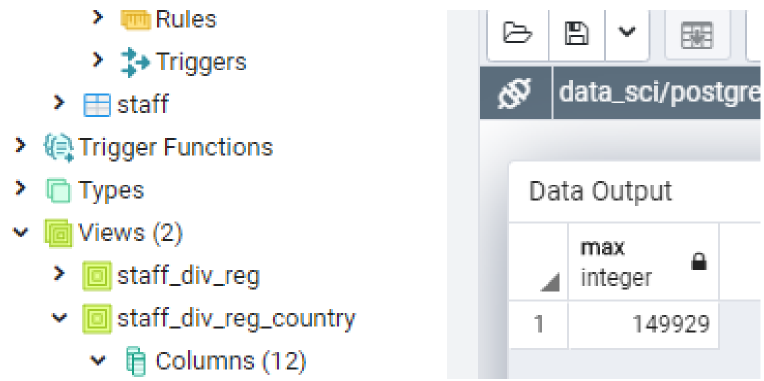
/* Show the number of staff members in each department */

```
select
  department, count(*)
from
  staff
group by
  department;
```

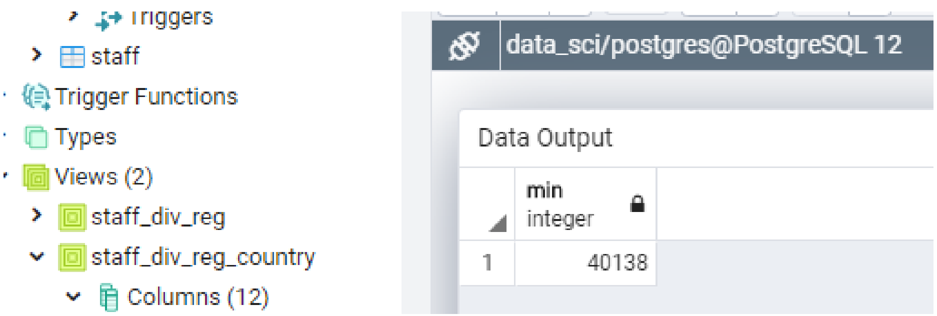


	department character varying (100)	count bigint
1	Tools	39
2	Electronics	49
3	Sports	40
4	Books	47
5	Clothing	53
6	Kids	38
7	Music	37
8	Automotive	46
9	Outdoors	48
10	Garden	47
11	Toys	41
12	Industrial	47
13	Health	46
14	Grocery	46
15	Movies	36
16	Home	52

```
/* Show the maximum salary of all staff member */
select
  max(salary)
from
  staff;
```




```
select
  min(salary)
from
  staff;
```



```
/* Show the minimum salary of all staff member */
select
   min(salary), max(salary),last_name,gender,department
from
   staff
group by last_name,gender,department;
```

data_sci/postgres@PostgreSQL 12

Data Output

	min integer	max integer	last_name character varying (100)	gender character varying (10)	department character varying (100)
1	142103	142103	Andrews	Female	Beauty
2	58420	58420	Nguyen	Female	Clothing
3	116355	116355	Bowman	Female	Baby
4	149114	149114	Riley	Male	Jewelery
5	135786	135786	Arnold	Male	Movies
6	62799	78755	Young	Female	Kids
7	140276	140276	Rodriguez	Female	Outdoors
8	124215	124215	Snyder	Female	Grocery
9	134114	148860	Reyes	Female	Garden
10	108854	108854	Boyd	Male	Clothing
11	93665	93665	Hunt	Male	Electronics
12	78828	78828	Palmer	Male	Sports
13	122108	122108	Walker	Male	Computers
14	104517	104517	Greene	Male	Clothing
15	48050	48050	Kennedy	Male	Industrial
16	92879	92879	Payne	Female	Home

Query Editor Query History **Data Output** Explain Messages Notifications

```
/* Show the minimum and maximum salary in each department */
select
   department, min(salary), max(salary)
from
   staff
group by
   department
   limit 10;
```

Data Output

	department character varying (100)	min integer	max integer
1	Tools	44788	149586
2	Electronics	40218	149597
3	Sports	40418	147166
4	Books	42714	146745
5	Clothing	42797	148408
6	Kids	43097	149351
7	Music	42759	144608
8	Automotive	42602	146167
9	Outdoors	43366	148906
10	Garden	50057	148860

```
/* Show the minimum and maximum salary by gender */
select
   gender, min(salary), max(salary)
from
   staff
group by
   gender;
```

Data Output

	gender character varying (10)	min integer	max integer
1	Female	40138	149929
2	Male	40194	149835

```
/* get anything */
select job_title, start_date, max(salary), gender
from staff
group by job_title, start_date, gender
order by max(salary) desc
limit 25;
```

Data Output

	job_title character varying (100)	start_date date	max integer	gender character varying (10)
1	Director of Sales	2001-02-10	149929	Female
2	Geologist III	2002-03-23	149835	Male
3	VP Product Management	2013-10-02	149598	Female
4	Structural Analysis Engineer	2005-02-05	149597	Female
5	Account Coordinator	2005-07-18	149586	Female
6	Design Engineer	2013-02-13	149351	Female
7	Senior Sales Associate	2010-03-14	149336	Female
8	Senior Financial Analyst	2013-01-17	149221	Female
9	Business Systems Developme...	2012-06-19	149114	Male
10	Executive Secretary	2007-02-12	149099	Male
11	Software Engineer III	2005-02-16	149085	Female
12	Speech Pathologist	2011-05-16	148993	Male
13	Structural Analysis Engineer	2008-04-29	148986	Female
14	Programmer IV	2011-05-04	148952	Female
15	Senior Financial Analyst	2012-01-07	148940	Female
16	Project Manager	2008-03-05	148906	Female

Query Editor Query History **Data Output** Explain Messages Notifications

Statistical functions:02_04 file

- [Narrator] Data scientists often use statistics to better understand data sets. SQL includes functions for describing data sets, so let's see how to use some of the most commonly-used statistic functions, to better understand our data. I'm going to start by selecting from the staff column, I will select star, from staff, and I'll limit this to 10, cause I'm just trying to get a feel for the data. And we'll notice that this table includes information about employees, which includes some string characters and some dates, and some integer values to work with. Now, one of the things I might like to know is, how much does the company pay in salary to all of its staff, across a given year? Well, the way to do that is to first of all, we'll remove the limit clause, cause we want to look at everyone, and we want to use the sum function, which is one of those statistic functions, and it's applied to a column, like salary, and I'll execute the query, and notice that the result is approximately $97.3 million is paid per year, across all of the staff. Now next, I might like to understand how much does each department pay in salary? Well, to do that, I'd want to use the group by clause again. So I will group by department, and I'll be sure to include the department name in the results set. I'll execute that query, and I'll notice that I have 22 rows returned, one for each department, and also, a value of the total amount spent in salary each year for that department, okay, that's really useful information. Different departments may have different number of employees so it's also helpful to understand what's the average salary paid per employee in department. To find that, I can use the AVG or average function, and ask for the average salary, and since I'm grouping by department, the average function will calculate the average salary per department, and I'll simply execute the statement, and now we'll notice that I have a third column added. And I'm just going to spread it out here, to make it easier to read. And this is the average salary per department. There are a couple of other statistics that are sometimes used by statisticians and others who are familiar with statistics. These are called the variance, and standard deviation. We won't go into details about what these mean, but I'll roughly describe them as both good ways of measuring how spread out data is around an average. So, let's look first at the variance, and how to calculate that. SQL has a function called **var pop,** short for **variance of population, and that's applied to a numeric column,** like salary. And when we execute this statement, we'll get a fourth column added to our results set, and this is known as the variance. I won't go into details about it, but it's a good way of measuring how spread out the data is. A more commonly used statistic for understanding the distribution of data is something called the standard deviation. In SQL, that function is **stddev**, underscore pop, which is short for standard deviation of population. And we want to know the standard deviation over the salary, so we'll use that function, and execute, and again, another column is added to our result set, and as we can see, this number indicates the standard deviation over the salary, by department. Now you'll notice that when we use functions like average, and standard deviation, we're getting a lot of decimal places after the decimal point. Since we're talking about salaries, we're talking about monetary values, it's probably helpful to round these to two decimal points. To do that, we simply use the **round function**, and apply it to the average, the variance, and the standard deviation. So I'll do that right now, I'll add round, and one of the things I want to do is tell the round function to round the average salary, and keep just two decimal points, and I'll do the same thing with variance, I'll ask to round it and keep just two decimal points, and finally I'll do the same for standard deviation. And when I execute, you'll notice that each column that has the round function applied to it, has only two decimal places, this makes the results a little easier to read. Now I will say, if you're not familiar with statistics, and terms like variance and standard deviations are not things you've come across before, that's nothing to worry about, it's just important to know that if you do need them in the future, the functions are available to you in SQL.

```
/* Show the first 10 rows of the staff table */
select * from staff limit 10;
```

data_sci/postgres@PostgreSQL 12

Data Output

	id [PK] integer	last_name character varying (100)	email character varying (200)	gender character varying (10)	department character varying (100)	start_date date	salary integer
1	1	Kelley	rkelley0@soundcloud.com	Female	Computers	2009-10-02	67470
2	2	Armstrong	sarmstrong1@infoseek.co.jp	Male	Sports	2008-03-31	71869
3	3	Carr	fcarr2@woothemes.com	Male	Automotive	2009-07-12	101768
4	4	Murray	jmurray3@gov.uk	Female	Jewelery	2014-12-25	96897
5	5	Ellis	jellis4@sciencedirect.com	Female	Grocery	2002-09-19	63702
6	6	Phillips	bphillips5@time.com	Male	Tools	2013-08-21	118497
7	7	Williamson	rwilliamson6@ted.com	Male	Computers	2006-05-14	65889
8	8	Harris	aharris7@ucoz.com	Female	Toys	2003-08-12	84427
9	9	James	rjames8@prnewswire.com	Male	Jewelery	2005-09-07	108657
10	10	Sanchez	rsanchez9@cloudflare.com	Male	Movies	2013-03-13	108093

Query Editor Query History Data Output Explain Messages Notifications

Successfully run. Total query runtime: 252 msec. 10 rows af

```
/* Sum the total amount of salary paid to staff */

select sum(salary)
from staff;
```

Data Output

	sum bigint
1	97331223

```
/* Sum the total amount of salary paid to staff by department */
select
  department, sum(salary)
from
   staff
group by
    department;
```

data_sci/postgres@PostgreSQL 12

Data Output

	department character varying (100)	sum bigint
1	Tools	4095808
2	Electronics	4489141
3	Sports	3756041
4	Books	4459837
5	Clothing	5037890
6	Kids	3543027
7	Music	3274767

```
/* Sum the total, and average (or mean) amount of salary paid to staff by department */
select
  department, sum(salary), avg(salary)
from
   staff
group by
    department;
```

Data Output

	department character varying (100)	sum bigint	avg numeric
1	Tools	4095808	717948717949
2	Electronics	4489141	122448979592
3	Sports	3756041	025000000000
4	Books	4459837	148936170213
5	Clothing	5037890	528301886792
6	Kids	3543027	552631578947
7	Music	3274767	216216216216
8	Automotive	4584268	000000000000

```
/* Sum the total, average (or mean) and variance of salary paid to staff by department */
select
  department, sum(salary), avg(salary),
var_pop(salary)
from
   staff
group by
    department;
```

Data Output

	department character varying (100)	sum bigint	avg numeric	var_pop numeric
1	Tools	4095808	717948717949	561.02301118
2	Electronics	4489141	122448979592	518.02582257
3	Sports	3756041	025000000000	239.87437500
4	Books	4459837	148936170213	035.82888185
5	Clothing	5037890	528301886792	590.36240655
6	Kids	3543027	552631578947	422.93144044
7	Music	3274767	216216216216	601.52081812
8	Automotive	4584268	000000000000	811.86956522
9	Outdoors	5378660	416666666667	798.45138889

```
/* Sum the total, average (or mean), variance and standard deviation of salary paid to staff by
department */
select
  department, sum(salary), avg(salary), var_pop(salary), stddev_pop(salary)
from
   staff
group by
    department;
```

Data Output

	department character varying (100)	sum bigint	avg numeric	var_pop numeric	stddev_pop numeric
1	Tools	4095808	717948717949	561.02301118	28211.14249766
2	Electronics	4489141	122448979592	518.02582257	32768.55990162
3	Sports	3756041	025000000000	239.87437500	32390.58875467
4	Books	4459837	148936170213	035.82888185	30753.29308918
5	Clothing	5037890	528301886792	590.36240655	31627.90840954
6	Kids	3543027	552631578947	422.93144044	32756.48673059

```
/* Use round function to round to 2 decimal places */
select
  department, sum(salary), round(avg(salary),2), round(var_pop(salary),2),
round(stddev_pop(salary),2)
from
   staff
group by
    department;
```

Data Output

	department character varying (100)	sum bigint	round numeric	round numeric	round numeric
1	Tools	4095808	105020.72	795868561.02	28211.14
2	Electronics	4489141	91615.12	1073778518.03	32768.56
3	Sports	3756041	93901.03	1049150239.87	32390.59
4	Books	4459837	94890.15	945765035.83	30753.29
5	Clothing	5037890	95054.53	1000324590.36	31627.91
6	Kids	3543027	93237.55	1072987422.93	32756.49
7	Music	3274767	88507.22	1001696601.52	31649.59
8	Automotive	4584268	99658.00	929902811.87	30494.31
9	Outdoors	5378660	112055.42	745849798.45	27310.25
10	Garden	4792085	101959.26	909562157.08	30158.95

Filtering and grouping data: 02_05 file

- [Instructor] Two common tasks in data science work are **filtering data** so we can work with just the data we're interested in, and the other is grouping data so we can calculate values for an entire group such as a product category or a department. Let's start with filtering data using numeric values. We'll use the staff table for this example. Let's assume **we want to list everything with a salary greater than $100,000.** So I'll use the select command, and for this example, we'll include the last name, department, and of course, salary. And we will select this from the staff table. Now, we only want to include a certain subset of the employees, so we use a WHERE clause. We're going to say WHERE salary is greater than 100,000, and if we execute that statement, we'll get a list of employees who work in various departments, but all of whom earn salaries over $100,000. **Filtering based on character strings** works in similar ways. Now, let's generate a list of employees who work in the tools department, and we'll keep the columns the same. So I'll just change the WHERE clause to say WHERE department = tools, and I'll execute that statement. And now all of the employees listed are in the tools department, and if we were to browse a little bit, we'd notice that yes, in fact, they are all in the tools department. Now let's combine the two filters. It's pretty **common to have more than one condition we want to filter on**. In fact, sometimes WHERE clauses can be the longest part of a select statement, because there are so many conditions that need to be satisfied. To filter for rows that **satisfy both conditions**, we use the AND keyword. So I'll include that in my WHERE clause, and I'll say WHERE department = tools AND salary > 100,000. And we'll execute this statement, and we'll notice that sure enough, everyone listed here is in the tools department and has a salary greater than $100,000. Now, if we want **at least one** of the conditions to be met but maybe not both, then we can use the OR keyword. And I'll change the AND to OR, execute that statement, an you'll notice, now we have some people who work in departments other than tools, but all of the people who aren't in the tools department have salaries over $100,000. And if we browse a little bit, we'll notice that some of the people in the tools department are listed, but they have salaries less than $100,000. For example, in this case, Watson is in the tools department and earns a salary of $81,870. Now, in SQL, the LIKE **operator is used to match patterns**. There are many options for working with the LIKE operator. For example, if we wanted to look at all of the departments that begin with the letter B, we could change this WHERE clause to say WHERE department LIKE 'B%.' And this means select all staff from the departments where the department name begins with a B and has **zero or more characters after it.** And let's, to make this a little more interesting, let's add a GROUP BY statement here, and let's GROUP BY department, and we'll change this. Instead of doing less than the department, we'll remove the last_name, we'll GROUP BY department, and we will sum the salary. And this will allow us to see how much we're spending on salaries in each department that begins with a B. We'll execute that statement, and you'll notice, there are three departments listed, and then the sum of the salaries is listed as well. Now, we can **make the LIKE operator actually be more specific.** For example, if we just wanted to see departments that began with the letter **B followed by the letter O**, we could specify that, and if we execute the statement, as we expected, we see just one department, and that's the one that begins with the letters B-O. The special symbol percent can be used in different parts of the string. It does not have to be at the end of the string. For example, if you want information on the departments that **begin with B and end with Y**, you could use the following query: like that, we'll have a percent in the middle, and we'll end with Y. With this query, any department that begins with B and ends with Y will match. So if we execute this statement, as we expect, we have two departments. Both begin with B and both end with Y. One note, **be careful when using percent at the beginning of a string**. When this happens, the database cannot use indexes that may be on the column. This will cause the database to scan every row in the table looking for matches. That's okay, it just may take a lot longer than you expect.

```
/* Filter based on numeric values */
select
  last_name, department, salary,gender,start_date
from
  staff
where
  salary < 50000 And salary > 35000;
```

Data Output

	last_name character varying (100)	department character varying (100)	salary integer	gender character varying (10)	start_date date
1	Black	Clothing	44179	Male	2003-02-04
2	Oliver	Clothing	42797	Female	2013-08-30
3	Garcia	Health	48360	Female	2002-03-19
4	Washington	Home	47206	Female	2000-07-11
5	Grant	Electronics	49296	Female	2000-07-23
6	Gardner	Kids	47879	Male	2009-08-06
7	Austin	Computers	47494	Female	2012-06-13
8	Martin	Movies	49644	Male	2012-10-02
9	Robinson	Books	45456	Male	2007-09-06
10	Sanders	Movies	41898	Male	2010-12-24
11	Meyer	Shoes	48829	Male	2012-11-09
12	Andrews	Home	48684	Male	2009-03-18
13	Foster	Music	42759	Female	2008-01-17
14	Fowler	Kids	43097	Female	2011-02-25
15	Gonzalez	Electronics	44917	Male	2009-04-24
16	Duncan	Jewelery	47439	Female	2011-01-16

Query Editor Query History **Data Output** Explain Messages Notifications

```
/* Filter based on character values */
select
  last_name, department, salary
from
  staff
where
  department = 'Tools';
```

Data Output

	last_name character varying (100)	department character varying (100)	salary integer
1	Phillips	Tools	118497
2	Willis	Tools	113507
3	Watson	Tools	81870
4	Daniels	Tools	139061
5	Gomez	Tools	103806
6	Gutierrez	Tools	58805
7	Harvey	Tools	138179
8	Thomas	Tools	128239

```
/* Filter based on multiple attributes, all filters must be met*/
select
  last_name, department, salary
from
  staff
where
  department = 'Tools'
and
  salary > 100000;
```

Data Output

	last_name character varying (100)	department character varying (100)	salary integer
1	Phillips	Tools	118497
2	Willis	Tools	113507
3	Daniels	Tools	139061
4	Gomez	Tools	103806
5	Harvey	Tools	138179
6	Thomas	Tools	128239
7	Jenkins	Tools	113599
8	Harris	Tools	148940
9	Bishop	Tools	110744
10	Ferguson	Tools	119385

```
/* Filter based on multiple attributes,
at least one filter must be met*/

select
  last_name, department, salary
from
  staff
where
  department = 'Tools'
And salary <50000 and salary > 30000
or
  department = 'Books'
And salary < 50000;
```

Data Output

	last_name character varying (100)	department character varying (100)	salary integer
1	Robinson	Books	45456
2	Harris	Books	47131
3	Owens	Books	42714
4	Perez	Tools	44788
5	James	Tools	47271

```
/* Filter based on patterns in character strings */
select
  department, sum(salary)
from
  staff
where
  department like 'C%'
group by
  department;
```

Data Output

	department character varying (100)	sum bigint
1	Clothing	5037890
2	Computers	5152963

```
/* Filter based on patterns in character strings */
select
  department, sum(salary)
from
  staff
where
  department like 'Bo%' /*exact case match*/
group by
  department;
```

Data Output

	department character varying (100)	sum bigint
1	Books	4459837

```
/* Filter based on patterns in character strings */
select
  department, sum(salary)
from
  staff
where
  department like 'B%y'
group by
  department;
```

Data Output

	department character varying (100)	sum bigint
1	Baby	4218724
2	Beauty	5481063

Reformatting character data:03_01 file

- [Instructor] We often need to collect data from multiple sources. Sometimes, the same data is stored differently in different systems. For example, one database might use abbreviations for department names, while another database spells out the full name. We can reformat data to get it into a consistent format. Anytime you start working with a new data set, it is helpful to browse through the data to get a sense of how the data's formatted. So let's quickly look at how department names are stored in our database. Since we just need to see each name once, we'll use a distinct keyword. So we'll start with a SELECT statement, and we'll select department FROM staff, and as I'd mentioned, we want to use a DISTINCT keyword, so we'll put that in there. There, now we'll see each department once, and as you can see, we have a set of department names appearing as they are stored in the database. If we wanted to **reformat the department names to be in capitals or uppercase**,

```
select distinct
  upper(department)
from
  staff;
```

Data Output

	upper text
1	MUSIC
2	CLOTHING
3	BEAUTY
4	GAMES
5	BOOKS
6	HOME
7	GROCERY

```
/* Convert the names of departments to lower case */
select distinct
 lower(department)
from
 staff;
```

Data Output

	lower text
1	books
2	home
3	health
4	electronics
5	toys
6	jewelery

we could use the UPPER keyword. And you'll notice the names are returned in uppercase. Similarly, if we wanted the department names in all lowercase, we'd use the LOWER keyword. Now, changing case is pretty simple. Sometimes, we might have data in two or more columns that we'd like to have in a single column. For example, if we want to have a job title combined with a department name, we could concatenate two columns. Let's do that. Let's select job_title, and we'll use the concatenation operator, which is two pipes,

```
select
  job_title || '-' || department
from
  staff;
```

Data Output

	?column? text
1	Structural Engineer-Computers
2	Financial Advisor-Sports
3	Recruiting Manager-Automotive
4	Desktop Support Technician-Jewelery
5	Software Engineer III-Grocery
6	Executive Secretary-Tools
7	Dental Hygienist-Computers
8	Safety Technician I-Toys
9	Sales Associate-Jewelery
10	Sales Representative-Movies
11	Community Outreach Specialist-Jewelery
12	Data Coordiator-Clothing
13	Compensation Analyst-Baby
14	Software Test Engineer III-Computers
15	Community Outreach Specialist-Games
16	Web Developer III-Baby

Query Editor Query History **Data Output** Ex

and I want to put a dash in between the job title and the department name, so I'll concatenate that, and then I'll specify department, and we'll select this FROM staff. And what you'll notice is we have a job title followed by a dash, followed by the department name. Now, a nice feature of SQL is that was can give names to columns we create, such as when we reformat. Let's call this new column Title_Department, and I do that by placing the alias, title_department, and I'll abbreviate that as D-E-P-T, after the concatenation string. Now, when I execute, you'll notice the title is changed to title_department.

```
select
   job_title || '-' || department  title_dept
from
   staff;
```

Data Output

	title_dept text
1	Structural Engineer-Computers
2	Financial Advisor-Sports
3	Recruiting Manager-Automotive
4	Desktop Support Technician-Jewelery
5	Software Engineer III-Grocery
6	Executive Secretary-Tools
7	Dental Hygienist-Computers
8	Safety Technician I-Toys

Depending on how data is stored in or extracted from other databases, you may find data has **extra white spaces**, like spaces and tabs. We can use the **trim function** to remove those extra characters. So, for example, let's just create a column here, and we'll select the trim of a string value that has some spaces in there. Let's call it Software Engineer, and we have a couple spaces in the front, so let's add a couple spaces at the end, and let's execute that statement. Now what happens is, we return just the value of the string without leaving or trailing white spaces. Now, let's just verify that we've actually cut off those spaces. Let's first check the length of the string. So, with the extra spaces, we have 21 characters.

Now, let's check the length after applying the trim function.

```
select
  trim('  Software Engineer  ');
```

Data Output

	btrim text
1	Software Engineer

```
select
 length('   Software Engineer   ');
```

Data Output

	length integer
1	23

```
select
  length(trim('   Software Engineer   '));
```

Data Output

	length integer
1	17

Now, this should reduce it by the number of white spaces, and it is. So we drop about four white spaces there. Reformatting can also entail adding new types of information, such as category columns that make it easier to select rows that meet some criteria. For example, let's list all employees with a title that begins with the word Assistant. And to do that, we'll create a new SELECT statement, and we'll select job_title, FROM staff, WHERE job title is **like** Assistant, and we'll use the **wildcard**, and we'll execute, and we'll see a number of titles have the word Assistant in the beginning.

```
/* Show all job titles that start wtih Assistant
select
  job_title
from
  staff
where
 job_title like 'Assistant%'
*/
```

Data Output

	job_title character varying (100)
1	Assistant Manager
2	Assistant Media Planner
3	Assistant Manager
4	Assistant Professor
5	Assistant Media Planner
6	Assistant Manager
7	Assistant Media Planner
8	Assistant Professor

```
/* Create a new boolean column indicating if a staff person has the term Assistant
 anywhere in their title.  */
select
  job_title, (job_title like '%Assistant%') is_asst
from
  staff;
```

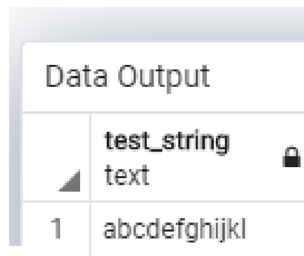
Data Output

	job_title character varying (100)	is_asst boolean
1	Structural Engineer	false
2	Financial Advisor	false
3	Recruiting Manager	false
4	Desktop Support Technician	false
5	Software Engineer III	false
6	Executive Secretary	false
7	Dental Hygienist	false
8	Safety Technician I	false

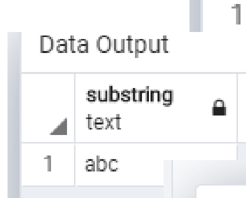
Now, we might want to make it obvious in a results set if someone is an assistant, so we can create a new **Boolean** column called Is Assist which will be true or false, depending on whether someone is actually an assistant. So, let's select job title and let's put in a Boolean expression here. Job_title like, and actually, we'll put a wildcard in front as well as after the term Assistant, that way, if someone has the word Assistant in their title, but it's not in the beginning, we can still catch it. There. So, anytime a job title has the word Assistant in it, this Boolean will return true, and let's give it the name Is Assistant, which we'll abbreviate as Is_Asst, and we'll just execute it from the staff, and we'll do it for the entire staff, so we'll delete that where clause. Now what we find is that we have returned a value, Is_Asst, which is false when the word Assistant does not appear in the job title. But it is true, however, if, for example, we have Assistant Media Planner or Assistant Manager as the title. So that's one way of adding new columns that can make it easier for other people to filter without having to understand all of the logic that might go behind, determining whether or not someone is an assistant.

Extracting strings from character data:03_02

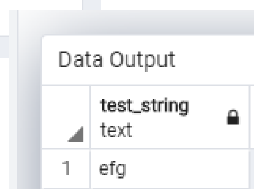
- [Instructor] In addition to **matching** and **reformatting** strings, we sometimes need to take them apart and **extract** pieces of stings. SQL provides some general purpose functions for extracting and overriding strings. Let's start with a simple string that's easy to experiment. We'll use the first twelve letters of the alphabet, and I can do that by simply saying, SELECT, and then typing out the first 12 letters, and we'll call this test_string, and I can execute that, and we'll see I receive back a simple 12 character string. Now, one of the things we can do with SQL is replace parts of a string, and we can use, for example, the **substring** function. The substring function takes a string and a range, and returns the characters from the string that are in that range. So, I'll specify that I want to use the substring function, and I want to apply this to this 12 character string, and I want to specify that I want to start extracting from position one for a total of three characters, and if I execute this, I will see that I get three characters back, the first three. That makes sense. Ranges can start anywhere in the string. They don't have to start at the beginning. So, for example, I could start from position five, and go for three characters and extract, and get back the letters efg.




```
/* Use a string of 12 characters for experimenting with string extraction */
select
  'abcdefghijkl' test_string;
```




```
/* Select the first three characters of the string */
select
  substring('abcdefghijkl' from 1 for 3) test_string;
```




```
/* Select a subset from the middle of the string */
select
  substring('abcdefghijkl' from 5 for 3) test_string;
```

Now, if you don't specify a length of a string to extract, the substring function will return the rest of the string starting at the position specified by the FROM keyword. So, for example, if we removed FOR 3, and I simply said, take the substring starting from position five, and I executed that statement, I would get all of the string starting at position five and going to the end. Okay, let's go back to working with our tables, and in particular, let's look at the job title column in the staff table. The assistant job title includes a main job category after the word Assistant. Let's see some examples. So, let's select job title FROM staff, WHERE job_title LIKE 'Assistant%', and what we'll notice is that assistant is the first word, and then there's another set of words like manager, media planner, or professor, these can act as a category, so if we wanted to extract the category, we could do that by noticing the word assistant is nine characters long, and there's a space after the word assistant. So, if we extract the job title starting at position 10, we should get back all of these categories. So, let's try that. So, I will select **SUBSTRING** of jobtitle FROM 10, and we'll execute that, and as expected, we get the words that follow assistant, such as manager, media planner, and professor. So that gives us what we expected. Now, **sometimes it's**

useful to replace parts of a string. We can do this with the **overlay function**. So, let's again keep it working with the job title. Let's say for example that we want to change the word assistant to an abbreviation like Asst, we can do that by replacing the full word with the abbreviation using the overlay function. So, let's change this to OVERLAY, and the overlay function takes the keyword PLACING, and we're going to place an abbreviation, in this case Asst., with whatever is in position from one, starting at one for 10 characters, and we'll do this only for job titles that are like Assistant%.

Data Output

	job_title character varying (100)	asst_category text
1	Assistant Manager	Manager
2	Assistant Media Planner	Media Planner
3	Assistant Manager	Manager
4	Assistant Professor	Professor
5	Assistant Media Planner	Media Planner
6	Assistant Manager	Manager
7	Assistant Media Planner	Media Planner
8	Assistant Professor	Professor
9	Assistant Media Planner	Media Planner
10	Assistant Professor	Professor
11	Assistant Professor	Professor
12	Assistant Media Planner	Media Planner
13	Assistant Professor	Professor
14	Assistant Manager	Manager
15	Assistant Media Planner	Media Planner
16	Assistant Media Planner	Media Planner

Query Editor Query History **Data Output** Explain

```
/* Change Assistant to Asst in job title */
select
  job_title, overlay(job_title placing 'Asst. ' from 1
for 10)
from
  staff
where
  job_title like 'Assistant%';
```

Data Output

	job_title character varying (100)	overlay text
1	Assistant Manager	Asst. Manager
2	Assistant Media Planner	Asst. Media Planner
3	Assistant Manager	Asst. Manager
4	Assistant Professor	Asst. Professor
5	Assistant Media Planner	Asst. Media Planner
6	Assistant Manager	Asst. Manager
7	Assistant Media Planner	Asst. Media Planner
8	Assistant Professor	Asst. Professor
9	Assistant Media Planner	Asst. Media Planner
10	Assistant Professor	Asst. Professor
11	Assistant Professor	Asst. Professor
12	Assistant Media Planner	Asst. Media Planner
13	Assistant Professor	Asst. Professor
14	Assistant Manager	Asst. Manager
15	Assistant Media Planner	Asst. Media Planner
16	Assistant Media Planner	Asst. Media Planner

Query Editor Query History **Data Output** Explain Messages

So, let's execute that. Now, what we'll notice here is that we have replaced Assistant with Asst., now, you'll also notice we lost the space. Now, why did that happen? That's because the word Assistant is nine characters long, we used 10 in previous examples because we wanted to include the space. In this case we don't want to overlay the space, so we need to change the length of the string we're replacing from 10 to length nine, and we'll execute again, and we'll notice that now we have what we would expect, the Assistant replaced by Asst, but we're not overriding the space. So, that's something important to watch out for as you're working with overlay in substring. It's very easy to commit what were called off by one errors, and this is just one example of those.

Filtering with regular expressions:03_03 file

- [Narrator**] Regular expressions** are patterns for describing how to match strings in a **WHERE** clause. Many programming languages support regular expressions that use slightly different syntax from what is used with the SQL **LIKE** operator. In this course, when we refer to regular expressions, we're referring to the patterns used with the SQL LIKE operator. Let's start by creating a list of job titles that have the word assistant in them. So to do that, we'll SELECT job_title from the staff table, WHERE job_title LIKE, and I want to search for the pattern where the word Assistant appears **anywhere** in the job title. So I'll put **% wildcards before and after** the word assistant. Then when we execute, you'll notice we have some titles where the word assistant is at the end, and some where the word assistant is in the beginning, and some where it is in the middle. Now, we'll notice there are different levels of assistant. Let's select just assistants at levels three or four. To do that, we're going to change the LIKE operator to the **SIMILAR TO** operator, because that has slightly more expressive syntax.

Data Output

	job_title character varying (100)
1	Assistant Professor
2	Assistant Media Planner
3	Assistant Manager

```
select distinct
   job_title
from
   staff
where
  job_title similar to '%Assistant%(I)*';/*returns ‘Assistant ‘ and ‘Assistant I’ */
```

Now I want to match anything that has the word assistant, and has **either the roman numeral III or the roman numeral IV** and to do that, I create a list, and within that list, I list the patterns I want to match, in this case roman numeral III or roman numeral IV, and I separate them with the or character, or the pipe. Now let's execute and see what happens. These results are all either an assistant at level four, or an assistant at level three.

Data Output

	job_title character varying (100)
1	Office Assistant I
2	Office Assistant III
3	Research Assistant II
4	Assistant Manager
5	Research Assistant I
6	Administrative Assistant IV
7	Research Assistant III
8	Office Assistant IV
9	Physical Therapy Assistant
10	Accounting Assistant III
11	Accounting Assistant IV
12	Administrative Assistant II
13	Accounting Assistant II
14	Administrative Assistant III
15	Human Resources Assistant I
16	Administrative Assistant I

Query Editor   Query History

```
select distinct
   job_title
from
   staff
where
  job_title similar to '%Assistant%(II|IV)'; /* ‘Assistant II’,
‘Assistant III’, and ‘Assistant IV returned */
```

Now we can also select a list of jobs that includes the assistant three, four, or any other two characters starting with the letter I. We do this by using the **underscore symbol**, which matches any **one** character. So I'm going to change the list with roman numeral III, and roman numeral IV, and instead, I'm going to look for assistant, followed by roman numeral I, and then any other roman numeral. This will match assistant two, or assistant four. And, as expected, all of the job titles listed, include the roman numeral IV or the roman numeral II.

Data Output

	job_title character varying (100)
1	Accounting Assistant IV
2	Administrative Assistant II
3	Research Assistant IV
4	Administrative Assistant IV
5	Accounting Assistant II
6	Human Resources Assistant IV
7	Research Assistant II
8	Human Resources Assistant II
9	Office Assistant IV
10	Office Assistant II

```
select distinct
    job_title
from
    staff
where
   job_title similar to '%Assistant I_';

/* returns only I or IV after Assistant */
```

Now, regular expressions can also be used to **match on a list of characters.** For example, we can use **square brackets** to specify a list of characters, any of which can match. For example, let's look at a pattern for matching any jobs that begin with the letter E, P, or S. We'll use the **SIMILAR TO** clause, and we'll list the characters we're interested in, E, P, and S in square brackets, and then that can be followed by any number of characters, so we use a **percent sign**, and we execute, and here again, we see that all of the job titles start with an S, a P, or E.

```
select distinct
    job_title
from
    staff
where
   job_title similar to '[EPS]%';
```

Data Output

	job_title character varying (100)
1	Social Worker
2	Software Engineer IV
3	Senior Sales Associate
4	Sales Associate
5	Executive Secretary
6	Programmer IV
7	Engineer I
8	Staff Accountant I
9	Software Test Engineer II
10	Environmental Tech
11	Structural Engineer
12	Software Consultant
13	Statistician IV
14	Structural Analysis Engineer
15	Staff Scientist
16	Paralegal

Query Editor Query History D

Reformatting numeric data:03_04 file

- [Instructor] Sometimes we will need to **reformat numbers**. This is especially true when we use calculations that have results with large numbers of decimal digits. For example, when we calculate the average salary by department, we will get a default format which produces more digits than really are necessary. Let's see that for an example. We'll select departments and the average salary. We'll select this from the staff table, and because we're using an aggregate function, in this case the average, we're going to want to make sure we have a group by statement, and in this case, we're going to group by department. When we execute this, we'll see we get a department list, and for each department, we have an average. Typically we would expect to see two decimal places when we're dealing with currencies, but in this case, we get more than is needed. There are a few different ways we can address this. If we're only interested in working with whole dollar amounts, then one option is to use the **truncate function**, or **trunc.** It truncates, or drops, the decimal portion of the number. Now I just want to point out that the truncate function, or trunc, does not round. It just ignores the decimal value. So let's add a column that truncates, or truncs, the average salary. Now we'll notice that, as expected, the decimal portion is truncated, or dropped. Now if we want to **round** to the nearest dollar when averaging salary, we should use the **round function**. Notice sometimes the truncated value is the same as the rounded, and sometimes it's not, so let's see that. Let's replace trunc with round and execute that. We'll notice the rounding sometimes rounds up and sometimes rounds down.

Data Output

	department character varying (100)	avg numeric
1	Tools	105020.717948717949
2	Electronics	91615.122448979592
3	Sports	93901.025000000000
4	Books	94890.148936170213
5	Clothing	95054.528301886792
6	Kids	93237.552631578947
7	Music	88507.216216216216
8	Automotive	99658.000000000000
9	Outdoors	112055.416666666667
10	Garden	101959.255319148936
11	Toys	96187.170731707317
12	Industrial	92900.851063829787
13	Health	98975.652173913043
14	Grocery	101113.934782608696
15	Movies	100911.805555555556
16	Home	92734.711538461538

Query Editor Query History **Data Output** Explain Me

```
select
  department, avg(salary), trunc(avg(salary)),
round(avg(salary))
from
  staff
group by
   department;
```

If we would like to always return the next larger integer, so sort of the opposite of trunc, you could use the **ceiling function** which is **ceil**, C-E-I-L. If the decimal part of the number is greater than zero, then the ceiling function returns the next integer up in size, so let's try that. Let's replace round with the ceiling function and execute. We'll notice that the numeric value returned by the ceiling is always the next larger integer.

Data Output

	department character varying (100)	avg numeric	trunc numeric
1	Tools	105020.717948717949	105020
2	Electronics	91615.122448979592	91615
3	Sports	93901.025000000000	93901
4	Books	94890.148936170213	94890
5	Clothing	95054.528301886792	95054
6	Kids	93237.552631578947	93237
7	Music	88507.216216216216	88507
8	Automotive	99658.000000000000	99658
9	Outdoors	112055.416666666667	112055
10	Garden	101959.255319148936	101959
11	Toys	96187.170731707317	96187
12	Industrial	92900.851063829787	92900
13	Health	98975.652173913043	98975
14	Grocery	101113.934782608696	101113
15	Movies	100911.805555555556	100911
16	Home	92734.711538461538	92734

Query Editor Query History **Data Output** Explain Messages

```
select
  department, avg(salary), trunc(avg(salary)),
ceil(avg(salary))
from
  staff
group by
   department;
```

When working with currencies, we often want to use two decimal places. We can use the round function for this. Round takes an optional parameter indicating how many digits to have after the decimal point. For example if we said round the average salary and return two decimal places, we would, as expected, have the number rounded to two decimal places. Sometimes this will round up to a larger number, and sometimes it'll round down to the smaller number, but it always rounds to the decimal point that you indicate using that optional parameter. Similarly, trunc which is used for truncating can also take an optional number of decimal places, so let's just add that. **Trunc of the average salary, and let's specify that to two decimal places**, and we'll execute, and once again we'll notice that sometimes trunc will be the same, and sometimes it won't. Now both the round and the trunc functions can be used to truncate or round to a variable number of decimal places. It doesn't have to be two, so let's round to three decimal places and truncate to four decimal places and see what that looks like. Again, we have three decimal places in the case of the round function and four decimal places in the case of the trunc function. The exception being if the final digit is zero, it can be dropped.

Data Output

	department character varying (100)	avg numeric	trunc numeric	round numeric
1	Tools	105020.717948717949	105020	105021
2	Electronics	91615.122448979592	91615	91615
3	Sports	93901.025000000000	93901	93901
4	Books	94890.148936170213	94890	94890
5	Clothing	95054.528301886792	95054	95055
6	Kids	93237.552631578947	93237	93238
7	Music	88507.216216216216	88507	88507
8	Automotive	99658.000000000000	99658	99658
9	Outdoors	112055.416666666667	112055	112055
10	Garden	101959.255319148936	101959	101959
11	Toys	96187.170731707317	96187	96187
12	Industrial	92900.851063829787	92900	92901
13	Health	98975.652173913043	98975	98976
14	Grocery	101113.934782608696	101113	101114
15	Movies	100911.805555555556	100911	100912
16	Home	92734.711538461538	92734	92735

Query Editor Query History **Data Output** Explain Messages Notifications

/* Both round and trunc can be used to truncate to a variable number of decimal places */

```
select
  department, avg(salary),
round(avg(salary), 3), trunc(avg(salary), 4)
from
  staff
group by
   department;
```

Subqueries in SELECT clauses:04_01 file

- [Narrator] We usually select data directly from tables and views but sometimes it's helpful to be able to get data from the results of another select statement. Now we can do this using what are called **Sub Queries**. Sub Queries can be used in **three different parts of a select statement**. In the **list of values returned**, in the **From Clause** and in the **Where Clause**. Let's work through an example of each of these three. We'll do the first in this video. Let's start by building a basic query. So let's select oh the last name, the salary, and the department from the staff table.

```
/* Select name, salary and department from staff table */
select
  last_name,
  salary,
  department
from
  staff;
```

Data Output

	last_name character varying (100)	salary integer	department character varying (100)
1	Kelley	67470	Computers
2	Armstrong	71869	Sports
3	Carr	101768	Automotive
4	Murray	96897	Jewelery
5	Ellis	63702	Grocery
6	Phillips	118497	Tools
7	Williamson	65889	Computers
8	Harris	84427	Toys
9	James	108657	Jewelery
10	Sanchez	108093	Movies
11	Jacobs	121966	Jewelery
12	Black	44179	Clothing
13	Schmidt	85227	Baby
14	Webb	59763	Computers
15	Jacobs	141139	Games
16	Medina	106659	Baby

Query Editor Query History **Data Output** Explain Messages Notific

Now, because we'll have multiple select statements within a single query we'll want to make sure the database can tell which table each value comes from. So to do this, we'll use a **table alias** and include that alias as a prefix for each value we'll return. I'll assign an alias of S1 to the staff table and add S1 as a prefix to each column I want to return from that table.

/* Use an alias on table names so that one table can be queried in *//* subqueries and top level queries */

```
select   s1.last_name,   s1.salary,
s1.department from   staff s1;
```

Now when we execute this, we'll get the data we expect. Last names, salaries, and department names.

Data Output

	last_name character varying (100)	salary integer	department character varying (100)
1	Kelley	67470	Computers
2	Armstrong	71869	Sports
3	Carr	101768	Automotive
4	Murray	96897	Jewelery
5	Ellis	63702	Grocery
6	Phillips	118497	Tools
7	Williamson	65889	Computers
8	Harris	84427	Toys
9	James	108657	Jewelery
10	Sanchez	108093	Movies
11	Jacobs	121966	Jewelery
12	Black	44179	Clothing
13	Schmidt	85227	Baby
14	Webb	59763	Computers
15	Jacobs	141139	Games
16	Medina	106659	Baby

Query Editor Query History **Data Output** Explain Messages

Now, in addition to the last name, salary, and department, I'd like to include the average salary for each of those departments. This will let us easily compare a person's salary with the average for their department. Now to do that we'll use a **Sub Query that calculates the average salary for the department that matches the department of the employee's row**. So to do that I'm going to add another value here but instead of the value coming from the table it's going to come from a select statement. And I will select the average salary. Now average can return a lot a decimal points so I'm going to actually round that to the nearest dollar and I'm going to select the salary or the average salary from the staff table. Now I've already mentioned staff table once and have an alias for it. So I'm going to use a different alias to indicate that I'm talking about it, sort of a different reference to the staff table. And the other thing I want to do is make sure that I include in the average just the salaries for the department that I'm working with. And to do that I'll say anytime I refer to the department for S2, I want that department to be the same as the department that's listed up above in S1. Let's take a step back and look at this. We have our **outer query** which includes a last name, a salary, and a department. And then we have this **inner query** or **sub query** that calculates the average salary from the staff table but only using rows where the department is equal to the department of the employee we're currently looking at. So that's how the sub query works and we'll execute this statement. And we'll see now that we've added an extra column. This column lists the average salary for the corresponding department, which in the first row is computers. The second row is sports, third is automotive and so on. The important thing to note is that **we use a sub query where we'd usually use a column name.** Now within that sub query, we need a **Where Clause that references a table in the top level query so that the sub query knows which row is referenced**.

```
/* Include department's average
salary in each row with staff */

/* Use an alias on table names so that
one table can be queried in */
/* subqueries and top level queries */
select
  s1.last_name,
  s1.salary,
  s1.department,
  (select round(avg(salary))
from staff s2
where s1.department =
s2.department) dept_avg
from
  staff s1;
```

Data Output

	last_name character varying (100)	salary integer	department character varying (100)	dept_avg numeric
1	Kelley	67470	Computers	99095
2	Armstrong	71869	Sports	93901
3	Carr	101768	Automotive	99658
4	Murray	96897	Jewelery	87812
5	Ellis	63702	Grocery	101114
6	Phillips	118497	Tools	105021
7	Williamson	65889	Computers	99095
8	Harris	84427	Toys	96187
9	James	108657	Jewelery	87812
10	Sanchez	108093	Movies	100912
11	Jacobs	121966	Jewelery	87812
12	Black	44179	Clothing	95055
13	Schmidt	85227	Baby	93749
14	Webb	59763	Computers	99095
15	Jacobs	141139	Games	103884
16	Medina	106659	Baby	93749

Query Editor Query History **Data Output** Explain Messages Notifications

Subqueries in FROM clauses:04_01 file

- [Instructor] Let's look at another example of how to use subqueries. Assume that anyone who earns more than $100,000 per year is an executive. And that we'd like to find the average executive salary by department. We can do this by creating a subquery that returns the department and the salary of executives only. We then group by department and average the salaries of those executives. So let's start by building the subquery. We'll select and we want to be able to select a department and a salary. We want to select this from the staff table. Now we want to limit this to executives so we'll have a where clause that says where salary is greater than $100,000. Now I'm going to turn this into a **subquery** so I'll **wrap it** in parentheses and I want to make sure I give it an alias. So I'll call this S-one. Now I want to treat that almost like a table. I want to select from this, I want to select the department, include the alias in that, and I want to select the average salary. Select the average of the salary. Now again average sometimes returns a large number of decimal points, so I'm going to use the round function to round to the nearest dollar. And I want to select the department and the average salary from this query, this set of data about executives. So I'm going to indent a little bit just to make it obvious that this is a subquery. Now what you'll notice is the subquery has been labeled S-one, and I'm selecting from it because I'm referencing S-one up above. I am using an aggregate function and I do want to group by department so I'm going to include that as well. And of course I want to include the alias.

Data Output

	department character varying (100)	round numeric
1	Tools	124637
2	Electronics	124825
3	Sports	131570
4	Clothing	125692
5	Books	125114
6	Kids	127565
7	Music	124875
8	Automotive	124300
9	Toys	126293
10	Outdoors	126402
11	Garden	128793
12	Industrial	125143
13	Health	128508
14	Grocery	129747
15	Movies	120877
16	Home	128761

Query Editor Query History **Data Output**

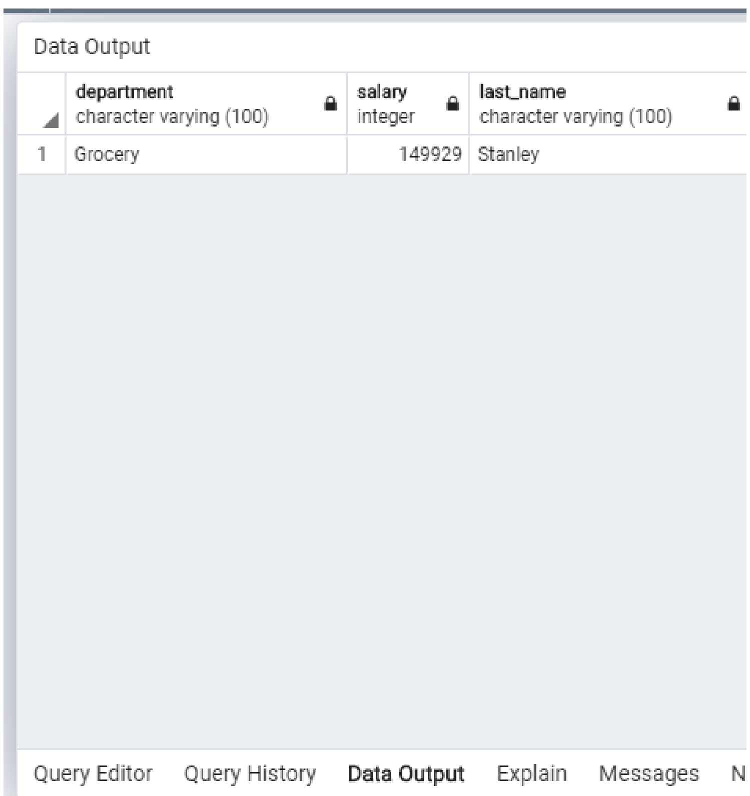
```
/* Select columns from a subquery instead of a table */
/* Find the average of executive salaries, defined as salaries >
100,000 */

select
  department,
  round(avg(salary))
from
   (select
       s2.department,
       s2.salary
    from
       staff s2
    where
       salary > 100000) s1
group by
   department;
```

There, now when we execute, what we find is we get a list of departments as expected, we get an average salary, but we'll notice all of the average salaries are above $100,000. Well above, and that indicates that we are on the right track. We have selected the salary of only executives and those are people that make more than $100,000 and we've done that by using a sub-select statement. The important thing to note here is that **we can use a subquery where we usually use a table name in the FROM clause.**

Subqueries in WHERE clauses:04_01 file

- [Male Voice] We can use subqueries in where clauses. These are **useful when we want to make comparisons within a single table**. For example, if we want to find the department of the person with the highest salary, we can use a subquery in the where clause that finds the maximum salary of all staff. Let's start by building the subquery. We want to select maximum or max salary from the staff table. Now, because we're working with **subqueries**, we want to make sure we use an **alias**, so I'll assign this the **alias s2,** and I'll make sure that **I reference s2** above and it's a **subquery so I will wrap this in parentheses**. So now I have a subquery that returns the **maximum salary** that is listed in the staff table. Now I want to build out the rest of my query, which is to **select the department** from the staff table, and I'll use **s1 as the reference** there, where the salary in s1 is equal to this maximum salary. So I'll move things around a little bit to make it obvious what I'm doing here. So what I'm doing is I have a subquery, which finds the maximum salary in the staff table. I then have a **top level** query which returns the department for whatever row has a salary that is equal to the maximum salary. Now instead of just showing the department name, let's show the last name and the salary. Now, of course, I want to include the aliases. So there. So this will show us the department, the last name and the salary of the person in the staff table who makes the maximum salary. So it turns out to be somebody named Stanley who works in the grocery department and makes almost $150,000 a year. So that's an example of how to use the subquery in a where clause.




```
/* Select the department that has the
employee with the highest salary */

select
  s1.department, s1.salary, s1.last_name
from
  staff s1
where
  (select max(salary) from staff s2) = salary;
```

Joining tables:04_04 file

- [Narrator] When working with SQL, we will sometimes need to retrieve data from multiple tables. For example, the Staff table includes a department for each employee. Departments are organized into divisions. Since we don't keep division information in the staff table, we have to look it up somewhere else. In our example data set, we have a table called Company Divisions. Let's take a look at that. We'll do that by typing Select + *, to get all the columns From, company divisions and that's plural. And when we execute it, we'll notice that first of all, 21 rows are returned and the table has two columns. One listed departments that we are familiar with, we've seen those before. And the other column lists a higher level grouping or division. Which include things like Auto and Hardware, Domestic and Electronic Equipment. Now, let's join the Staff and Company Divisions tables. So, let's replace the Select + * with a Select. Oh, let's select from the Staff table a last name and a department and let's from the Company Divisions table let's select Company Division. Now we'll have to do this from two tables, from the Staff table and the Company Divisions table. And I'll make sure I alias both of those. Now, to join these, I need to use the **Join** keyword. And so now I'm joining the Staff table to the Company Division and I have to specify which columns to look at to join. And to do that, I specify the **On** keyword and from the Staff table, I want to look at Department. And similarly, from the Company Divisions table, I want to look at the Department, as well. So, when the departments are equal in both cases, then we join those rows. So again, this Select statement, chooses two columns from the Staff table, one column from the Company Divisions table and it joins using the Department columns in both. And when we execute, we get these results. So we have the last name and the department from the Staff table and the company division from the Company Division table.


Data Output

	department [PK] character varying (100)	company_division character varying (100)
1	Automotive	Auto & Hardware
2	Baby	Domestic
3	Beauty	Domestic
4	Clothing	Domestic
5	Computers	Electronic Equipment
6	Electronics	Electronic Equipment
7	Games	Domestic
8	Garden	Outdoors & Garden
9	Grocery	Domestic
10	Health	Domestic
11	Home	Domestic
12	Industrial	Auto & Hardware
13	Jewelery	Fashion
14	Kids	Domestic
15	Movies	Entertainment
16	Music	Entertainment

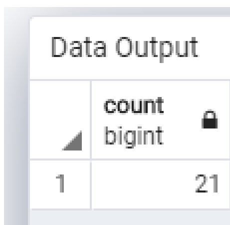
Query Editor Query History Data Output Explain Mes

/* Select all columns in the company_division table to review contents of the table */

```
select
  *
from
  company_divisions;
```

/* And get a count to understand the size of the table */

```
select
  count(*)
from
  company_divisions;
```


Data Output

	count bigint
1	21

```
/* Join staff and department. If the staff department is not found in */
/* company_divisions, then no row will be returned. */
select
  s.last_name, s.department, cd.company_division
from
  staff s
join
  company_divisions cd
on
  s.department = cd.department;
```

Data Output

	last_name character varying (100)	department character varying (100)	company_division character varying (100)
937	Lopez	Sports	Games & Sports
938	Williams	Jewelery	Fashion
939	Stevens	Electronics	Electronic Equipment
940	Ryan	Beauty	Domestic
941	Lawrence	Tools	Auto & Hardware
942	Cole	Beauty	Domestic
943	Fields	Music	Entertainment
944	Thomas	Kids	Domestic
945	Turner	Electronics	Electronic Equipment
946	Edwards	Outdoors	Outdoors & Garden
947	Hamilton	Health	Domestic
948	Wood	Jewelery	Fashion
949	James	Games	Domestic
950	Reynolds	Computers	Electronic Equipment
951	Walker	Games	Domestic
952	Kennedy	Industrial	Auto & Hardware

Query Editor Query History **Data Output** Explain Messages Notifications

Now, I'm going to execute this query again and point out that 953 rows were returned. Now, there are 1000 rows in the Staff table. So something has not gone right here. How did we lose 47 rows? What we want to do now, if we want to return all rows in the Staff table, even when no corresponding row is found in the Company Divisions table, then we use an outer join to return all rows. And that's what we want to do here because for some reason, 47 rows were not returned. Now, **outer joins** can either be **left** outer joins or **right** outer joins. And that has to do with **the ordering of the tables in the join statement**. We're going to use a left outer join because that will return all rows from the table referenced on the left side of the keyword. I'm going to make this a little explicit and put the join on the same line as the two tables that we're working with. And now I'm going to change this join to a left join and this says select these three columns we've specified, Last Name, Department and Company Division from the Staff table, left join to the Company Divisions table. Again, by left joining we're going to take all of the rows in the Staff table, even if there isn't a corresponding row in the Company Divisions table.

So, let's execute that. We have 1000 rows that were returned in this case. We have some rows in this result set that don't have a company division. So, let's find out where those are. Let's select Where Company Division is null. Which is how SQL indicates there is no value for a column.

```
/* The previous query did not return 1,000 rows. What rows are missing? */
select distinct
  department
from
  staff
where
  department not in
   (select
       department
    from
       company_divisions);
```

Data Output

	department character varying (100)
1	Books

/* Use an outer join to return all rows, even it a corresponding row in */
/* company_divsion does not exist. */

```
select
  s.last_name, s.department,
cd.company_division
from
  staff s
left join
  company_divisions cd
on
  s.department = cd.department;
```

Data Output

	last_name character varying (100)	department character varying (100)	company_division character varying (100)
984	Williams	Jewelery	Fashion
985	Stevens	Electronics	Electronic Equipment
986	Ryan	Beauty	Domestic
987	Lawrence	Tools	Auto & Hardware
988	Cole	Beauty	Domestic
989	Fields	Music	Entertainment
990	Thomas	Kids	Domestic
991	Turner	Electronics	Electronic Equipment
992	Edwards	Outdoors	Outdoors & Garden
993	Anderson	Books	[null]
994	Hamilton	Health	Domestic
995	Wood	Jewelery	Fashion
996	James	Games	Domestic
997	Reynolds	Computers	Electronic Equipment
998	Walker	Games	Domestic
999	Kennedy	Industrial	Auto & Hardware

Query Editor Query History **Data Output** Explain Messages Notifications

And what we notice here is, we have 47 rows returned, as expected. They're all from the Books department so it must be the case that Books is not included in the Company Divisions table.

Creating a view:04_05 file

- [Narrator] We can **group data** in a variety of ways using Sequel. Now we'll be working with data from the staff, company divisions and company regions table. So let's create a view using a select statement with the appropriate joins. This will save us from having to type long joined statements repeatedly. So lets start by defining the select statement that we want. So I'll start by saying, select. I want to select all of the columns from the staff table. And I want to select company division. And I want to select company region, and I want to select this from the staff table, which we'll alias as s. And, we'll left join on the company divisions table, and we'll do this on the staff table department. And, company divisions department. And we'll left join company regions. On, staff, region id and the company regions. Region id. And before I run this, it looks like I missed the s at the end of the cr.company_regions column. And, what we have here is a select statement that uses two left joins and it selects all the rows from the staff table, it selects the company division, and the company regions name. And, it returns them all as a single table. So, rather than re-typing this select statement every time we need this combination of these three tables, we can simply **create a view** and we do that by issuing the command, create, view, and let's give it a name, we'll call this, staff_div_reg AS. So now, we're able to specify a view which we will call, staff_div_reg. And it will be associated with this select statement that we just created. So if I execute this command, instead of getting the results returned, I actually create the view. And so that view has been created as the message implies. Now, let's just do a quick double check and make sure everything worked as expected. So we'll clear this out, and now what I want to do, is I want to select, count(*), from that view I just created. So I'm going to select it FROM staff_div_reg, and if all goes well, we will have a thousand returned, and we do. So all thousand rows in the staff table, have been joined to company regions and company divisions, and now we have a view that we can work with.

```
/* Create a view to minimize the amount of typing and reduce the risk of making a mistake */
create view staff_div_reg as
   select
     s.*, cd.company_division, cr.company_regions
   from
     staff s
   left join
     company_divisions cd
   on
     s.department = cd.department
   left join
     company_regions cr
   on
      s.region_id = cr.region_id;

/* remove view is already created */
DROP view staff_div_reg;
```

/* Verify the view has 1,000 rows */
```
select
  count(*)
from
  staff_div_reg;
```

Data Output

	count bigint
1	1000

/* Get the number of employees in each division within each region */
```
select
   company_division, company_regions,
count(*)
from
   staff_div_reg
group by
   company_division, company_regions
```

Data Output

	company_division character varying (100)	company_regions character varying (20)	count bigint
1	Electronic Equipment	Northeast	15
2	Entertainment	Southeast	7
3	Auto & Hardware	British Columbia	14
4	Fashion	British Columbia	7
5	Fashion	Northwest	4
6	Fashion	Quebec	5
7	Entertainment	Northeast	13
8	Games & Sports	British Columbia	9
9	Fashion	Nova Scotia	11
10	Entertainment	Southwest	13
11	Domestic	Nova Scotia	72
12	Games & Sports	Northeast	13
13	Outdoors & Garden	Southeast	13
14	Fashion	Southeast	3
15	[null]	British Columbia	4

Query Editor Query History **Data Output** Explain Messages No

Grouping and totaling:04_05 file

- [Instructor] Let's use the view we just created to get a count of the number of employees in each region. So we'll enter a SELECT query and let's select the company region and let's get a count of company regions. And we'll get this from our view, staff_div_reg, now we're aggregating, we're using a count, so we'll have to have a GROUP BY statement, and in this case we'll want to group by company region. Let's add an ORDER BY so we get the company region counts listed alphabetically.

/* Add an order by clause to make it easier to read */
```
select
   company_division, company_regions,
count(*)
from
   staff_div_reg
group by
   company_division, company_regions
order by
   company_regions, company_division;
```

Data Output

	company_division character varying (100)	company_regions character varying (20)	count bigint
1	Auto & Hardware	British Columbia	14
2	Domestic	British Columbia	58
3	Electronic Equipment	British Columbia	10
4	Entertainment	British Columbia	14
5	Fashion	British Columbia	7
6	Games & Sports	British Columbia	9
7	Outdoors & Garden	British Columbia	13
8	[null]	British Columbia	4
9	Auto & Hardware	Northeast	21
10	Domestic	Northeast	57
11	Electronic Equipment	Northeast	15
12	Entertainment	Northeast	13
13	Fashion	Northeast	3
14	Games & Sports	Northeast	13
15	Outdoors & Garden	Northeast	14

Query Editor Query History **Data Output** Explain Messages No

And I'll execute, and we'll notice we have our list of seven regions, with a count of the number of staff in each. Now if we want counts by both region and division, we can use a feature called **grouping sets**. Here, let's look at an example that returns employee counts by division and by region. So the first thing I'll do is I will add company division to my list of columns that I'm returning. **In the GROUP BY clause, I'm going to add the phrase, grouping sets, and then I'm going to give it a list of columns that I would like to group by**. Well I'd like to group by company division, and by company region, so I'll close the list. Now I've also added company division to the list of columns that I returned so I'll want to be sure to include that in my ORDER BY clause. Now let's execute and see the results.

```
/* Get employee counts by division and by region */
select
  company_division, company_regions, count(*)
from
  staff_div_reg
group by
  grouping sets (company_division, company_regions)
order by
  company_regions, company_division;
```

Data Output

	company_division character varying (100)	company_regions character varying (20)	count bigint
1	[null]	British Columbia	129
2	[null]	Northeast	144
3	[null]	Northwest	129
4	[null]	Nova Scotia	159
5	[null]	Quebec	117
6	[null]	Southeast	154
7	[null]	Southwest	168
8	Auto & Hardware	[null]	132
9	Domestic	[null]	425
10	Electronic Equipment	[null]	101
11	Entertainment	[null]	73
12	Fashion	[null]	46
13	Games & Sports	[null]	81
14	Outdoors & Garden	[null]	95
15	[null]	[null]	47

Now what you'll notice here is we still get our seven company regions, those are listed first, but we also get totals by company division, those are the next set. Now we'll notice one of these down at the bottom has no name for company division**, that's because the books department didn't have a corresponding division.** Now let's just add in gender just to generate some additional results. So to do that, I'll add gender to my list of columns that I want to return. And I will add it to my list that follows the phrase grouping sets. And of course I'll want it in the ORDER BY clause.

/* Now, add in gender to break down even further */
/* Get employee counts by division and by region */
/* There are null values in all columns, but the relevant fields have values */

Data Output

	company_division character varying (100)	company_regions character varying (20)	gender character varying (10)	count bigint
1	[null]	British Columbia	[null]	129
2	[null]	Northeast	[null]	144
3	[null]	Northwest	[null]	129
4	[null]	Nova Scotia	[null]	159
5	[null]	Quebec	[null]	117
6	[null]	Southeast	[null]	154
7	[null]	Southwest	[null]	168
8	Auto & Hardware	[null]	[null]	132
9	Domestic	[null]	[null]	425
10	Electronic Equipment	[null]	[null]	101
11	Entertainment	[null]	[null]	73
12	Fashion	[null]	[null]	46
13	Games & Sports	[null]	[null]	81
14	Outdoors & Garden	[null]	[null]	95
15	[null]	[null]	Female	496
16	[null]	[null]	Male	504

Query Editor Query History **Data Output** Explain Messages Notifications

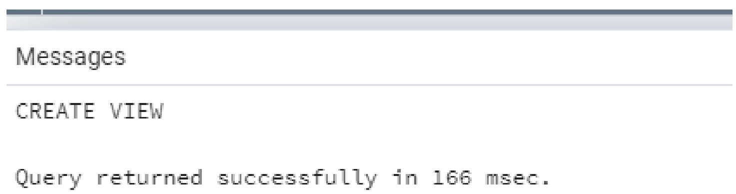
```
select
  company_division, company_regions,
gender, count(*)
from
  staff_div_reg
group by
  grouping sets (company_division,
company_regions, gender)
order by
  company_regions, company_division,
gender;
```

And now we'll notice again the first set of results is the seven company regions, followed by company divisions, followed by gender.

ROLLUP and CUBE to create subtotals:04_07 file

- [Instructor] Let's continue our look at ways to **group and aggregate** data with two other operators**, ROLLUPs and CUBE**s. Now first, let's modify the staff division region view we created to include country code. To do that, I'm going to use the command **CREATE OR REPLACE VIEW**, and as the name implies, if this view doesn't exist, it will simply create it. If there is a version of this view that already exists, it'll replace that version with the version I'm about to specify. And we'll call this staff_div_reg_country, and we'll define this as the SELECT of all the columns in the staff table, plus company_division, company_regions, and country. And we'll select these columns FROM staff table, which we'll alias as s. And the staff table will LEFT JOIN to the company_divisions table, which we'll alias as cd, and that left join will be performed on department in both state and company_division. And then we'll take the results of that left join operation and apply another left join, this time to company_regions, and that will be on the staff table region_id, and the company_region's region_id.

```
/* create or replace view */
create or replace view staff_div_reg_country as
   select
     s.*, cd.company_division, cr.company_regions,cr.country
   from
     staff s
   left join
     company_divisions cd
   on
     s.department =
cd.department
   left join
     company_regions cr
   on
      s.region_id = cr.region_id;
```



So, let's execute that to create that view. Now, let's select the number of employees by company, region, and country. So the first thing I would do is just clean up this. I'll remove the text that I had before to create the view, and I'll specify a SELECT statement. And what I want to select is company_regions, country, and then a count of those. And I'll get this from the view I just created, staff_div_reg_country. Now I have an aggregate of the count, so I'll need a GROUP BY statement, and I'm going to group by company_regions and country, and let's specify an ORDER BY clause. Let's order by country first, and then by company_regions.

```
/* Select number of employees by
company_region and country */
select
  company_regions, country, count(*)
from
  staff_div_reg_country
group by
  company_regions, country
order by
  country, company_regions
```

Data Output

	company_regions character varying (20)	country character varying (20)	count bigint
1	British Columbia	Canada	129
2	Nova Scotia	Canada	159
3	Quebec	Canada	117
4	Northeast	USA	144
5	Northwest	USA	129
6	Southeast	USA	154
7	Southwest	USA	168

Now let's execute this SELECT. This shows the totals for each regions in each of the two countries.

Now, if we wanted to see the totals from each country as well, we can use the **ROLLUP operation on the GROUP BY clause**. This will calculate sums in the hierarchy for regions and countries. So, in the GROUP BY clause, we're going to specify ROLLUP, and we're going to rollup by country and then by company_regions.

```
/* Use rollup operation on the group by clause to create hierarchical sums */
select
  company_regions, country,
count(*)
from
  staff_div_reg_country
group by
  rollup (country, company_regions)
order by
  country, company_regions
```

Data Output

	company_regions character varying (20)	country character varying (20)	count bigint
1	British Columbia	Canada	129
2	Nova Scotia	Canada	159
3	Quebec	Canada	117
4	[null]	Canada	405
5	Northeast	USA	144
6	Northwest	USA	129
7	Southeast	USA	154
8	Southwest	USA	168
9	[null]	USA	595
10	[null]	[null]	1000

And so you see, for example, we have the total for Canada as well as the total for US and a total for both countries combined.

Now, for more advanced breakdowns, we can use the **CUBE** operation on the GROUP BY clause. This tells SQL to create all possible combinations of sets of grouping columns. For example, for each division, show results by region. So we will move the ORDER BY here and we'll remove ROLLUP, and we'll specify CUBE, and we're going to use not country, but we're going to use a cube of company_division. So I'll need to specify company_division here in the SELECT statement. So I'll have company_division, company_region, and count, but we no longer need country, so I'll remove that. So now, my SELECT statement is to select company_division and company_regions along with a count FROM the staff_div_reg_country view, and then apply the CUBE operation.

/* Use cube operation on the group by clause to create all possible combination of sets of grouping columns */

```
select
  company_division, company_regions, count(*)
from
  staff_div_reg_country
group by
  cube (company_division, company_regions);
```

Data Output

	company_division character varying (100)	company_regions character varying (20)	count bigint
1	[null]	[null]	1000
2	Electronic Equipment	Northeast	15
3	Entertainment	Southeast	7
4	Auto & Hardware	British Columbia	14
5	Fashion	British Columbia	7
6	Fashion	Northwest	4
7	Fashion	Quebec	5
8	Entertainment	Northeast	13
9	Games & Sports	British Columbia	9
10	Fashion	Nova Scotia	11
11	Entertainment	Southwest	13
12	Domestic	Nova Scotia	72
13	Games & Sports	Northeast	13
14	Outdoors & Garden	Southeast	13
15	Fashion	Southeast	3
16	[null]	British Columbia	4

Query Editor Query History **Data Output** **Explain** Messages Nc

This shows totals by division and regions with totals by division.

FETCH FIRST to find top results:04_08 file

- [Narrator] When working with large data sets, we're sometimes interested only in the top results, based on some sort criteria. For example, we might want to list the employees with the top 10 salaries. Let's see how easy it is to work with the top results in SQL. Now I'm going to build a select statement, and let's select the last name, job title, and salary. Let's pull this from the staff table, and we want to know about salary and top salaries, so we're going to order by salary, and we want the top salary, so we want to start at the highest so we'll use the descending, or **DESC** keyword. Now, what I'm going to do is add a clause called **fetch** first. Now, **fetch first works with the order by clause to sort and limit results.** Fetch first is like the limit keyword, in that only a fixed number of rows are returned, but **with fetch first, the ordering is performed before choosing the rows to return**. So, I'll specify fetch first, 10 rows only, so this will return only 10 rows. And as we'll see, there are 10 rows returned, and they're in descending order, by salary. An important point to remember is that fetch first works with the order by clause to sort the results before selecting the rows to return. This is different from the way the limit clause works. Limit actually limits the number of rows, and then performs the operations.

/* Use order by and fetch first to limit the number of rows returned */

```
select
  last_name, job_title, salary
from
  staff
order by
  salary desc
fetch first
  10 rows only;
```

Data Output

	last_name character varying (100)	job_title character varying (100)	salary integer
1	Stanley	Director of Sales	149929
2	Greene	Geologist III	149835
3	Morales	VP Product Management	149598
4	King	Structural Analysis Engineer	149597
5	Allen	Account Coordinator	149586
6	Freeman	Design Engineer	149351
7	Stewart	Senior Sales Associate	149336
8	Cox	Senior Financial Analyst	149221
9	Riley	Business Systems Developme...	149114
10	Long	Executive Secretary	149099

So now let's build a simple aggregation query. So I'll remove most of this part of the query, and let's select company division, and count from a staff div reg country view, and let's group by company division, and we'll order by count, and we'll execute.

Data Output

	company_division character varying (100)	count bigint
1	[null]	47
2	Entertainment	73
3	Outdoors & Garden	95
4	Games & Sports	81
5	Domestic	425
6	Auto & Hardware	132
7	Fashion	46
8	Electronic Equipment	101

```
select
  company_division, count(*)
from
  staff_div_reg_country
group by
  company_division;
```

And we have a list here, so let's tweak this list a little bit, let's make it in descending order, and I'll do that by adding the descending, or DESC keyword, so now we have it in descending order, so let's finally add a fetch first and limit the results to the top five divisions by staff count, and we'll do that by specifying fetch first five rows only, and when we execute, five rows are returned, and it happens to be the top five.

```
/* Use fetch first with order by to select top 5 divisions by staff count */
select
  company_division, count(*)
from
  staff_div_reg
group by
  company_division
order by
  count(*) desc
fetch first
  5 rows only;
```

Data Output

	company_division character varying (100)	count bigint
1	Domestic	425
2	Auto & Hardware	132
3	Electronic Equipment	101
4	Outdoors & Garden	95
5	Games & Sports	81

5. Window Functions and Ordered Data

Window functions: OVER PARTITION:05_01 file

- [Instructor] **Window functions** allow us to make SQL statements about rows that are related to the current row during processing. This is somewhat like the way subqueries work. They let us do an operation that's related to the current row that SQL is processing. For example, instead of using a subquery to calculate an average salary for an employee's department, we can use a windowing function on rows called **OVER PARTITION**. Let's take a look at an example. Let's select department, last_name, and salary. So that'll list for us the departments, last name, and salaries of each of the employees, but I also want to look at an average salary for each department. I can specify the average salary aggregate and then say I would like to have the average salary over a partition. And I use the phrase **PARTITION BY**, and then specify how I want to do my grouping. In this case, I want to do it by department, and I'll select this from the staff table. So again, what I have is I've selected department, last_name, and salary, and I'm going to also display an average of salary for each department, and that's what the **OVER PARTITION BY** statement does. Let's see what we have here.

```
/* Select individual salary and average department salary */
select
   department,
   last_name,
   salary,
   avg(salary) over (partition by department)
from
   staff;
```

Data Output

	department character varying (100)	last_name character varying (100)	salary integer	avg numeric
1	Automotive	Reed	126001	99658.000000000000
2	Automotive	Ortiz	91296	99658.000000000000
3	Automotive	Mcdonald	111041	99658.000000000000
4	Automotive	Torres	120875	99658.000000000000
5	Automotive	Peterson	53964	99658.000000000000
6	Automotive	Burns	44377	99658.000000000000
7	Automotive	Edwards	140194	99658.000000000000
8	Automotive	Nichols	110589	99658.000000000000
9	Automotive	Ross	71363	99658.000000000000
10	Automotive	Gordon	136448	99658.000000000000
11	Automotive	Hall	83177	99658.000000000000
12	Automotive	Thomas	116487	99658.000000000000
13	Automotive	Jenkins	84356	99658.000000000000
14	Automotive	Morris	91932	99658.000000000000
15	Automotive	Carroll	128885	99658.000000000000
16	Automotive	Boyd	69936	99658.000000000000

Query Editor Query History **Data Output** **Explain** Messages Notifications

What we'll notice is we have a list ordered by department, and then we have individual's last names, their salary, and then the average salary for each person in that department. So we'll notice for the automotive department, the average salary is 99,658. Now, let's scroll down and see a different department. We'll notice in the baby department the average salary now changes. It's about $93,750.

We can use other aggregate functions too. For example, let's change the average to max, and I'll leave everything else the same. And I'll execute that query. Now what I'm listing includes the department name, the employee's name, the employee's salary, and then the maximum salary in that department. So for example in automotive, the maximum salary is 146,167, and we'll scroll down again to a different department to see that it shifts. In the baby department, the maximum salary is 148,687.

```
/* Use a windowing operation with a different aggregate function */
select
   department,
   last_name,
   salary,
   max(salary) over (partition by department)
from
   staff;
```

Data Output

	department character varying (100)	last_name character varying (100)	salary integer	max integer
39	Automotive	Duncan	45774	146167
40	Automotive	Mcdonald	69594	146167
41	Automotive	Alexander	144724	146167
42	Automotive	Foster	63364	146167
43	Automotive	Meyer	42602	146167
44	Automotive	Owens	135326	146167
45	Automotive	Ortiz	113231	146167
46	Automotive	Fox	87134	146167
47	Baby	Wallace	65216	148687
48	Baby	Mcdonald	141464	148687
49	Baby	Williams	131273	148687
50	Baby	Dixon	106281	148687
51	Baby	Day	125914	148687
52	Baby	Barnes	112837	148687
53	Baby	Scott	86497	148687
54	Baby	Price	96388	148687

Query Editor   Query History   **Data Output**   **Explain**   Messages   Notifications

Now, let's change both the aggregate function and the column we partition by. So I'm going to clean up. I'm going to use the view that we had created, and what I'd like to do is SELECT company_regions, last_name, salary, and the minimum salary. And here, I'm going to do it **OVER a PARTITION BY** company_regions. And since I'm referencing company_regions, I'll select from our view which we created earlier called staff_div_reg_country. And I'll execute, and here, what we have is a similar list, but instead of department, we have company_regions, and we have the minimum salary that's earned by anyone in that region. So for example, in British Columbia, the minimum salary is 40,194, but let's scroll down to another region. For example, here in the northeast, the minimum salary is $41,026.

```
/* Use a windowing operation with a different aggregate function and different grouping */
select
  company_regions,
  last_name,
  salary,
  min(salary) over (partition by company_regions)
from
  staff_div_reg_country;
```

Data Output

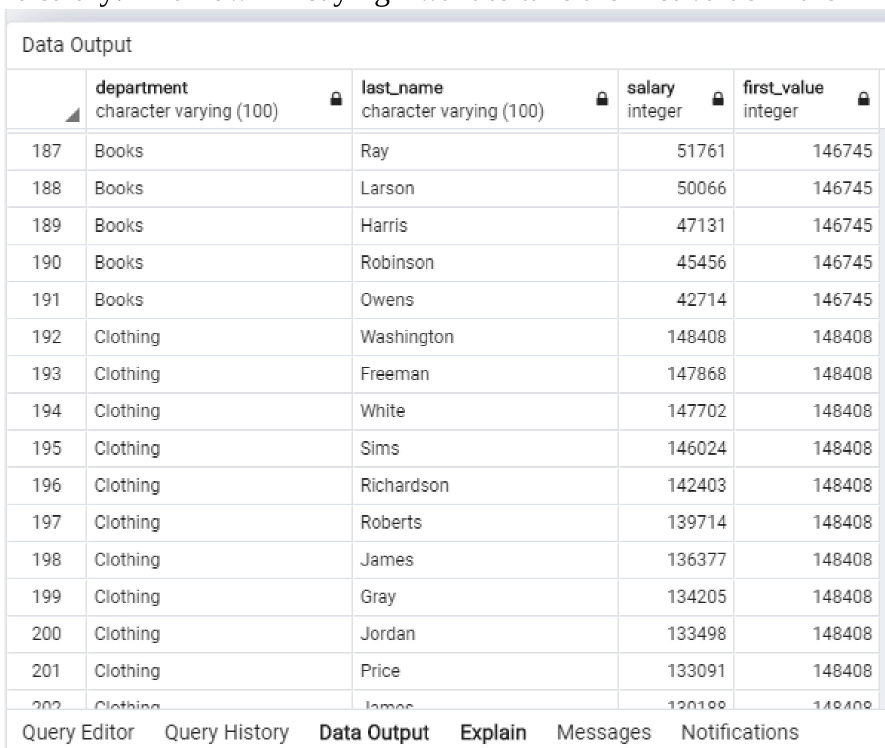
	company_regions character varying (20)	last_name character varying (100)	salary integer	min integer
126	British Columbia	Turner	130849	40194
127	British Columbia	Foster	54007	40194
128	British Columbia	Jones	128048	40194
129	British Columbia	Parker	148906	40194
130	Northeast	Cruz	61739	41026
131	Northeast	Vasquez	94596	41026
132	Northeast	Schmidt	125465	41026
133	Northeast	Lewis	74191	41026
134	Northeast	Richards	80939	41026
135	Northeast	Jackson	41516	41026
136	Northeast	Rose	50060	41026
137	Northeast	Ellis	146256	41026
138	Northeast	Welch	49463	41026
139	Northeast	Reyes	71559	41026
140	Northeast	Austin	48840	41026

Query Editor Query History **Data Output** **Explain** Messages Notifications

Window functions: FIRST_VALUE:05_01 file

- [Narrator] We could also select a set of attributes grouped by department and include the first value by department in each row using something called **The First Value Function**. Let's take a look at an example. Let's select department, last name, salary, and now let's put in our function here. I want to select First Value. And I want to select the First Value from the list of salaries and I want that list to be over a **PARTITION** by department. So that's grouping by department. And I want to order it by salary itself and I would like to start at the highest salary and go down to the lowest. So I'm going to use the Descending, or **DESC** keyword here. Now I'm going to select from staff and let's just review this one more time. We have the three columns that we've been using, department, last name and salary. And now I'm saying I want to take the first value in the list of salaries where that list is grouped by or partitioned by department in descending order by salary. So let's execute that and see what happens.

```
/* Select a set of attributes grouped by department, include the first value by department in each row */
select
  department,
  last_name,
  salary,
  first_value(salary) over (partition by department order by salary desc)
from
  staff;
```


Data Output

	department character varying (100)	last_name character varying (100)	salary integer	first_value integer
187	Books	Ray	51761	146745
188	Books	Larson	50066	146745
189	Books	Harris	47131	146745
190	Books	Robinson	45456	146745
191	Books	Owens	42714	146745
192	Clothing	Washington	148408	148408
193	Clothing	Freeman	147868	148408
194	Clothing	White	147702	148408
195	Clothing	Sims	146024	148408
196	Clothing	Richardson	142403	148408
197	Clothing	Roberts	139714	148408
198	Clothing	James	136377	148408
199	Clothing	Gray	134205	148408
200	Clothing	Jordan	133498	148408
201	Clothing	Price	133091	148408
202	Clothing	James	130188	148408

Query Editor Query History **Data Output** **Explain** Messages Notifications

Well, what we have here is a list with departments, employee's last names, and then the salary ordered in descending order. So for example, Sanchez in Automotive earns 146,167 and then each of the employees listed below there is in order of their salary in descending order. But the First Value Function always returns the salary of the first person in the list, grouped by department. So as you'd expect as we scroll down, we'll notice when we shift to the baby department now the top name is Howard, the maximum salary, or in this case the First Value is 148,687 and that's the value that's shown in the First Value Column. Now in this case, First Value returns the same ordering as if we had used the MAX Function. **What's different about First Value** is that we can **change the Order By** Claus. So instead of ordering by salary, let's order by last name and execute. Now what we have is a grouping by departments, so we still have Automotive grouped together. It's ordered by last name, in this case it's in alphabetical order. **The first value is the value of the salary of the person who's name is first in alphabetical order in that department**. So in this case it's someone named Adam who earns 79,045. And what you'll notice is the last names are ordered in alphabetical order, the first value is always the value of Adam's salary. So the first value refers to the first line that appears in the grouping. And as we scroll down into the baby department what we'll notice here is this person earns a salary of 66,847 and that salary continues to show in the First Value Column for everyone in the baby department. So that's the difference between First Value and Max. We can change which value appears as the first value in the list by changing the Order by Claus.

Window functions: RANK:05_01 file

- Another useful function is the **Rank function**. It works with the **partition** operation to order results and assign a Rank value based on the way the partition data is sorted. Let's look at an example. Let's select department, last name, salary, and now let's introduce the Rank function. And it works with the over partition by operator. And we're going to say we're going to partition by, recruit by department. And we're going to order that by salary. And I'd like to use descending again. So I'll add that. And we'll select this from staff. Now let's execute and see what kind of values we get.

```
/* Order results and include the relative rank by row */
select
   department,
   last_name,
   salary,
   rank() over (partition by department order by salary desc)
from
   staff;
```

Data Output

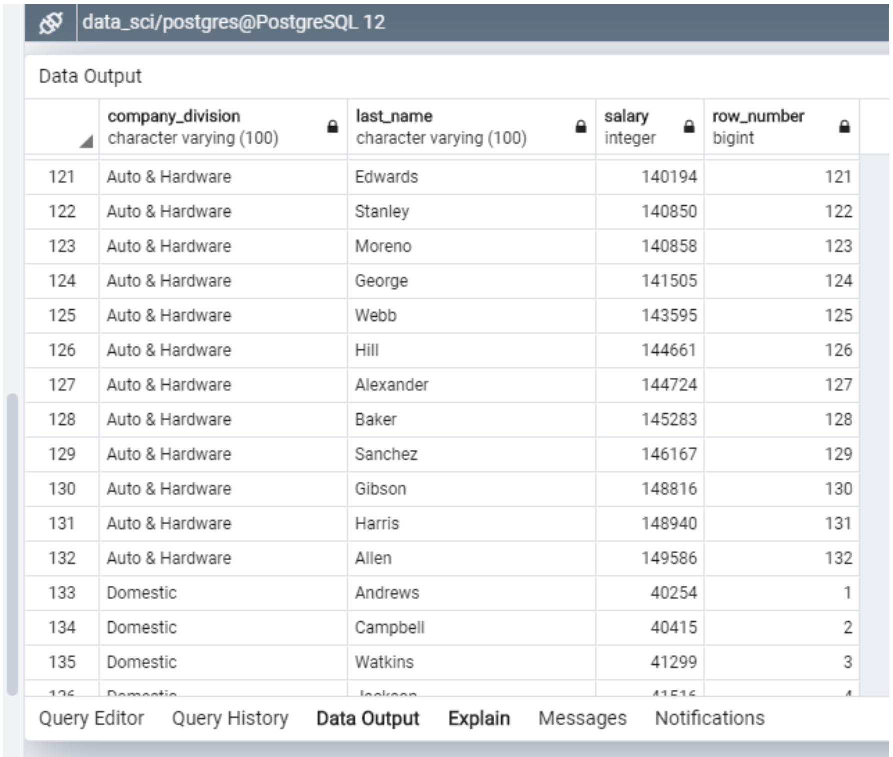
	department character varying (100)	last_name character varying (100)	salary integer	rank bigint
38	Automotive	Gutierrez	67800	38
39	Automotive	Rose	66063	39
40	Automotive	Foster	63364	40
41	Automotive	Simmons	58555	41
42	Automotive	Peterson	53964	42
43	Automotive	Marshall	47281	43
44	Automotive	Duncan	45774	44
45	Automotive	Burns	44377	45
46	Automotive	Meyer	42602	46
47	Baby	Howard	148687	1
48	Baby	Tucker	148573	2
49	Baby	Mcdonald	141464	3
50	Baby	Richards	140004	4
51	Baby	Dixon	138970	5
52	Baby	Harrison	134334	6

Query Editor Query History **Data Output** **Explain** Messages Notifications

Now what we'll notice here is that we have a familiar set of data. We have a listing by department, with the employees last name and the employees salary. The salaries are sorted in descending order. So the highest is first. And then we also have this column called **Rank**. And that lists the number one through the number of employees in the department. So Rank tells us where each salary falls in the department. And we'll notice as we move into the next department, in this case, the baby department, the ranking restarts again at number one. And that number one is assigned to the person in the department who has the highest salary. Window functions like Rank can streamline complex queries that would otherwise require sub-queries or multiple queries to process.

/* Window functions can be used to add ranked row numbers */

```
select
  company_division,
  last_name,
  salary,
  row_number()  over
(partition by
company_division
order by salary asc)
from
  staff_div_reg;
```



	company_division character varying (100)	last_name character varying (100)	salary integer	row_number bigint
121	Auto & Hardware	Edwards	140194	121
122	Auto & Hardware	Stanley	140850	122
123	Auto & Hardware	Moreno	140858	123
124	Auto & Hardware	George	141505	124
125	Auto & Hardware	Webb	143595	125
126	Auto & Hardware	Hill	144661	126
127	Auto & Hardware	Alexander	144724	127
128	Auto & Hardware	Baker	145283	128
129	Auto & Hardware	Sanchez	146167	129
130	Auto & Hardware	Gibson	148816	130
131	Auto & Hardware	Harris	148940	131
132	Auto & Hardware	Allen	149586	132
133	Domestic	Andrews	40254	1
134	Domestic	Campbell	40415	2
135	Domestic	Watkins	41299	3

LAG and LEAD:05_04 file

- [Narrator] Newer versions of SQL provide additional features for operating on rows related to the currently processed row. For example, if you want to know a person's salary and the next lower salary in the department, we could use the **lag function** to reference rows relative to the currently processed rows. So, let's look at an example. We'll select, department, last name, salary. And now we'll introduce the lag function. And I would like to lag on salary over a partition by department and let's order that by salary and we'll use the descending keyword and let's select this from the Staff table.

```
/*  Use lag to reference rows behind */
select
  department,
  salary,
  lag(salary,3) over (partition by department
order by salary desc)
from
  staff
```

Data Output

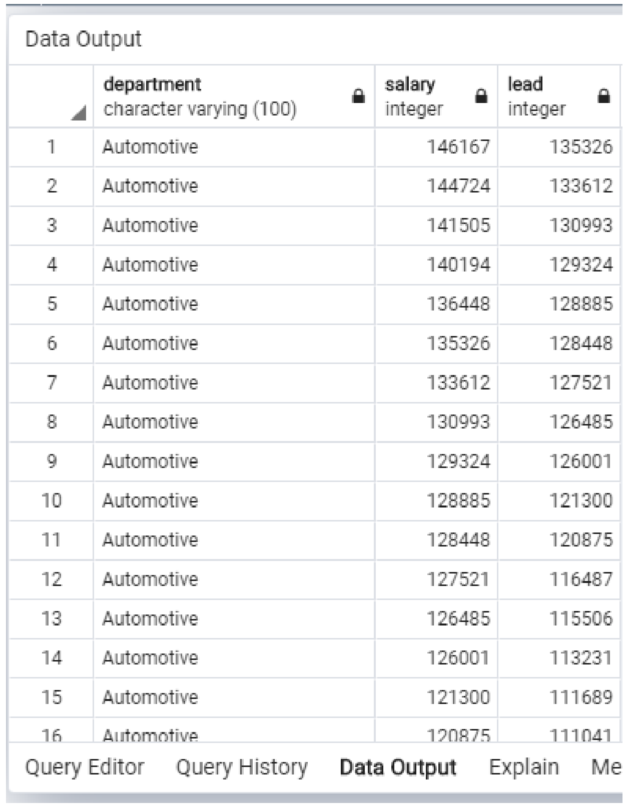
	department character varying (100)	salary integer	lag integer
1	Automotive	146167	[null]
2	Automotive	144724	[null]
3	Automotive	141505	[null]
4	Automotive	140194	146167
5	Automotive	136448	144724
6	Automotive	135326	141505
7	Automotive	133612	140194
8	Automotive	130993	136448
9	Automotive	129324	135326
10	Automotive	128885	133612
11	Automotive	128448	130993
12	Automotive	127521	129324
13	Automotive	126485	128885
14	Automotive	126001	128448
15	Automotive	121300	127521
16	Automotive	120875	126485

Query Editor Query History **Data Output** Explain Mes

Now, it's best, let's just execute this and look at the results. What we have is a result set that includes the department name, an employee's name, their salary and then a column labeled lag. And what lag is referring to is to the row that came before the currently processed row. So for example, the very first one doesn't have a currently processed row so the lag value is null. But when we move to row two, which is Automotive, the person's name is Alexander. That person's salary is $144,724. The lag or the salary that came before is $146,167. Let's scroll down to the next department and we'll notice the switch again. Here, there's a break when we shift from Automotive to the new department, Baby. Salaries are ordered in a descending order and the first row in this case, Howard, with a salary of $148,687, doesn't have a previous salary. So there's no value there. However, when we move to the second name in the department, Tucker, we'll notice that the lag column has the value of the salary from the previous row. So, lag is one of those operators that works relative to the current row that we're referring to.

Another operator is called **Lead**. Let's take a look at that. Lead is essentially the opposite of lag. Because it refers to the column that comes after the currently processed column.

```
/* Use lead to reference rows ahead */
select
  department,
  salary,
  lead(salary,5) over (partition by
department order by salary desc)
from
  staff
```



	department character varying (100)	salary integer	lead integer
1	Automotive	146167	135326
2	Automotive	144724	133612
3	Automotive	141505	130993
4	Automotive	140194	129324
5	Automotive	136448	128885
6	Automotive	135326	128448
7	Automotive	133612	127521
8	Automotive	130993	126485
9	Automotive	129324	126001
10	Automotive	128885	121300
11	Automotive	128448	120875
12	Automotive	127521	116487
13	Automotive	126485	115506
14	Automotive	126001	113231
15	Automotive	121300	111689
16	Automotive	120875	111041

So for example, in row one Sanchez has a salary of $146,167. The lead, the one ahead of it, is $144,724. And again, each row has a department, a last name of an employee, that employee's salary and now it has the leading, or next salary in the list. And we'll see that again, when we switch departments from Automotive to Baby. We'll notice that things are reset. For example, Howard has a salary of $148,687. The next row, or the leading row, has a salary of $148,573 and so on. And we'll notice that the last line of the department, in this case someone named Meyer, with a salary of $42,602, does not have a lead value because there's no other row following it.

NTILE functions:05_04 file

- [Narrator] Sometimes we want to group rows into some number of buckets or ordered groups. We can use **the ntiles function** to assign buckets to rows. This allows us to easily calculate statistics like core tiles over sets of rows. Here's an example that labels each salary with a core tile of one to four with **one being the group with the largest salaries**, and **four being the smallest set of salaries**. So let's select department, last name, salary. And now let's introduce the ntile function. **Ntile takes a number, and it will be the number of buckets that we want or ordered groups that we want.** So we're going to use core tiles or four groups so I'll specify four and I would like these ntiles over a group that is partitioned by department. And I'd like this ordered by salary, and we'd like that in descending order. Then we'll query this from the staff table.

```
/* Use ntiles to assign "buckets" to rows */
/* Include quartiles in list of salaries by department */
select
   department,
   salary,
   ntile(4) over (partition by department order by salary desc) as quartile
from
   staff;
```

data_sci/postgres@PostgreSQL 12

Data Output

	department character varying (100)	salary integer	quartile integer
11	Automotive	128448	1
12	Automotive	127521	1
13	Automotive	126485	2
14	Automotive	126001	2
15	Automotive	121300	2
16	Automotive	120875	2
17	Automotive	116487	2
18	Automotive	115506	2
19	Automotive	113231	2
20	Automotive	111689	2
21	Automotive	111041	2
22	Automotive	110589	2
23	Automotive	108378	2
24	Automotive	101768	2
25	Automotive	101006	3

Query Editor Query History Data Output Explain Mess

data_sci/postgres@PostgreSQL 12

Data Output

	department character varying (100)	salary integer	quartile integer
128	Beauty	83144	3
129	Beauty	79718	3
130	Beauty	79419	3
131	Beauty	76052	3
132	Beauty	74191	4
133	Beauty	72948	4
134	Beauty	72016	4
135	Beauty	71448	4
136	Beauty	69045	4
137	Beauty	66313	4
138	Beauty	63918	4
139	Beauty	55081	4
140	Beauty	50060	4
141	Beauty	48791	4
142	Beauty	47716	4
143	Beauty	41299	4

Now when we execute, what we'll notice is, we have a list of departments, each employee in that department and then there's salary and this employees are ordered with the highest salary first and then going down. They also have an integer value assigned to them. In this case, the top earners in the automotive department are assigned number one. Now, the second group is assigned the value of number two. So this is the second group based on the order of salary. And the third group is assigned number three. And then the fourth group has the lowest set of salaries and that's assigned number four. It is important to note, the number of rows in each group is the same, plus or minus one. If the total number of rows is not evenly divisible by the number of buckets, then some of the buckets will have one more row than others. Now also, as you notice, as we shift to a new department, we start the tiling again. In this case, in the baby department, Howard is in the first group as is Tucker and other high earners. And again, if you were to scroll down, you will notice that about a quarter of the way down, we shift to the second tile, or ntile of two. We move another quarter down, and we shift to the ntile of three. And finally, ntile of four. **Ntile is the window function we use when we want to group rows into some number of buckets or ordered groups.**

Data Output

	department character varying (100)	salary integer	quartile integer
36	Automotive	69936	8
37	Automotive	69594	8
38	Automotive	67800	8
39	Automotive	66063	9
40	Automotive	63364	9
41	Automotive	58555	9
42	Automotive	53964	9
43	Automotive	47281	10
44	Automotive	45774	10
45	Automotive	44377	10
46	Automotive	42602	10
47	Baby	148687	1
48	Baby	148573	1
49	Baby	141464	1
50	Baby	140004	1

Query Editor Query History Data Output Explain Mes

```
/* You can change to any number of ntiles or buckets */

select
  department,
  salary,
  ntile(10) over (partition by department order by salary desc) as quartile
from
  staff;
```

Tips for using SQL for data science

- [Instructor] The goal of data science is to tell stories using data. The stories we tell start with a business problem, like customers switching to competitors or products not selling as well as they use to. We need data to tell these stories. This data is often stored in multiple databases and file systems so our first task is to pull data from these different data sources and prepare them like a cook preparing ingredients for a dish. As with cooking, if you spend the time getting your data organized and properly prepared, the actual analysis is more enjoyable and goes more smoothly. Here are some tips as you work with SQL for data science. First, data collection and preparation will likely take more time than any other part of the data science project. Be sure to count all data sources when estimating and add extra time if you'll be joining data across different data sources. You may have inconsistent coding schemes or other data quality problems that will require additional time to correct. Aggregate and statistical functions in SQL can help you understand your data. For example, do you have attributes that fall into a bell curve? Or are there many different values and the curve has a long tail? Are there many outliers? Knowing the answers to these questions can help determine the kinds of analyses you try later on. Save yourself some time and do as much formatting and quality checks before attempting to join data from different sources. Use consistent coding schemes for things like states and country names. This helps avoid problems like not including all rows in a group count because some country codes are USA and others are US. Remember that an inner jointed SQL will return rows only when both tables have corresponding rows. If you want all the rows from one table even if there is not a corresponding row in the other table, then use an outer join. SQL statements get complicated especially when using joins, subqueries, and complicated grouping and filtering clauses. Use views to capture this logic so that you can query the view instead of having to repeatedly type in long SQL statements. It's easy to make a mistake when typing in such statements so use views that you can test and verify. And besides, views make it easy to share your SQL logic with others as well. Cubes, rollups, and grouping sets are useful when you need to produce cross-tabulations and subtotals. Now you could use multiple select statements to get the same results, but these operators can be more efficient. And finally, use window functions. They help us focus on sets of related rows such as all rows in a single department or company region. Window functions can help simplify select statements that would otherwise require subqueries.