[Skip to Content](https://learning.oreilly.com/library/view/sql-for-data/9781789807356/C11861_02_Commercial_Final_SZ_ePub.xhtml" \l "main)

**2. The Basics of SQL for Analytics**

**Learning Objectives**

By the end of this chapter, you will be able to:

* Describe the purpose of SQL
* Analyze how SQL can be used in an analytics workflow
* Apply the basics of a SQL database
* Perform operations to create, read, update, and delete a table

In this chapter, we will cover how SQL is used in data analytics. Then, we will learn the basics of SQL databases and perform CRUD (create, read, update, and delete) operations on a table.

**Introduction**

In *Chapter 1*, *Understanding and Describing Data*, we discussed analytics and how we can use data to obtain valuable information. While we could, in theory, analyze all data by hand, computers are far better at the task and are certainly the preferred tool for storing, organizing, and processing data. Among the most critical of these data tools is the relational database and the language used to access it, **Structured Query Language** (**SQL**). These two technologies have been cornerstones of data processing and continue to be the data backbone of most companies that deal with substantial amounts of data.

Companies use SQL as the primary method for storing much of their data. Furthermore, companies now take much of this data and put it into specialized databases called **data warehouses** and **data lakes** so that they can perform advanced analytics on their data. Virtually all of these data warehouses and data lakes are accessed using SQL. We'll be looking at working with SQL using analytics platforms such as data warehouses.

We assume that every person following this chapter has had some basic exposure to SQL. However, for those users who have very limited exposure to SQL, or who have not used it for some time, this chapter will provide a basic refresher of what relational databases and SQL are, along with a basic review of SQL operations and syntax. We will also go over a number of practice exercises to help reinforce these concepts.

**Relational Databases and SQL**

A **relational database** is a database that utilizes the **relational model** of data. The relational model, invented by Edgar F. Codd in 1970, organizes data as relations, or sets of tuples. Each tuple consists of a series of attributes, which generally describe the tuple. For example, we could imagine a customer relation, where each tuple represents a customer. Each tuple would then have attributes describing a single customer, giving information such as first name, last name, and age, perhaps in the format (John, Smith, 27). One or more of the attributes is used to uniquely identify a tuple in a relation and is called the **relational key**. The relational model then allows logical operations to be performed between relations.

In a relational database, relations are usually implemented as tables, as in an Excel spreadsheet. Each row of the table is a tuple, and the attributes are represented as columns of the table. While not technically required, most tables in a relational database have a column referred to as the **primary key**, which uniquely identifies a row of the database. Every column also has a **data type**, which describes the data for the column.

Tables are then usually assimilated in common collections in databases called **schemas**. These tables usually are loaded via processes known as **Extract, Transform, and Load** jobs (**ETL**).

**Note**

*Tables are usually referred to in queries in the format***[schema].[table]***. For example, a***product***table in the analytics schema would be generally referred to as***analytics.product***. However, there is also a special schema called***public***. This is a default schema where, if you do not explicitly mention a schema, the database uses the public schema, for example, the***public.products***table and***product***table are similar.*

The software used to manage relational databases on a computer is referred to as a **relational database management system** (**RDBMS**). SQL is the language utilized by users of an RDBMS to access and interact with a relational database.

**Note**

*Technically, virtually all relational databases that use SQL deviate from the relational model in some basic ways. For example, not every table has a specified relational key. Also, the relational model does not technically allow for duplicate rows, but you can have duplicate rows in a relational database. These differences are minor and will not matter for the vast majority of readers of this book. For more information on why most relational databases are not technically relational, refer to this article:*[*https://www.periscopedata.com/blog/your-database-isnt-really-relational*](https://www.periscopedata.com/blog/your-database-isnt-really-relational)*.*

**Advantages and Disadvantages of SQL Databases**

Since the release of Oracle Database in 1979, SQL has become an industry standard for data in nearly all computer applications – and for good reason. SQL databases provide a ton of advantages that make it the de facto choice for many applications:

* **Intuitive**: Relations represented as tables is a common data structure that almost everyone understands. As such, working with and reasoning about relational databases is much easier than doing so with other models.
* **Efficient**: Using a technique known as normalization, relational databases allow the representation of data without unnecessarily repeating it. As such, relational databases can represent large amounts of information while utilizing less space. This reduced storage footprint also allows the database to reduce operation costs, making well-designed relational databases quick to process.
* **Declarative**: SQL is a declarative language, meaning that when you write code, you only need to tell the computer what data you want, and the RDBMS takes care of determining how to execute the SQL code. You never have to worry about telling the computer how to access and pull data in the table.
* **Robust**: Most popular SQL databases have a property known as **atomicity, consistency, isolation, and durability** (**ACID**) compliance, which guarantees the validity of the data, even if the hardware fails.

That said, there are still some downsides to SQL databases, which are as follows:

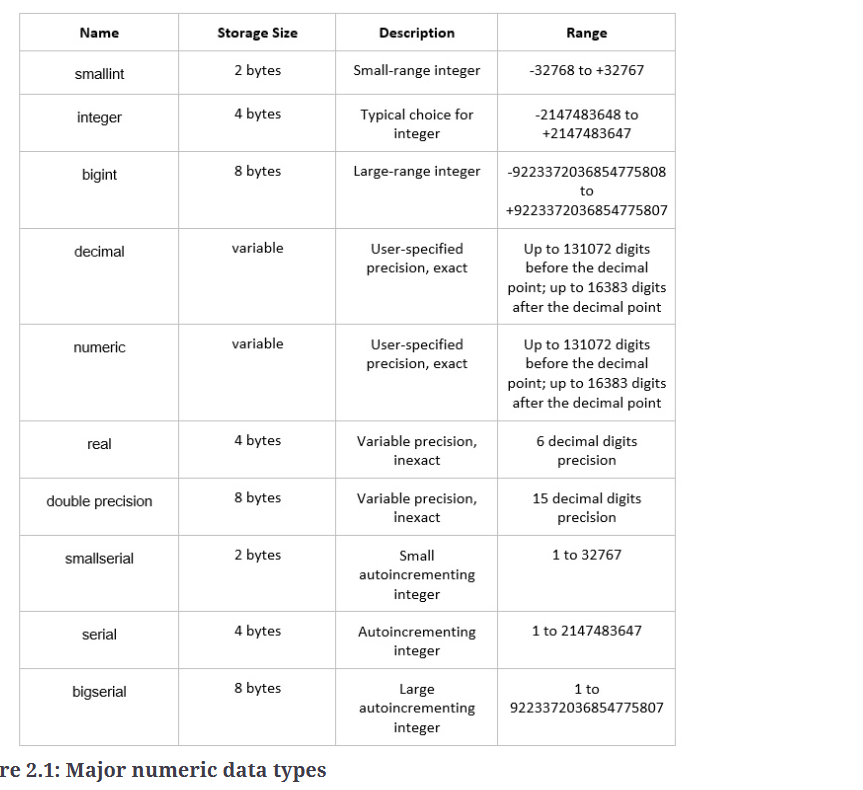
* **Lower specificity**: While SQL is declarative, its functionality can often be limited to what has already been programmed into it. Although most popular RDBMS software is updated constantly with new functionality being built all the time, it can be difficult to process and work with data structures and algorithms not programmed into an RDBMS.
* **Limited scalability**: SQL databases are incredibly robust, but this robustness comes at a cost. As the amount of information, you have doubles, the cost of resources more than doubles. When very large volumes of information are involved, other data stores, such as NoSQL databases, may actually be better.
* **Object-relation mismatch impedance**: While tables are a very intuitive data structure, they are not necessarily the best format for representing objects in a computer. This primarily occurs because objects often have attributes that have many-to-many relationships. For instance, a customer for a company may own multiple products, but each product may have multiple customers. For an object in a computer, we could easily represent this as a **list** attribute under the **customer** object. However, in a normalized database, a customer's products would potentially have to be represented using three different tables, each of which must be updated for every new purchase, recall, and return.

**Basic Data Types of SQL**

As previously mentioned, each column in a table has a data type. We review the major data types here.

**Numeric**

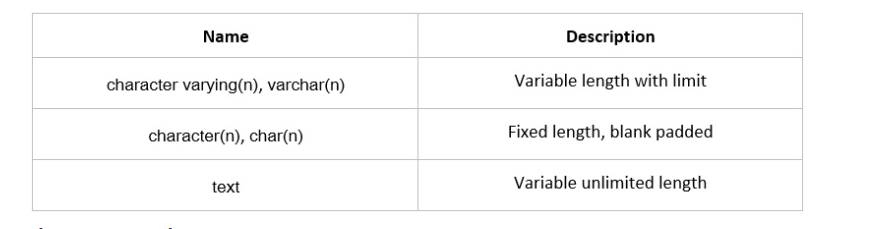
Numeric data types are data types that represent numbers. The following diagram provides an overview of some of the major types:



**Figure 2.1: Major numeric data types**

**Character**

**Character** data types store text information. The following diagram summarizes the character data types:

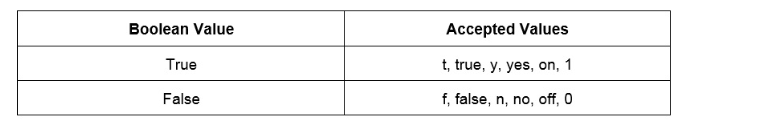


**Figure 2.2: Major character data types**

Under the hood, all of the character data types use the same underlying data structure in PostgreSQL and many other SQL databases, and most modern developers do not use **char(n)**.

**Boolean**

Booleans are a data type used to represent **True** or **False**. The following table summarizes values that are represented as a Boolean when used in a query with a Boolean data column type:

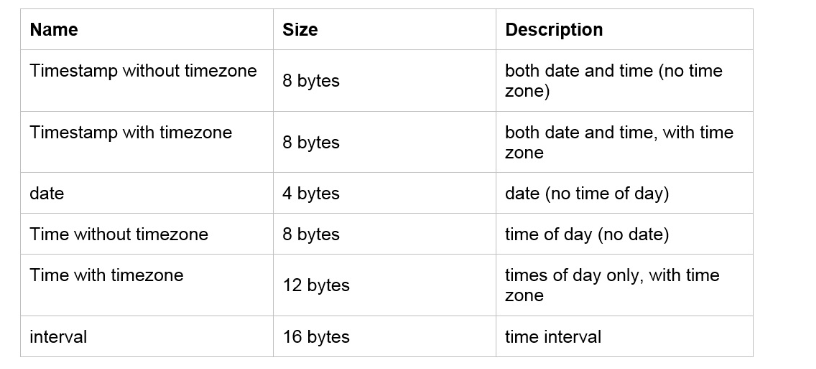


**Figure 2.3: Accepted Boolean values**

While all of these values are accepted, the values **True** and **False** are considered compliant with best practice. Booleans columns can also have **NULL** values.

**Datetime**

The **datetime** data type is used to store time-based information such as dates and times. The following are some of the datetime data types:



**Figure 2.4: Major datetime data types**

We will discuss this data type more in *Chapter 7*, *Analytics Using Complex Data Types*.

**Data Structures: JSON and Arrays**

Many versions of modern SQL also support data structures such as **JavaScript Object Notation** (**JSON**) and arrays. Arrays are simply listing of data usually written as members enclosed in square brackets. For example, ['cat', 'dog', 'horse'] is an array. A JSON object is a series of key-value pairs that are separated by commas and enclosed in curly braces. For example, {'name': 'Bob', 'age': 27, 'city': 'New York'} is a valid JSON object. These data structures show up consistently in technology applications and being able to use them in a database makes it easier to perform many kinds of analysis work.

We will discuss data structures more in *Chapter 7*, *Analytics Using Complex Data Types*.

We will now look at the basic operations in an RDBMS using SQL.

**Reading Tables: The SELECT Query**

The most common operation in a database is reading data from a database. This is almost exclusively done through the use of the **SELECT** keyword.

**Basic Anatomy and Working of a SELECT Query**

Generally speaking, a query can be broken down into five parts:

* **Operation**: The first part of a query describes what is going to be done. In this case, this is the word **SELECT**, followed by the names of columns combined with functions.
* **Data**: The next part of the query is the data, which is the **FROM** keyword followed by one or more tables connected together with reserved keywords indicating what data should be scanned for filtering, selection, and calculation.
* **Conditional**: A part of the query that filters the data to only rows that meet a condition usually indicated with **WHERE**.
* **Grouping**: A special clause that takes the rows of a data source, assembles them together using a key created by a **GROUP BY** clause, and then calculates a value using the values from all rows with the same value. We will discuss this step more in *Chapter 4*, *Aggregate Functions for Data Analysis*.
* **Post-processing**: A part of the query that takes the results of the data and formats them by sorting and limiting the data, often using keywords such as **ORDER BY** and **LIMIT**.

The steps of a **SELECT** query are as follows:

1. Create a data source by taking one or more tables and combining them in one large table.
2. Filter the table based on the large data source created in step 1 by seeing which rows meet the **WHERE** clause.
3. Calculate values based on columns in the data source in step 1. If there is a **GROUP BY** clause, divide the rows into groups and then calculate an aggregate statistic for each group. Otherwise, return a column or value calculated by performing functions on one or more columns together.
4. Take the rows returned and reorganize them based on the query.

To break down these steps, let's look at a typical query and follow the logic we've described:

SELECT first\_name

FROM customers

WHERE state='AZ'

ORDER BY first\_name

The operation of this query follows a sequence:

1. We start with the **customers** table.
2. The **customers** table is filtered to where the **state** column equals **'AZ'**.
3. We capture the **first\_name** column from the filtered table.
4. The **first\_name** column is then ordered in alphabetical order.

Here, we've shown how a query can be broken down into a series of steps for the database to process.

We will now look at the query keywords and patterns found in a **SELECT** query.

**Basic Keywords in a SELECT Query**

**SELECT and FROM**

The most basic **SELECT** query follows the pattern **SELECT…FROM <table\_name>;**. This query is the way to pull data from a single table. For example, if you want to pull all the data from the **products** table in our sample database, simply use this query:

SELECT \*

FROM products;

This query will pull all data from a database. The **\*** symbol seen here is shorthand to return all columns from a database. The semicolon operator (**;**) is used to tell the computer it has reached the end of the query, much like a period is used for a normal sentence. It's important to note that the rows will be returned in no specific order. If we want to return only specific columns from a query, we can simply replace the asterisk with the names of the columns we want to be separated in the order we want them to be returned in. For example, if we wanted to return the **product\_id** column followed by the **model** column of the **products** table, we would write the following query:

SELECT product\_id, model

FROM products;

If we wanted to return the **model** column first and the **product\_id** column second, we would write this:

SELECT model, product\_id

FROM products;

**WHERE**

The **WHERE** clause is a piece of conditional logic that limits the amount of data returned. All the rows returned in a **SELECT** statement with a **WHERE** clause in it meet the conditions of the **WHERE** clause. The **WHERE** clause can usually be found after the **FROM** clause of a single **SELECT** statement.

The condition in the **WHERE** clause is generally a Boolean statement that can either be **True** or **False** for every row. In the case of numeric columns, these Boolean statements can use equals, greater than, or less than operators to compare the columns against a value.

We will use an example to illustrate. Let's say we wanted to see the model names of our products with the model year of 2014 from our sample dataset. We would write the following query:

SELECT model

FROM products

WHERE year=2014;

**AND/OR**

The previous query had only one condition. We are often interested in multiple conditions being met at once. For this, we put multiple statements together using the **AND** or **OR** clause.

Now, we will illustrate this with an example. Let's say we wanted to return models that not only were built in 2014, but also have a **manufacturer's suggested retail price** (**MSRP**) of less than $1,000. We can write:

SELECT model

FROM products

WHERE year=2014

AND msrp<=1000;

Now, let's say we wanted to return models that were released in the year 2014 or had a product type of **automobile**. We would then write the following query:

SELECT model

FROM products

WHERE year=2014

OR product\_type='automobile';

When using more than one **AND**/**OR** condition, use parentheses to separate and position pieces of logic together. This will make sure that your query works as expected and that it is as readable as possible. For example, if we wanted to get all products with models in the years between 2014 and 2016, as well as any products that are scooters, we could write:

SELECT \*

FROM products

WHERE year>2014

AND year<2016

OR product\_type='scooter';

However, to clarify the **WHERE** clause, it would be preferable to write:

SELECT \*

FROM products

WHERE (year>2014 AND year<2016)

OR product\_type='scooter';

**IN/NOT IN**

As mentioned earlier, Boolean statements can use equals signs to indicate that a column must equal a certain value. However, what if you are interested in returning rows where a row has a column that can be equal to any of a group of values? For instance, let's say you were interested in returning all models with the year 2014, 2016, or 2019. You could write a query such as this:

SELECT model

FROM products

WHERE year = 2014

OR year = 2016

OR year = 2019;

However, this is long and tedious to write. Using **IN**, you can instead write:

SELECT model

FROM products

WHERE year IN (2014, 2016, 2019);

This is much cleaner to write and makes it easier to understand what is going on.

Conversely, you can also use the **NOT IN** clause to return all values that are not in a list of values. For instance, if you wanted all products that were not produced in the years 2014, 2016, and 2019, you could write:

SELECT model

FROM products

WHERE year NOT IN (2014, 2016, 2019);

**ORDER BY**

As previously mentioned, SQL queries will order rows as the database finds them if more specific instructions to do otherwise are not given. For many use cases, this is acceptable. However, you will often want to see rows in a specific order. Let's say you want to see all of the products listed by the date when they were first produced, from earliest to latest. The method for doing this in SQL would be as follows:

SELECT model

FROM products

ORDER BY production\_start\_date;

If an order sequence is not explicitly mentioned, the rows will be returned in ascending order. Ascending order simply means the rows will be ordered from the smallest value to the highest value of the chosen column or columns. In the case of things such as text, this means alphabetical order. You can make the ascending order explicit by using the **ASC** keyword. For our last query, this would be achieved by writing:

SELECT model

FROM products

ORDER BY production\_start\_date ASC;

If you would like to extract data in greatest-to-least order, you can use the **DESC** keyword. If we wanted to fetch manufactured models ordered from newest to oldest, we would write:

SELECT model

FROM products

ORDER BY production\_start\_date DESC;

Also, instead of writing the name of the column you want to order by, you can instead refer to what number column it is in the natural order of the table. For instance, say you wanted to return all the models in the **products** table ordered by product ID. You could write:

SELECT model

FROM products

ORDER BY product\_id;

However, because **product\_id** is the first column in the table, you could instead write:

SELECT model

FROM products

ORDER BY 1;

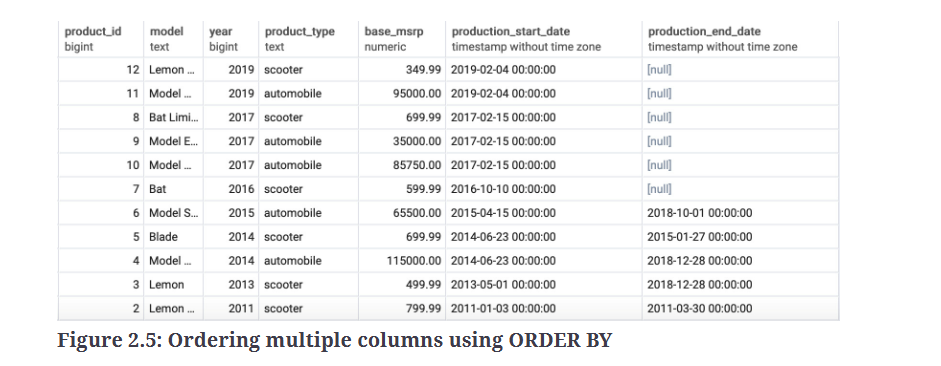
Finally, you can order by multiple columns by adding additional columns after **ORDER BY** separated with commas. For instance, let's say we wanted to order all of the rows in the table first by the year of the model, from newest to oldest, and then by the MSRP from least to greatest. We would then write:

SELECT \*

FROM products

ORDER BY year DESC, base\_msrp ASC;

The following is the output of the preceding code:



**Figure 2.5: Ordering multiple columns using ORDER BY**

**LIMIT**

Most tables in SQL databases tend to be quite large, and therefore returning every single row is unnecessary. Sometimes, you may want only the first few rows. For this scenario, the **LIMIT** keyword comes in handy. Let's imagine that you wanted to only get the first five products that were produced by the company. You could get this by using the following query:

SELECT model

FROM products

ORDER BY production\_start\_date

LIMIT 5;

As a general rule, you probably want to use a **LIMIT** keyword for a table or query you have not worked with.

**IS NULL/IS NOT NULL**

Often, some entries in a given column may be missing. This could be for a variety of reasons. Perhaps the data was not collected or not available at the time that the data was collected. Perhaps the ETL job failed to collect and load data into a column. It may also be possible that the absence of a value is representative of a certain state in the row and actually provides valuable information. Whatever the reason, we are often interested in finding rows where the data is not filled in for a certain value. In SQL, blank values are often represented by the **NULL** value. For instance, in the **products** table, the **production\_end\_date** column having a **NULL** value indicates that the product is still being made. In this case, if we want to list all products that are still being made, we can use the following query:

SELECT \*

FROM products

WHERE production\_end\_date IS NULL

If we are only interested in products that are not being produced, we can use the **IS NOT NULL** clause, as in the following query:

SELECT \*

FROM products

WHERE production\_end\_date IS NOT NULL

**Exercise 6: Querying the Salespeople Table Using Basic Keywords in a SELECT Query**

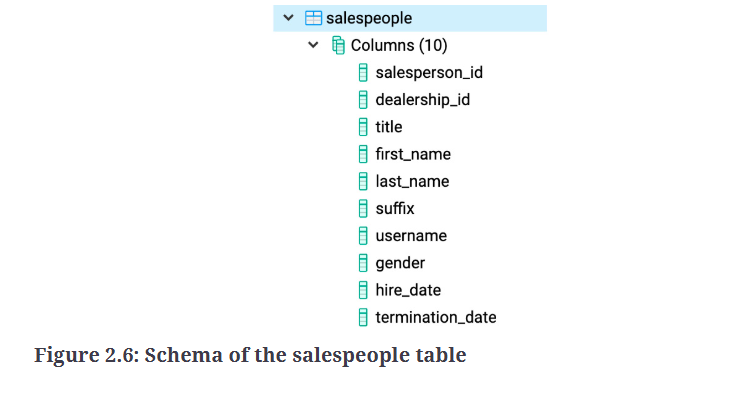
In this exercise, we will create various queries using basic keywords in a **SELECT** query. Let's say that, after a few days at your new job, you finally get access to the company database. Today, your boss has asked you to help a sales manager who does not know SQL particularly well. The sales manager would like a couple of different lists of salespeople. First, create a list of the online usernames of the first 10 female salespeople hired, ordered from the first hired to the latest hired.

**Note**

*For all exercises in this book, we will be using pgAdmin 4. Codes for all the exercises and activities can also be found on GitHub:*[*https://github.com/TrainingByPackt/SQL-for-Data-Analytics/tree/master/Lesson02*](https://github.com/TrainingByPackt/SQL-for-Data-Analytics/tree/master/Lesson02)*.*

Perform the following steps to complete the exercise:

1. Open your favorite SQL client and connect to the **sqlda** database.
2. Examine the schema for the **salespeople** table from the schema dropdown. Notice the names of the columns in the following diagram:



**Figure 2.6: Schema of the salespeople table**

1. Execute the following query to get the usernames of female salespeople sorted by their **hire\_date** values and set **LIMIT** as **10**:

SELECT username

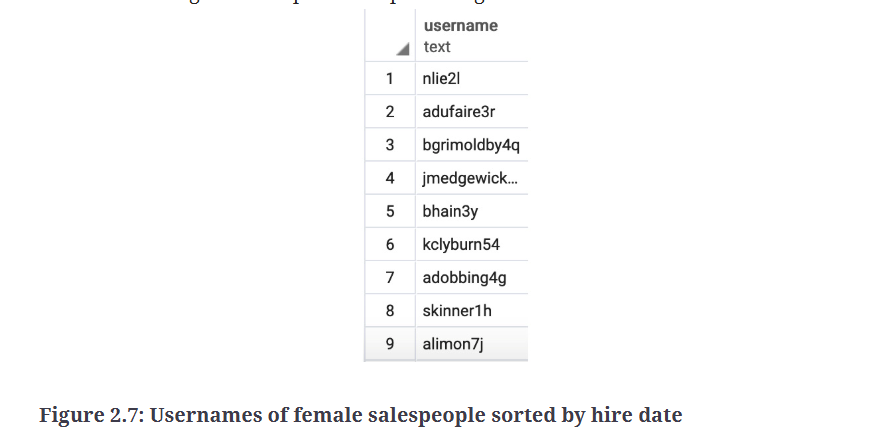
FROM salespeople

WHERE gender= 'Female'

ORDER BY hire\_date

LIMIT 10

The following is the output of the preceding code:



**Figure 2.7: Usernames of female salespeople sorted by hire date**

We now have a list of usernames for female salespeople ordered from the earliest hire to the most recent hire.

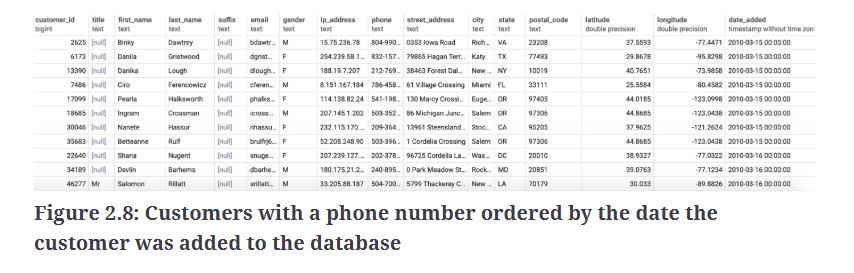
In this exercise, we used different basic keywords in a **SELECT** query to help the sales manager to get the list of salespeople as per their requirements.

**Activity 3: Querying the customers Table Using Basic Keywords in a SELECT Query**

One day, your manager at ZoomZoom calls you in and tells you that the marketing department has decided that they want to do a series of marketing campaigns to help promote a sale. You will need to send queries to the manager to pull the data. The following are the steps to complete the activity:

1. Open your favorite SQL client and connect to the **sqlda** database. Examine the schema for the **customers** table from the schema dropdown.
2. Write a query that pulls all emails for ZoomZoom customers in the state of Florida in alphabetical order.
3. Write a query that pulls all the first names, last names and email details for ZoomZoom customers in New York City in the state of New York. They should be ordered alphabetically by the last name followed by the first name.
4. Write a query that returns all customers with a phone number ordered by the date the customer was added to the database.

**Expected Output:**



**Figure 2.8: Customers with a phone number ordered by the date the customer was added to the database**

**Note**

*The solution for this activity can be found via*[*this link*](https://learning.oreilly.com/library/view/sql-for-data/9781789807356/C11861_Solution_Commercial_Final_SZ_ePub.xhtml#_idTextAnchor244)*.*

In this activity, we used various basic keywords in a **SELECT** query and helped the marketing manager to get the data they needed.

**Creating Tables**

Now that we know how to read data from tables, we will now look at how to create new tables. There are fundamentally two ways to create tables: creating blank tables or using **SELECT** queries.

**Creating Blank Tables**

To create a new blank table, we use the **CREATE TABLE** statement. This statement takes the following structure:

CREATE TABLE {table\_name} (

{column\_name\_1} {data\_type\_1} {column\_constraint\_1},

{column\_name\_2} {data\_type\_2} {column\_constraint\_2},

{column\_name\_3} {data\_type\_3} {column\_constraint\_3},

…

{column\_name\_last} {data\_type\_last} {column\_constraint\_last},

);

Here **{table\_name}** is the name of the table, **{column\_name}** is the name of the column, **{data\_type}** is the data type of the column, and **{column\_constraint}** is one or more optional keywords giving special properties to the column. Before we discuss how to use the **CREATE TABLE** query, we will first discuss column constraints.

**Column Constraints**

Column constraints are keywords that give special properties to a column. Some major column constraints are:

* **NOT NULL**: This constraint guarantees that no value in a column can be null.
* **UNIQUE**: This constraint guarantees that every single row for a column has a unique value and that no value is repeated.
* **PRIMARY KEY**: This is a special constraint that is unique for each row and helps to find the row quicker. Only one column in a table can be a primary key.

Suppose we want to create a table called **state\_populations**, and it has columns with states' initials and populations. The query would look like this:

CREATE TABLE state\_populations (^state VARCHAR(2) PRIMARY KEY,

population NUMERIC

);

**Note**

*Sometimes, you may run a***CREATE TABLE***query and get the error "relation***{table\_name}***already exists". This simply means that a table with the same name already exists. You will either have to delete the table with the same name or change the name of your table.*

We will now discuss the next way to create a table, which is by using a SQL query.

**Exercise 7: Creating a Table in SQL**

In this exercise, we will create a table using the **CREATE TABLE** statement. The marketing team at ZoomZoom would like to create a table called **countries** to analyze the data of different countries. It should have four columns: an integer key column, a unique name column, a founding year column, and a capital column.

Perform the following steps to complete the exercise:

1. Open your favorite SQL client and connect to the **sqlda** database.
2. Run the following query to create the **countries** table:

CREATE TABLE countries (

key INT PRIMARY KEY,

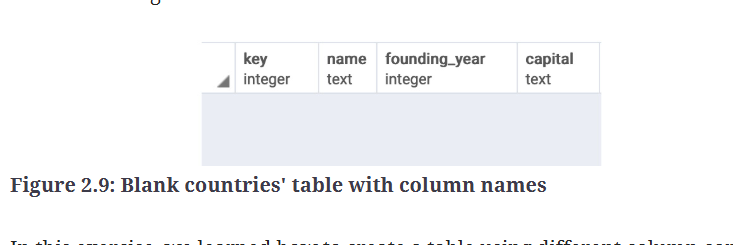
name text UNIQUE,

founding\_year INT,

capital text

);

You should get a blank table as follows:



**Figure 2.9: Blank countries' table with column names**

In this exercise, we learned how to create a table using different column constraints and the **CREATE TABLE** statement.

**Creating Tables with SELECT**

We know how to create a table. However, say you wanted to create a table using data from an existing table. This can be done using a modification of the **CREATE TABLE** statement:

CREATE TABLE {table\_name} AS (

{select\_query}

);

Here, **{select\_query}** is any **SELECT** query that can be run in your database. For instance, say you wanted to create a table based on the **products** table that only had products from the year 2014. Let's call this table **products\_2014**. You could then write the following query:

CREATE TABLE products\_2014 AS (

SELECT \*

FROM products

WHERE year=2014

);

This can be done with any **SELECT** query, and the table will inherit all the properties of the output query.

**Updating Tables**

Over time, you may also need to modify a table by adding columns, adding new data, or updating existing rows. We will discuss how to do that in this section.

**Adding and Removing Columns**

To add new columns to an existing table, we use the **ADD COLUMN** statement as in the following query:

ALTER TABLE {table\_name}

ADD COLUMN {column\_name} {data\_type};

Let's say, for example, that we wanted to add a new column to the **products** table that we will use to store the products' weight in kilograms called **weight**. We could do this by using the following query:

ALTER TABLE products

ADD COLUMN weight INT;

This query will make a new column called **weight** in the **products** table and will give it the integer data type so that only numbers can be stored within it.

If you want to remove a column from a table, you can use the **DROP** column statement:

ALTER TABLE {table\_name}

DROP COLUMN {column\_name};

Here, **{table\_name}** is the name of the table you want to change, and **{column\_name}** is the name of the column you want to drop.

Let's imagine that you decide to delete the **weight** column you just created. You could get rid of it using the following query:

ALTER TABLE products

DROP COLUMN weight;

**Adding New Data**

You can add new data in a table using several methods in SQL.

One method is to simply insert values straight into a table using the **INSERT INTO…VALUES** statement. It has the following structure:

INSERT INTO {table\_name} ({column\_1], {column\_2}, …{column\_last})

VALUES ({column\_value\_1}, {column\_value\_2}, … {column\_value\_last});

Here, **{table\_name}** is the name of the table you want to insert your data into, **{column\_1}, {column\_2}, … {column\_last}** is a list of the columns whose values you want to insert, and **{column\_value\_1}, {column\_value\_2}, … {column\_value\_last}** is the values of the rows you want to insert into the table. If a column in the table is not put into the **INSERT** statement, the column is assumed to have a **NULL** value.

As an example, let's say you wanted to insert a new scooter into the **products** table. This could be done with the following query:

INSERT INTO products (product\_id, model, year, product\_type, base\_msrp, production\_start\_date, production\_end\_date)

VALUES (13, "Nimbus 5000", 2019, 'scooter', 500.00, '2019-03-03', '2020-03-03');

Another way to insert data into a table is to use the **INSERT** statement with a **SELECT** query using the following syntax:

INSERT INTO {table\_name} ({column\_1], {column\_2}, …{column\_last})

{select\_query};

Here, **{table\_name}** is the name of the table into which you want to insert the data, **{column\_1}, {column\_2}, … {column\_last}** is a list of the columns whose values you want to insert, and **{select query}** is a query with the same structure as the values you want to insert into the table.

Take the example of the **products\_2014** table we discussed earlier. Imagine that instead of creating it with a **SELECT** query, we created it as a blank table with the same structure as the **products** table. If we wanted to insert the same data as we did earlier, we could use the following query:

INSERT INTO products (product\_id, model, year, product\_type, base\_msrp, production\_start\_date, production\_end\_date)

SELECT \*

FROM products

WHERE year=2014;

**Updating Existing Rows**

Sometimes, you may need to update the values of the data present in a table. To do this, you can use the **UPDATE** statement:

UPDATE {table\_name}

SET {column\_1} = {column\_value\_1},

    {column\_2} = {column\_value\_2},

    ...

    {column\_last} = {{column\_value\_last}}

WHERE

{conditional};

Here, **{table\_name}** is the name of the table with data that will be changed, **{column\_1}, {column\_2},… {column\_last}** is the columns whose values you want to change, **{column\_value\_1}, {column\_value\_2},… {column\_value\_last}** is the new values you want to insert into those columns, and **{WHERE}** is a conditional statement like one you would find in a SQL query.

To illustrate its use of the update statement, let's say that for the rest of the year, the company has decided to sell all scooter models before 2018 for $299.99. We could change the data in the **products** table using the following query:

UPDATE products

SET base\_msrp = 299.99,

WHERE

product\_type = 'scooter'

AND year<2018;

**Exercise 8: Updating Tables in SQL**

Our goal in this exercise is to update the data in a table using the **UPDATE** statement. Due to the higher cost of rare metals needed to manufacture an electric vehicle, the new 2019 Model Chi will need to undergo a price hike of 10%. Update the **products** table to increase the price of this product.

Perform the following steps to complete the exercise:

1. Open your favorite SQL client and connect to the **sqlda** database.
2. Run the following query to update the price of **Model Chi** in the **products** table:

UPDATE products

SET base\_msrp = base\_msrp\*1.10

WHERE model='Model Chi'

and year=2019;

1. Now, write the **SELECT** query to check whether the price of **Model Chi** in **2019** has been updated:

SELECT \*

FROM products

WHERE model='Model Chi'

AND year=2019;

The following is the output of the preceding code:



**Figure 2.10: The updated price of Model Chi in 2019**

As seen in the output, the price of **Model Chi** is now **104,500**, which was previously $95,000.

In this exercise, we learned how to update a table using the **UPDATE** statement.

We will now discuss how to delete tables and data from tables.

**Deleting Data and Tables**

We often discover that data in a table is incorrect, and therefore can no longer be used. At such times, we need to delete data from a table.

**Deleting Values from a Row**

Often, we will be interested in deleting a value in a row. The easiest way to accomplish this task is to use the **UPDATE** structure we already discussed and to set the column value to **NULL** like so:

UPDATE {table\_name}

SET {column\_1} = NULL,

    {column\_2} = NULL,

    ...

    {column\_last} = NULL

WHERE

{conditional};

Here, **{table\_name}** is the name of the table with the data that needs to be changed, **{column\_1}, {column\_2},… {column\_last}** is the columns whose values you want to delete, and **{WHERE}** is a conditional statement like one you would find in a SQL query.

Let's say, for instance, that we have the wrong email on file for the customer with the customer ID equal to **3**. To fix that, we can use the following query:

UPDATE customers

SET email = NULL

WHERE customer\_id=3;

**Deleting Rows from a Table**

Deleting a row from a table can be done using the **DELETE** statement, which looks like this:

DELETE FROM {table\_name}

WHERE {conditional};

DELETE FROM customers

WHERE email='bjordan2@geocities.com';

If we wanted to delete all the data in the **customers** table without deleting the table, we could write the following query:

DELETE FROM customers;

Alternatively, if you want to delete all the data in a query without deleting the table, you could use the **TRUNCATE** keyword as follows:

TRUNCATE TABLE customers;

**Deleting Tables**

To delete the table along with the data completely, you can just use the **DROP TABLE** statement with the following syntax:

DROP TABLE {table\_name};

Here, **{table\_name}** is the name of the table you want to delete. If we wanted to delete all the data in the **customers** table along with the table itself, we would write:

DROP TABLE customers;

**Exercise 9: Unnecessary Reference Table**

The marketing team has finished analyzing the potential number of customers they have in every state, and no longer need the **state\_populations** table. To save space on the database, delete the table.

Perform the following steps to complete the exercise:

1. Open your favorite SQL client and connect to the **sqlda** database.
2. Run the following query to drop the **state\_populations** table:

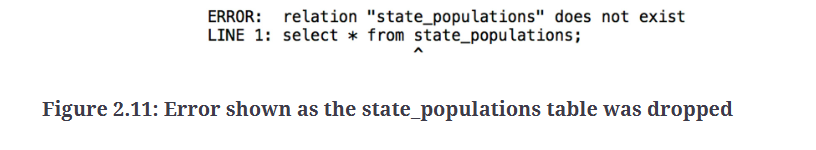
DROP TABLE state\_populations;

The **state\_populations** table should now be deleted from the database.

1. Since the table has just been dropped, a **SELECT** query on this table throws an error, as expected:

SELECT \* FROM state\_populations;

You will find the error shown in the following diagram:



**Figure 2.11: Error shown as the state\_populations table was dropped**

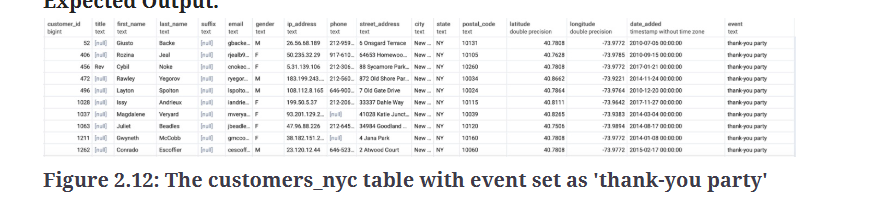
In this exercise, we learned how to delete a table using the **DROP TABLE** statement.

**Activity 4: Marketing Operations**

You did a great job pulling data for the marketing team. However, the marketing manager, who you so graciously helped, realized that they had made a mistake. It turns out that instead of just the query, the manager needs to create a new table in the company's analytics database. Furthermore, they need to make some changes to the data that is present in the **customers** table. It is your job to help the marketing manager with the table:

1. Create a new table called **customers\_nyc** that pulls all rows from the **customers** table where the customer lives in New York City in the state of New York.
2. Delete from the new table all customers in postal code **10014**. Due to local laws, they will not be eligible for marketing.
3. Add a new text column called **event**.
4. Set the value of the event to **thank-you party**.

**Expected Output:**



**Figure 2.12: The customers\_nyc table with event set as 'thank-you party'**

1. You've told the manager that you've completed these steps. He tells the marketing operations team, who then uses the data to launch a marketing campaign. The marketing manager thanks you and then asks you to delete the **customers\_nyc** table.

**Note**

*The solution for this activity can be found via*[*this link*](https://learning.oreilly.com/library/view/sql-for-data/9781789807356/C11861_Solution_Commercial_Final_SZ_ePub.xhtml#_idTextAnchor245)*.*

In this activity, we used different CRUD operations to modify a table as requested by the marketing manager. We will now come full circle to talk about how SQL and analytics connect.

**SQL and Analytics**

In this chapter, we went through the basics of SQL, tables, and queries. You may be wondering, then, what SQL has to do with analytics. You may have seen some parallels between the first two chapters. When we talk about a SQL table, it should be clear that it can be thought of as a dataset. Rows can be considered individual units of observation and columns can be considered features. If we view SQL tables in this way, we can see that SQL is a natural way to store datasets in a computer.

However, SQL can go further than just providing a convenient way to store datasets. Modern SQL implementations also provide tools for processing and analyzing data through various functions. Using SQL, we can clean data, transform data to more useful formats, and analyze data with statistics to find interesting patterns. The rest of this book will be dedicated to understanding how SQL can be used for these purposes productively and efficiently.

**Summary**

Relational databases are a mature and ubiquitous technology that is used to store and query data. Relational databases store data in the form of relations, also known as tables, which allow for an excellent combination of performance, efficiency, and ease of use. SQL is the language used to access relational databases. SQL is a declarative language that allows users to focus on what to create, as opposed to how to create it. SQL supports many different data types, including numeric data, text data, and even data structures.

When querying data, SQL allows a user to pick which fields to pull, as well as how to filter the data. This data can also be ordered, and SQL allows for as much or as little data as we need to be pulled. Creating, updating, and deleting data is also fairly simple and can be quite surgical.

Now that we have reviewed the basics of SQL, we will discuss how SQL can be used to perform the first step in data analytics, cleaning, and the transformation of data, in the next chapter.

* **Explain this concept**
* Highlight
* Add Note
* Copy

table of contentssearch

Settings

**Previous chapter**

[**1. Understanding and Describing Data**](https://learning.oreilly.com/library/view/sql-for-data/9781789807356/C11861_01_Commercial_Final_SZ_ePub.xhtml)

**Next chapter**

[**3. SQL for Data Preparation**](https://learning.oreilly.com/library/view/sql-for-data/9781789807356/C11861_03_Commercial_Final_SW_ePub.xhtml)