SQL and Databases

- Welcome to week 4; SQL and Databases
- Introduction to databases
- SQL filtering versus Linux filtering
- Adedayo: SQL in cybersecurity
- Basic queries
- Query a database
- find table name and comumns definintion for SQL and variences
- Basic filters on SQL queries
- The WHERE clause and basic operators
- Filter dates and numbers
- Operators for filtering dates and numbers
- Filters with AND, OR, and NOT
- More on filters with AND, OR, and NOT
- Join tables in SQL
- Types of joins
- Compare types of joins
- Continuous learning in SQL
- Wrap-up; Glossary terms from week 4

Welcome to week 4; SQL and Databases

In the world of security, diversity is important.
Diverse perspectives are often needed to find effective solutions.
This is also true of the tools we use.
Your job will often require you to use a lot of diverse tools.
In the last section, we studied the Linux command line and learned how this tool can help you search and filter through data, navigate through the Linux file system, and authenticate users.
Now, we'll learn about another tool.

In this section, we'll explore SQL and how it allows you to analyze data in a way needed for your role as a security analyst.
We're going to start off by learning about relational databases and how they're structured.
From there, we're going to introduce SQL queries and how to use them to access data from databases.
We then move on to SQL filters, which help us refine our queries to get the exact information we need.
Lastly, we'll explore SQL joins, which allow you to combine tables together.

When I'm presented with a problem or a project at work, I often have to sift through a large amount of data.
When I use SQL, I'm able to review data quickly and provide results with confidence since the queries are
consistent and easily executed.

SQL is a very powerful and flexible tool.
Throughout this section, you'll learn how to use the parts of it you need as a security analyst and gain hands-on experience.
Good luck, and I'll join you for the rest of the course!

Introduction to databases

Our modern world is filled with data and that data almost always guides us in making important decisions.
When working with large amounts of data, we need to know how to store it, so it's organized and quick to access and process.
The solution to this is through databases, and that's what we're exploring in this video!

To start us off, we can define a database as an organized collection of information or data.
Databases are often compared to spreadsheets.
Some of you may have used Google Sheets or another common spreadsheet program in the past.
While these programs are convenient ways to store data, spreadsheets are often designed for a single user or a small team to store less data.
In contrast, databases can be accessed by multiple people simultaneously and can store massive amounts of data.
Databases can also perform complex tasks while accessing data.
As a security analyst, you'll often need to access databases containing useful information.
For example, these could be databases containing information on login attempts, software and updates, or machines and their owners.

Now that we know how important databases are for us, let's talk about how they're organized and how we can interact with them.
Using databases allow us to store large amounts of data while keeping it quick and easy to access.
There are lots of different ways we can structure a database, but in this course, we'll be working with relational databases.
A relational database is a structured database containing tables that are related to each other.

Let's learn more about what makes a relational database.
We'll start by examining an individual table in a larger database of organizational information.
Each table contains fields of information.
For example, in this table on employees, these would include fields like employee_id, device_id, and username.

These are the columns of the tables.
In addition, tables contain rows also called records.
Rows are filled with specific data related to the columns in the table.
For example, our first row is a record for an employee whose id is 1,000 and
who works in the marketing department.

Relational databases often have multiple tables.
Consider an example where we have two tables from a larger database, one with employees of the company and another with machines given to those employees.
We can connect two tables if they share a common column.
In this example, we establish a relationship between them with a common employee_id column.

The columns that relate two tables to each other are called keys.
There are two types of keys.
The first is called a primary key.
The primary key refers to a column where every row has a unique entry.
The primary key must not have any duplicate values, or any null or empty values.
The primary key allows us to uniquely identify every row in our table.
For the table of employees, employee_id is a primary key.
Every employee_id is unique and there are no employee_ids that are duplicate or empty.

The second type of key is a foreign key.
The foreign key is a column in a table that is a primary key in another table.
Foreign keys, unlike primary keys, can have empty values and duplicates.
The foreign key allows us to connect two tables together.
In our example, we can look at the employee_id column in the machines table.
We previously identified this as a primary key in the employees table, so we can use this to connect every machine to their corresponding employee.

It's also important to know that a table can only have one primary key, but multiple foreign keys.

With this information, we're ready to move on to the basics of SQL, the language that lets us work with databases.
Throughout this section, we'll gain hands-on experience working with the concepts we just covered!

SQL filtering versus Linux filtering

Previously, you explored the Linux commands that allow you to filter for specific information contained within files or directories. And, more recently, you examined how SQL helps you efficiently filter for the information you need. In this reading, you'll explore differences between the two tools as they relate to filtering. You'll also learn that one way to access SQL is through the Linux command line.

Accessing SQL

There are many interfaces for accessing SQL and many different versions of SQL. One way to access SQL is through the Linux command line.

To access SQL from Linux, you need to type in a command for the version of SQL that you want to use. For example, if you want to access SQLite, you can enter the command *sqlite3* in the command line.

After this, any commands typed in the command line will be directed to SQL instead of Linux commands.

Differences between Linux and SQL filtering

Although both Linux and SQL allow you to filter through data, there are some differences that affect which one you should choose.

Structure

SQL offers a lot more structure than Linux, which is more free-form and not as tidy.

For example, if you wanted to access a log of employee log-in attempts, SQL would have each record separated into columns. Linux would print the data as a line of text without this organization. As a result, selecting a specific column to analyze would be easier and more efficient in SQL.

In terms of structure, SQL provides results that are more easily readable and that can be adjusted more quickly than when using Linux.

Joining tables

Some security-related decisions require information from different tables. SQL allows the analyst to join multiple tables together when returning data. Linux doesn't have that same functionality; it doesn't allow data to be connected to other information on your computer. This is more restrictive for an analyst going through security logs.

Best uses

As a security analyst, it's important to understand when you can use which tool. Although SQL has a more organized structure and allows you to join tables, this doesn't mean that there aren't situations that would require you to filter data in Linux.

A lot of data used in cybersecurity will be stored in a database format that works with SQL. However, other logs might be in a format that is not compatible with SQL. For instance, if the data is stored in a text file, you cannot search through it with SQL. In those cases, it is useful to know how to filter in Linux.

Key takeaways

To work with SQL, you can access it from multiple different interfaces, such as the Linux command line. Both SQL and Linux allow you to filter for specific data, but SQL offers the advantages of structuring the data and allowing you to join data from multiple tables.

Adedayo: SQL in cybersecurity

My name is Adedayo, and I'm a Security Engineer at Google.
A lot of people think you need to have a degree in computer science, right to be able to get into cybersecurity. I don't think that's true.
Take me for an example, I started learning IT from Lagos, Nigeria where I was born and raised, and then I'm all the way here now
in Silicon Valley, working for Google.
I think that's just amazing and a dream come true.
You taking this certificate is a first step to you making a commitment to switching your career to cybersecurity. Kudos to you on that.
SQL is one of the skillset you need to have in your toolbox as a cybersecurity professional because you can very quickly make decisions, not just off the bat, but make decisions with data backing you, and be able to communicate with your team, with stakeholders about why you made a decision because it's one thing to be able to say, we need to do this, it's
another thing to say we need to do this and here's the data that I wrote my SQL statements about.
I learned SQL by, first, as a coursework in school, that was really great, but I think I forgot everything about that after school.
The next step that I took was taking online courses, such as the one you're taking right now to learn SQL and the fundamentals
about it and how to really use it.
Then the first time I used SQL practically was at Google.
You really need to practice.
I think with anything else, practice makes perfect.
Being able to, even if it's just a few hours a week, put aside time to practice writing SQL statement.
Having that skill is something that will be very applicable to your first job, and you can use that to make data-driven decisions.
I feel very fulfilled working in cybersecurity.
I feel very energized, come into work every day.
Not only because I get to work on really complex problems and try to figure out solutions for them, but I also have great teammates that we all come together and tackle the problem.
Being able to go to bed at night, knowing that the work that I do is for the better of Google users and Google employees
is a very rewarding feeling for me.

Basic queries

In this video, we're going to be running our very first SQL query!
This query will be based on a common work task that you might encounter as a security analyst.
We're going to determine which computer has been assigned to a certain employee.
Let's say we have access to the employees table.
The employees table has five columns.
Two of them, employee_id and device_id, contain the information that we need.
We'll write a query to this table that returns only those two columns from the table.

The two SQL keywords we need for basic SQL queries are SELECT and FROM.
SELECT indicates which columns to return.
FROM indicates which table to query.
The use of these keywords in SQL is very similar to how we would use these words in everyday language.
For example, we can ask a friend to select apples and bananas from the big box when going out to buy fruit.
This is already very similar to SQL.

So let's go ahead and use SELECT and FROM in SQL to return the information we need on employees and the computers they use.
We start off by typing in the SQL statement.
After FROM, we've identified that the information will be pulled from the employees table.
And after SELECT, employee_id and device_id indicate the two columns we want to return from this table.
Notice how a comma separates the two columns that we want to return.

It's also worth mentioning a couple of key aspects related to the syntax of SQL here.
Syntax refers to the rules that determine what is correctly structured in a computing language.
In SQL, keywords are not case-sensitive, so you could also write select and from in lowercase, but we're placing them in capital letters because it makes the query easier to understand.
Another aspect of this syntax is that semicolons are placed at the end of the statement.

And now, we'll run the query by pressing Enter.
The output gives us the information we need to match employees to their computers.
We just ran our very first SQL query!

Suppose you wanted to know what department the employee using the computer is from, or their username, or the office they work in.
To do that, we can use SQL to make another statement that prints out all of the columns from the table.
We can do this by placing an asterisk after SELECT.
This is commonly referred to as select all.
Now, let's run this query to the employees table in SQL.

And now we have the full table in the output.

You just made it through a basic query in SQL, congratulations!
In the next video, we'll learn how to add filters to our queries, so I'll meet you there!

Query a database

Previously, you explored how SQL is an important tool in the world of cybersecurity and is essential when querying databases. You examined a few basic SQL queries and keywords used to extract needed information from a database. In this reading, you'll review those basic SQL queries and learn a new keyword that will help you organize your output. You'll also learn about the *Chinook* database, which this course uses for queries in readings and quizzes.

Basic SQL query

There are two essential keywords in any SQL query: *SELECT* and *FROM*. You will use these keywords every time you want to query a SQL database. Using them together helps SQL identify what data you need from a database and the table you are returning it from.

The video demonstrated this SQL query:

SELECT employee_id, device_id

FROM employees;

In readings and quizzes, this course uses a sample database called the *Chinook* database to run queries. The *Chinook* database includes data that might be created at a digital media company. A security analyst employed by this company might need to query this data. For example, the database contains eleven tables, including an *employees* table, a *customers* table, and an *invoices* table. These tables include data such as names and addresses.

As an example, you can run this query to return data from the *customers* table of the *Chinook* database:

SELECT customerid, city, country
FROM customers;

```
+------------+--------------------+----------------+
| CustomerId | City               | Country        |
+------------+--------------------+----------------+
|          1 | São José dos Campos | Brazil         |
|          2 | Stuttgart          | Germany        |
|          3 | Montréal           | Canada         |
|          4 | Oslo               | Norway         |
|          5 | Prague             | Czech Republic |
|          6 | Prague             | Czech Republic |
|          7 | Vienne             | Austria        |
|          8 | Brussels           | Belgium        |
```

```
|          9 | Copenhagen          | Denmark        |
|         10 | São Paulo           | Brazil         |
|         11 | São Paulo           | Brazil         |
|         12 | Rio de Janeiro      | Brazil         |
|         13 | Brasília            | Brazil         |
|         14 | Edmonton            | Canada         |
|         15 | Vancouver           | Canada         |
|         16 | Mountain View       | USA            |
|         17 | Redmond             | USA            |
|         18 | New York            | USA            |
|         19 | Cupertino           | USA            |
|         20 | Mountain View       | USA            |
|         21 | Reno                | USA            |
|         22 | Orlando             | USA            |
|         23 | Boston              | USA            |
|         24 | Chicago             | USA            |
|         25 | Madison             | USA            |
+------------+---------------------+----------------+
(Output limit exceeded, 25 of 59 total rows shown)
```

The *SELECT* keyword indicates which columns to return. For example, you can return the *customerid* column from the *Chinook* database with

SELECT customerid

You can also select multiple columns by separating them with a comma. For example, if you want to return both the *customerid* and *city* columns, you should write *SELECT customerid, city*.

If you want to return all columns in a table, you can follow the *SELECT* keyword with an asterisk (*). The first line in the query will be *SELECT* *.

Note: Although the tables you're querying in this course are relatively small, using *SELECT* * may not be advisable when working with large databases and tables; in those cases, the final output may be difficult to understand and might be slow to run.

FROM

The *SELECT* keyword always comes with the *FROM* keyword. *FROM* indicates which table to query. To use the *FROM* keyword, you should write it after the *SELECT* keyword, often on a new line, and follow it with the name of the table you're querying. If you want to return all columns from the *customers* table, you can write:

*SELECT **

FROM customers;

When you want to end the query here, you put a semicolon (;) at the end to tell SQL that this is the entire query.

Note: Line breaks are not necessary in SQL queries, but are often used to make the query easier to understand. If you prefer, you can also write the previous query on one line as

*SELECT * FROM customers;*
and here would be the databases answer for
*SELECT * FROM customers ORDER BY country, city;*

CustomerId	FirstName	LastName	Company	Addr
56	Diego	Gutiérrez	None	307
55	Mark	Taylor	None	421
7	Astrid	Gruber	None	Rote
8	Daan	Peeters	None	Grét
13	Fernanda	Ramos	None	Qe 7
12	Roberto	Almeida	Riotur	Praç
1	Luís	Gonçalves	Embraer - Empresa Brasileira de Aeronáutica S.A.	Av.
10	Eduardo	Martins	Woodstock Discos	Rua
11	Alexandre	Rocha	Banco do Brasil S.A.	Av.
14	Mark	Philips	Telus	8210
31	Martha	Silk	None	194A
3	François	Tremblay	None	1498
30	Edward	Francis	None	230
29	Robert	Brown	None	796
15	Jennifer	Peterson	Rogers Canada	700
32	Aaron	Mitchell	None	696
33	Ellie	Sullivan	None	5112
57	Luis	Rojas	None	Call
5	František	Wichterlová	JetBrains s.r.o.	Klar
6	Helena	Holý	None	Rils
9	Kara	Nielsen	None	Sønc
44	Terhi	Hämäläinen	None	Port
42	Wyatt	Girard	None	9, F
43	Isabelle	Mercier	None	68,
41	Marc	Dubois	None	11,

(Output limit exceeded, 25 of 59 total rows shown)

ORDER BY

Database tables are often very complicated, and this is where other SQL keywords come in handy. *ORDER BY* is an important keyword for organizing the data you extract from a table.

ORDER BY sequences the records returned by a query based on a specified column or columns. This can be in either ascending or descending order.

Sorting in ascending order

To use the *ORDER BY* keyword, write it at the end of the query and specify a column to base the sort on. In this example, SQL will return the *customerid*, *city*, and *country* columns from the *customers* table, and the records will be sequenced by the *city* column:

```
SELECT customerid, city, country
FROM customers
ORDER BY city;
```

```
+------------+--------------+----------------+
| CustomerId | City         | Country        |
+------------+--------------+----------------+
|         48 | Amsterdam    | Netherlands    |
|         59 | Bangalore    | India          |
|         36 | Berlin       | Germany        |
|         38 | Berlin       | Germany        |
|         42 | Bordeaux     | France         |
|         23 | Boston       | USA            |
|         13 | Brasília     | Brazil         |
|          8 | Brussels     | Belgium        |
|         45 | Budapest     | Hungary        |
|         56 | Buenos Aires | Argentina      |
|         24 | Chicago      | USA            |
|          9 | Copenhagen   | Denmark        |
|         19 | Cupertino    | USA            |
|         58 | Delhi        | India          |
|         43 | Dijon        | France         |
|         46 | Dublin       | Ireland        |
|         54 | Edinburgh    | United Kingdom |
|         14 | Edmonton     | Canada         |
|         26 | Fort Worth   | USA            |
|         37 | Frankfurt    | Germany        |
|         31 | Halifax      | Canada         |
|         44 | Helsinki     | Finland        |
|         34 | Lisbon       | Portugal       |
|         52 | London       | United Kingdom |
|         53 | London       | United Kingdom |
+------------+--------------+----------------+
(Output limit exceeded, 25 of 59 total rows shown)
```

The *ORDER BY* keyword sorts the records based on the column specified after this keyword. By default, as shown in this example, the sequence will be in ascending order. This means

- if you choose a column containing numeric data, it sorts the output from the smallest to largest. For example, if sorting on *customerid*, the ID numbers are sorted from smallest to largest.
- if the column contains alphabetic characters, such as in the example with the *city* column, it orders the records from the beginning of the alphabet to the end.

Sorting in descending order

You can also use the *ORDER BY* with the *DESC* keyword to sort in descending order. The *DESC* keyword is short for "descending" and tells SQL to sort numbers from largest to smallest, or alphabetically from Z to A. This can be done by following *ORDER BY* with the *DESC* keyword. For example, you can run this query to examine how the results differ when *DESC* is applied:

SELECT customerid, city, country
FROM customers
ORDER BY city DESC;

```
+------------+---------------------+----------------+
| CustomerId | City                | Country        |
+------------+---------------------+----------------+
|         33 | Yellowknife         | Canada         |
|         32 | Winnipeg            | Canada         |
|         49 | Warsaw              | Poland         |
|          7 | Vienne              | Austria        |
|         15 | Vancouver           | Canada         |
|         27 | Tucson              | USA            |
|         29 | Toronto             | Canada         |
|         10 | São Paulo           | Brazil         |
|         11 | São Paulo           | Brazil         |
|          1 | São José dos Campos | Brazil         |
|          2 | Stuttgart           | Germany        |
|         51 | Stockholm           | Sweden         |
|         55 | Sidney              | Australia      |
|         57 | Santiago            | Chile          |
|         28 | Salt Lake City      | USA            |
|         47 | Rome                | Italy          |
|         12 | Rio de Janeiro      | Brazil         |
|         21 | Reno                | USA            |
|         17 | Redmond             | USA            |
|          5 | Prague              | Czech Republic |
|          6 | Prague              | Czech Republic |
|         35 | Porto               | Portugal       |
|         39 | Paris               | France         |
|         40 | Paris               | France         |
|         30 | Ottawa              | Canada         |
+------------+---------------------+----------------+
```

```
(Output limit exceeded, 25 of 59 total rows shown)
```

Now, cities at the end of the alphabet are listed first.

Sorting based on multiple columns

You can also choose multiple columns to order by. For example, you might first choose the *country* and then the *city* column. SQL then sorts the output by *country*, and for rows with the same *country*, it sorts them based on *city*. You can run this to explore how SQL displays this:

SELECT customerid, city, country
FROM customers
ORDER BY country, city;

```
+------------+---------------------+----------------+
| CustomerId | City                | Country        |
+------------+---------------------+----------------+
|         56 | Buenos Aires        | Argentina      |
|         55 | Sidney              | Australia      |
|          7 | Vienne              | Austria        |
|          8 | Brussels            | Belgium        |
|         13 | Brasília            | Brazil         |
|         12 | Rio de Janeiro      | Brazil         |
|          1 | São José dos Campos | Brazil         |
|         10 | São Paulo           | Brazil         |
|         11 | São Paulo           | Brazil         |
|         14 | Edmonton            | Canada         |
|         31 | Halifax             | Canada         |
|          3 | Montréal            | Canada         |
|         30 | Ottawa              | Canada         |
|         29 | Toronto             | Canada         |
|         15 | Vancouver           | Canada         |
|         32 | Winnipeg            | Canada         |
|         33 | Yellowknife         | Canada         |
|         57 | Santiago            | Chile          |
|          5 | Prague              | Czech Republic |
|          6 | Prague              | Czech Republic |
|          9 | Copenhagen          | Denmark        |
|         44 | Helsinki            | Finland        |
|         42 | Bordeaux            | France         |
```

```
|         43 | Dijon               | France         |
|         41 | Lyon                | France         |
+------------+---------------------+----------------+
(Output limit exceeded, 25 of 59 total rows shown)
```

Key takeaways

SELECT and *FROM* are important keywords in SQL queries. You use *SELECT* to indicate which columns to return and *FROM* to indicate which table to query. You can also include *ORDER BY* in your query to organize the output. These foundational SQL skills will support you as you move into more advanced queries.

find table name and comumns definintion for SQL and variences

Standard SQL:
For databases that support the ANSI SQL standard and have the `INFORMATION_SCHEMA` views available, you can use the following query:

```
SELECT table_name, column_name
FROM information_schema.columns;
```

you can append if you want to specify where

```
WHERE table_schema = 'your_database_name';
```

if you want **Database-specific Queries:** If you are working with a specific database system and the standard SQL approach doesn't work, you can try the following methods:

MySQL/MariaDB:

```
SELECT table_name, column_name
FROM information_schema.columns;
```

or

```
SHOW TABLES;
DESCRIBE table_name;
```

PostgreSQL:

```
 SELECT table_name, column_name
FROM information_schema.columns;
```

SQLite:

```
SELECT name AS table_name, sql AS column_definition
FROM sqlite_master
WHERE type = 'table';
```

You would run this SQLite command when you want to list all the tables in your SQLite database along with their SQL schema.

SQLite keeps a system table, `sqlite_master`, where it stores metadata about the database. Each row of `sqlite_master` represents an object (table, index, etc.) in the database.

The columns are:

- type: the type of the database object, such as 'table' or 'index'.
- name: the name of the object.
- tbl_name: the name of the table to which the object is associated. For a table, it's the same as `name`.
- rootpage: the page number in the database file where the root B-tree page for the object is stored.
- sql: the SQL statement that created the object.

This command specifies `type = 'table'` in the `WHERE` clause, so it only selects tables, not other types of objects like indices. For each table, it selects the name (renamed as `table_name` for clarity) and the SQL statement that created the table (as `column_definition`).

So this command is useful when you need to know the structure of all tables in your SQLite database, such as the table names and their corresponding column definitions. It's a handy tool for exploring a database when you don't have the schema in front of you or when you've inherited a database and need to understand its structure.

Basic filters on SQL queries

One of the most powerful features of SQL is its ability to filter.
In this video, we're going to learn how this helps us make better queries and select more specific pieces of data from a database.

Filtering is selecting data that match a certain condition.
Think of filtering as a way of only choosing the data we want.
Let's say we wanted to select apples from a fruit cart.
Filtering allows us to specify what kind of apples we want to choose.
When we go buy apples, we might explicitly say, "Choose only apples that are fresh."
This removes apples that aren't fresh from the selection.
This is a filter!
As a security analyst, you might filter a log-in attempts table to find all attempts from a specific country.
This could be done by applying a filter on the country column.
For example, you could filter to just return records containing Canada.

Before we get started, we need to focus on an important part of the syntax of SQL.
Let's learn about operators.
An operator is a symbol or keyword that represents an operation.
An example of an operator would be the equal to operator.
For example, if we wanted to find all records that have USA in the country column, we use country = 'USA'
To filter a query in SQL, we simply add an extra line to the SELECT and FROM statement we used before.
This extra line will use a WHERE clause.
In SQL, WHERE indicates the condition for a filter.
After the keyword WHERE, the specific condition is listed using operators.
So if we wanted to find all of the login attempts made in the United States, we would create this filter.
In this particular condition, we're indicating to return all records that have a value in the country column that is equal to USA.

Let's try putting it all together in SQL.
We're going to start with selecting all the columns from the log_in_attempts table. And then add the WHERE filter.
Don't forget the semicolon!
This tells us we finished the SQL statement.
Now, let's run this query! Because of our filter, only the rows where the country of the log-in attempt was USA are returned.

In the previous example, the condition for our filter was based simply on returning records that are equal to a particular value.

We can also make our conditions more complex by searching for a pattern instead of an exact word.
For example, in the employees table, we have a column for office.
We could search for records in this column that match a certain pattern.
Perhaps we might want all offices in the East building.
To search for a pattern, we used the percentage sign to act as a wildcard for unspecified characters.
If we ran a filter for 'East%', this would return all records that start with East --
for example, the offices East-120, East-290, and East-435.

When searching for patterns with the percentage sign, we cannot use the equals operator.
Instead, we use another operator, LIKE.
LIKE is an operator used with WHERE to search for a pattern in a column.

Since LIKE is an operator, similar to the equal sign, we use it instead of the equal sign.
So, when our goal is to return all values in the office column that start with the word East, LIKE would appear in a WHERE clause.

Let's go back to the example in which we wanted to filter for log-in attempts made in the United States.
Imagine that we realize that our database contains inconsistencies with how the United States is represented.
Some entries use US while others use USA.
Let's get into SQL and apply this new type of filter with LIKE.
We're going to start with the same first two lines of code because we want to select all columns from the log-in attempts table.
And we're going to add a filter with LIKE so that records will be returned if they contain a value in the country column beginning with the characters US.
This includes both US and USA.
Let's run this query to check if the output changes. This returns all the entries where the user location was in the United States.
And now we can use the LIKE clause to filter columns based on a pattern!

Wow, we've already learned how to get very precise with our database and get exactly the data we need with one single query.
I'm excited for what's next!

The WHERE clause and basic operators

Previously, you focused on how to refine your SQL queries by using the *WHERE* clause to filter results. In this reading, you'll further explore how to use the *WHERE* clause, the *LIKE* operator and the percentage sign (%) wildcard. You'll also be introduced to the underscore (_), another wildcard that can help you filter queries.

How filtering helps

As a security analyst, you'll often be responsible for working with very large and complicated security logs. To find the information you need, you'll often need to use SQL to filter the logs.

In a cybersecurity context, you might use filters to find the login attempts of a specific user or all login attempts made at the time of a security issue. As another example, you might filter to find the devices that are running a specific version of an application.

WHERE

To create a filter in SQL, you need to use the keyword *WHERE*. *WHERE* indicates the condition for a filter.

If you needed to email employees with a title of IT Staff, you might use a query like the one in the following example. You can run this example to examine what it returns:

```
SELECT firstname, lastname, title, email
FROM employees
WHERE title = 'IT Staff';
```

```
+-----------+----------+----------+------------------------+
| FirstName | LastName | Title    | Email                  |
+-----------+----------+----------+------------------------+
| Robert    | King     | IT Staff | robert@chinookcorp.com |
| Laura     | Callahan | IT Staff | laura@chinookcorp.com  |
+-----------+----------+----------+------------------------+
```

Rather than returning all records in the *employees* table, this *WHERE* clause instructs SQL to return only those that contain '*IT Staff*' in the *title* column. It uses the equals sign (=) operator to set this condition.

Note: You should place the semicolon (;) where the query ends. When you add a filter to a basic query, the semicolon is after the filter.

Filtering for patterns

You can also filter based on a pattern. For example, you can identify entries that start or end with a certain character or characters. Filtering for a pattern requires incorporating two more elements into your *WHERE* clause:

- a wildcard
- the *LIKE* operator

Wildcards

A **wildcard** is a special character that can be substituted with any other character. Two of the most useful wildcards are the percentage sign (%) and the underscore (_):

- The percentage sign substitutes for any number of other characters.
- The underscore symbol only substitutes for one other character.

These wildcards can be placed after a string, before a string, or in both locations depending on the pattern you're filtering for.

The following table includes these wildcards applied to the string '*a*' and examples of what each pattern would return.

Pattern	Results that could be returned
'*a%*'	*apple123, art, a*
'*a_*'	*as, an, a7*
'*a__*'	*ant, add, a1c*
'*%a*'	*pizza, Z6ra, a*
'*_a*'	*ma, 1a, Ha*
'*%a%*'	*Again, back, a*
'*_a_*'	*Car, ban, ea7*

LIKE

To apply wildcards to the filter, you need to use the *LIKE* operator instead of an equals sign (=). *LIKE* is used with *WHERE* to search for a pattern in a column.

For instance, if you want to email employees with a title of either '*IT Staff*' or '*IT Manager*', you can use *LIKE* operator combined with the % wildcard:

```
SELECT lastname, firstname, title, email
FROM employees
WHERE title LIKE 'IT%';
```

```
+----------+-----------+------------+------------------------+
| LastName | FirstName | Title      | Email                  |
+----------+-----------+------------+------------------------+
| Mitchell | Michael   | IT Manager | michael@chinookcorp.com |
| King     | Robert    | IT Staff   | robert@chinookcorp.com  |
| Callahan | Laura     | IT Staff   | laura@chinookcorp.com   |
+----------+-----------+------------+------------------------+
```

This query returns all records with values in the *title* column that start with the pattern of '*IT*'. This means both '*IT Staff*' and '*IT Manager*' are returned.

As another example, if you want to search through the invoices table to find all customers located in states with an abbreviation of '*NY*', '*NV*', '*NS*' or '*NT*', you can use the '*N_*' pattern on the *state* column:

```
SELECT firstname,lastname, state, country
FROM customers
WHERE state LIKE 'N_';
```

```
+-----------+----------+-------+---------+
| FirstName | LastName | State | Country |
+-----------+----------+-------+---------+
| Michelle  | Brooks   | NY    | USA     |
| Kathy     | Chase    | NV    | USA     |
| Martha    | Silk     | NS    | Canada  |
| Ellie     | Sullivan | NT    | Canada  |
+-----------+----------+-------+---------+
```

This returns all the records with state abbreviations that follow this pattern.

Key takeaways

Filters are important when refining what your query returns. *WHERE* is an essential keyword for adding a filter to your query. You can also filter for patterns by combining the *LIKE* operator with the percentage sign (%) and the underscore (_) wildcards.

Filter dates and numbers

In this video, we're going to continue using SQL queries and filters, but now we're going to apply them to new data types.
First, let's explore the three common data types that you will find in databases:
string, numeric, and date and time.
String data is data consisting of an ordered sequence of characters.
These characters could be numbers, letters, or symbols.
For example, you'll encounter string data in user names, such as a user name: analyst10.
Numeric data is data consisting of numbers, such as a count of log-in attempts.
Unlike strings, mathematical operations can be used on numeric data, like multiplication or addition.
Date and time data refers to data representing a date and/or time.

Previously, we applied filters using string data, but now let's work with numeric and date and time data.
As a security analyst, you'll often need to query numbers and dates.
For example, we could filter patch dates to find machines that need an update, or we could filter log-in attempts to return only
those made in a certain period of time.
We learned about operators in the last video, and we're going to use them again for numbers and dates.

Common operators for working with numeric or date and time data types include: equals, greater than, less than, not equal to, greater than or equal to, and less than or equal to.
Let's say you want to find the log-in attempts made after 6 pm.
Because this is past normal business hours, you want to look for suspicious patterns.
You can identify these attempts by using the greater than operator in your filter.
We'll start writing our query in SQL.
We begin by indicating that we want to select all columns FROM the log_in_attempts table.
Then we'll add our filter with WHERE.

Our condition indicates that the value in the time column must be greater than, or for dates and times, later than '18:00', which is how 6 pm is written in SQL.
Let's run this and examine the output.
Perfect! Now we have a list of log-in attempts made after 6 pm.

We can also filter for numbers and dates by using the BETWEEN operator.
BETWEEN is an operator that filters for numbers or dates within a range.
An example of this would be when looking for all patches installed within a certain range.
Let's do this! Let's find all the patches installed between March 1st, 2021 and September 1st, 2021.
In our query, we start with selecting all records FROM the machines table.

And we add the BETWEEN operator in the WHERE statement.

Let's break down the statement.
First, after WHERE, we indicate which column to filter, in our case, OS_patch_date.
Next, comes our operator BETWEEN.
We then add the beginning of our range, type AND, then finish by adding the end of our range and a semicolon.
Now, let's run this and explore the output.
And now we have a list of all machines patched between those two dates!

Before we wrap up, an important thing to note is that when we filter for strings,
dates, and times, we use quotation marks to specify what we're looking for.
However, for numbers, we don't use quotation marks.
With this new knowledge, you're now ready to work on all sorts of interesting filters for numbers and dates.
In the next video, we'll be able to expand our filtering even further by using multiple conditions in one query.

Operators for filtering dates and numbers

Previously, you examined operators like less than (<) or greater than (>) and explored how they can be used in filtering numeric and date and time data types. This reading summarizes what you learned and provides new examples of using operators in filters.

Numbers, dates, and times in cybersecurity

Security analysts work with more than just **string data**, or data consisting of an ordered sequence of characters.

They also frequently work with **numeric data**, or data consisting of numbers. A few examples of numeric data that you might encounter in your work as a security analyst include:

- the number of login attempts
- the count of a specific type of log entry
- the volume of data being sent from a source
- the volume of data being sent to a destination

You'll also encounter **date and time data**, or data representing a date and/or time. As a first example, logs will generally timestamp every record. Other time and date data might include:

- login dates
- login times
- dates for patches
- the duration of a connection

Comparison operators

In SQL, filtering numeric and date and time data often involves operators. You can use the following operators in your filters to make sure you return only the rows you need:

operator	use

<	less than
>	greater than
=	equal to
<=	less than or equal to
>=	greater than or equal to
<>	not equal to

Note: You can also use *!=* as an alternative operator for not equal to.

Incorporating operators into filters

These comparison operators are used in the *WHERE* clause at the end of a query. The following query uses the > operator to filter the *birthdate* column. You can run this query to explore its output:

```
SELECT firstname, lastname, birthdate
FROM employees
WHERE birthdate > '1970-01-01';
```

```
+-----------+----------+---------------------+
| FirstName | LastName | BirthDate           |
+-----------+----------+---------------------+
| Jane      | Peacock  | 1973-08-29 00:00:00 |
| Michael   | Mitchell | 1973-07-01 00:00:00 |
| Robert    | King     | 1970-05-29 00:00:00 |
+-----------+----------+---------------------+
```

This query returns the first and last names of employees born after, but not on, '*1970-01-01*' (or January 1, 1970). If you were to use the >= operator instead, the results would also include results on exactly '*1970-01-01*'.

In other words, the > operator is exclusive and the >= operator is inclusive. An **exclusive operator** is an operator that does not include the value of comparison. An **inclusive operator** is an operator that includes the value of comparison.

BETWEEN

Another operator used for numeric data as well as date and time data is the *BETWEEN* operator. *BETWEEN* filters for numbers or dates within a range. For example, if you want to find the first and last names of all employees hired between January 1, 2002 and January 1, 2003, you can use the

BETWEEN operator as follows:

```
SELECT firstname, lastname, hiredate
FROM employees
WHERE hiredate BETWEEN '2002-01-01' AND '2003-01-01';
```

```
+-----------+----------+---------------------+
| FirstName | LastName | HireDate            |
+-----------+----------+---------------------+
| Andrew    | Adams    | 2002-08-14 00:00:00 |
| Nancy     | Edwards  | 2002-05-01 00:00:00 |
| Jane      | Peacock  | 2002-04-01 00:00:00 |
+-----------+----------+---------------------+
```

Note: The *BETWEEN* operator is inclusive. This means records with a *hiredate* of January 1, 2002 or January 1, 2003 are included in the results of the previous query.

Key takeaways

Operators are important when filtering numeric and date and time data. These include exclusive operators such as < and inclusive operators such as <=. The *BETWEEN* operator, another inclusive operator, helps you return the data you need within a range.

Filters with AND, OR, and NOT

In the previous lesson, we learned about even more ways to filter queries in SQL to work with some typical security analyst tasks.
However, when working with real security questions, we often have to filter for multiple conditions.
Vulnerabilities, for instance, might depend on more than one factor.
For example, a security vulnerability might be related to machines using a specific email client on a specific operating system.
So, to find the possible vulnerabilities, we need to find machines using both the email client and the operating system.

To make a query with multiple conditions that must be met, we use the AND operator between two separate conditions.
AND is an operator that specifies that both conditions must be met simultaneously.
Bringing this back to our fruit and vegetable analogy, this is the same as asking someone to select apples from the big box where the apples are large and fresh.
This means our results won't include any small apples even if they're fresh, or any rotten apples even if they're large.
They'll only include large fresh apples.
The apples must meet both conditions.

Going back to our database, the machines table lists all operating systems and email clients.
We want a list of machines running Operating System 1 and a list of machines using Email Client 1.
We'll use the left and right circles in the Venn diagram to represent these groups.
We need SQL to select the machines that have both OS 1 and Email Client 1.
The filled-in area at the intersection of these circles represents this condition.
Let's take this and implement it in SQL.

First, we're going to start by building the first lines of the query, telling SQL to SELECT* all columns FROM the machines table.
Then, we'll add the WHERE clause.
Let's examine this more closely.
First, we indicate the first condition that it must meet, that the operating system column has a value of '0S 1'

Then, we use AND to join this to another condition.
And finally, we enter the other condition, in this case that the email client column should have a value of 'Email Client 1'
And this is how you use the AND operator in SQL!
Let's run this to get the query results.
Perfect! All the results match both our conditions!

Let's keep going and explore more ways to combine different conditions by working with the OR operator.

The OR operator is an operator that specifies that either condition can be met.
In a Venn diagram, let's say each circle represents a condition.
When they are joined with OR,
SQL would select all rows that satisfy one of the conditions.
And it's also ok if it meets both conditions.

Let's run another query and use the OR operator.
Let's say that we wanted the filter to identify machines that have either OS 1 or OS 3 because both types need a patch.
We'll type in these conditions.

Let's examine this more closely.
After WHERE, our first condition indicates we want to filter, so that the query selects machines with 'OS 1'
We use the OR operator because we also want to find records that match another condition.
This additional condition is placed after OR and indicates to also select machines running 'OS 3'
Executing the query, our results now include records that have a value of either OS 1 or OS 3 in the operating system column.
Good job, we're running some complex queries.

The last operator we're going to go into is the NOT operator.
NOT negates a condition.
In a diagram, we can show this by selecting every entry that does not match our condition.
The condition is represented by the circle.
The filled-in portion outside the circle represents what gets returned.
This is all data that does not match the condition.
For example, when picking out fruit, you can be looking for any fruit that is not an apple.
That is a lot more efficient than telling your friend you want a banana or an orange or a lime, and so on.

Suppose you wanted to update all of the devices in your company except for the ones using OS 3.
Bringing this into SQL, we can write this query.

We place NOT after WHERE and before the condition of the filter.
Executing these queries gives us the list of all the machines that aren't running OS 3, and now we know which machines to update.

That was a lot of new content that we just looked into, but you're learning more and more SQL that you can use on your journey to become an analyst!
In the next video, we'll be learning how to combine and join two tables together to expand the kinds of queries we can run. I'll meet you there!

More on filters with AND, OR, and NOT

Previously, you explored how to add filters containing the *AND*, *OR*, and *NOT* operators to your SQL queries. In this reading, you'll continue to explore how these operators can help you refine your queries.

Logical operators

AND, *OR*, and *NOT* allow you to filter your queries to return the specific information that will help you in your work as a security analyst. They are all considered logical operators.

AND

First, *AND* is used to filter on two conditions. *AND* specifies that both conditions must be met simultaneously.

As an example, a cybersecurity concern might affect only those customer accounts that meet both the condition of being handled by a support representative with an ID of 5 and the condition of being located in the USA. To find the names and emails of those specific customers, you should place the two conditions on either side of the *AND* operator in the *WHERE* clause:

```
SELECT firstname, lastname, email, country, supportrepid
FROM customers
WHERE supportrepid = 5 AND country = 'USA';
```

```
+-----------+----------+------------------------+---------+-------------+
| FirstName | LastName | Email                  | Country | SupportRepId |
+-----------+----------+------------------------+---------+-------------+
| Jack      | Smith    | jacksmith@microsoft.com | USA     |            5 |
| Kathy     | Chase    | kachase@hotmail.com     | USA     |            5 |
| Victor    | Stevens  | vstevens@yahoo.com      | USA     |            5 |
| Julia     | Barnett  | jubarnett@gmail.com     | USA     |            5 |
+-----------+----------+------------------------+---------+-------------+
```

Running this query returns four rows of information about the customers. You can use this information to contact them about the security concern.

OR

The *OR* operator also connects two conditions, but *OR* specifies that either condition can be met. It returns results where the first condition, the second condition, or both are met.

For example, if you are responsible for finding all customers who are either in the USA or Canada so that you can communicate information about a security update, you can use an *OR* operator to find all the needed records. As the following query demonstrates, you should place the two conditions on either side of the *OR* operator in the *WHERE* clause:

```
SELECT firstname, lastname, email, country
FROM customers
WHERE country = 'Canada' OR country = 'USA';
```

```
+-----------+----------+------------------------+---------+--------------+
| FirstName | LastName | Email                  | Country | SupportRepId |
+-----------+----------+------------------------+---------+--------------+
| Jack      | Smith    | jacksmith@microsoft.com | USA     |            5 |
| Kathy     | Chase    | kachase@hotmail.com     | USA     |            5 |
| Victor    | Stevens  | vstevens@yahoo.com      | USA     |            5 |
| Julia     | Barnett  | jubarnett@gmail.com     | USA     |            5 |
+-----------+----------+------------------------+---------+--------------+
```

The query returns all customers in either the US or Canada.

Note: Even if both conditions are based on the same column, you need to write out both full conditions. For instance, the query in the previous example contains the filter *WHERE country = 'Canada' OR country = 'USA'*.

NOT

Unlike the previous two operators, the *NOT* operator only works on a single condition, and not on multiple ones. The *NOT* operator negates a condition. This means that SQL returns all records that don’t match the condition specified in the query.

For example, if a cybersecurity issue doesn't affect customers in the USA but might affect those in other countries, you can return all customers who are not in the USA. This would be more efficient than creating individual conditions for all of the other countries. To use the *NOT* operator for this task, write the following query and place *NOT* directly after *WHERE*:

```
SELECT firstname, lastname, email, country
FROM customers
```

```
WHERE NOT country = 'USA';
```

```
+-----------+-------------+-------------------------------+----------------+
| FirstName | LastName    | Email                         | Country        |
+-----------+-------------+-------------------------------+----------------+
| Luís      | Gonçalves   | luisg@embraer.com.br          | Brazil         |
| Leonie    | Köhler      | leonekohler@surfeu.de         | Germany        |
| François  | Tremblay    | ftremblay@gmail.com           | Canada         |
| Bjørn     | Hansen      | bjorn.hansen@yahoo.no         | Norway         |
| František | Wichterlová | frantisekw@jetbrains.com      | Czech Republic |
| Helena    | Holý        | hholy@gmail.com               | Czech Republic |
| Astrid    | Gruber      | astrid.gruber@apple.at        | Austria        |
| Daan      | Peeters     | daan_peeters@apple.be         | Belgium        |
| Kara      | Nielsen     | kara.nielsen@jubii.dk         | Denmark        |
| Eduardo   | Martins     | eduardo@woodstock.com.br      | Brazil         |
| Alexandre | Rocha       | alero@uol.com.br              | Brazil         |
| Roberto   | Almeida     | roberto.almeida@riotur.gov.br | Brazil         |
| Fernanda  | Ramos       | fernadaramos4@uol.com.br      | Brazil         |
| Mark      | Philips     | mphilips12@shaw.ca            | Canada         |
| Jennifer  | Peterson    | jenniferp@rogers.ca           | Canada         |
| Robert    | Brown       | robbrown@shaw.ca              | Canada         |
| Edward    | Francis     | edfrancis@yachoo.ca           | Canada         |
| Martha    | Silk        | marthasilk@gmail.com          | Canada         |
| Aaron     | Mitchell    | aaronmitchell@yahoo.ca        | Canada         |
| Ellie     | Sullivan    | ellie.sullivan@shaw.ca        | Canada         |
| João      | Fernandes   | jfernandes@yahoo.pt           | Portugal       |
| Madalena  | Sampaio     | masampaio@sapo.pt             | Portugal       |
| Hannah    | Schneider   | hannah.schneider@yahoo.de     | Germany        |
| Fynn      | Zimmermann  | fzimmermann@yahoo.de          | Germany        |
| Niklas    | Schröder    | nschroder@surfeu.de           | Germany        |
+-----------+-------------+-------------------------------+----------------+
(Output limit exceeded, 25 of 46 total rows shown)
```

SQL returns every entry where the customers are not from the USA.

Pro tip: Another way of finding values that are not equal to a certain value is by using the <> operator or the *!=* operator. For example, *WHERE country <> 'USA'* and *WHERE country != 'USA'* are the same filters as *WHERE NOT country = 'USA'*.

Combining logical operators

Logical operators can be combined in filters. For example, if you know that both the USA and Canada are not affected by a cybersecurity issue, you can combine operators to return customers in all countries besides these two. In the following query, *NOT* is placed before the first condition, it's joined to a second condition with *AND*, and then *NOT* is also placed before that second

condition. You can run it to explore what it returns:

```
SELECT firstname, lastname, email, country
FROM customers
WHERE NOT country = 'Canada' AND NOT country = 'USA';
```

```
+-----------+-------------+-------------------------------+----------------+
| FirstName | LastName    | Email                         | Country        |
+-----------+-------------+-------------------------------+----------------+
| Luís      | Gonçalves   | luisg@embraer.com.br          | Brazil         |
| Leonie    | Köhler      | leonekohler@surfeu.de         | Germany        |
| Bjørn     | Hansen      | bjorn.hansen@yahoo.no         | Norway         |
| František | Wichterlová | frantisekw@jetbrains.com      | Czech Republic |
| Helena    | Holý        | hholy@gmail.com               | Czech Republic |
| Astrid    | Gruber      | astrid.gruber@apple.at        | Austria        |
| Daan      | Peeters     | daan_peeters@apple.be         | Belgium        |
| Kara      | Nielsen     | kara.nielsen@jubii.dk         | Denmark        |
| Eduardo   | Martins     | eduardo@woodstock.com.br      | Brazil         |
| Alexandre | Rocha       | alero@uol.com.br              | Brazil         |
| Roberto   | Almeida     | roberto.almeida@riotur.gov.br | Brazil         |
| Fernanda  | Ramos       | fernadaramos4@uol.com.br      | Brazil         |
| João      | Fernandes   | jfernandes@yahoo.pt           | Portugal       |
| Madalena  | Sampaio     | masampaio@sapo.pt             | Portugal       |
| Hannah    | Schneider   | hannah.schneider@yahoo.de     | Germany        |
| Fynn      | Zimmermann  | fzimmermann@yahoo.de          | Germany        |
| Niklas    | Schröder    | nschroder@surfeu.de           | Germany        |
| Camille   | Bernard     | camille.bernard@yahoo.fr      | France         |
| Dominique | Lefebvre    | dominiquelefebvre@gmail.com   | France         |
| Marc      | Dubois      | marc.dubois@hotmail.com       | France         |
| Wyatt     | Girard      | wyatt.girard@yahoo.fr         | France         |
| Isabelle  | Mercier     | isabelle_mercier@apple.fr     | France         |
| Terhi     | Hämäläinen  | terhi.hamalainen@apple.fi     | Finland        |
| Ladislav  | Kovács      | ladislav_kovacs@apple.hu      | Hungary        |
| Hugh      | O'Reilly    | hughoreilly@apple.ie          | Ireland        |
+-----------+-------------+-------------------------------+----------------+
(Output limit exceeded, 25 of 38 total rows shown)
```

Key takeaways

Logical operators allow you to create more specific filters that target the security-related information you need. The *AND* operator requires two conditions to be true simultaneously, the *OR* operator requires either one or both conditions to be true, and the *NOT* operator negates a condition. Logical operators can be combined together to create even more specific queries.

Join tables in SQL

You've already learned a lot about SQL queries and filters. Nice work!
The last concept we're introducing in this section is joining tables when querying a database.
This is helpful when you need information from two different tables in a database.
Let's say we have two tables: one that tells us about security vulnerabilities of different operating systems, and one about different machines in our company, including their operating systems.
Having the ability to combine them gives us a list of vulnerable machines.
That's pretty cool, right?

First, let's start talking about the syntax of joins.
Since we're working with two tables now, we need a way to tell SQL what table we're picking columns from.
In our example database, we have an employee_id column in both the employees table and the machines table.
In SQL statements that contain two columns, SQL needs to know which column we're referring to.
The way to resolve this is by writing the name of the table first, then a period, and then the name of a column.
So, we would have employees followed by a period, followed by the column name.
This is the employee_id column for the employees table.
Similarly, this is the employee_id column for the machines table.
Now that we understand this syntax, let's apply it to a join!

Imagine that we want to get a deeper understanding of the employees accessing the machines in our company.
By joining the employees and the machines tables, we can do this!
We first need to identify the shared column that we'll use to connect the two tables.
In this case, we'll use a primary key and one table to connect to another table where it's a foreign key.
The primary key of the employees table is employee_id, which is a foreign key in the machines table.
employee_id is a primary key in the employees table because it has a unique value for every row in the employees table, and no empty values.
We don't have a guarantee that the employee_id column in the machines table follows the same criteria since it's
a foreign key and not a primary key.

Next, we'll use a type of join called an INNER JOIN.
An INNER JOIN returns rows matching on a specified column that exists in more than one table.
Tables usually contain many more rows, but to further explain what we mean by INNER JOIN, let's focus on just four rows from the employees table and four rows from the machines table.
We'll also look at just a few columns of each table for this example.
Let's say we choose employee_id in both tables to perform an INNER JOIN.
Let's look at the two rows where there is a match.

Both tables have 1188 and 1189 in their respective employee_id columns, so they are considered a match.
The results of the join is the two rows that have 1188 and 1189 and all columns from both tables.
Before we move on to the queries, we have to talk about the NULL values in the tables.
In SQL, NULL represents a missing value due to any reason.
In this case, this might be machines that are not assigned to any employee.
Now, let's bring this into SQL and do an INNER JOIN on the full tables.
Let's imagine we want to join these tables in order to get a list of users and their office location that also shows
what operating system they use on their machines.
employee_id is a common column between these tables, and we can use this to join them.
But we won't need to show this column in the results.
First, let's start with a basic query that indicates we want to select the username, office, and operating_system columns.
We want employees to be our first or left table, so we'll use that in our FROM statement.
Now, we write the part of the query that tells SQL to join the machines table with the employees table.

Let's break down this query.
INNER JOIN tells SQL to perform the INNER JOIN.
Then, we name the second table we want to combine with the first.
This is called the right table.
In this case, we want to join machines with the employees table that was already identified after FROM.
Lastly, we tell SQL what column to base the join on.
In our case, we're using the employee_id column.
Since we're using two tables, we have to identify the table and follow that with the column name.
So, we have employees.employee_id. And machines.employee_id.

Let's review the output.
Perfect! We have now joined two tables.
The results of our query displays the records that match on the employee_id column.
Notice that these records contain columns from both tables, but only the ones we've indicated through our SELECT statement.
There are other types of joins that don't require a match to join two tables, and we're going to discuss those in
the next video. I'll meet you there!

Types of joins

Welcome back. I hope you enjoyed working on inner joins.
In the previous video and exercises, we saw how inner joins can be useful by only returning records that share a value in specify columns.
However, in some situations, we might need all of the entries from one or both of our tables.
This is where we need to use outer joins.

There are three types of outer joins: LEFT JOIN, RIGHT JOIN, and FULL OUTER JOIN.
Similar to inner joins, outer joins combine two tables together; however, they don't necessarily need a match between columns to return a row.
Which rows are returned depends on the type of join.
LEFT JOIN returns all of the records of the first table, but only returns rows of the second table that match on a specified column.
Like we did in the previous video, let's examine this type of join by looking at just four rows of two tables with a small number of columns.
Employees is the left table, or the first table, and machines is the right table, or the second table.
Let's join on employee_id.
There's a matching value in this column for two of the four records.
When we execute the join, SQL returns these rows with the matching value, all other rows from the left table, and all columns from both tables.
Records from the employees table that didn't match but were returned through the LEFT JOIN contain NULL values
in columns that came from the machines table.
Next, let's talk about right joins.
RIGHT JOIN returns all of the records of the second table but only returns rows from the first table that match on a specified column.
With a RIGHT JOIN on the previous example, the full result returns matching rows from both, all the rows from the second table, and all the columns in both tables.
For the values that don't exist in either table, we are left with a NULL value.
Last, we'll discuss full outer joins.

FULL OUTER JOIN returns all records from both tables. Using our same example, a FULL OUTER JOIN returns all columns from all tables.
If a row doesn't have a value for a particular column, it returns NULL.
For example, the machines table do not have any rows with employee_id 1190, so the values for that row and the
columns that came from the machines table is NULL.
To implement left joins, right joins, and full outer joins in SQL, you use the same syntax structure as the INNER JOIN but use these keywords:
LEFT JOIN, RIGHT JOIN, and FULL OUTER JOIN.
As a security analyst,
you're not required to know all of these from memory.
Once you understand the type of join you need,

you can quickly search and find
all the information you need to execute these queries.
With this information on joins,
we've now covered some very important information
you'll need as a security analyst using SQL.
Thank you for joining me in this video.

Compare types of joins

Previously, you explored SQL joins and how to use them to join data from multiple tables when these tables share a common column. You also examined how there are different types of joins, and each of them returns different rows from the tables being joined. In this reading, you'll review these concepts and more closely analyze the syntax needed for each type of join.

Inner joins

The first type of join that you might perform is an inner join. *INNER JOIN* returns rows matching on a specified column that exists in more than one table.



It only returns the rows where there is a match, but like other types of joins, it returns all specified columns from all joined tables. For example, if the query joins two tables with *SELECT* *, all columns in both of the tables are returned.

Note: If a column exists in both of the tables, it is returned twice when *SELECT* * is used.

The syntax of an inner join

To write a query using *INNER JOIN*, you can use the following syntax:

*SELECT **

FROM employees

INNER JOIN machines ON employees.device_id = machines.device_id;

SELECT thing1, thing2, thingX FROM table1 inner join table2 ON table1.commun_colomn = table2.common_colomn;

You must specify the two tables to join by including the first or left table after *FROM* and the second or right table after *INNER JOIN*.

After the name of the right table, use the *ON* keyword and the = operator to indicate the column you are joining the tables on. It's important that you specify both the table and column names in this portion of the join by placing a period (.) between the table and the column.

In addition to selecting all columns, you can select only certain columns. For example, if you only want the join to return the *username*, *operating_system* and *device_id* columns, you can write this query:

SELECT username, operating_system, employees.device_id

FROM employees

INNER JOIN machines ON employees.device_id = machines.device_id;

it makes more sense for it to be all in one row for me so heres an explanation of each part how it works
SELECT thing_1, thing_2, thing_X FROM table1 INNER JOIN table2 ON table1.common_column = table2.common_column;

Note: In the example query, *username* and *operating_system* only appear in one of the two tables, so they are written with just the column name. On the other hand, because *device_id* appears in both tables, it's necessary to indicate which one to return by specifying both the table and column name (*employees.device_id*).

Outer joins

Outer joins expand what is returned from a join. Each type of outer join returns all rows from either one table or both tables.

Left joins

When joining two tables, *LEFT JOIN* returns all the records of the first table, but only returns rows of the second table that match on a specified column.



The syntax for using *LEFT JOIN* is demonstrated in the following query:

*SELECT **

FROM employees

LEFT JOIN machines ON employees.device_id = machines.device_id;

As with all joins, you should specify the first or left table as the table that comes after *FROM* and the second or right table as the table that comes after *LEFT JOIN*. In the example query, because *employees* is the left table, all of its records are returned. Only records that match on the *device_id* column are returned from the right table, *machines*.

Right joins

When joining two tables, *RIGHT JOIN* returns all of the records of the second table, but only returns rows from the first table that match on a specified column.



The following query demonstrates the syntax for *RIGHT JOIN*:

*SELECT **

FROM employees

RIGHT JOIN machines ON employees.device_id = machines.device_id;

RIGHT JOIN has the same syntax as *LEFT JOIN*, with the only difference being the keyword *RIGHT JOIN* instructs SQL to produce different output. The query returns all records from *machines*, which is the second or right table. Only matching records are returned from *employees*, which is the first or left table.

Note: You can use *LEFT JOIN* and *RIGHT JOIN* and return the exact same results if you use the tables in reverse order. The following *RIGHT JOIN* query returns the exact same result as the *LEFT JOIN* query demonstrated in the previous section:

*SELECT **

FROM machines

RIGHT JOIN employees ON employees.device_id = machines.device_id;

All that you have to do is switch the order of the tables that appear before and after the keyword used for the join, and you will have swapped the left and right tables.

Full outer joins

FULL OUTER JOIN returns all records from both tables. You can think of it as a way of completely merging two tables.



You can review the syntax for using *FULL OUTER JOIN* in the following query:

*SELECT **

FROM employees

FULL OUTER JOIN machines ON employees.device_id = machines.device_id;

The results of a *FULL OUTER JOIN* query include all records from both tables. Similar to *INNER JOIN*, the order of tables does not change the results of the query.

Key takeaways

When working in SQL, there are multiple ways to join tables. All joins return the records that match on a specified column. *INNER JOIN* will return only these records. Outer joins also return all other records from one or both of the tables. *LEFT JOIN* returns all records from the first or left table, *RIGHT JOIN* returns all records from the second or right table, and *FULL OUTER JOIN* returns all records from both tables.

Continuous learning in SQL

You've explored a lot about SQL, including applying filters to SQL queries and joining multiple tables together in a query. There's still more that you can do with SQL. This reading will explore an example of something new you can add to your SQL toolbox: aggregate functions. You'll then focus on how you can continue learning about this and other SQL topics on your own.

Aggregate functions

In SQL, **aggregate functions** are functions that perform a calculation over multiple data points and return the result of the calculation. The actual data is not returned.

There are various aggregate functions that perform different calculations:

- *COUNT* returns a single number that represents the number of rows returned from your query.
- *AVG* returns a single number that represents the average of the numerical data in a column.
- *SUM* returns a single number that represents the sum of the numerical data in a column.

Aggregate function syntax

To use an aggregate function, place the keyword for it after the *SELECT* keyword, and then in parentheses, indicate the column you want to perform the calculation on.

For example, when working with the *customers* table, you can use aggregate functions to summarize important information about the table. If you want to find out how many customers there are in total, you can use the *COUNT* function on any column, and SQL will return the total number of records, excluding *NULL* values. You can run this query and explore its output:

```
SELECT COUNT(firstname)
FROM customers;
```

```
+------------------+
| COUNT(firstname) |
+------------------+
|               59 |
+------------------+
```

The result is a table with one column titled *COUNT(firstname)* and one row that indicates the count.

If you want to find the number of customers from a specific country, you can add a filter to your query:

```
SELECT COUNT(firstname)
FROM customers
WHERE country = 'USA';
```

```
+------------------+
| COUNT(firstname) |
+------------------+
|               13 |
+------------------+
```

With this filter, the count is lower because it only includes the records where the *country* column contains a value of '*USA*'.

There are a lot of other aggregate functions in SQL. The syntax of placing them after *SELECT* is exactly the same as the *COUNT* function.

Continuing to learn SQL

SQL is a widely used querying language, with many more keywords and applications. You can continue to learn more about aggregate functions and other aspects of using SQL on your own.

Most importantly, approach new tasks with curiosity and a willingness to find new ways to apply SQL to your work as a security analyst. Identify the data results that you need and try to use SQL to obtain these results.

Fortunately, SQL is one of the most important tools for working with databases and analyzing data, so you'll find a lot of support in trying to learn SQL online. First, try searching for the concepts you've already learned and practiced to find resources that have accurate easy-to-follow explanations. When you identify these resources, you can use them to extend your knowledge.

Continuing your practical experience with SQL is also important. You can also search for new databases that allow you to perform SQL queries using what you've learned.

Key takeaways

Aggregate functions like *COUNT*, *SUM*, and *AVG* allow you to work with SQL in new ways. There are many other additional aspects of SQL that could be useful to you as an analyst. By continuing to explore SQL on your own, you can expand the ways you can apply SQL in a cybersecurity context.

Wrap-up; Glossary terms from week 4

Congratulations! We've made it together through the end of our focus on SQL.
You've put in a lot of work and learned an important tool that will help you on your journey as a security analyst.
Let's take a moment to go through all of the topics you learned in this section.

We started by learning about the structure of relational databases and how we can access them by using the query language SQL.
We then got hands-on practice with writing our own SQL queries.
We used SQL to bring up information you might need on the job when working as an analyst.
We then focused on SQL filters.
We started with simple conditions with strings, and by the end, we learned how to use multiple filters in one query.
We concluded the unit with SQL joins and learned how to join multiple tables, giving us even more information at once.

By completing this course, you just took a very big step in your future career as a security analyst.
You have been introduced to a powerful tool that can help you in your work.
Whenever you need to, I encourage you to revisit the materials in this course.
Learning a querying language like SQL takes time.
Thank you again for joining me in this journey.
I hope you'll enjoy using SQL as much as I do.

Terms and definitions from Course 4, Week 4

Database: An organized collection of information or data

Date and time data: Data representing a date and/or time

Exclusive operator: An operator that does not include the value of comparison

Filtering: Selecting data that match a certain condition

Foreign key: A column in a table that is a primary key in another table

Inclusive operator: An operator that includes the value of comparison

Log: A record of events that occur within an organization's systems

Numeric data: Data consisting of numbers

Operator: A symbol or keyword that represents an operation

Primary key: A column where every row has a unique entry

Query: A request for data from a database table or a combination of tables

Relational database: A structured database containing tables that are related to each other

String data: Data consisting of an ordered sequence of characters

SQL (Structured Query Language): A programming language used to create, interact with, and request information from a database

Syntax: The rules that determine what is correctly structured in a computing language

Wildcard: A special character that can be substituted with any other character