WEEK 2

LEARNING OBJECTIVES

- Use Hue to execute SQL statements
- Use SQL utility commands to explore and navigate databases and tables
- Provide some examples of interfaces other than Hue
- (Honors) Use command line interfaces to run SQL commands on big data systems

REVIEW AND PREPARATION

This course is the second in the specialization Modern Big Data Analysis with SQL from Cloudera. The first course in this specialization, Foundations for SQL on Big Data, teaches the key concepts behind relational databases, SQL, and big data. In this video, I'll do a whirlwind review of that first course to ensure you're prepared for this second course. If you already took the first course, that's great, and this will just be a quick recap for you. If not, then this video should help you to understand what it's about so you can decide whether to take that first course before continuing with this second course. So here's that first course in a nutshell. Data analysis begins with data. Somewhere in the data, you'll often find information that you need to answer questions. Finding that information isn't always easy, but organizing the data allows you to find it more quickly and efficiently. Relational database systems and SQL are popular because they allow you to do just that, organize data, so it's easier to work with. SQL enables you to perform four categories of activities with relational databases.
Designing using SQL's Data Recognition Language, or DDL.
Updating using Data Manipulation Language, DML.
Retrieving using Data Query Language, DQL. And
managing using Data Control Language, DCL.
SQL statements fall into these four categories. It's common to use SQL with two different types of relational databases. Operational databases, which are mostly used to store rapidly changing data about the current state of a process or business. And analytic databases, which are mostly used for answering questions about more static data, sometimes going back many years. The design of the tables in a database affects the type of data you can store and use.

So the way you intend to use your data will inform how the database should be designed. For an operational database, you'll typically use a well normalized design. But for an analytic database, you will often de-normalize your tables. When it comes to analysis on big data, there's often a much larger volume and a greater variety in the data. And this drives many of the decisions about how data is stored and processed. Traditional database systems have their place with smaller datasets, but they become too slow and storage becomes too expensive for very large scale data. These traditional systems also require the data to be highly structured. But there is a type of newer distributed SQL engine that can work with much larger datasets and can handle semi-structured and unstructured data as well, enabling you to answer a larger variety of questions.

Modern distributed engines use their own dialects of SQL, which provide features that are helpful for working with big data. This includes different options for storing data, a separation of the metadata from the data itself, and loose coupling between the SQL engine and the metadata and data it operates on. However, the basics of the data query language, the select statement, that's used with these distributed engines is mostly the same as with traditional relational databases. So if you have some experience with SQL queries, you'll be able to reuse many of those skills with these engines. Two examples of this type of distributed query engine are Hive and Impala. Both are designed for use as analytic databases. These are two different SQL engines, but they can both operate on a shared set of metadata and data.

Some of the parts of the SQL language that are important when you're working with a traditional relational database are not relevant when you're working with these distributed big data query engines like Hive and Impala. This specialization focuses on the skills that do apply when you're doing data analysis in this big data world. The final part of the first course shows how to set up the virtual machine, or VM, that you'll use for hands-on exercises and practice in this course. If you haven't yet set up your VM, the document attached to this video provides instructions to help you do that. If the topics I've described in this video all sound familiar to you, then you're ready to take this course. But if some of these topics were unfamiliar, then you probably need more of a conceptual foundation before you're ready to take this course. If that's the case, I recommend completing the first course in this specialization, Foundations for SQL on Big Data, before you continue with this course. SQL and want to know how it relates to this course.

(Optional) What about Spark SQL?

The first video in this course listed several open source distributed SQL engines that are capable of querying extremely large datasets:

- Apache Hive
- Apache Impala
- Presto
- Apache Drill

This list did not include Spark SQL. This optional reading briefly explains what Spark is, what Spark SQL is, and why Spark SQL was not included on this list.

WHAT IS APACHE SPARK, AND WHAT IS SPARK SQL?

Apache Spark is a large-scale data processing engine. It is capable of running a wide range of different data processing workloads. Apache Spark provides several libraries for performing different kinds of work. One of these libraries is Spark SQL.

Spark SQL is Spark's library for working with structured data. The name "Spark SQL" seems to suggest that the SQL query language is the central piece of this library, but it is not. Support for the SQL query language is just one part of what Spark SQL provides. Spark SQL also provides programming interfaces for several programming languages (Scala, Java, Python, and R) that are not based on the SQL query language.

Who Uses Spark SQL?

Spark SQL is most often used by data scientists, data engineers, and big data application developers. Spark SQL helps those types of users work with structured data inside their Spark applications.

Spark SQL is not widely used by data analysts. Compared to Hive and Impala, Spark SQL is not as well-incorporated into the ecosystem of tools that data analysts use. The lack of integration, tooling, and support for Spark SQL has limited its use by data analysts. Furthermore, the architecture of Apache Spark makes Spark SQL inherently less efficient as a query engine for data analysts than purpose-built query engines like Impala.

However, there have been some recent efforts to make Spark SQL a more viable alternative for data analysts running interactive SQL queries. If these efforts prove successful, we will consider adding more details about Spark SQL to this course. But at the current time, the number of data analysts using Spark SQL remains relatively small, and there are obstacles to its broader use. As a result, we recommend that data analysts focus on learning Hive and Impala.

Spark SQL Is Compatible with Hive and Impala

The good news is that Spark SQL was designed to be highly compatible with Hive and Impala. Spark SQL can query the same tables that Hive and Impala can, and the Spark SQL query syntax is almost entirely compatible with Hive's query syntax. So even though this course does not mention Spark SQL by name (except in this reading), you can take the skills you'll learn in this course and apply them directly to Spark SQL.

For more information about Spark SQL compatibility with Hive, see https://spark.apache.org/docs/latest/sql-migration-guide-hive-compatibility.html (but note that many of the details described there are beyond the scope of this course).

To help ensure you're in the right place, this reading will describe what's expected in terms of where you've been and where you're going, and will provide some advice for how to proceed depending on your current knowledge and experience.

If you want to earn a certificate showing that you completed the whole Coursera specialization, then please start with Course 1, and proceed to this course after you've completed Course 1. Even if you are not hoping to earn this certificate, Course 1 is recommended unless you already have significant experience with the structure of traditional RDBMSs and already understand the basics of big data warehousing. See the “Review and Preparation” to help you judge if you have enough background on this. (This is not about understanding SQL, but about understanding how traditional and big data systems are structured.)

If, after these strong encouragements to complete Course 1, you still wish to skip that course and continue with this one, you should be sure to download and install the virtual machine (VM) that provides a hands-on environment for testing and practicing the skills you'll learn in this course. Learners who completed Course 1 should already have this VM installed, and they should have done some browsing of databases and tables using the VM. Instructions for downloading and installing it are available in a separate reading, in this week's materials.

This course is the second course in the specialization. It is suitable for beginners with no prior experience with SQL. In this course, you'll learn fundamental skills, not advanced tricks. Some of the more complex and confusing parts of SQL have been saved for fourth and final course in this specialization. To use a Star Wars analogy: At the end of this course, you’ll be a SQL Padawan, not a full-fledged SQL Jedi. Becoming a full-fledged SQL Jedi would require taking the remaining courses in the specialization, plus gaining some real-world experience.

If you have some familiarity with SQL already, there is still value in taking this course:

- It will help give you more comprehensive knowledge—unless you’re a total SQL pro (Jedi!), you’ll definitely learn something you didn’t know.
- It will help you to intuitively understand SQL and to be more fluent in it.
- It will help you to go from using SQL on RDBMSs to using SQL on distributed big data query engines.

However, you might want to make a few adjustments as you proceed with this course:

- Move at a faster pace through the materials.
- Start with the assessments to verify your skills and find the pieces that might be new to you.
- Focus on the parts of SQL that are different in big data query engines like Hive and Impala than in RDBMSs.

If, on the other hand, you want to learn about SQL so you can use an RDBMS or some other SQL engine (not specifically Hive and Impala), that's OK too! The VM includes MySQL and PostgreSQL engines for you to work with. Most of the datasets used in the materials is available to query using those SQL engines, though the largest datasets (in the **fly** database) are not. If you skip over the pieces that are specific to big data engines, you probably will not pass all the quizzes, though you should be able to pass most of them.

Finally, each week in this course concludes with an honors lesson. These lessons mostly focus on using the command line as an interface to the big data SQL engines (instead of the browser-based graphical interface, called Hue). These honors lessons are intended for learners who are interested in the Cloudera Certified Associate (CCA) Data Analyst certification exam. If you would like to pursue this Cloudera certification (which is different from the Coursera certificate you receive from passing this course and specialization), these honors lessons will help prepare you, by helping you work more efficiently (the CCA exam is time-sensitive) and by providing information and practice using resources you might not be familiar with.

USING THE HUE QUERY EDITORS

At this point in the course, you should have the VM set up and running. Throughout the course, you will need to use the VM to follow along with demonstrations and to complete quizzes and assignments. The two SQL engines you'll use in this course, Hive and Impala, are both installed on the VM. And all the data you'll need to query with those engines is preloaded in tables on the VM. To interact with this data and these query engines, you'll be using Hue. Hue is a web browser based analytics works bench that provides a user interface to Hive and Impala. With the VM set up and running, you can open the web browser in the VM, and click the link in the bookmarks toolbar to access Hue. Hue includes a number of different interfaces, many of which you will not use in this course. There are just a few interfaces that you will use.

If you completed the first course in this specialization, you should recall that one of the Hue interfaces is the Table Browser. Click the icon in the upper left corner, then under browsers, click Tables. Here in the Table Browser, you can click Databases to see databases exist. You can see there are databases named default, fly, fun, toy, and wax. You can click the name of the database to see what tables are in it. I'll click fun, you can see that the fun database has tables named card_rank, card_suit, games, and inventory. Then you can click the name of a table to see more details about that table. I'll click games. If you go to the columns tab you can see that this games table has eight columns, id, name, inventor, year, min_age, min_players, max_players, and list_price.

You can also click the Sample tab to see a sample of the data in this table. So the Table Browser in Hue provides a convenient interface for browsing tables, through simple point and click actions. And if you completed the first course in the specialization, you should recall it well. In this course, you'll go beyond simple point and click actions and learn how to run SQL statements to query the tables. Hue has a different interface for that. Notice the big Query button in the top bar in Hue. When you click the right side of this button, a dropdown menu opens. Under Editor in the dropdown menu, you'll see options for Impala and Hive. There are some other options below that, you can ignore those for now. I'll click Impala and this opens the query editor for Impala.

The query for Hive is nearly identical, it says Hive at the top instead of Impala, but besides that it looks the same. All the features I'll describe here are also available in the Hive query editor. When you're in the query editor, you can use this assist panel on the left side to browse the databases and tables. If the assist panel is hidden, you can click to show it. And you should make sure that the SQL mode is active at the top of the assist panel. Click this database icon to make sure that's the active mode. You can use this assist panel to see what databases exist. If you're already in one of the databases you'll have to click this back arrow to go back to the list of all the databases. You can click the name of the data bases to see what tables are in it. If you click the name of a table you'll see the columns in it. And you can click the letter i icon to the right of a table name to see more details about the table.

This is all very similar to what you can do through the table browser I showed earlier in this video. The most important feature of the query editor is this text area in the centre, where you can enter and edit SQL statements then run them. We'll be using this extensively throughout the course. When you click inside this text area, another assist panel opens up on the right side. We will not be using this assist panel at this point in the course, so you can click to hide it for now. Throughout the remainder of this course you want to keep the Hue query editor open in the web browser on your VM.

You'll need to use it in almost every lesson for the rest of the course. So Hue has sequel query editors for both Hive and Impala, for most of this course, it doesn't matter which one you use. Most of the SQL statements you'll learn about will work the same with both Hive and Impala and with most SQL engines too, including relational databases and data warehouse systems. Whenever I show a SQL statement that is not broadly compatible across different engines, I'll be sure to indicate that. Typically, Impala returns query results faster than Hive, so I'd encourage you to use the Impala query editor throughout this course. The Impala query editor is the default one.

So if you click the centre of the big Query in the top bar in Hue, that will take you directly to the Impala query editor. Keep in mind that Hive and Impala are both accessing the same tables with the same data. These are two different engines operating on one set of underlying tables and data. So on the VM when you choose to use the Hive query editor or the Impala query editor, you're simply choosing which SQL engine will run the queries on that shared set of tables. Also a quick comment about terminology, the word database as I've used it in this video, refers to a logical container for a group of tables.

If you're familiar with the concept of a named space, then you can think of a database as the same thing, it's just an abstract container. In this case, a container that holds tables. Within one database the tables all need to have different names. But two different tables can have the same name if they're in different databases. But this word database also has some broader meanings. Any organized collection of data can be called a database. SQL engines in general are often called databases, and one specific instance of a SQL engine is often called a database. To try to resolve this confusion sometimes people use the word schema, to refer specifically to a group of tables. In this course, I'll use the word database not schema, but keep in mind that you might see this word schema used to mean to same thing, a logical container for a group of tables.

RUNNING SQL UTILITY STATEMENTS

Recall from the previous video, that Hue enables you to see what databases exist, switch into a particular database, see what tables are in it, and look at the columns in those tables, all through point and click actions. In this video, I'll show how for each of those tasks, you can write and run a SQL Utility Statement that does the same thing as the point and click action. So if you could do something by pointing and clicking, why would you want to write a SQL statement to do it?

Well, for one, not all SQL interfaces have a graphical user interface like Hue does. Sometimes the only way to perform simple tasks like these is by entering and running SQL statements. Also, an interface like Hue enables you to perform some simple tasks without using SQL. But as you'll see beginning in the next video, you can achieve many more types of tasks by entering and running SQL statements. The first SQL Utility Statement I'll talk about is SHOW DATABASES. This is often the very first statement you would run when connecting to an instance of a SQL engine for the first time. It tells you what databases exist. I'll run this statement in the Impala Query Editor in Hue on the VM to show you what it returns. In the editor, I'll enter SHOW DATABASES and I'll terminate the statement with a semicolon.

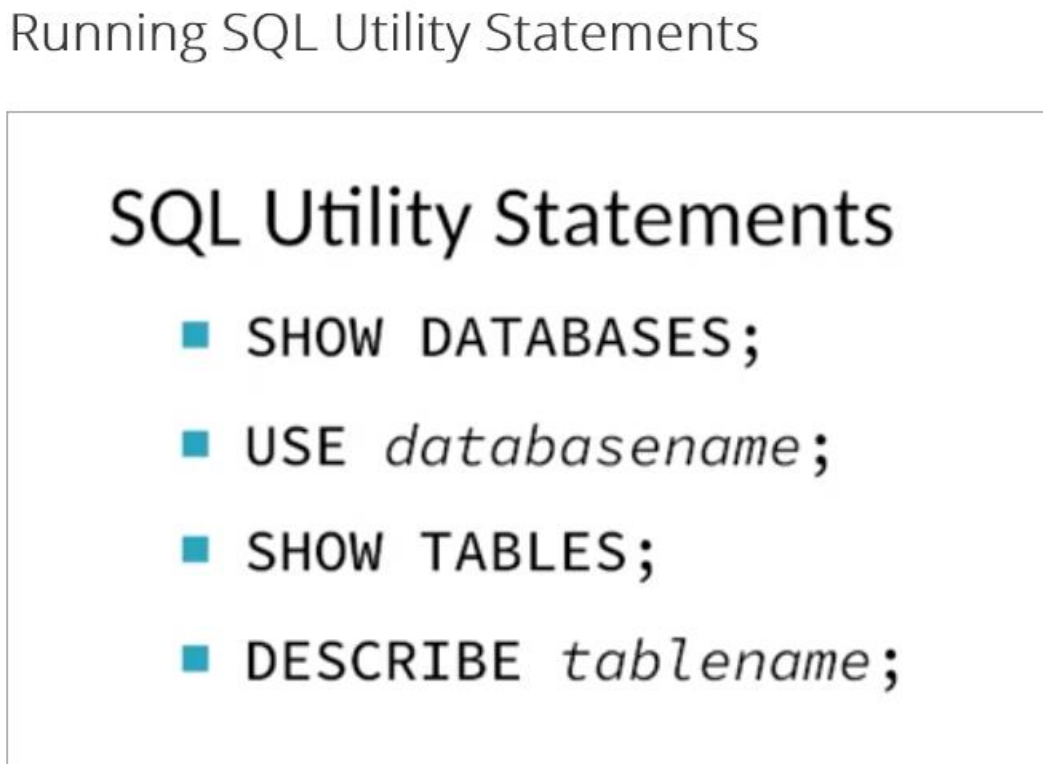
The convention is SQL is to use a semicolon to indicate the end of every statement. But here in Hue, if you're just running a single statement, the semicolon is optional. So you could leave it off. Then I'll click this Execute button to run the statement. You can also use the keyboard shortcut Control, Enter to do this. After the statement runs, the result appears directly below. Running this statement with Impala, you can see the first row of the results list a system database named, Impala builtins. You can safely ignore that. Below that, you can see the actual databases. Each one has a name and optionally a comment. You can see there are databases named default, fly, fun, toy, and wax. When you're using a SQL engine, there is always one particular database that you're connected to. This is called the current database or the active database. When you first log into Hue and open the Hive or Impala Query Editor, the current database is typically the one named default. Other SQL engines also have particular databases that they connect to by default at the start of a new session, but they're not generally named default, and they can vary from user to user.

Usually, if you're going to be working with a particular table, you'll want to set the current database to the database that contains that table. To set which database is the current database, you can run a USE statement. The USE statement is very simple, it's just the keyword USE followed by the name of a database. Hue, which you'll use throughout this course actually does not support the USE statement. Instead, in Hue, you always use point and click actions to set the current database. Just above the editor, there is an active database selector. You can use that to see what the current database is, right now it's default, and to change the current database. I'll select fun, and now the current database is fun.

Alternatively, you can click the name of a database in the assist panel on the left side, and Hue will set that as the current database. In the assist panel, I'll click the back arrow to go back to the list of all the databases, then I'll click the wax database. You can see in the active database selector that wax is now the current database. The current database persists for the duration of your session or until you change it again. I'll change it back to fun. If you're using a SQL interface that lacks a point and click interface for switching databases, then you would instead need to execute a USE statement, like USE fun. Also keep in mind that your selection of the current database only affects the particular session in which it is run. Other users in other sessions have their own current databases. Recall that a database in SQL is just a logical container for a group of tables. So after you see what databases exist and change the current database, often the next step is to see what tables exists in the current database. To do this, run the statement SHOW TABLES. I'll enter and run SHOW TABLES in Hue. The result shows the names of the four tables in the current database, which remember is the fun database.

The tables are card_rank, card_suit, games, and inventory. The final utility statement I'll talk about is the DESCRIBE statement. You can use the DESCRIBE statement to see what columns are in a table. The syntax is simple, following the keyword DESCRIBE, you put the name of the table whose columns you want to see. I know that one of the tables in the fun database is named games. So in Hue, I'll enter and run DESCRIBE games. The result shows that this table has eight columns; id, name, inventor, year, min_age, min_players, max_players, and list_price. Each column also has a datatype and optionally a comment.

You'll learn about data types later in this course. So for now, the column names and the order they're in are what you should pay attention to. So now you've seen how to use SQL Utility Statements to explore and navigate databases and tables. SHOW DATABASES shows you what databases exist, the USE statement changes the current database, SHOW TABLES shows what tables are in the current database, and the DESCRIBE statement shows what columns are in a table. In the next video, you'll see how to run SQL statements to look at the rows of data in a table.


Running SQL Utility Statements

SQL Utility Statements

- `SHOW DATABASES;`
- `USE` *`databasename`*`;`
- `SHOW TABLES;`
- `DESCRIBE` *`tablename`*`;`

RUNNING SQL SELECT STATEMENTS

The Select statement is the most important part of the SQL language. The options for what you can do with a Select statement are so extensive, that Select forms it's own category of SQL statements called queries. In this video, I'll show you how to run some very simple Select statements and view the results. The purpose here is not for you to understand what's possible with the select statement. We have the whole rest of the course to do that. In this video, you should just focus on the mechanics of running select statements and viewing the results. On the VM we're using for this course, you will write and run select statements in Hue, using the Hive or Impala Query Editor. These queries will execute on Hive or Impala, and you'll view the results in Hue. In the Impala Query Editor in Hue, I'll enter and run a simple query to return all the data in a table. The current database is Fun.

I know there's a table named games in this database. To return all the columns and all the rows from this table, I'll enter the query, "Select star from game." The star means all the columns. Notice the editor has some auto-complete features that suggest available database table and column names, and other query syntax. I'll press the "Execute" button, to run this query. The data that's returned by a SQL statement is called the result set or just the result. This result set has all five rows and eight columns from the games table. You can see that this table contains some information about five different board games, Monopoly, Scrabble, Clue, Candy Land, and Risk. You can see the columns ID, name, inventor, year, min age, min players, max players, and list price.

It's important to pause here for a minute to talk about the order of the columns and rows in the result set. The order of the columns in a result set is determined by your query, or by the structure of the table you're querying. There's nothing random about the order of the column. In the example I just showed, the query returned all the columns. So the result set showed them all. Their order from left to right was determined by the structure of the games table, ID, name, inventor and so on. Recall that in the previous video, when I ran the statement, "Describe games" to see what columns were in the games table, that returned the names of the same column, in the same order from top to bottom, ID, name, inventor, and so on. I also have the option to specify in the select statement which columns to return, and what order I want them in. I'll change, "Select star" to "Select name, year, inventor".

Then, when I run this query, the results set has just these three columns in the order I specify, name, year, and inventor, from left to right. So the order of the columns in a result set is deterministic, but the order of the rows is not. When you run a select statement using a distributed SQL engine, the order of the rows in the results set is arbitrary and unpredictable. You could run the exact same query twice on data that has not changed, and get the rows in a different order each time. The result overall will be the same, but the order of the rows might vary. So if you run this query, don't be surprised if the rows you see are in a different order than when I ran it. This is normal and expected when you're using a distributed SQL engine.

But you also shouldn't be surprised if the order of the rows in your results are the same as mine. On the VM for this course, these distributed SQL engines, Hives, and Impala, are not actually distributed across multiple computers or multiple processors. We configured the VM to use just one processor on your computer, and this takes away much of the randomness that causes the rows to get shuffled around. I'll talk more about row order later in the course. But for now, you should just remember that the order of the rows is arbitrary. So at this point in the course, you should have a basic sense of how to enter and run a simple SQL statement using Hive or Impala through Hue. In the next lesson, we'll talk about how Hue is not the only application that provides a SQL interface to Hive and Impala. There are many others. Of course, there are many other SQL engines too. Then in next week's lessons, we'll go beyond the very simple SQL statements I showed in this lesson, and we'll begin to reveal the full extent of what's possible with the select statement.

UNDERSTANDING DIFFERENT SQL INTERFACES

In the previous videos in this lesson, I presented Hue as the SQL interface to Hive and Impala. But actually, Hue is only one of many interfaces to Hive and Impala. Besides Hue, there are a variety of other SQL query tools, sometimes called SQL clients, that can be set up to work with Hive or Impala, as well as other SQL query engines. Like the query editors in Hue, these utilities allow you to enter SQL statements, run them, and see the results. Some of these are Web browser based, others can be installed on Windows or Mac or Linux. Some are command line applications, others have a graphical user interface.

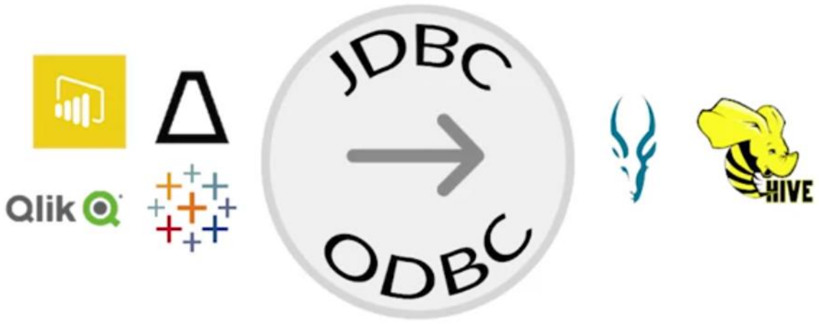
There are too many of these to name. But just as one example, there's an open source one called SQuirreL SQL Client that's been around for a long time. It can be installed on different platforms and can be configured to run SQL statements on an instance of Hive or Impala. Many data analysts today need to do more with the results of a query than just display it in a table. So in the last 20 years or so there has been enormous growth in the use of analytics and business intelligence, or BI software, which can create charts and graphs and maps and other data visualizations based on SQL query results. Again, there are far too many of these to name, but one of the best known ones is Tableau. Many of these applications allow the user to manually enter SQL statements to run on Hive or Impala or other SQL engines.

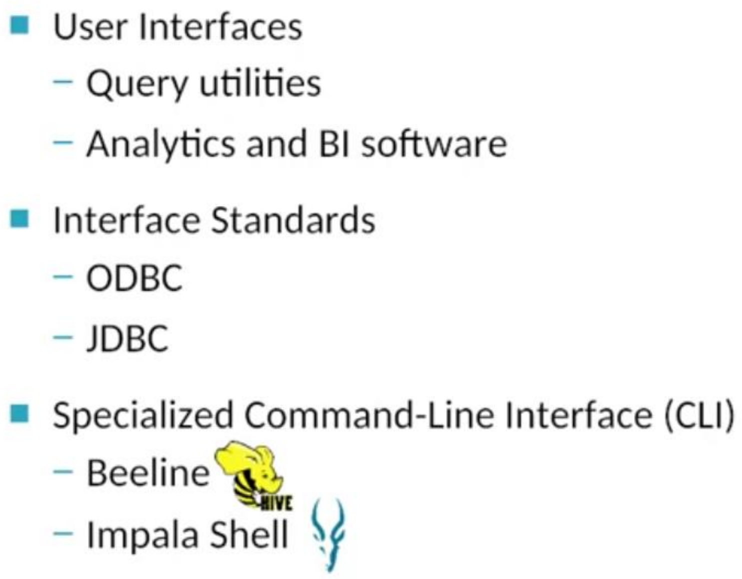
And many of them can also automatically generate SQL statements as you interact with them through a graphical user interface. These applications can be used to create reports, interactive dashboards, and more. All these tools and applications from the most basic SQL clients, through the most sophisticated analytics software need some way to connect to an instance of Hive or Impala to be able to run SQL statements on them. There are a few standard interface protocols that can be used to do this. The two best known of these are ODBC and JDBC.

These are two different interface standards that virtually any software can use to connect to virtually any SQL engine. Both Hive and Impala support both ODBC and JDBC. Different SQL interface applications typically use one or the other of these standards. For example, Tableau uses ODBC, and SQuirreL SQL Client uses JDBC. Also, custom code written by data scientists, data engineers, and developers can use ODBC or JDBC to connect to SQL engines. Scripts written in popular languages like Python and R can use either of these standard interfaces to run SQL queries on Hive or Impala and then process the results. So I mentioned two kinds of user interfaces, query tools and analytics and BI software, and two underlying interface standards, ODBC and JDBC.

They can work with most with any SQL engine, including Hive and Impala. But many SQL engines also have specialized interface applications that are designed to work only with that one SQL engine. Hive and Impala also have that. Hive includes a specialized command line interface, or CLI, called Beeline, and Impala has a specialized CLI called Impala Shell. Both of these are accessed from the command line or the terminal. And both of these are installed on the VM, so they're available for you to use in this course.

In the honors lesson, you can learn how to use Beeline and Impala Shell. But the main point here is that although you'll be using Hue throughout this course, the SQL statements you'll write can be run in numerous different applications. Hue is just an interface, and in this course you shouldn't focus too much on the interface. You should focus on the SQL statements.





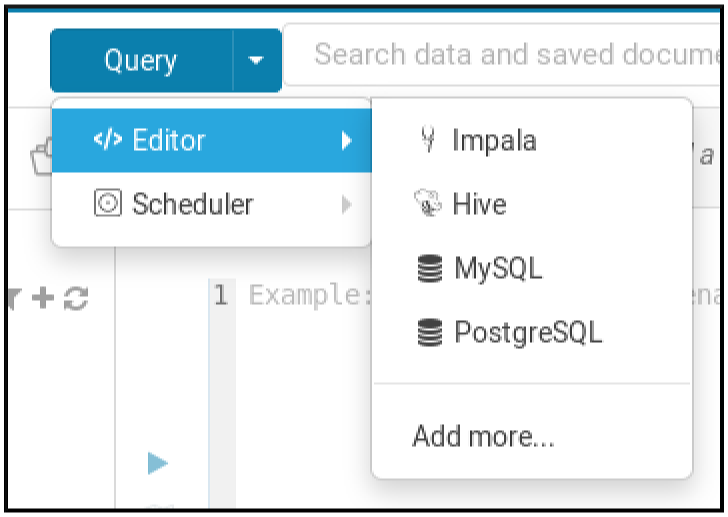
- User Interfaces
  - Query utilities
  - Analytics and BI software
- Interface Standards
  - ODBC
  - JDBC
- Specialized Command-Line Interface (CLI)
  - Beeline
  - Impala Shell

(OPTIONAL) USING OTHER SQL ENGINES

Apache Hive and Apache Impala are open source big data SQL engines that are the primary focus of this course and specialization, but you might be interested in learning or using some other SQL engines, as well.

Two very popular SQL engines are MySQL (pronounced “my ess cue ell”) and PostgreSQL (pronounced “post gress cue ell”). Both are traditional open source relational database management systems (RDBMSs); they cannot work with really big data like Hive and Impala can, but they are often used to store small- to medium-sized data. You can run SELECT queries on that data, just like you can with Hive and Impala.

In the VM, we have installed MySQL and PostgreSQL. All the tables from the **default**, **fun**, and **wax** databases in Hive and Impala have been loaded into tables in MySQL and PostgreSQL so you can query them there. The tables from the **fly** database in Hive and Impala have *not* been loaded into MySQL and PostgreSQL; those are only available to query with Hive or Impala.
Hue has been configured with query editors for these other SQL engines also, so you can practice using them. You can access them from the **Query** drop-down menu. Click the down arrow beside the **Query** button, and choose which editor you want to use.



When you first enter the editor, you need to switch to either the **mydb** database (for MySQL) or the **public** database (for PostgreSQL) by selecting it as the active database. MySQL doesn't really have a default database in Hue. It will not let you run any queries, even those that have no table references, if **default** is still the active database. PostgreSQL will still allow you to run queries, but it will be much easier if you switch databases.

Learning Objectives

- Construct working SELECT statements as a foundation for more advanced statements
- Write expanded SELECT statements that include expressions and functions
- Use the FROM clause to specify the table from which the SELECT statement retrieves data
- Identify rules and conventions regarding keywords and identifiers in SQL
- (Honors) Use Beeline and Impala Shell in non-interactive modes

Introduction

Welcome to week two of analyzing big data with SQL. In the previous week's lessons, you ran some simple SELECT statements, but you did not yet use the SELECT statement to do data analysis. You just use it to do some simple data retrieval. Data retrieval is important and it's definitely something you'll do with the SELECT statement, but what the SELECT statement can do goes far beyond just data retrieval. Data analysis is when you try to answer questions using the data or you tried to discover things in the data like patterns and outliers.

Sometimes the term data mining is used to refer to the practice of discovering things in the data, but I'll stick to the broader more popular term, data analysis. To do data analysis, you need to do more than just to retrieve data, you also need to manipulate or transform data in different ways.

These two terms, to manipulate and transform, refer broadly and generically to any operations performed on data. Later, you'll learn some related terms that have more narrow specific meanings. In this course when I talk about manipulating or transforming data, I'm not talking about changing the existing data in the tables. I'm talking about reading the existing data from the tables essentially making a copy of it and then manipulating the copy, and returning the manipulated copy as the result. So for this entire course, you can consider the data in the tables to be immutable, it cannot be altered or modified in place. Throughout this course you will never alter or modify the data in the tables.

Also, this word manipulate or manipulation has some negative connotations in common use like currency manipulation or psychological manipulation. It can be used to mean falsify or modify unfairly. But in the context of SQL and data analysis, it has no negative connotations, it just means to perform some operations on data to generate a result. So typically, the practice of data analysis is prompted by questions or by the need to make decisions that are informed by the data. In this week of the course and throughout the remaining weeks, I'll teach you how to answer different types of questions using SELECT statements. You'll learn the fundamentals of data analysis with SQL including some rules and conventions, and you'll do some simple data retrieval too.

SQL SELECT Building Blocks

In this video, you'll learn about the building blocks of a SELECT statement called clauses. A SELECT statement is made up of one or more clauses. The next several weeks of the course are structured around these different clauses. The order in which I'll teach them matches their correct order within a SELECT statement. First, you'll learn about the SELECT clause, which specifies what columns should be returned in your query result.

Next, you'll learn about the FROM clause, which specifies where the data you are querying should come from. Then you'll learn about the WHERE clause, which filters the rows of the data based on one or more conditions. Next is the GROUP BY clause and the related topic of aggregation, which can be used to split the data into groups and then reduce each group down to a single value. Then you'll learn about the HAVING clause, which filters the data based on aggregates. Next is the ORDER BY clause, which sorts or arranges the results of a query. Finally, the LIMIT clause, which controls how many rows a query can return.

As you learn about these parts of a SELECT statement, you'll see how the different clauses and combinations of them can enable you to answer different kinds of questions. Most of what I'll teach is applicable to any SQL engine; Hive and Impala, other big data SQL engines, and traditional relational database systems. But there are some differences in the SELECT syntax across the dialect of SQL that these different engines use. I'll teach you about these differences with a particular focus on the SELECT syntax for Hive and Impala.

If you're new to SQL, this tour through the clauses of the SELECT statement should give you a straightforward intuitive introduction to the language. Each week of the course builds on the clauses taught in the previous week. So take your time and make sure you understand each clause and all the related topics before moving on to the next one. If you are already familiar with SQL and you're looking to get proficient with modern distributed SQL engines, then you can proceed more quickly through the course, and look out for the places where I call out the important differences between the SQL dialects.

Introduction to the SELECT List

A SELECT statement begins with the keyword SELECT. The part of the statement starting at the beginning with the keywords SELECT and ending before the keyword FROM is called the SELECT clause. Everything that comes after the keyword SELECT in this clause is called the SELECT list. Recall some simple examples from the lessons in the previous week. In the statement, SELECT star FROM games, the SELECT list is simply the star, the asterisk. In the statement, SELECT name, year, inventor FROM games, the SELECT list is name comma year comma inventor.

It's a list of the names of three specific columns separated by commas. You'll also recall from the previous lessons what the purpose of the SELECT list is. It's to specify what columns should be returned in the result set. In a SELECT list, the asterisk symbol which is universally pronounced "star", has the special meaning all the columns. So when you use the star as your SELECT list, the results will contain all the columns from the table in the same order that they're defined in in the table. If you want to return some but not all of the columns, then instead of using the star, you use a list of the column names separated by commas. When you do this, then the order of the columns in the results set is determined by the order of the column names in the SELECT list.

Always remember that the order of the columns in a result set is deterministic, but the order of the rows is arbitrary. So if you run these statements and the order of the rows in your results set doesn't match what I show, that's perfectly fine. This all seems pretty simple so far. But in this lesson and the next one, you'll learn that there's much more you can do with the SELECT list, going beyond basic data retrieval and into data manipulation and data analysis. In all of this, the SELECT list is really important.

So I'm going to take time to cover it thoroughly. During these lessons about the SELECT list, you'll also learn about some topics that you'll use again later in other clauses. With many SQL engines, the SELECT clause is actually the only part of a SELECT statement that's strictly required. With these engines, you can use SELECT without a FROM clause, but there's a catch. This will only work if you include only literal values or literals and no column references in the SELECT list. First, here are a couple of examples that will not work. SELECT star with no FROM clause. Star means all the columns, but there is no table for the columns to come from, so this will fail. SELECT name, year, inventor with no FROM clause.

This again will fail because name, year, and inventor are column references, but the source of these columns is not specified. Here are a couple of examples that will work. SELECT 42. This returns a single row and a single column containing the integer number 42. Because 42 is a literal value not a reference to a column in a table, the statement will run successfully in many SQL engines. SELECT 'foo', 'bar'. The quoted strings foo and bar here are both literal string values.

The statement returns a single row with two columns containing the three character strings, foo and bar. The single quotes around these strings tell the SQL engine to interpret them as literal strings, not column references. In both of these examples that have no FROM clause, the resultant has just one row. Whenever you omit the FROM clause, the SQL engine acts like you're querying an imaginary table with one row and no columns. Since there are no columns, you can reference, the only thing you can include in the SELECT list are literals. You can also mix column references and literal values in the SELECT list. But when you do this, you need a FROM clause to specify what table the columns should come from.

For example, SELECT 'Board Game', name, list_price FROM games. The single quotes around Board Game mean that it's a literal string, but name and list_price with no quotes around them are both column references. The result of this statement has five rows, that's the number of rows in the games table, and it includes the name and list_price columns from the games table. It also has the literal string value Board Game in the first column repeated in every row.
This is what happens when you use literal values in the SELECT list with a FROM clause. The number of rows in the result is controlled by the table in the FROM clause and the literal value is repeated in all these rows. In the examples here, I used single quotes to make literal strings in the SELECT list, and no quotes around column names in the SELECT list. In a later lesson, you'll learn more about the rules and conventions for quoting literal strings and column references, and you'll learn about how different SQL engines do this differently. For now, just remember literal strings are enclosed in single quotes and column references are just bare words with no quotes around them.

This is the convention I'll use in this course, and it's a convention that works with all the major SQL engines. So in this video, I showed how you can use column references and literal values in the SELECT list, as well as the asterisk symbol meaning all the columns. But those are not the only things you can include in the SELECT list, and in the next video, I'll introduce expressions and show how you can use those in the SELECT list. To successfully answer the in-video questions and to complete the quizzes in this lesson and in upcoming lessons, you will need to write and run SELECT statements.

You'll need to run these statements in the VM, in the Hue Query Editor or perhaps in Beeline or Impala shell, and then use the results to get the right answers. I'll start by asking you to query some of the smaller tables in the VM which are in the default database and the toy and fun databases. The tables in these databases have just a few rows and a few columns. Then in later lessons, I'll ask you to query the table in the wax database which is a little bit bigger, then the tables in the fly database, some of which are much, much bigger. One of the tables there has tens of millions of rows. You can also go ahead and explore these tables on your own and try running various SELECT statements on them but for now, you should not yet try to query the tables in the fly database because I have not yet introduced the clauses you'll need to work with very large tables.

EXPRESSIONS AND OPERATORS

In the previous video, you learned that a select list can include column references, literal values, and the asterisk symbol, which means all the columns. In this video, you'll learn how the select list can also include expressions. An expression in SQL is a combination of literal values, column references, operators, and functions. I'll demonstrate this with a simple example. The games table in the fun database has a column named list_price.

Say I wanted to return the names of these games and their list prices, but with a five dollar shipping fee added to the price. To do this, I would use the query SELECT name, list_price plus five FROM games. Then in the results set, the first column is just the names of the games, but the second column gives the values in the list_price column with five added to each one. $19.99 plus $5 is $24.99, 17.99 plus 5 is 22.99, and so on. In this SELECT list, name is just a column reference, but list_price plus five is an expression. It consists of a column reference, list_price, an operator, the plus sign, and a literal value, five.

When the SQL engine evaluates the expression, it returns a result column in which the value in each row is the value of list_price in that row plus five. Here's another example of an expression. This one uses a function instead of an operator. SELECT name, round list_price FROM games. The round function rounds the decimal numbers to the nearest whole number. So in this case, it rounds each list_price to the nearest dollar, 19.99 rounds to 20, 17.99 rounds to 18, and so on. The SQL engine applies the round function to the value of list_price in each row. These example expressions were very simple, but expressions in SQL can be arbitrarily complex so long as they're composed of literal values, column references, operators, and functions put together in a valid way. Round is an example of a built-in function. You'll learn more about built-in functions in the next lesson. But in this lesson, I'll talk more about operators like the plus sign in the first example. The plus sign is of course the addition operator.

It adds together numeric values. This is just one of many operators in SQL. There are too many for me to cover them all right now, so in this lesson, I'll just cover the common arithmetic operators, and I'll cover other types of operators later in the course. In addition to the plus sign, the other common arithmetic operators are the minus sign for subtraction, the asterisk for multiplication, the front slash for division, and the percent sign which is the modulo operator. Plus and minus can both be used as unary operators or binary operators. Unary means having only one operand, or only one argument. Here's an example of the minus sign being used as a unary operator. The minus sign before list_price flips the sign of the numbers in this column, and since all these numbers are positive, they're all changed to negative in the results set. There's only one operand in this expression list_price, and the operator, the minus sign comes before it.

You can also use the plus sign in this way as a unary operator before a numeric operand, but it just returns the numeric column with its sign unchanged. So there's no point in using it. All four of the common arithmetic operators can be used as binary operators, meaning that there's an operand on both sides of the operator: the left and right sides. With binary numeric operators, the operand on both sides can be a literal numeric value, or a reference to a numeric column. So for example, using the games table, you could use any of these expressions: two plus five, max_players minus min_players, list_price divided by 2, 1.05 times list_price, list_price modulo one.

are all valid expressions. You can also use an expression on one or both sides of the operand, like list_price divided by two times 1.05. That takes list_price and divides it by two, and then take that result and multiplies it by 1.05. But when you're using expressions like this that have multiple operations, be mindful of the rules of order of operations. See the reading for more information about that. Also, a warning about the division operator in SQL.

In Hive, and Impala, and MySQL, and some other SQL engines, the division operator, the front slash, always performs decimal division. That means if both operands are integer numbers and the numerator on the left side does not divide evenly in whole numbers into the denominator on the right side, then the result will be a decimal number. For example, 5 divided by 2 is 2.5. That's what Hive, and Impala, and MySQL all return, but not all SQL engines work this way.

Some engines like PostgreSQL will perform integer division in this case and will return two instead of two and a half. See the reading for more information about that. The reading also explains what the modulo operator does. An easy way to try out these different operators is to use them with literal operands in a SELECT statement that has no FROM clause. As I mentioned in the previous video, many SQL engines do not require a FROM clause, and this is especially useful when you're trying out operators and functions. This works with Hive, Impala, MySQL, PostgreSQL, and others. There are other SQL engines that do not allow you to omit the FROM clause, and in these, there's usually some other technique you can use, like using a dummy table, or some other special syntax in the FROM clause. If you're using one of these other engines, search for information about how to select without a table with the specific engine you're using. The example expressions I showed in this video were very simple, but expressions in SQL can be arbitrarily complex, so long as they're composed of literal values, column references, operators, and functions, and they're put together in a valid way. Throughout the remainder of the course, you'll have a chance to practice writing progressively more complex expressions.

ORDER OF OPERATIONS

When using arithmetic expressions, care must be taken to ensure they're evaluated the way you want them to be. For example, someone might think this expression:

2 + 3 * 3 * 5

would be evaluated left to right. First 2 + 3 is 5, then multiply that by 3 to get 15, and finally multiply that by 5 to get 75. Some four-function calculators (with only the four basic operations) still work this way.

However, operators typically are given a conventional of order precedence that you probably have seen before. First, multiplication and division, including the modulo operator, are evaluated from left to right. Then addition and subtraction are evaluated from left to right. You can add parentheses to the expression to override this, so that addition or subtraction can be done first.
Note that functions are evaluated with the same precedence as parentheses. (This is easy to remember because function calls also use parentheses.)

Here are some examples:

Expression	**Evaluation**	**Explanation**
2 + 3 * 3 * 5	**2 + 9 * 5 = 2 + 45 = 47**	The multiplication 3 * 3 * 5 would be done first, so the expression becomes 2 + 45, which is 47.
(2 + 3) * 3 * 5	**5 * 3 * 5 = 15 * 5 = 75**	The parentheses force the addition be done first: 2 + 3 is 5, so the expression becomes 5 * 3 * 5, which is 75.
(2 + 3 * 3) * 5	**(2 + 9) * 5 = 11 * 5 = 55**	The expression in the parentheses is done first, but within that expression, the multiplication is done first. So 2 + 3 * 3 becomes 2 + 9, still in parentheses, which is 11, and the 11 is multiplied by 5 to get 55.
SUM(numbers) * 3 * 5 (where **numbers** is a	**5 * 3 * 5 = 15 * 5 = 75**	The **SUM** function adds all the values in the column, which in this case has only

Expression	Evaluation	Explanation
column with two values, **2** and **3**)		two values, 2 and 3. The sum is 5 and the expression becomes 5 * 3 * 5.

Division is unlike the other three basic arithmetic operations (addition, subtraction, and multiplication), because while you can add, subtract, or multiply integers and still get another integer, you can't always divide two integers and get back an integer. Because of that, many programming languages, including SQL, offer more than one division operator.

There are three approaches that one can take with division. For an example, think about separating 17 items into 5 groups (as equally as possible). The three approaches you can take are:

1. Use ***decimal division.*** That is, the result of the division can be something other than an integer. For example, 17 divided by 5 is 3.4. If the items can be split, like sharing 17 biscuits with 5 people, this could work—each person gets 3 whole biscuits and 0.2 of another. But if the items are people (maybe you're trying to transport 17 people in 5 cars) this doesn't quite work.
2. Use ***integer division.*** That is, you only care about the whole number part of the division. In the biscuit example, you would have 3 items for each of the 5 groups. This works if you're sharing 17 biscuits with 5 people, and anything left over can be given to your dog. With the car example, each car would have at least 3 people, but some will have to have more (or you leave a couple of people behind).
3. Use ***modulo*** (or ***modulus***)***.*** That is, use only the ***remainder*** part of the division. This is what you would use if you want to know how many biscuits the dog will get, or how many of the 5 cars will need to take one extra person.

The operator you use in an expression will depend partly on which of these approaches you intend, but it also might depend on which query engine you are using. There are three operators used for these three types of division:

/ DIV %

The operator for dividing decimal operands is always **/**, the forward slash (sometimes called a ***solidus*** or ***stroke***). For example:
25.2 / 3.2 = 7.875
However, this gets more complicated when both of the operands are integers! Many engines, including Hive, Impala, and MySQL will always consider this decimal division, regardless of whether the operands are decimal types or integer types.
Hive, Impala, MySQL: **17 / 5 = 3.4**
Other engines, including PostgreSQL and Presto, will use ***integer division*** if both operands are integers. If you want to divide integers using decimal division in these engines, you must make at least one of the operands a decimal type, for example by adding **.0** to the end of the literal integer.
PostgreSQL, Presto: **17 / 5 = 3**
17.0 / 5 = 3.4
17 / 5.0 = 3.4
(You will see another way to convert an integer into a decimal type later in this course.)
For the engines that always perform decimal division using the **/** operator, even when the operands are integers, you can use the **DIV** operator to perform integer division.
Hive, Impala, MySQL: **17 DIV 5 = 3**
If you attempt to use **DIV** with non-integers, you will get an error message.
Hive, Impala, MySQL: **17.0 DIV 5 = [ERROR]**
Finally, modulo is done using the **%** operator. This can be confusing, because you are probably used to interpreting this as "percent" (or "out of 100"). But many computer languages use this as the modulo operator:

17 % 5 = 2
25 % 3 = 1
72 % 8 = 0

Notice that modulo is ***cyclical***, and the only possible results of ***a* % *b*** are numbers from **0** to **b - 1**.

3 % 3 = 0
4 % 3 = 1

5 % 3 = 2
6 % 3 = 0
7 % 3 = 1
…

Using % with a non-integer probably should not be needed often, but it is possible—the result is the remainder after subtracting (or adding if only one operand is negative) as many of the divisor as possible, without having a negative remainder. That is, if $n = q * x + r$ where x is an integer and $0 \leq r < q$, then $n \% q = r$. For example,
25.2 = 3.2 * 7 + 2.8
so
25.2 % 3.2 = 2.8

PRACTICE

Here are a few examples for you to try. Try running **SELECT** statements in the VM for these, using Impala or PostgreSQL (or both). Click here for the answers.

A. Use ***decimal division*** for these, if possible:

Divide 38.5 by 8

Divide 29 by 5

B. Use ***integer division*** for these, if possible:

Divide 38.5 by 8

Divide 29 by 5

C. Use ***modulo*** for these, if possible:

Find 38.5 modulo 8

Find 29 modulo 5

DATA TYPES

In the previous lesson, I showed examples of some simple expressions. Like 2 plus five, max_players minus min_players, and list_price divided by 2. These expressions use arithmetic operators with numeric operands. In other words, the literal values and the columns in these expressions are all numeric. If you try to use arithmetic operators with other types of literal values and columns like character strings, you will have problems. For example, you should not try to add five to the literal character string "hello." You should not try to divide the name "column" in the games table by two, because name is a character string column. You should not try to take the negative of the name "column". You should not try to multiply the character string column inventor by the numeric column list_price.

None of these expressions will work. Most SQL engines will throw an error if you try to use invalid expressions like this in a select statement. Some others like MySQL will not necessarily throw errors, but will return unexpected results. So using arithmetic operators forces us to think about the datatype of columns and a literal values. Datatypes in SQL is a very rich topic. I will not try to cover it in full detail here, but I will describe some of the fundamentals. In this course, every column and literal value you'll work with, will fall into two high-level categories of data types, numeric and character.

I'll first talk about numeric data types. Within the numeric category, there are a couple of subcategories. First, there are datatypes for integer numbers. These are whole numbers, positive or negative with no decimal parts. For example, in the games table, the min_age, min_players, and max_player's columns are all integer columns. Also SQL engines interpret literal numeric values as integers, if they do not have any decimal part. So for example, just the bare number five in an expression would be interpreted as an integer.

Within this integer subcategory, there are some specific integer data types that can represent different ranges of numbers. Roughly speaking, numbers up to a couple a 100 can be represented with the tiny int data type, into the thousands with the small int type, millions with the integer or int type, and billions in beyond with the big int type. It's most efficient to use the smallest datatype that will fit the range of numbers you have, that's why there are different integer types for efficiency. But the specifics of these different integer types are beyond the scope of this course. They're covered in the next course, it's part of this specialization. The other subcategory of numeric types is decimal types. These can represent numbers that have a decimal part, a fractional part.

The list price column in the games table is an example of a decimal column. Also, SQL engines interpret literal numeric values as decimals, if they have a decimal part. So for example, the literal number 2.5 in an expression would be interpreted as a decimal number. Within this decimal subcategory, there are specific types named decimal, float, and double, that can store different sizes of both fixed precision, decimal numbers, and floating point decimal numbers. But the differences between these are beyond the scope of this course. Some SQL engines, notably MySQL, can support both signed and unsigned numeric data types. Signed means that the datatype has a built-in plus or minus sign before the number. So it can represent both positive and negative numbers.

Unsigned means it can only represent positive numbers. But all the major open-source distributed SQL engines, Hive, Impala, Presto, Drill, only support signed numeric types where all the numbers can be positive or negative. So that's all you need to know about the numeric category of data types. The other high level category is character data types, which can represent strings of characters. In the games table, the name and inventor columns are both character string columns. Of course you can use literal character strings in expressions, by enclosing characters in quotes.

Within this character category of datatypes, there are some specific data types that can represent different lengths of character strings. There are types names string, char, and varchar. But for this course, you do not need to understand the differences between them. The specific type that Hive and Impala most often use for character strings, is the one named string. Also, be aware that the word string is often used in a general sense to mean any character string datatype, not just the specific datatype named string. There are some additional data types beyond those I mentioned here, including boolean types, date and time types, and complex or nested types. I'll mention some of those later in the course. Others are introduced in later courses that are a part of this specialization.

Column Aliases

In this video, I'll show you how to control the names of the columns in result sets. So as you've been running select statements in Hue, or perhaps in Beeline or Impala shell, you've probably seen that results set has a header at the top, giving the names of the columns in the result set. When the select list contains only column references, like in this example, then the result set column names are unsurprisingly just the same column names you used in the select list. But if you include literal values in the select list, those don't have names. They're just literal values. So the SQL engine makes up names for the resulting columns.

For example, in the Impala Query Editor, if I add a third item to this select list, which is just the literal number five, I'll add it between name and list price, and I execute the query, the name of the resulting column is five. So Impala names literal value columns using the literal value itself. Now I will also modify the select list to change the column reference list price into an expression, list price plus five. When I execute this query, now the name of the third column in the results set is listed price plus five. Impala names the columns generated by an expression using the expression. The behavior of Hive though is different. I'll copy this select statement, and go to the Hive Query Editor. I'll paste the statement here, and make sure the current database is fun and execute it. Hive gives names to unnamed columns in a different way. It names them _c, then a number indicating which column it is. The leftmost column is zero, the next one is one and so on. The two unnamed columns in this statement are the second and third from the left. So they're numbered one and two.

If I reorder the select list, to put the literal value five first before the name column, then I re-run this statement, now this column containing the five's is the leftmost column so it's named _c0. So that's how Impala and Hive name the unnamed columns in a result set. Some other SQL engines do it other ways, and the exact naming behavior for Hive and Impala also might vary if you're using some different client instead of Hue or Beeline or Impala shell. Fortunately, there is a way to control the names of the columns in a result set. You can do this in a select statement by using Column Aliases. I'll return to the Impala Query Editor, and recall the previous query from the Query History.

After the literal value five in the select list, I'll add space AS space shipping_fee. After the expression list price plus five, I'll add space AS space “price with shipping”, and I'll execute the query. Now, the names of the columns in the results set are these names I specified in the select list. These names are called column aliases. You can also use column aliases with column references. The first column here name, already has a name, but I can use a column alias to give it a different name in the results set. In the select list after name, I'll add space AS space game_name. Then, after I execute the query, the first column in the result set is named game name. With most SQL engines, the AS keyword before a column alias is optional.

So I can remove the AS before each alias and execute the query again and it succeeds and gives the same result. But using the AS keyword helps make select statements more easily readable. So I will typically include the AS keyword before column aliases throughout this course. There are some rules about what you can use as a column alias. I'll explain that in detail later in the course. But for now, you can just remember a few simple rules. You should use only letters, digits, and underscores in column aliases. You should not use only digits. Also, you should not use words that have special meaning in SQL as the aliases. For instance, you should not use the word select as a column alias. You'll learn some more details about this in a later lesson.

BUILT-IN FUNCTIONS

In an earlier lesson, I described how an expression in SQL is a combination of literal values, column references, operators, and functions, and I showed some examples of simple expressions including one that used the round function. The round function rounds decimal numbers to the nearest whole number. In this example, it rounds each list price to the nearest dollar. A function is invoked in a SQL statement by using the function name followed by a pair of parentheses enclosing the arguments or parameters to the function. When there are two or more arguments, they're separated by commas. Function names in SQL are typically not case sensitive, and by convention we write them in all lowercase. The round function is just one of many functions in SQL.

Functions like round are often called built-in functions because they're built right into the SQL engine. Different SQL engines offer different sets of built-in functions, numbering anywhere from about 100 after maybe several 100 different functions depending upon the engine. In this video, I'll cover some common mathematical functions, most of which are built into all the major SQL engines. Round is one of the most basic mathematical functions. Rounding to the nearest whole number is an elementary concept, but it's a bit more complicated than you might imagine. When you use the round function with just one argument, then it rounds to the nearest whole number, to the nearest integer.

So round 19.37 returns 19. But you can also use the round function with two arguments. The second one specifies how many decimal places to round two. So round with the first argument 19.37 and the second argument one, we'll round to one decimal place and return 19.4. Recall that, an easy way to try out these built-in functions, is to use them with literal arguments in a select statement that has no FROM clause, just the same way you tried out arithmetic operators in an earlier lesson. In addition to round, there are two related functions, floor and ceil for ceiling. Floor rounds down to the nearest integer value and ceil rounds up to the nearest integer value. So for example, ceil 19.37 returns 20, taking the ceiling of 19 point anything will return 20, and taking the floor of 19 point anything will return 19.

So whereas round we'll round up or down depending on whether the number is closer to the integer above or below, floor will always round to the nearest integer below, and ceil will always round to the nearest integer above. Another note about the round function. With positive numbers, if the number you're rounding is exactly in the middle, equally far from the rounded value above and the one below, then the round function will round to the one above. For example, 4.5 which is right in the middle between four and five, rounds to five. But with negative numbers, it will round to the one below. For example, negative 4.5 rounds to negative five. So in the case where a value is equally far from the one above and the one below, it's always rounded to one that's farther from zero, positive up, negative down.

Two other common mathematical functions are ABS and SQRT. ABS returns the absolute value of a number, SQRT returns the square root of a number. Both of these take just one argument. There is also the pow or power function, which returns the first argument raised to the power of the second argument. Raising a number to a power is also called exponentiation. Some SQL dialects and many programming languages support an exponentiation operator. Often it's the caret symbol or a double asterisk. But Hive and Impala and many other SQL dialects, do not have an exponentiation operator. Instead, you use the pow or power function for exponentiation. With most major SQL engines, including Hive or Impala, you can use either pow or power, but some SQL engines might only support one or the other.

Whatever you do, just do not try to use the caret operator for exponentiation in Hive or Impala. This operator does something completely different in those engines called the bitwise exclusive OR, that's beyond the scope of this course. The rand or random function, returns a pseudo-random decimal number between zero and one. If you're looking for a pseudo-random numbers in a different range, you can use the arithmetic operators to shift and scale the output, and you can round it using the round or floor or ceil function. For example, to get a pseudo-random integer between one and 10, you could write an expression that takes the ceiling of rand times 10.

You can see from this example, that the arguments to a function can themselves be expressions not just column references and literals. The rand or random function does not require any arguments but with many SQL engines, you can supply a seed value to control whether its output is predictable or not. Different SQL engines do this in different ways, the details are beyond the scope of this course. As for the two different names, rand and random, some SQL engines like Hive and MySQL will only recognize rand. Others like PostgreSQL will only recognize random and some like Impala will recognize either and that's the case with many built-in functions. There's a great deal of inconsistency between different SQL dialects, so you should always test functions to check whether they work with the particular SQL engine you're using.

There are many more mathematical functions including trigonometric and logarithmic functions. The details of those are beyond the scope of this course, but the principles of using them are the same. There are also other types of built-in functions, like string functions for working with character strings. See the reading to learn about those. Also, notice that the built-in functions I've talked about so far, return a column with the same number of rows as the table you're querying from. In other words, they do not reshape the data. With these functions, when you use them in a select list and you have a from clause, then the results that we'll always have the same number of rows as the table in the FROM clause.

When you use them with literal arguments with no FROM clause, then the result always has just one row. But you'll see later in the course, that there is another type of function that does reshape the data.

Common String Functions

There are many string functions available for use in SQL statements. These are useful for working with text or character string data types. The following list is not exhaustive, but it does present some of the more common ones you might want to use.

Unless otherwise noted, the function takes a string argument and returns a string. Note the bottom square box character (␣) is sometimes used to represent whitespace, which could be, for example, a space or a tab character. For some of these, it would be difficult to see the effect of the function if regular spaces are used here.

length(str)
This returns an integer value equal to the number of characters in the string argument **str**.

Notes:
The name of this function is different, depending on the SQL engine you're using. For example, some engines use **len(str)** or **char_len(str)**. (The other functions described below have the same name across all the major SQL engines.)
For Apache Hive and Apache Impala, use the **length** function; it works as described here.

Some SQL engines have functions that are similar to **length** as described above, but that return the number of bytes or other units of information that are required to store a character string. If you're using some other SQL engine besides Hive or Impala, check the documentation to be sure you understand what the length function returns and to see what other similar functions are available.
Examples:
length('Common String Functions') = 23
length(' Common String Functions ') = 25
length('') = 0
reverse(str)

This returns the characters within the string argument **str**, but in the reverse order. Try it with your favorite palindrome!

Examples:
reverse('Common String Functions') = 'snoitcnuF gnirtS nommoC'
reverse('never odd or even') = 'neve ro ddo reven'
upper(str), lower(str)

These return the string **str** but with all characters converted either to uppercase or lowercase. These can be useful for doing case-insensitive string comparisons (by converting the string to be compared to one case, for example, **WHERE upper(fname) = 'BOB'** or **WHERE lower(fname) = 'bob')**.

Examples:
upper('Common String Functions') = 'COMMON STRING FUNCTIONS'
lower('Common String Functions') = 'common string functions'
trim(str), ltrim(str), rtrim(str)

These remove whitespace at the ends of the argument **str**. You can choose to remove only leading whitespace (**ltrim** for left trim), trailing whitespace (**rtrim** for right trim), or both (**trim**). If there is no whitespace on the specified end, the string is unchanged.

Examples:
trim('␣Common String Functions␣␣␣') = 'Common String Functions'
ltrim('␣Common String Functions␣␣␣') = 'Common String Functions␣␣␣'
rtrim('␣Common String Functions␣␣␣') = '␣Common String Functions'
ltrim('Common String Functions␣␣␣') = 'Common String Functions␣␣␣'
rtrim('␣Common String Functions') = '␣Common String Functions'
lpad(str, n, padstr), rpad(str, n, padstr)

These functions take a string **str** and an integer **n** and return a string of length **n**. If the original string **str** is shorter than **n** characters, the returned string will be **str** with characters from **padstr** added at the left (**lpad**) or the right (**rpad**) to make it length **n**. (This is called ***padding*** the string, and is the opposite of trimming.) These functions are often used to add zeros to the left or right of numbers that are represented in strings (this is called ***zero-padding***). If necessary, the pad string will be repeated. If the length of **str** is longer, however, the function will return a truncated version of the string. Truncated characters will be taken from the right, regardless of which function you specify.

Examples:
lpad('.50', 4, '0') = '0.50'
rpad('0.5', 4, '0') = '0.50'
rpad('Common', 13, ' String') = 'Common String'
rpad('Common', 17, ' String') = 'Common String Str'
lpad('Common', 17, ' String') = ' String StrCommon'
rpad('Common String', 6, ' Function') = 'Common'
lpad('Common String', 6, ' Function') = 'Common'
substring(str, index, max_length)

This function takes a string and two integers, and returns a portion of the original string. The argument **index** indicates where to start the substring (indexing the original string **str** starting at 1) and **max_length** is how many characters to include (though it might be fewer, if the end of the original string is reached). With many SQL engines, you can also use **substr** which is an alias for **substring**.

Examples:
substring('Common String',1,6) = 'Common'
substring('Common String',8,3) = 'Str'
substring('Common String',8,6) = 'String'
substring('Common String',8,10) = 'String'
concat(str1, str2[, str3, …]), concat_ws(sep, str1, str2[, str3, …])

These functions ***concatenate*** strings—that is, they put them together into a single string. The ***ws*** in **concat_ws** stands for "with separator," the first argument in that case is placed between each pair of strings. In both cases, the arguments are concatenated in the order given.
Notes:
Both **concat** and **concat_ws** must include at least two strings to concatenate. They can take more than two, as well.

Some SQL engines have an ***operator*** for string concatenation, usually **+** or **||**. However, Hive and Impala do ***not*** have concatenation operators; one of these functions must be used.

Examples:
concat('Common','String') = 'CommonString'
concat_ws(' ','Common','String') = 'Common String'
concat('Common','String',' Functions') = 'CommonString Functions'
concat_ws(' ','Common','String','Functions') = 'Common String Functions'
concat_ws(', ','Common','String','Functions') = 'Common, String, Functions'

Non-ASCII characters

Note that the string functions in different SQL engines can differ in their handling of non-ASCII characters. For example: In most SQL engines, **upper('é')** returns **É**, but in others it might return **é** or throw an error. You should test or consult the documentation to see how this works.

Other String Functions

Many more string functions are available in most SQL engines. For example, there are functions for splitting strings, extracting parts of strings, and finding and replacing specific characters or substrings within strings. If you are interested in them, check the documentation of the SQL engine you are using (probably under "String Functions").

DATA TYPE CONVERSION

In an earlier video, I talked about how arithmetic operators in SQL expect their operands you have numeric datatypes. So for example, you should not try to write an expression like 'hello' plus five. Functions in SQL also expect their arguments to have certain data types. Some arguments of some functions are expected to be numeric, others are expected to be character strings. In some cases, you might need to convert one type of data to another, so you can apply an operator or function.

For example, you might have some numeric values and some string values that you want to concatenate together into strings, or you might have a string column whose values actually represent numbers. Converting one data type to another is called type conversion or casting. Some SQL engines do most type conversions automatically, this is called implicit casting. Other engines require you to do type conversion manually, this is called explicit casting. Here are some examples to demonstrate this.

The games table in the fun database has information about five board games including, the name of the game in the name column, and the minimum player age in the min age column. Say you want to concatenate these two columns together along with some literal string values to create a sentence like, blank game is more players age blank or older. You can do this with an expression like this using the concat function. But the concat function expects all its arguments to have a string data type, and the min age column has an integer data type. Some SQL engines including HIV will implicitly cast min age as a string column. The query will run successfully and will return the expected results. But with other SQL engines including Impala, this query will fail. With engines like that, you need to modify the SQL statement to explicitly cast min_age as a string column. To do this, you use the cast function. Cast is a special function that you use in a slightly different way than the other built-in functions. You enclose the column or literal value or expression that you want to convert as the argument to the cast function.

Then before the closing parenthesis, you put a space the keyword "AS" another space, and the name of the data type you want to convert to. In this example, I need to cast min_age as a string column. So it's cast min_age AS STRING. With the statement modified to include this explicit cast, it will now run successfully in Impala and return the expected result. Here's another example. In the games table there's a column named year, representing the year that each game was invented. The values in the year column are all four digit years, but it's actually a string of column. You can see this by looking at the column information a queue or by running the utility statement to describe games. It's pretty common to have numbers in string columns like this.

Columns containing zip codes or numeric postal codes are a classic example of this. Say you wanted to add some number to each of the years in the year column. For that, the column would need to be converted to a numeric type. Since the years are whole numbers, it would be the integer type. If you're using a SQL engine like Impala that requires explicit casting, then you can do this by changing the column reference year to cast year AS INT. The explicit casting syntax described in this video, works in HIV, Impala, and many other SQL engines.

Some other engines support different syntax, for instance, there's a function named convert in some SQL dialects.

So if this cast value as type syntax doesn't work with a specific engineer using, search for how to convert datatypes with that engine. It's a good practice to use explicit type conversion even if the SQL engine you're using can do it implicitly. Explicit cast can make the queries more portable between different SQL engines, they can make the queries run more efficiently, and they can help you to avoid unexpected results.   In an earlier video, I talked about how arithmetic operators in SQL expect their operands you have numeric datatypes. So for example, you should not try to write an expression like 'hello' plus five. Functions in SQL also expect their arguments to have certain data types.

Some arguments of some functions are expected to be numeric, others are expected to be character strings. In some cases, you might need to convert one type of data to another, so you can apply an operator or function. For example, you might have some numeric values and some string values that you want to concatenate together into strings, or you might have a string column whose values actually represent numbers. Converting one data type to another is called type conversion or casting. Some SQL engines do most type conversions automatically, this is called implicit casting. Other engines require you to do type conversion manually, this is called explicit casting. Here are some examples to demonstrate this. The games table in the fun database has information about five board games including, the name of the game in the name column, and the minimum player age in the min age column. Say you want to concatenate these two columns together along with some literal string values to create a sentence like, blank game is more players age blank or older. You can do this with an expression like this using the concat function. But the concat function expects all its arguments to have a string data type, and the min age column has an integer data type.

Some SQL engines including HIV will implicitly cast min age as a string column. The query will run successfully and will return the expected results. But with other SQL engines including Impala, this query will fail. With engines like that, you need to modify the SQL statement to explicitly cast min_age as a string column. To do this, you use the cast function. Cast is a special function that you use in a slightly different way than the other built-in functions. You enclose the column or literal value or expression that you want to convert as the argument to the cast function. Then before the closing parenthesis, you put a space the keyword "AS" another space, and the name of the data type you want to convert to. In this example, I need to cast min_age as a string column. So it's cast min_age AS STRING. With the statement modified to include this explicit cast, it will now run successfully in Impala and return the expected result.

Here's another example. In the games table there's a column named year, representing the year that each game was invented. The values in the year column are all four digit years, but it's actually a string of column. You can see this by looking at the column information a queue or by running the utility statement to describe games. It's pretty common to have numbers in string columns like this. Columns containing zip codes or numeric postal codes are a classic example of this. Say you wanted to add some number to each of the years in the year column. For that, the column would need to be converted to a numeric type. Since the years are whole numbers, it would be the integer type. If you're using a SQL engine like Impala that requires explicit casting, then you can do this by changing the column reference year to cast year AS INT.

The explicit casting syntax described in this video, works in HIV, Impala, and many other SQL engines. Some other engines support different syntax, for instance, there's a function named convert in some SQL dialects. So if this cast value as type syntax doesn't work with a specific engineer using, search for how to convert datatypes with that engine. It's a good practice to use explicit type conversion even if the SQL engine you're using can do it implicitly. Explicit cast can make the queries more portable between different SQL engines, they can make the queries run more efficiently, and they can help you to avoid unexpected results.

INTRODUCTION TO THE FROM CLAUSE

In the previous two lessons, you learned about the select list, which specifies what columns should be returned in your query results. The next part of a select statement that comes right after the select list is the FROM clause. That's the subject of this lesson. The FROM clause specifies which table the data you're querying should come from. You'll see later in this course that the FROM clause can do some other things, like combine two tables together. But for now we'll consider only the case where the FROM clause refers to a single table. Earlier in the course, I described how the FROM clause is optional in many sequel dialects.

But you can't do much without it aside from testing out different functions and operators.

So from here on, I'll mostly treat the FROM clause like a required part of the select statement. As you already saw in previous lessons, the basic syntax of the form clause is very simple. It's just from table reference. Table reference is often simply the name of the table in the current database. So if the current database is fun, then to query data from the games table, you just use the FROM clause from games. Recall that a database is just a logical container for a group of tables. It's sometimes called a schema. And recall that the current database or the active database is the particular one that you're in, that you're connected to. When you use just a table name in the from clause, this makes your query dependent on which database is the current database.

The current database needs to be the one that contains the table you're querying. So if you're querying a table in one of the non defaults databases, you need to remember first to connect to that database. Or to switch into that database by using the active database selector in ue or by running a ue's statement if you’re in a command line sequel interface. It's easy to forget to do this. You might have already stumbled on this when attempting to run queries in the VM. I do this all the time. You get an error like table not found or could not resolve table reference. Or worse than that, you might forget to switch into the intended database, and the current database might have a table with the same name as the one you intended to be in. In that case, you might get a more cryptic error, or your query might succeed, but you'd be querying a completely different table than the one you thought you were querying.

This has happened to me and caused me a lot of confusion. Also, if you're sending a SQL statement to someone else for them to execute, it could be ambiguous which database you intended for them to use. Fortunately there is a way to avoid this. Instead of using just the table name in the FROM clause, you can qualify the table name with the database name. The syntax for this is FROM databasename.tablename. For example, FROM fun.games. If you use this qualified form of a table reference, and it doesn't matter what the current database is, the statement will always query the table in the specified database. In this course, I will often just use unqualified table names in the FROM clause to keep things concise. But it's generally a good practice to qualify table names when you're running SQL statements in the real world. In the previous two lessons, you learned about the select list, which specifies what columns should be returned in your query results. The next part of a select statement that comes right after the select list is the FROM clause. That's the subject of this lesson. The FROM clause specifies which table the data you're querying should come from.

You'll see later in this course that the FROM clause can do some other things, like combine two tables together. But for now we'll consider only the case where the FROM clause refers to a single table. Earlier in the course, I described how the FROM clause is optional in many sequel dialects. But you can't do much without it aside from testing out different functions and operators. So from here on, I'll mostly treat the FROM clause like a required part of the select statement. As you already saw in previous lessons, the basic syntax of the form clause is very simple. It's just from table reference. Table reference is often simply the name of the table in the current database. So if the current database is fun, then to query data from the games table, you just use the FROM clause from games. Recall that a database is just a logical container for a group of tables.

It's sometimes called a schema. And recall that the current database or the active database is the particular one that you're in, that you're connected to. When you use just a table name in the from clause, this makes your query dependent on which database is the current database. The current database needs to be the one that contains the table you're querying. So if you're querying a table in one of the non defaults databases, you need to remember first to connect to that database. Or to switch into that database by using the active database selector in ue or by running a ue's statement if you’re in a command line sequel interface. It's easy to forget to do this. You might have already stumbled on this when attempting to run queries in the VM. I do this all the time. You get an error like table not found or could not resolve table reference. Or worse than that, you might forget to switch into the intended database, and the current database might have a table with the same name as the one you intended to be in. In that case, you might get a more cryptic error, or your query might succeed, but you'd be querying a completely different table than the one you thought you were querying. This has happened to me and caused me a lot of confusion.

Also, if you're sending a SQL statement to someone else for them to execute, it could be ambiguous which database you intended for them to use. Fortunately there is a way to avoid this. Instead of using just the table name in the FROM clause, you can qualify the table name with the database name. The syntax for this is FROM databasename.tablename. For example, FROM fun.games. If you use this qualified form of a table reference, and it doesn't matter what the current database is, the statement will always query the table in the specified database. In this course, I will often just use unqualified table names in the FROM clause to keep things concise. But it's generally a good practice to qualify table names when you're running SQL statements in the real world.

IDENTIFIERS

The database names and table names you use in the FROM clause are types of identifiers. The column reference is used in the select list are also identifiers. This is a good time to pause, to clarify some things about identifiers, and some related topics like keywords and case as in uppercase and lowercase. There are some rules governing what is a valid identifier in SQL. The rules vary depending on what SQL engine you're using. But here is a stringent set of rules that should be safe to follow with any of the major SQL engines including Hive and Impala. Identifiers can consist of alphabetic characters, that's the letters a to z, digits, zero to nine and underscores.

Other things like Unicode characters, punctuation, emoji, and so on should not be used in identifiers. The first character of an identifier should be a letter, a to z. The letters in identifiers should all be lowercase, and identifiers can be as short as one character long. The maximum length varies, but it's a good idea to limit them to 30 characters or fewer. Those rules are pretty stringent and some of them can be relaxed depending on which SQL engine you're using. There are also some particular words that you cannot use as an identifier even though they do follow these rules. These are called Reserved words. An example is the keyword SELECT. SELECT is a special keyword in SQL that signals the start of a clause. So you should never try to use it by itself as an identifier. Making it lowercase doesn't change this, reserved words are reserved regardless of their case. Other familiar keywords like FROM, AS, DISTINCT, SHOW, and USE, are also reserved words. The full list depends on what SQL engine you're using. For Hive and Impala, you can follow the provided links to see the full list of reserved words. The list for Hive distinguishes reserved keywords from a non reserved keywords.
It's just the reserved ones that you need to avoid. For Impala under the list of current reserved words, there's also a list of possible future reserved words. You

should also avoid using any of those as identifiers. It is possible with most SQL engines to break some of the rules for identifiers. But to do this, you need to enclose the offending identifier in some quote characters. Different SQL engines use different quote characters around identifiers.

Hive, Impala, Presto, Drill, and MySQL, all use back ticks. But in Post SQL you use double quotes around identifiers. If you're using another SQL engine, you should check what characters it uses. Here's a silly example for Hive or Impala. If someone has made the terrible choice of creating a database named 'use' containing a table named 'from' with a column named 'select' then to switch into that database and to query it you would need to enclose all the identifiers in back ticks as shown here.

USE 'use' then SELECT 'select' FROM 'from'. It's a joke among SQL experts to see how long you can keep going like this having each identifier be the same as the keyword before it. But if you ever see identifiers like this in databases in the real-world, you should probably mistrust the person who created them. Even if identifier don't break the rules, you can always enclose them in quote characters. In fact, it's often considered a good practice to quote all your identifiers. This is especially true if you're writing SQL statements that might be used for many years or built into an application. But for ad hoc queries that will just execute once, it's probably not worth the fas. So identifier's in SQL include database names, table names, and column names. But another type of identifier, is the column aliases that you can use in SELECT list.

Recall that you can use these aliases to control the names of the columns in the result set. If you use unquoted aliases, then the rules for what is a valid alias are essentially the same as the rules for other types of identifiers. Only lowercase letters, digits, and underscores, starts with the letter no reserved words. But if you quote an alias, then the rules for what you can use are typically much looser. When you quote an alias, you can use reserved words, and you can often use things like spaces and punctuation characters too. When enclosing an alias in quotes, you should use the same quote characters you use to quote database names, table names, and column names. Again, for Hive and Impala, it's back ticks. To continue the silly example from earlier, you could use the quoted reserved word AS as a column alias as shown here, SELECT 'select' AS 'as' FROM

'from'. This again would be a terrible choice. When you're choosing aliases it's a good practice to follow the stringent rules I described earlier.

With Hive and Impala and many other SQL engines, identifiers are case insensitive. That means, that the statement SELECT NAME FROM FUN.GAMES; in all capital letters will work exactly the same as SELECT name FROM fun. Games. You could use any combination of upper and lowercase letters in the database name, table name, and column name, and it would still work. This is true whether or not you quote your identifiers, but I strongly encourage you to only use lowercase letters in identifiers for consistency. Some databases are fussier about the case of identifiers. For example, Post SQL is sensitive to the case of identifiers but only when they're enclosed in double quotes. If you're using some other SQL engine, search for the details regarding case sensitivity of identifiers. Also, please see the reading for some more information about case in SQL. If you need help understanding the meanings of some of the terms I used in this video like identifier and keyword, see the glossary in Course Resources which includes definitions for these and many other terms.

FORMATTING SELECT STATEMENTS

In the example SELECT statements presented in this course, you might have noticed that sometimes the SELECT clause and the FROM clause are both on the same line, and other times the FROM clause is on a separate line below the SELECT clause. Either way is fine because in SQL extra white-space is generally ignored. You can include as many new lines, spaces, and tab characters as you like between the keywords and names and literals and operators that make up the statement, and SQL engines will ignore at all. The only time when extra white-space is not ignored is when it's used inside key words or names or literals.

For example, you cannot just put a whitespace character in the middle of a keyword like SELECT or in the middle of a function name or table name or number. If you add a whitespace character inside a quoted literal string, then it will represent that literal whitespace character within the string. Of course, there are many places where you need to use at least one whitespace character like to separate the keywords SELECT from the list that comes after it and to separate the keyword FROM and the table reference that comes after it. In these places, any extra white-space you use is ignored. Because extra white-space has no meaning to a SQL engine, you can use it to format SQL statements to make them easier to read.

With very short statements, this is usually not necessary, you can just put the whole statement on one line. But with longer statements, it really helps with readability, if you use some extra white-space. For longer statements, a convention is to put each clause on a new line and indent all the clauses after the first one with a tab or several spaces. Also, if there's an individual clause that's too long to fit on a single line, then you can split it up using new lines and use double indentation at the beginning of the lines that the cause continues onto. Here's an example. Notice that this statement has a very long SELECT list and it's been split up into three lines by adding new lines after some of the commas. The FROM clause is on its own line too and is indented with two spaces.

The lines that the SELECT list continues onto are indented with four spaces. Using four spaces instead of two to indent those lines, makes it easier to see when the SELECT clause ends and the FROM clause begins. That's the convention I'll use for formatting longer SQL statements in this course, except in cases where there's not enough room and I need to format it differently to fit on the screen. I encourage you to also follow this convention or perhaps some other convention of your choice that keeps your SELECT statements looking tidy and easily readable. Again, using new lines or extra spaces is never necessary, but it's helpful for readability and it will be even more helpful as you use additional clauses and write longer and more complex SQL statements in the later parts of this course. Another thing you should keep in mind as you learn about the other clauses is that the order of the clauses in the SELECT statement is important.

In general, you cannot go moving around the clauses. They need to be in the correct order or the SQL statement will be invalid. There are some exceptions to this. For example, HIVE will actually allow you to put the FROM clause before the SELECT clause, which is fine if you like to write SQL like the way Yoda speaks. But you should not generally do this. It will work with HIVE, but not with many other SQL engines, and it might not even work with HIVE depending on what client application you're using. In the example SELECT statements presented in this course, you might have noticed that sometimes the SELECT clause and the FROM clause are both on the same line, and other times the FROM clause is on a separate line below the SELECT clause. Either way is fine because in SQL extra white-space is generally ignored. You can include as many new lines, spaces, and tab characters as you like between the keywords and names and literals and operators that make up the statement, and SQL engines will ignore at all. The only time when extra white-space is not ignored is when it's used inside key words or names or literals.

For example, you cannot just put a whitespace character in the middle of a keyword like SELECT or in the middle of a function name or table name or number. If you add a whitespace character inside a quoted literal string, then it will represent that literal whitespace character within the string. Of course, there are many places where you need to use at least one whitespace character like to separate the keywords SELECT from the list that comes after it and to separate the keyword FROM and the table reference that comes after it. In these places, any extra white-space you use is ignored. Because extra white-space has no meaning to a SQL engine, you can use it to format SQL statements to make them easier to read. With very short statements, this is usually not necessary, you can just put the whole statement on one line. But with longer statements, it really helps with readability, if you use some extra white-space. For longer statements, a convention is to put each clause on a new line and indent all the clauses after the first one with a tab or several spaces.

Also, if there's an individual clause that's too long to fit on a single line, then you can split it up using new lines and use double indentation at the beginning of the lines that the cause continues onto. Here's an example. Notice that this statement has a very long SELECT list and it's been split up into three lines by adding new lines after some of the commas. The FROM clause is on its own line too and is indented with two spaces. The lines that the SELECT list continues onto are indented with four spaces. Using four spaces instead of two to indent those lines, makes it easier to see when the SELECT clause ends and the FROM clause begins. That's the convention I'll use for formatting longer SQL statements in this course, except in cases where there's not enough room and I need to format it differently to fit on the screen.

I encourage you to also follow this convention or perhaps some other convention of your choice that keeps your SELECT statements looking tidy and easily readable. Again, using new lines or extra spaces is never necessary, but it's helpful for readability and it will be even more helpful as you use additional clauses and write longer and more complex SQL statements in the later parts of this course. Another thing you should keep in mind as you learn about the other clauses is that the order of the clauses in the SELECT statement is important. In general, you cannot go moving around the clauses. They need to be in the correct order or the SQL statement will be invalid. There are some exceptions to this. For example, HIVE will actually allow you to put the FROM clause before the SELECT clause, which is fine if you like to write SQL like the way Yoda speaks. But you should not generally do this. It will work with HIVE, but not with many other SQL engines, and it might not even work with HIVE depending on what client application you're using.

CASE (IN)SENSITIVITY IN SQL

As you've seen in the course videos, whether the case of words in SQL matters varies. In general, *case sensitive* means that case matters—a letter or group of letters that are uppercase (capital, for example **'A'** or **'FROM'**) is considered different from the same letter or group of letters that are lowercase (for example, **'a'** or **'from'**). *Case insensitive* means these things are the same—it doesn't matter if you write something with uppercase letters, lowercase letters, or even a mix of cases. For case insensitive matters, **'FROM'** is the same as **'from'**.

In SQL, there are many different things that could be case sensitive or not: keywords, function names, column and table references, strings (when comparing). Unfortunately, it's difficult to be definitive, because different SQL engines have different behaviors. For example:

- In PostgreSQL, Apache Hive, and Apache Impala, table and column names are always lowercase. Even when you create a table, if you use uppercase, any results showing the table or column names will show them lowercase. However, table and column references are essentially case insensitive, because you can still use uppercase or lowercase in your queries. The engine will automatically convert them to lowercase. Results will show them using lowercase letters.
- In MySQL, table and column names will retain how they are defined on creation (or for column names, however they might be altered later). If you define them with uppercase, they will be stored as uppercase; if you define them with lowercase, they will be stored as lowercase. However, the table names are case sensitive: You must match the case to get a result. If the

table is named **TABLE_NAME**, then **FROM table_name** will not match that table. On the other hand, column names are case insensitive. Any references using the correct letters will match regardless of case. The results will use the case you use for the query, so **SELECT NAME** will have **NAME** in the result header, but **SELECT name** will have **name** in the result header. Otherwise the results will be the same.

Even string comparisons can work differently:

- In PostgreSQL, Hive, and Impala, string comparisons are case sensitive. For example, **'this' = 'This'** returns **false**.
- MySQL string comparisons are *not* case sensitive. For example, **'this' = 'This'** returns **1**(true), as does **'this'='THIS'**. (But **'this' = 'that'** returns **0**, or false.)

This course emphasizes Hive and Impala. In Hive and Impala, all keywords, function names, and identifiers are case insensitive. Only string comparisons are case sensitive. However, we will use the following conventions. These are merely *conventions;* although they are widely used, they are not essential:

- Keywords (like **SELECT** and **FROM**) are in uppercase.
- Most other things are all lowercase, including identifiers and most function names.

Making keywords the only things uppercase makes it much easier to quickly identify the keywords.

Other tools that access Hive or Impala, such as business intelligence applications, might impose their own case conventions and might have their own rules for identifier names. So if you’re using some tool like that, be sure to also consider its conventions and rules, not only the conventions and rules imposed by Hive or Impala.

WEEK 3

Learning Objectives

- Construct query statements that filter results to provide more efficient analysis
- Use comparison operators, logical operators, and conditional functions in expressions
- Create queries with desired handling of NULL values
- (Honors) Format and save query results

INTRODUCTION

Welcome to Week 3 of Analyzing Big Data with SQL. In the previous week's lessons, you learned about two of the clauses that make up a select state. The SELECT clause, which specifies what column should be returned in your result set, and the FROM clause, which specifies where the data you're querying should come from. The SELECT clause is a required part of every SELECT statement, and the FROM clause is required to query actual rows of data from tables. In this week of the course and throughout the remaining weeks, you'll learn about several more clauses that are not required but that enable you to answer different types of questions.

This week is all about the WHERE clause, which filters the rows of data based on some specified conditions. Even though we’re moving on to a new clause this week, you’ll see that we will revisit many of the topics covered in the previous week, including data types, expressions, operators, and functions. These are just important in the WHERE clause as they are in the SELECT clause. And in this week of the course, I'll extend to some of those topics. You'll learn about Boolean data types, how to write Boolean expressions, how to use logical operators and conditional function. Understanding each of these topics will help you specify the conditions to control which rows of data are returned in a result set.

About the Datasets

In this course so far I've asked you to query only the smaller tables on the VM, which are in the defaults database and the toy, and fun databases. That's because I had not yet introduced the clauses that you need to work with very large tables.

But beginning in this week of the course you'll start to learn about those clauses. So I'll start asking you to query the larger tables on the VM. First, I'll ask you to query the table in the wax database, which is just a little bit bigger than the tables in the fly database, some of which are much bigger.

Even though you'll be learning how to work with larger tables, I'll still be using the tiny little tables to teach and demonstrate. With these tiny tables you can see the table, see the select statement, see the result all at a glance. And this is great for understanding what different kinds of select statements do. So we're not done with these tiny tables yet, but we'll add the larger tables to the mix. In this video I'll explain what's in these databases named wax and fly. The wax database has just one table named, crayons. This table describes 120 different crayon colors. The columns give the name of the color, its hexadecimal code, and its red, green and blue values, which each range between zero and 255. The hex code and the red, green and blue values assume we're representing the colors using the RGB color model. There's also a column named pack, which gives the smallest pack that the crayon comes in. The basic colors like blue and yellow come in all the packs even the smallest, but some of the more exotic colors only come in the bigger packs. So this pack column gives the number of crayons that's in the smallest pack that the color comes in. The fly database has four tables named, flights, airlines, planes, and airports. The flights table is the biggest of these. It has more than 60 million rows representing passenger flights by major airlines in the United State for a full decade from 2008 through 2017.

This table is big, but it's not really so big by today's standards. There are real-world tables stored on clusters and in cloud storage today that are many thousands of times bigger. But the data in the flights table is big enough that if you tried to use traditional tools to work with it, you would have trouble. It would be impossible at worst and slow, or inefficient at best. It's with data this large that the value of distributed SQL engines like Hive and Impala start to become evident. The other three tables in the fly database, airlines, planes and airports are all related to the flight's table. They have more information about the airlines that operated these flights, the planes that flew them, and the airports they departed from and arrived at.

The data in this fly database is real-world data from the US Department of Transportation. So it might contain records representing airports you've been to or flights that you flew on. When you're analyzing the data in a table as large as some of those in the fly data base. You should almost never run a select statement that returns all the rows in the table. But if you use the WHERE clause to filter rows, or if you use some of the other clauses you will learn about later in the course. Then you can query a table this large and get a results that it's small enough to work with or to interpret. So as you learn how to use the WHERE clause and these other clauses, more of the questions and quizzes will ask you to query these bigger tables in the fly database. And you'll see that the remainder of the course will start to have a bit of an aviation theme.

This will culminate with the final assignment in which you will write and execute some more complex queries on these tables in the fly database. To analyze the data and answer some important business questions. It will help for you to get familiar with the tables in the fly database. Please see the reading for more information about these tables and about the tables in the other databases, too. Including how they're structured and where the data comes from. The reading include descriptions of every column, and some of them do require further explanation. So see the descriptions there to understand what they all represent.

Different SQL engines accept different character sets for string data. Two common sets are Unicode and ASCII, but there are other character sets as well.

CHARACTER SETS

The acronym ASCII (pronounced “AS-key”) stands for “American Standard Code for Information Interchange” and dates back to the 1960s. It was the first character set widely used on computers and consists of 128 characters, including the English alphabet (both uppercase and lowercase letters), numbers, and punctuation marks. Each character is assigned a number, from 0 to 127.

The extended ASCII set adds another 127 characters, for a total of 256. It includes diacritic characters (such as é, ç, or ö) and some other common symbols, such as the degree symbol (°).

Unicode is a newer industry standard with well over 130,000 characters. It is a superset of the ASCII set; that is, Unicode characters 0–127 are the same as the ASCII characters with the same numbers. Unicode includes not only diacritic characters (such as é, ç, or ö), it also includes completely different character sets, such as Cyrillic and Brahmic language characters, and symbols such as mathematics symbols, advanced punctuation, and dingbats.

Here are some examples of Unicode characters in a literal string, using different languages:

'Bishop à G5' (French)

'Бишоп до Г5' (Macedonian)

'ប៊ីស្សពដើម្បី G5' (Khmer)

'♗to G5' (English, using a dingbat for ***bishop***)

Support in SQL Engines

Most major SQL engines allow the use of Unicode characters in character strings. These can be values within a string column or field, or literal strings. ***However, using Unicode characters in identifiers (such as column names or aliases) is inadvisable, even if allowed.***

With some SQL engines, you might need to modify a setting to enable handling of Unicode characters, but even with handling enabled, there can be limitations or special considerations to using these in your data. For example, some systems may require including the letter N before a literal value (for example, **N'Бишоп до Г5'**), to alert the system that the following string might use Unicode characters (or even characters not supported by the Unicode set). Other systems require that the string be enclosed in double quotes (not single quotes). For data values, some (such as Microsoft SQL Server*) require using separate data types that allow Unicode characters.

The SQL engines available on the VM all have limited support for Unicode characters within literal strings and the standard string variable types. For example, string functions might work with non-ASCII Unicode characters, but they might not. The function **lower('ÉTIENNE')** will produce **'étienne'** with Apache Hive and PostgreSQL, but **'Étienne'** with Apache Impala and **'��tienne'** with MySQL (using the command line—see below).

Impala and MySQL treat each Unicode character as a byte array, so one character will appear to be more than one. They incorrectly identify **length('ÉTIENNE')** as 8 (treating É as two characters). Also, **substr('ÉTIENNE',1,1)** will instead return �, but **substr('ÉTIENNE',1,2)** will return **'É'**. Hive and PostgreSQL will handle both of these instances correctly.

The MySQL editor in Hue on the VM has some difficulty with Unicode characters, particularly when they appear in a column heading above a result set. You may, however, use MySQL on the command line on the VM if you are familiar with it (use **mydb** as the database, with user **training** and password **training**).

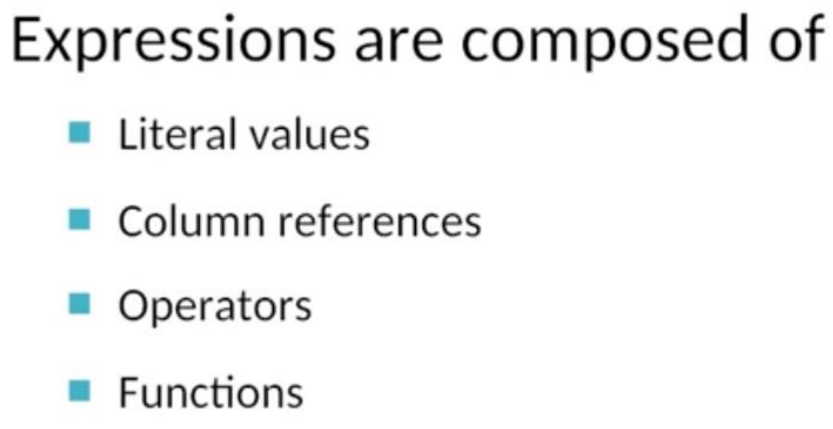
If there is a possibility that your data will include Unicode characters, check what your system's support for Unicode is. Test your SQL statements with values that include Unicode characters so that you are aware of any issues that might arise.

* Collation and Unicode Support. Retrieved from https://docs.microsoft.com/en-us/sql/relational-databases/collations/collation-and-unicode-support?view=sql-server-2017on August 17, 2018.

INTRODUCTION TO THE WHERE CLAUSE

The next clause in a SQL statement after the Select clause and the From clause is the Where clause. The Where clause filters the rows of data based on one or more conditions. Typically these conditions are tests of the values in specified columns for all the rows of the data. In other words, the Where clause takes all the data in the table, tests which rows meet some criteria, and returns only those rows. The Where clause has no effect on which columns are returned only on which rows are returned. The Where clause is optional. If you run a Select statement that has a Select clause and a From clause but no Where clause, then you get a result set that has as many rows as the table specified in the From clause.

The one exception to this that you've learned so far is if you use the distinct keyword, which removes the duplicate rows from the results set. Recall the earlier videos where I talked about data retrieval versus data analysis. The Where clause is where you really start to do data analysis, because you can use it to answer questions in the form in which rows are these conditions true. These conditions are the question you're asking about the data, and the result you get back contains the answer to that question ready for you to interpret. For example, looking at the games table in the fund database, you could ask, which games are priced at below $10? To answer that question, you would filter the rows to return only the games where list_price is less than 10, and the result you would get is Clue and Candy Land. That's a very simple example of the kind of logic you'll learn how to express in the Where clause.



Expressions are composed of

- Literal values
- Column references
- Operators
- Functions

So the WHERE Clause takes all the rows in a table, checks each row against some conditions and returns only the rows in which the conditions are true. The way that you specify these conditions in the WHERE clause is with an expression. Recall what you learned about expressions in the previous week of the course. When you learned how to use them in the SELECT list. An expression is a combination of literal values, column references, operators and functions. When a SQL engine evaluates an expression, it produces a column of values.

When you use an expression in the SELECT list, that column of values is returned as a column in the results set. You can use one or more expressions in the SELECT list separated by commas. These expressions can produce columns with different data types including numeric columns and character string columns. When you use an expression in the WHERE clause it is not returned in the results set, instead it is used to determine which rows are returned. You can use only one expression in the WHERE clause not to multiple expressions separated by commas.

The expression must evaluate to a single column of Boolean values also called logical values. These are true and false values. Only the rows in which the expression evaluates to true, are returned in the results set. I will demonstrate this with some examples. First recall the question from the previous video. Which games are priced at under $10? Looking at the columns in the games table you can see that the column with price information is named list_price. Once you know that, it's straightforward to express this question in the form of an expression that returns true or false for each row in the games table. The expression is simply list_price less than 10. To make this Boolean expression into a WHERE clause you simply put the keyword WHERE before it, WHERE list_price less than 10.

But you can't execute a WHERE clause by itself. It needs to be part of a complete SELECT statement. So, first I'll add the FROM clause, to specify which table the data should come from, FROM fun.games. Then I'll add the SELECT clause to specify what columns should be returned in the results set. The only column I really want to return is the name column. So I'll make it SELECT name, and now have a complete SELECT statement including a WHERE clause. When I execute this statement, the results set contains one column, name and two rows with the values Clue and Candy Land. So those are the only games in this table priced below $10. It would be nice to see what the list prices are for these two games. We know they're both less than $10, but we don't know exactly what the prices are. So I'll add the list price column to the SELECT list after the name column. That's also returned in the SELECT list. The choice of which columns you include in the SELECT list is totally separate from the choice of which columns you evaluate in the WHERE clause. You can select star or select whatever you want independent of what columns are in the expression in the WHERE clause. Here's another example, which games were invented by Elizabeth Magie? The expression to test this is, inventor equals 'Elizabeth Magie'.

In a WHERE clause inside a complete SELECT statement its, SELECT name FROM fun.games WHERE inventor equals 'Elizabeth Magie'. The result shows that it's the one game monopoly. A third example, which games are suitable for a seven-year-old? The games table has a column named min_age which represents the minimum age for a player. So for a game to be suitable for a player age seven the value in this min age column must be less than or equal to seven. Making this into a complete SELECT statement, its SELECT name, min_age FROM Fun.games WHERE min_age less than or equal to seven. The result shows it's just one game Candy land, that's suitable for players as young as three. You'll learn more about these comparison operators like less than and less than or equal to you in another video later in this week of the course.

The questions I posed at the start of these examples, were all very simple. It didn't take a lot of hard thinking to figure out how to write an expression in the WHERE clause to answer each of these questions. But in the real world things are rarely so simple. Often, the hardest part of writing a WHERE clause is translating human language with its frequent ambiguities and possibility for misunderstanding into an unambiguous Boolean expression that answers the question as intended. As a data analyst the questions you're asked to translate into SELECT statements, will often lack sufficient detail and clarity.

USING EXPRESSIONS IN THE WHERE CLAUSE

You'll often need to ask for clarification or make some reasonable assumptions and then clearly communicate those assumptions when you share your result. You'll also need to be sure you're using the right columns in the right tables to answer the question. Here are some ways that my attempts to answer the questions in the examples I just showed could have been confounded. The first question was, which games are priced at under $10? I answered this question by querying the Fun.games table and filtering on the column list_price. But what if the person who asked this question intended for me to answer it using a different table?

There's a table named inventory in the fund database that has information about the board games that are in stock at a couple of shops. There's a column named price in this table, representing the price the game is being sold for at each shop. Maybe the person who asked me the question intended for me to query this inventory table. This would give a different result set. Only the game Clue, in the shop named Dicey, is for sale at a price less than $10. The second question was, which games were invented by Elizabeth Magie? I was lucky with this question because the inventor's name in the question, perfectly matched the value in the inventor column in the games table.

But all kinds of things could have gone wrong. Elizabeth Magie also went by Lizzy Maggie and Elizabeth J. Maggie, and later she got married and changed her name to Elizabeth J. Philips. If I heard the question verbally maybe I would have thought the last name was spelled MCGEE. Circumstances like this could have prevented me for writing a select statement that returned to the intended result. Obviously, since I'm working with a table that has only five rows, errors like this seem easy to avoid. I can just glance at the table and figure it out. But that would not be the case if the table had millions of rows. The third question was, which games are suitable for a seven year old? To answer this question, I evaluated the min age column in the games table.

What if there was also a max age column and I had neglected to notice it? Candy Land is a pretty juvenile game. Some retailers describe it as a game for players age three to six. If there were a max age column with the value six for Candy Land, then the SELECT statement I wrote, would incorrectly identify Candy Land as suitable for a seven-year-old. To get the correct answer in that case, I would need to modify the expression in the WHERE clause, to evaluate both the min age and max age columns. Then the result would indicate that none of these games are suitable for a seven-year-old. You'll learn about logical operators like the 'and' in this example in another video later this week.

These allows you to combine multiple conditions into a single Boolean expression. As if these examples weren't enough, you'll also need to consider the possibility of missing values in the data and all the misinterpretations that can arise from that. You'll learn more about that in one of the lessons this week. So in this week of the course, as you learn how to write expressions to specify different kinds of filtering conditions in the WHERE clause, don't forget to always carefully interpret the questions you're asked, and watch out for possible ambiguities and misinterpretations.

COMPARISON OPERATORS

In the simple examples in the previous video, I showed how you can use operators, like less than and equals in expressions in the WHERE Clause to compare column values with literal values. Recall the examples, like WHERE list_price less than 10, and WHERE inventor equals Elizabeth Maggie. In this video, I'll show the common comparison operators that you can use in Boolean expressions in SQL and I'll describe how you can use them. All of these are binary operators, meaning that there's an operand on both sides of the operator, and all of these work across all the major SQL engines. The equals sign in a SQL expression tests for equality of the operands.

You can use this operator with any data types, including numeric and character string if the operands on the left side and the right side, call them X and Y. If they are exactly equal, then the expression X equals Y evaluates to true. If you have any experience with programming languages, then you're probably more familiar with the use of double equals, two equals signs for equality comparison. That's because a single equal sign is used for variable assignment in many programming languages, but in a SQL SELECT Statement, there's no variable assignment, so you just use a single equal sign for equality comparison. The not-equals operator test for inequality of the operands. This is basically the opposite of the equal sign. When X equals Y is true, then X not-equals Y is false and vice versa. But you'll see in an upcoming video, that in the case of missing values, it is possible to have two operands that are neither equal nor unequal. There are two forms of the not-equals operator in SQL, exclamation mark equals, and less than, greater than. Both of these work with most major SQL engines.

The first form, exclamation mark equals is more widely used today and it's what I'll use in this course. Less than and greater than are two operators that should be familiar to almost everyone. These are typically used with numeric operands. The same is true of the less than or equal to and greater than or equal to operators. Each of these operators is two characters with the equals sign right after the less than or greater than symbol. At this stage of the course, you should not try to use these last four operators with non-numeric operands, like character strings. You'll learn more in a later week of the course about how SQL engines compare the ordinal values of different strings but for now, just avoid using these operators with non-numeric operands. When you're using these comparison operators in an expression in SQL, the operands on the left and right sides can be column references or literal values. For example, you can compare the values in one column to the values in the same rows in another column or you can compare the values in a column to a literal value. Here are some simple examples based on the data in the crayons table in the wax database. Recall that there are columns in that table named, red, green, and blue with integer values between zero and 255, these represent colors with the RGB color model.

You could query this table to return only the colors that have a green value larger than the red value. To do this, you would use the expression green greater than read, this compares each value in the green column to the value in the same row in the red column or you could answer the question which colors have a blue value of 50 or lower. To do this, you would use the expression blue less than or equal to 50, this compares each value in the blue column to the literal number 50.

Try writing some SELECT Statements to query the crayons table, using expressions like these in the WHERE Clause. The operands on the left and right sides of comparison operators can also be expressions.

The SQL engine evaluates the expressions on both sides of the comparison operator, then evaluates the whole Boolean expression to compare the left and right sides for each row. I'll demonstrate this with an example. In the RGB color model, very high values of red, green, and blue, yield colors that are very light, close to white. So we could find the lightest crayon colors by looking for the ones where the sum of the red, green, and blue values is large, say over 650. To do this, you would use the expression, red plus green plus blue greater than 650. Now, try using some expressions like this in the WHERE Clause, using arithmetic operators or built-in functions to create an expression on one or both sides of the comparison operator.

The example I just showed used an arithmetic expression, red plus green plus blue on one side of the comparison operator, and a literal number, 650 on the other side. When you use this expression in the WHERE Clause of a SELECT Statement, then the rows in the results set all have red plus green plus blue greater than 650, but the results set does not include a column giving these values of red plus green plus blue, the SQL engine calculate these sums and uses them to filter the data, but it does not return these sums in the results set. But it might be good to include them in the results set, maybe to see how close to 650 each one is. So you might try something like this, compute the sum of these colors in an expression in the select list, give this result column an alias, rgb_sum, and then use that alias in the expression in the WHERE Clause, rgb_sum greater than 650.

Unfortunately, this does not work. The reason is that SQL engines process the WHERE Clause before they compute the expressions in the select list. In other words, they filter the rows of the table before they build the columns for the result set. So aliases defined in the select list are not available for you to use in the WHERE Clause. There are some exceptions to this, but most SQL engines have this limitation. You can work around this by entering the expression again in the select list. If this seems a bit kludgy, needing to use the same expression twice, I agree, it is. In a later course in this specialization, you'll learn about other possible workarounds. In this video, I've focused on using these Boolean expressions in the

WHERE Clause, but you can use them in the select list too, then you get a Boolean column in your result set.
For example, here I moved the whole Boolean expression, red plus green plus blue greater than 650, out of the WHERE Clause and into the select list and I gave it the column alias, light because these colors with high RGB values are light colors. Then the results that includes a column named light, containing true or false values. You can see Almond has a light color, Antique Brass is not, and so on. The results set has not been filtered by this expression, instead, the expression just returns a column of Boolean values in the results set. With many SQL engines, Boolean is a datatype, just like the numeric and character string data types you learned about earlier. Instead of being numbers or strings, a Boolean column contains true and false values. Hive, Impala, Presto, Drill, and PostgreSQL, all have a Boolean data type. But many SQL engines actually do not have a Boolean data type, instead, they use integers to represent true and false values, zero is false and one is true, MySQL works this way. So if you ran a query like this with MySQL, you would get a column of ones and zeros in the results set.

DATA TYPES AND PRECISION

When you're using comparison operators in a boolean expression, it's important to pay attention to the data types of the operands. The left and right operands don't need to have exactly the same data type, but both operands should have the same high level category of data type, like both numeric or both character string. For example, it's okay to have an integer on one side, and a decimal number on the other, like this, 1=1.0. Integer and decimal are compatible data types and this expression will return true. When the operators do not have the same high-level data type, different SQL engines do different things. For example, the digit 1, in a coded string, equals the integer value 1.

Comparing these two values requires converting them to a common data type to enable an apples to apples comparison. Some SQL engines, like Impala, will not automatically perform this type conversion. So a query with an expression like this in it will fail. To get the query to succeed you would need to explicitly cast the left operand to a numeric type, or the right operand to a character string type, as described in an earlier video. But other sequel engines including Hive will perform and implicit cast and the query with an expression like this will run successfully. But it's hard to understand exactly what is happening in implicit casts like this. Is

the left operand being converted to a number for comparison purposes or the right operand to a string?
Unexpected things can happen with automatic type conversion. And it's better not to leave anything to chance. So you should explicitly cast operands to the same high level category of data type before using comparison operators to compare them. Refer back to the video about data type conversion in week two of this course, if you need a refresher on how to do this. Also, when you're working with decimal data types in SQL, and in particular with floating point numbers, some strange things can happen when you use comparison operators. For example, one-third represented as a decimal number is 0.3333, infinitely repeating. But of course, you can't write infinitely many digits after the decimal, you need to stop somewhere. Nor can a computer store infinitely many digits after the decimal.

So how many digits do you need? How many threes do I need for a SQL engine to say that 0.3333 and so on is equal to one third. With Impala the expression one third equals 0.3333 with four threes after the decimal, that evaluates to false. But if you put 20 threes after the decimal then it evaluates to true. Somewhere on the far right side of a number like 0.3333 with arbitrarily many threes. Somewhere in the smallest decimal places, SQL engines will start to ignore differences when comparing numbers. This is necessary because values like one third can never be represented with perfect precision regardless of how many decimal places you have.

But it's hard to know exactly how far into the less significant digits where a SQL engine will start to ignore the differences. One solution is to use rounding to make comparisons like this less ambiguous. One-third rounded to the fourth decimal place is equal to 0.3333, with four 3s after the decimal. An upcoming video shows another possible work around for this. And the next course in this specialization goes into more detail about the unexpected behaviors that can happen with floating point values. But for now just keep in mind that weird stuff can happen in the least significant digits. And use rounding to avoid ambiguous comparisons.

WORKING WITH LITERAL STRINGS

In the video lecture, “Comparison Operators,” the instructor presented how to use comparison operators in SQL statements. Most of these comparison

operators are probably already familiar to you. You probably already have an intuitive understanding of how they work with *numeric* operands.
You can also use them with *character string* operands. Just as numbers have order to them, characters have order to them, and comparison operators use this order to determine the truth of the comparison. Alphabetical order is a simple example, so for example, when using the English language, b < t because b comes before t in the English alphabet. It can get more complicated when you begin to consider things like case sensitivity, punctuation, Unicode characters, and mixing sets of alphabets. You'll read more about these complications in Week 3. For now, you can just think about the alphabetical order and focus on *equal* strings.
Within a SQL statement, you are not likely to compare two *literal strings* (also called *string constants*)—strings with a stated value, such as **"this"** or **'that'**. Instead, you are likely to compare a column with the **STRING** data type to a literal string (or occasionally to another string column). For example, this statement will filter the **inventory** table to provide only rows for the Dicey shop:

```
SELECT * FROM inventory
 WHERE shop = 'Dicey'
```

Even this can have some complications. For example, consider the same statement with just the name of the shop changed to the other shop in the table, Board 'Em:

```
SELECT * FROM inventory
WHERE shop = 'Board 'Em'
```

Do you see the issue? Because the name includes an apostrophe—which is the same character as the single quote being used to define the literal string—the string would be interpreted as '**Board** ' and the unexpected extra text **Em'** would most likely throw an error.
Many SQL engines allow you to use either single or double quotes around literal strings, so one way to fix this would be to use double quotes:

```
SELECT * FROM inventory
 WHERE shop = "Board 'Em"
```

Some versions of Impala, including the one installed on the course VM, require that you escape single quotes even within double-quoted literal strings. The backslash alerts the query engine that the next character has a different meaning than usual—in this case, that it should be taken literally and not as the end of the quote.

```
SELECT * FROM inventory
 WHERE shop = "Board \'Em"
```

or

```
SELECT * FROM inventory
 WHERE shop = 'Board \'Em'
```

Some SQL engines do *not* allow using double quotes for quoted strings. PostgreSQL, for example, requires single quotes around literal strings. Double quotes are reserved for a different purpose. In that case, you would need to escape the interior single quote. This is done by using the backslash as with Impala, or by putting *two*single quotes (which is not the same as a double quote) in its place:

```
SELECT * FROM inventory
 WHERE shop = 'Board ''Em'
```

This method will also work with most SQL engines that do allow both single and double quotes. Other methods may also work; see the section at the end of this reading.

For the greatest compatibility across SQL engines, we recommend escaping interior single quotes with the backslash (**\'**).

One final warning when working with literal strings: Be careful, especially when copying and pasting quoted strings from emails and documents, that the pair of single or double quotes that enclose the string are *straight quotes* and not *curly* or *smart quotes.* Compare the quotes in the following statement to the ones in the statements above:

```
SELECT * FROM inventory
 WHERE shop = “Board ’Em”
```

Can you tell how the double quotes are both slanted, and in different directions? SQL engines will not recognize this character as the straight double quotes (**"**) and so it will not work.

Notice that the single quote in the statement above is also different. In this case, because it's an interior quote and not being used to define the start and end of a string, this will not throw any errors. But note, it would not match the actual value **Board 'Em** in the data, because the value in the data uses a straight quote and not a curly quote. For example, this query will succeed in most SQL engines but will return 0 rows:

```
SELECT * FROM inventory
 WHERE shop = "Board ’Em"
```

But if curly single quotes were used to define the string, as in the following, it would not be accepted:

```
SELECT * FROM inventory
 WHERE shop = ‘Dicey’
```

In addition to the comparison operators mentioned in the video lecture, another comparison operator that can be used with string operands is the **LIKE** operator. This operator can be used to match partial strings. For example, this statement will filter the **inventory** table to provide only rows for the Dicey shop, because the string **Dicey** contains the substring **ice**.

```
SELECT * FROM inventory
 WHERE shop LIKE '%ice%'
```

The percent sign (**%**) is a wildcard that matches zero or more characters. You can also use an underscore (**_**) to match any single character. In Hive and Impala, matches with the **LIKE** operator are case-sensitive, but case sensitivity varies depending on the SQL engine.

LOGICAL OPERATORS

The WHERE clause if you use it, must contain only one boolean expression, as I described in an earlier video. You cannot have multiple expressions separated by commas in the WHERE clause. However, you can use what are called logical operators to combine multiple smaller boolean expressions or sub-expressions into a single boolean expression. I'll demonstrate this with some examples using the games table in the fund database. What if you're looking to buy a game that six people can play and you have $10 to spend on it. In the WHERE clause, you would need to combine these two conditions. Max_players greater than or equal to 6 and list_price less than or equal to 10. You would combine them with an AND operator in this case because you want to both of these conditions to be true. Here's another example.

What if you're looking for a game to play with no specific number of players in mind but you only have $10 to spend on it. Also, you already own monopoly. So Monopoly is also an option regardless of the price. In the WHERE clause, you would combine these two conditions, list price less than or equal to 10 and name equals Monopoly. You would combine them with the OR operator in this case because you're just looking for either one of these two conditions to be true not necessarily both. AND and OR in these examples are binary logical operators. Meaning that there's an operand on both sides of the operator. In SQL there's also a unary logical operator which comes before its operand, and that is the NOT operator. Here's a simple example using the NOT operator. Return all the games except Risk. You could do this by using the WHERE clause WHERE NOT name equals Risk. Of course, you could also use the NOT equals comparison operator to do this that gives the same results and that's a clearer way to write it for this example. If you've used a programming language, you might be accustomed to using symbols for these logical operators.

Often, it's an exclamation mark for the NOT operator, an ampersand for the AND operator, and a pipe character or the OR operator. But in SQL, you use the words NOT, AND, and OR as logical operators. Also, NOT, AND, and OR are case insensitive but the convention is to put them in all uppercase letters. Each of the examples I showed used just one logical operator; AND, OR, or NOT. But you can write arbitrarily complex expressions that use as many logical operators as you want in whatever combination you want. But if you use multiple logical operators together in an expression, you need to be mindful of order of operations. In SQL, the precedence of logical operators is NOT, AND, and OR in that order.

If you forget that you can end up with a totally incorrect result. For example, if you want to return all the games except Candy Land and Risk, you might write expressions like these. NOT name equals Candy Land AND name equals Risk. Or NOT name equals Candy Land OR name equals Risk. You might think that the NOT would be applied to both of the equality comparisons coming after it, but that is not what happens. The NOT operator applies only to the equality comparison immediately after it. So both of those statements return results that are not what you're looking for.

To get the result you're looking for, you need to use the NOT operator before both equality comparisons, and combine them with an AND operator. That gives the result you're looking for. Alternatively, you could just use the NOT equals comparison operator, that would also work fine. Another option is to use parentheses around parts of the expression to control the order of operations. Using parentheses, you can override the usual rules of operator precedence. Parenthesize sub-expressions are evaluated first. For example, you can enclose name equals Candy Land or name equals Risk in parentheses, then put the NOT operator before the opening parenthesis, so it negates the entire sub-expression in parenthesis. This will give the result you expect.

All the games except Candy Land and Risk. Here's an example that demonstrates the precedence of the AND operator over the OR operator. I'll combine the first two examples I showed in this video. You're looking to buy a game that's six people can play and you have $10 to spend on it, but you already own Monopoly, so that's an option too. To express this in a WHERE clause, you need to combine three comparisons into one boolean expression. You might try to do it like this, WHERE list_price less than or equal to 10 OR name equals Monopoly AND max_players greater than or equal to 6.

That might seem correct, but because the AND takes precedence over the OR, the max_player is greater than or equal to 6 part first gets combined with the name equals Monopoly part. This limits the results only to Monopoly. Then the list_price less than or equal to 10 part on the left side of the OR broadens this to include any game with list_price less than or equal to 10. So you end up with Candy Land in the results set, even though Candy Land has max_players four. The solution is to use parenthesis to express what you really intended. The $10 limit on the list price or the game being Monopoly, that's one condition that must be true, and the max_players being six or more, that's a separate condition that must be true. So you should put parentheses around the list price and name conditions to combine them into one condition.

That fixes the precedence problem and gives you the result you're looking for. It's important to clearly understand the conditions that you want to use to filter the data before you try to express those conditions in a WHERE clause. So as I've said before when you're doing a data analysis to answer a question that someone asked to you, then you might need to ask them to clarify the question. Or you might need to make some assumptions but then clearly communicate those assumptions when you share your results. Also, Boolean logic can be confusing.

So when you're working as a data analyst, don't be afraid to ask someone else to review your more complex boolean expressions to check your logic. Also, always do a sanity check on your result. Take a look at the number of rows in the results set and examined some of the values in it to try to catch errors that you might have made that would cause the result to be different than expected.

OTHER RELATIONAL OPERATORS

In an earlier video, I described the comparison operators like equals and less than, that you can use to make comparisons in an expression in SQL. Each of these takes a single operand on the left side and a single operand on the right side. There are a couple other operators in SQL that are similar to these but they take more than one operand on the right side, these are the IN and BETWEEN operators. The IN operator compares the operand on the left side to a set of operands on the right side. It returns true if the operand on the left matches any value in the set on the right. Here's an example, this statement filters the games table by the name column returning only the rows in which the value in the name column is in the specified set of three literal strings; Monopoly, Clue and Risk. On the left of the IN operator is the column reference, name, and on the right side is the set of three literal values separated by commas and enclosed in parentheses. This is the syntax you use with the IN operator. This example uses character string operands but you can use any datatype. However, the operands on the left and right should have compatible datatypes just like with the comparison operators. Also, this example uses a column reference on the left and literal values in the list on the right but you can use any combination of column references, literal values, and expressions on either side.

You could get the same result by using three equality tests with OR operators combining them into one Boolean expression. But as you can see using the IN operator is more concise and more readable especially if you have a large set on the right side. Also, some SQL engines can process a query more quickly and efficiently when it uses the IN operator instead of multiple equality tests. The BETWEEN operator compares the operand on the left side to a lower bound and an upper bound both specified on the right side. The comparison returns true if the operand on the left is greater than or equal to the lower bound and less than or equal to the upper bound.

Here's an example, this statement filters the games table by the min_age column, returning only the rows in which min_age is greater than or equal to eight and less than or equal to 10. The BETWEEN operator is typically used with numeric operands like in this example, you can use it with non-numeric operands like character strings, but I would not recommend doing that yet. You need to learn first about how SQL engines compare the ordinal values of character strings and that's a topic you'll learn about later in this course. Also, this example uses a column reference on the left and two literal values on the right, but you can use any combination of column references, literal values and expressions as long as the operands have compatible datatypes. You could get the same result by using two comparison operators combined with an AND.

In this example, it would be min_age greater than or equal to eight AND mean_age less than or equal to 10. But using the BETWEEN operator is typically more concise and more readable. One case where the BETWEEN operator is useful, is when you want to check a numeric value for approximate equality with some margin of error allowed. Recall the earlier discussion about the uncertain precision of decimal number comparisons, using BETWEEN is one way to work around that. Both IN and BETWEEN can be used with the word NOT immediately before them, that negates the result of the comparison. So NOT IN returns true if the operand on the left does not match any of the values in the set on the right. NOT BETWEEN returns true if the operand on the left is less than the lower bound or greater than the upper bound.

UNDERSTANDING MISSING VALUES

When you're filtering data based on some conditions, it's important to consider whether the conditions are known for all the rows or not. The presence of missing or unknown values in the data, can make it impossible to determine weather conditions are true or false. For example, if the price of some board game is unknown, then it's impossible to determine whether that price is less than $10. Maybe it's less, maybe it's more, maybe it's exactly $10, you have no way of knowing. Life would be easier if you never had to work with missing values. But the reality is that many datasets have them, and as a data analyst, you'll need to handle them properly.

So the next few videos and readings are all about how to work with and interpret missing or unknown values. In SQL, a null value is a value that's missing or unknown. This is represented by the keyword "null" in SQL expressions, and in result sets. The small tables I've used in the examples up to this point like the games table and the crayons table, do not have any null values in them. But some of the other tables do. Among the small tables the ones that have null values are the offices table in the default database, and the inventory and card_rank tables in the fun database.

The null values that are in these different tables all look the same, they're all just null. But the meaning or interpretation of null values, what they actually represent, can differ based on the context. In the offices table, the row representing these Singapore office has a null value in the state_province column. The reason that value is null, is that Singapore is not divided into states or provinces. So the null there means not applicable. In the inventory table, there is a null in the aisle column and another in the price column. In this table what these nulls mean are, we don't know what aisle the game clue is in the dicey shop and we don't know what the price of Candy Land is in the Board 'Em shop.

The nulls here mean unknown. In the card_rank table, there is a null in the value column, in the row representing ace. The reason that's a null, is that the value of an ace card can vary. For instance, in the card game blackjack and ace card can be worth either one point or 11 points. So this null means undefined or indeterminate. Real-world data sets often have lots of missing values in them. There are many datasets in the real world that have more missing values than non missing values. These are referred to as sparse datasets. None of the tables on the VM are like that, but some of the tables in the fly database which contain real-world data, do have many missing values. For example, in the flights table the actual departure time column, depth_time, and the actual arrival time column, ARR_time, have lots of nulls.

Some of these nulls represent flights were canceled, others are there because of data collection errors. There's one row in the flights table representing the famous US Airways Flight 1549. This flight took off from LaGuardia Airport in New York, ran into a flock of geese that caused both engines to fail, and ditched in the Hudson River. You might have heard about this. There was a movie about its starring Tom Hanks. No. Not that solely. Yes. That's the one. Anyway, try writing a query to find this flight in the flights table. It was on January 15th, 2009, the carrier is US, the flight number is 1549, and the origin airport is LGA.

After you write and run a query that returns that row, take a look at which column values are missing in this row. In each of the examples I showed so far in this video, you could see the null value in the table or in the results set. Null values are mostly straightforward to interpret when you can spot them. You might have some questions about what they mean, but at least you know they're there and you can interpret the results accordingly. However, null values become more of a tricky problem when you inadvertently filter them out of your results using a WHERE clause. Recall that when you use a WHERE clause, you specify the filtering conditions with a boolean expression, and only the rows for which the boolean expression evaluates to true are returned in the result set. So far we've assumed that for each row, a boolean expression will evaluate to either true or false. But if a table has null values in it, then there is a third possibility, a boolean expression could evaluate to null.

That's because as I said at the beginning of this video, missing values can make it impossible to determine whether some conditions are true or false. Rows in which the boolean expression in the WHERE clause evaluates to null, are omitted from the results just like the rows where it evaluates to false. This is a really important concept to understand so I'll repeat it. Any rows in which the expression in the WHERE clause evaluates to false and any rows in which it evaluates to null, are filtered out excluded from the result set.

This can have important implications for interpreting query results. Here's an example to demonstrate this using the inventory table in the fun database. This table contains information about board games that are in stock at a couple of shops. What if you wanted to know which games are available in these shops for less than $10? To find this, you would run a query like, SELECT star FROM fun.inventory WHERE price less than 10, and you would get a result with just one row, the game clue in the shop Dicey, which has the price $9.99. You might interpret this result and make a statement like, there is only one game available at the shop that's priced under $10. But this is not necessarily true.

There is a null value in the price column indicating that the price of Candy Land in the Board 'Em shop is unknown. When a numeric value is unknown, you cannot determine whether it's less than 10. There's just not enough information to know whether it is or not. So because of this null value in the price column, it's misleading to say there is only one game at these shops priced under $10. It would be better to make a statement like, there is at least one game at the shops possibly two for less than $10, or there is only one game at the shops that we know has a price of less than $10 or something like that.

As a data analyst, you can avoid a lot of trouble and blame by being mindful of the possibility of missing values, and phrasing your interpretations to account for that possibility. It's also important to explicitly check for null values and handle them in your queries, that's the subject of the next video. Also, if you took the first course that's part of this specialization, or if you have any past experience with traditional relational database systems, you might be familiar with the concept of null constraints. Null constraints can prevent data with null values from being loaded into the database in the first place, but distributed SQL engines like Hive and Impala, do not generally support these constraints. The way they work makes it impractical. So you often need to assume that any column in any table could have null values in it.

SQL engines provide several facilities for identifying and handling null values in tables. These includes several operators and functions that you can use in expressions in the SELECT clause or in the WHERE clause. In this video, I'll introduce the operators. Then in the next video, I'll introduce some of the functions. But there's one thing you need to understand first. You cannot test for null values using the standard comparison operators like equals, not equals, less than, and so on. For example, you might think that you could return the row with the missing price in the inventory table using the WHERE clause, where price equals null. You might think you could return all the other rows except this one using WHERE price not equals null. But neither of these work that way.

They would both return zero rows. That's because when you use the standard comparison operators in SQL expressions, any value compared to null always returns null. Recall from the last video, when the expression in the WHERE clause returns null for a row, that row is excluded from the results set. I want to emphasize this point, that whenever you use a standard comparison operator in an expression than any value compared to null always yields null. For example, five equals null, evaluates to null. Five not equals null, evaluates to null. Five less than null, evaluates to null. Even null equals null evaluates to null, and null not equals null also evaluates to null. Look at the last two examples here. The best way to understand these is to remember that null means some unknown value.

So does some unknown value equals some unknown value or are they different? There's just no way to know. These comparisons return unknown, they return null. In SQL statements, to check for null values you need to use a special operator. The IS NULL operator. You use this operator by putting the keywords IS NULL after a column reference or expression. There's another version of this operator, IS NOT NULL. Here's an example.

If you want to return only the rows of the inventory table where the price is null, you would use the WHERE clause, WHERE price is NULL. This returns the one row representing Candy Land in the boredom shop. If you want to return all the rows except the one where the price is null, you would use WHERE price is not NULL. That returns all the other rows from the table. Returning to the famous case of US Airways Flight 1549, recall that in the row in the flights table representing that flight, the departure time is not missing, but the arrival time is missing. Go ahead and modify the SELECT statement you wrote for the question about that in the last video, this time return the rows representing all the flights on that same day, January 15th, 2009 that have non-missing departure time in the depo_time column and missing arrival time in the ARR_time column.

You'll need to use both IS NULL and IS NOT NULL to do this. In addition to IS NULL and IS NOT NUL, there is another pair of operators that can help you to handle null values. They are, IS DISTINCT FROM and IS NOT DISTINCT FROM. I'll explain this with an example. The office's table in the default database has four rows representing four different offices. What if you wanted to write a WHERE clause to filter out the office in Illinois, to return the three offices that are not in Illinois. You might try to write a WHERE clause like WHERE state_province not equal to Illinois. But this returns only two rows. Though row representing the Singapore office is not in the result set. That's because the state province value in that row is null, and null not equal to Illinois evaluates to null. So that row is excluded from the results set. But in this case, you know from context that this null value doesn't mean unknown, it means not applicable because Singapore does not have states or provinces. This is the type of situation where the IS DISTINCT FROM operator is useful. If you replace not equals with IS DISTINCT FROM, then the results set includes the Singapore office.

The IS DISTINCT FROM operator is like the not equals operator, but it treats null values and non null values as explicitly unequal. Whenever the operand on one side is null and the operand on the other side is not null, it evaluates to true. Compare that to the not equals operator which evaluates to null in that case. Also, if both operands are null, then IS DISTINCT FROM evaluates to false. It treats any two null values as equivalent. So null, IS NOT DISTINCT FROM null, they are the same. Compare that with the not equals operator which evaluates to null when both operands are null. You can rewrite an expression that uses IS DISTINCT FROM to instead use a not equals comparison and one or two tests to determine if the operands are null or not. But it's more concise to use IS DISTINCT FROM.

There is also a version of this operator that negates the result of the comparison, IS NOT DISTINCT FROM. This is like the equals operator, except when it compares a null value with a non-null value, it returns false instead of null, and when it compares two null values it returns true instead of null. Some SQL engines offer a shorter way to write the IS NOT DISTINCT operator. It's less than sign, equals sign, greater than sign. This shorthand notation does exactly the same thing as IS NOT DISTINCT FROM and it's supported by Hive, Impala, and MySQL. There is no special shorthand notation for IS DISTINCT FROM.

Reading: Reading Missing Values in String Columns

This reading describes how the three logical ***operators***—**AND**, **OR**, and **NOT**—work when one or both of their operands are **NULL**.
Many misunderstandings about **NULL**s in Boolean logic arise when you confuse **NULL** with **false**. So remember: **NULL** does ***not*** mean **false**; it means "unknown."
The examples below use this sample table, which is not available to query in the VM:

name	age	siblings
An	8	1
Belinda	NULL	3
Chand	3	NULL
Delmar	NULL	NULL
Enise	1	2

The AND Operator

For an AND expression to return true, the operands on both sides must be true. On the other hand, if either expression is false, then the expression returns false. This means, if one operand is NULL and the other is true, then the AND expression returns NULL. If one operand is NULL and the other is false, then it returns false. If both operands are NULL, it returns NULL.

Expression	**Value**
true AND NULL	**NULL**
false AND NULL	**false**
NULL AND NULL	**NULL**

Look in the example table above for children that you know are under the age of two ***and*** have more than one sibling. Which can you say definitely ***do*** or ***do not*** match the criteria?
Here are the results:

name	**age < 2**	**siblings > 1**	**age < 2 AND siblings > 1**
An	false	false	false
Belinda	NULL	true	NULL
Chand	false	NULL	false
Delmar	NULL	NULL	NULL
Enise	true	true	true

The OR **Operator**

For an OR expression to return true, only one of the operands needs to be true. It is only false if both operands are false.
If one operand is NULL and the other is true, then the OR expression returns true. If one operand is NULL and the other is false, then it returns NULL. If both operands are NULL, it returns NULL.

Expression	**Value**
true OR NULL	**true**
false OR NULL	**NULL**
NULL OR NULL	**NULL**

For example, look in the table above for children who are under the age of two ***or*** have more than one sibling. Which can you say definitely ***do*** or ***do not*** match the criteria?

Here are the results:

name	age < 2	siblings > 1	age < 2 OR siblings > 1
An	false	false	false
Belinda	NULL	true	true
Chand	false	NULL	NULL
Delmar	NULL	NULL	NULL
Enise	true	true	true

The NOT Operator
When the unary operator **NOT** is applied to a **NULL** operand, the result remains **NULL**.

Expression	Value
NOT NULL	**NULL**

The expression **NOT NULL**in the table above does not represent the **IS NOT NULL** operator; it is simply the unary operator **NOT** applied to the literal Boolean value **NULL**.
Once again, look in the table, this time for children who are ***not*** under the age of two. Which can you say definitely ***do*** or ***do not*** match the criterion?
Here are the results:

name	age < 2	NOT age < 2
An	false	true
Belinda	NULL	NULL
Chand	false	true
Delmar	NULL	NULL
Enise	true	false

Try It!
For this table of data, what would be the result of each expression, for each row in the table? Click here for the answers.

title	**year**	**length**
If	1993	4:31
Security	1969	NULL
Coming Around Again	NULL	3:41
Seasons of Love	1996	2:52
Love So Soft	2017	2:52

1. year < 2000 AND length > 4:00
2. year < 2000 OR length > 4:00
3. NOT(year < 2000 OR length > 4:00)

When working with character string columns, a couple of misconceptions often arise around the issue of missing values.

An ***empty string***, also called a ***zero-length string***, is ***not*** the same thing as a **NULL** value. A literal empty string is written in a SQL expression as a pair of opening and closing quotes with nothing between them (**''**). When working with real-world data, watch out for string columns in which the absence of a known value is represented by an empty string instead of a **NULL**. The expressions required to find and handle empty strings are different than the expressions to find and handle **NULL**s. For example, to filter out the rows that have an empty string in the column named **string_column**, you would use: **WHERE string_column != ''** or **WHERE length(string_column) > 0** instead of: **WHERE string_column IS NULL** When you are working with real-world data, always inspect the data to determine how missing values are represented. If necessary, ask the person responsible for maintaining the data to tell you how missing values are represented.

The literal string **'NULL'** is also not the same as **NULL**. This literal string is ***not*** a missing value, it's a four-character string composed of the letters N, U, L, and L. The letters could also be in other cases: **'null'** or **'Null'** for example. Imagine being the technology journalist Christopher Null, whose last name (not a pseudonym!) often is not recognized by applications that don't distinguish between the literal string and the missing value **NULL**! (Read about it in "Hello, I'm Mr. Null. My Name Makes Me Invisible to Computers.")
Be mindful of these issues when you work with character string columns.

CONDITIONAL FUNCTIONS

In this video, I'll introduce some built-in functions, that are often used for handling null values. Some of these can be used for other purposes too. Some other functions I'll describe here are very common, and are built into virtually every SQL engine. Others are not universally implemented. When you're using any one of these functions, always do a test first to check for errors, and to check that the output is what you expect. The if function takes a Boolean expression as the first argument. If the expression evaluates to true, it returns the second argument. If not, it returns the third argument. Here's an example. This select statement uses the if function in an expression in the select list. It checks each row to see if the value in the price column is null. It returns the value 8.99 if it is null, otherwise it returns the actual value from the price column. The resulting column has the alias correct price.

In the results side, you can see that the one null value, in the price column has been replaced with 899. Here's another example using the if function. This one demonstrates that, when the expression in the first argument evaluates to null, the if function returns the third argument, the same as if the expression evaluated to false. The expression here price greater than 10 evaluates to true for the rows with monopoly, and risk. So for those the if function returns the second argument, high price. The expression evaluates to false for clue, and to null for Candy land. So for both of those it returns the third argument low or missing price. So the if function allows you to conditionally return one of two values.

But if you want to conditionally return more than two different values, you can use a case expression. This uses a different syntax, than the built-in functions. I'll demonstrate this with an example. In the previous example, I use the if function to give each game one of two designations; high price, or low or missing price. Here, I use a case expression to give them three designations; high price, low price, or missing price. The case expression begins with the keyword case. Following that there's a series of clauses, that each begin with the keyword when. After each 'When' keyword, there's a Boolean expression, then the keyword then, and an expression giving the result.

If the Boolean expression evaluates to true, then that result is returned. Otherwise, it continues to the next when clause. After all the when clauses, there is an else clause. That gives the result to return when none of the above Boolean expressions are true. Finally the case expression ends, with the word end. There's no punctuation between the when clauses, and you can write the whole case expression on one line, or you can use extra white-space like I did here to make it more readable. Case expressions are really useful for implementing complex conditional logic in an expression in the select list.

So the if function, and the case expression are both useful for handling null values, and they're also useful for some other kinds of cases where you need to implement conditional logic in a SQL expression. The remainder of the functions I'll describe in this video, are narrower in the scope of their application, and they pertain just few null value. The null if function, takes two arguments, and returns null if the two arguments are equal. If they are not equal, it returns the value of the first argument. That might seem like a puzzling thing for a function to do, and it probably doesn't sound very useful at first, but there are some particular cases where it is useful.

I'll demonstrate one such case with an example using the flights table in the fly database. You probably remember from elementary mathematics, that you're not supposed to divide by zero. If you try to divide by zero in an expression in a select statement, some SQL engines like Postgres SQL will throw an error. Others like Impala, will return a special value signifying that the result is infinite, or that it's not a number. The null if function can help you to avoid this. The expression in this example divides the flight distance in miles, by the time spent in the air in minutes. To compute the average speed, it multiplies that by 60 to get the units in miles per hour. The trouble is that there are some rows in the flights table, in which the air-time is zero. Air-time is in the denominator in this division. So you need to do something about those zeros to avoid dividing by them. If you use the nullif function as shown here that replaces these zeros with nulls. Dividing something by null, just returns null.

But that's okay, it's better than getting an infinite value or an error. The nullif function, is really just shorthand for a case expression, or an if function with the argument specified in a certain way, but it's shorter to write it using the nullif function. Another case where the nullif function is useful, is when there is some particular value that's used to represent a missing value, but it's not a null. For example, there are many data sets where values like 999 are used to represent missing values in numeric columns.

This kind of thing is unfortunately very common, and if you don't properly handle these values, by converting them to actual nulls, they can cause all kinds of problems. The nullif function can help you to work with data like this. There's another function that's named ifnull in some SQL engines. It's named NVL in some others. This function tests if an expression evaluates to null. If it's not null then it returns the value of the expression. If it is null then it returns some other value. Here's an example of this function. In the flights table, the air-time column has some null values in it. For whatever reason the value is missing for some of the flights.
Say you're looking specifically at flights from New York EWR, to San Francisco SFO, and you want to replace those missing air-time values, with an estimated value. I happen to know that a typical flight from New York to San Francisco is in the air for about 340 minutes. So I want to replace those nulls with 340. You can do this with the ifnull function as shown here. The first argument is the column reference air-time, and the second argument is the value to replace the nulls, 340. The result set from this query will not contain any nulls. The final function I'll introduce in this video is coalesce. Coalesce can take any number of arguments, and it returns the value of the first argument, that's not null. If they're all null, it returns null. Here's an example.

In the flights table, there is an actual arrival time column, and a scheduled arrival time column. There are a lot of missing values in the actual arrival time column but very few in the scheduled arrival time column. Suppose you want to return the actual arrival time for each flight, but if it's missing you want to fall back on the scheduled arrival time, and a return to that instead of a null if possible. To do this, you can use the coalesce function as shown here. The first argument is the column reference ARR time, and the second argument is the column reference, scaled ARR time. You could use more arguments.
Whenever you use these conditional functions, check that they're available with the SQL engine you're using, and test them out to verify that they're working as you expect, before you use them to produce a result that you analyze or send to someone else.

USING VARIABLES WITH BEELINE AND IMPALA SHELL

In the honors lessons in the previous weeks, you learned about Beeline and Impala Shell. You learned how to use these two command line tools interactively and non-interactively. In the videos in this honors lesson, you'll learn about some additional capabilities of these command line tools and you'll see how you can call them from shell scripts. You'll also learn about some different options for integrating Hive and Impala with scripts and applications. In this video, I'll show you how to use a feature in Beeline and Impala shell called variable substitution. This feature enables you to parameterize queries. In other words, it enables you to take names or values that are hard-coded into your SQL statements and replace them with variables. This feature is implemented differently in Hive and Impala and the syntax differs slightly between them. First, I'll describe variables substitution in Hive, I'll use a couple different examples to demonstrate different applications of it. One situation where variables substitution is useful is when you have two or more statements in a SQL script and there's a particular literal value that's used in multiple places in the statements. In the example shown here, there are two select statements in the SQL script file game_prices.sql and the literal string, Monopoly, is used in both of these statements.

The trouble with this is that if you want to change this literal string to something else, say from Monopoly to Clue, then you need to change it in more than one place. With only two places it's not so bad, but imagine there were dozens of places where you needed to change the value that would be cumbersome. Variable substitution provides a solution to this. At the top of the SQL scripts, you can add a set statement to assign a value to a variable. In this example, the variable named game is assigned the value Monopoly.

The syntax of the set statement is unlike other SQL syntax, it begins with the keyword SET followed by a space, then the word hivevar and a colon. After the colon is the name of the variable you want to assign, in this example, it's game. Then there's an equal sign then the value you want to assign to the variable. You should not use quotes around the value in a set statement even if it's a character string. Elsewhere in SQL, you do need to use quotes around literal strings but the SET statement is special. You can use spaces on either side of the equal sign if you want, Hive will trim any white-space from the beginning and end of the value after the equal sign.

White-space inside the value like spaces between words is retained. Finally, you terminate the set statement with a semicolon. On the lines below the SET statement you can use this variable in your SQL statements, to do this you use the syntax shown here, ${hivevar:game}. Then when you execute the statements in the SQL file using the Beeline command with the dash f option. Hive replaces each instance of this dollar sign curly brace hivevar placeholder with the value that's assigned to the variable. In other words, it substitutes the assigned value in each of these places. This example assigns just one variable then uses it in two places, but you can use more than one set statement to assign multiple variables with different names and you can use those variables in as many places as you need following the set statements.
A different situation where variable substitution is useful is when you have a SQL statement in a file and you want to run it many times but with a different literal value substituted in each time.

In the example shown here, there is just one SELECT statement that returns the hexadecimal color code for the crayon with the specified name, it's red in this case. If you wanted to run the same query for many different colors it would be cumbersome to keep editing the SQL script file to change the name of the color in the where clause. Variable substitution provides a solution to this. First, you replace the hard-coded value in the SQL script with a variable using the same dollar sign curly brace hivevar syntax I described earlier.

The variable is named color in this example. Then instead of using a SET statement to assign a value to the variable, you use a command line argument to assign it. At the command line, you use the beeline command with the dash f option to execute the statement in the SQL file. In this example, it's hex_color.sql. Right before the dash f option, you use dash dash hivevar followed by the name and the value of the variable. The syntax is dash dash hivevar a space, then the name of the variable an equals sign and the value of the variable enclosed in quotes. Unlike with the SET statement you should use quotes around the value in this case, so the operating system shell passes it correctly to Beeline.

When you run the beeline command, Hive replaces the dollar sign curly brace hivevar placeholder with the value specified on the command line. In this example, there was just one statement in the SQL file but you can have more than one. You can also use multiple different variables. To do that you need to specify dash dash hivevar on the command line once for each name-value pair as shown here. This example returns the name of the crayon that has the specified red green and blue values. So that's how you use variable substitution with Beeline. For Impala shell it's very similar, there's actually only one difference, with Impala shell you use var instead of hivevar.

Aside from that, everything works the same as I described with Beeline. When choosing a name for a variable, stick to the same rules for valid identifiers that I introduced earlier in the course and you'll be safe. You should use all lowercase letters to avoid any questions about case sensitivity. Also, whenever you are assigning literal string values that include apostrophes or quotation marks you should always escape them with a backslash. You should do this regardless of whether it's Beeline or Impala shell and regardless of whether you are assigning the variable with a SET statement or on the command line.

Calling Beeline and Impala Shell from Scripts

In this video, I'll close out the topic of Beeline and Impala shell, by showing how you can invoke these two utilities using shell scripts. In previous videos, demonstrating Beeline and Impala shell, I showed how you can run these utilities by issuing Beeline or Impala dash shell commands directly at the operating system command prompt. In this video, I'll show how you can put these commands inside a text file along with other shell commands to create a shell script. Putting shell commands in a script file, makes them easier to rerun later. Instead of entering and running a whole series of commands at the command prompt, or copying and pasting commands from a file you can simply issue a single command to run the whole shell script or you could use a scheduler to run the shell script at some designated times. Even if you're planning to run a series of commands only once, still saving them all in a shell script has some benefits.

You can write the commands using an editor instead of right at the command prompt so it's easier to see what you're doing. You'll have a clear record of what commands you ran in case any questions arise. Shell scripts are sometimes called Bash scripts because the most common command shell in the Linux and Unix family of operating systems is called Bash. When you open the terminal on the VM for this course, Bash is the shell you're using there. It's pretty straightforward to include a Beeline or Impala shell command, in a shell script. I'll demonstrate this with an example. Here's a simple shell script named email_results.SH, .SH is the usual file extension for shell scripts. The first line of the script has what's called a hash bin. It tells the operating system to use bash to execute this script. You should generally used that at the top of any shell script. Below that there are two commands. The first command invokes impala shell in a non interactive mode and queries a small subset of the rows from the Flights table. It saved the result to a comma delimited text file named zero_air_time.csv. The second command sends an email to fly at example.com with the specified subject and body and with this file zero_air_time.csv attached to the email. You can write each command on multiple lines in a shell script but you'll need to use a backslash at the end of the line whenever the command continues onto the next line like in this example. After you create a shell script and save it, you need to change the permissions on the script file to allow you to execute it. You can do this using a chmod command like the one shown here. Then to execute the shell script, use the syntax shown here. At the terminal in the current directory where you saved the shell script, Enter./ then the file name of the shell script and press enter to execute it. You can also schedule shell scripts to run at specific times, and you can run shell scripts from inside other scripts or programs or applications.

For example you can run a shell script with Python by using the call function in the subprocess module. If you would like to try running this example on the VM, you can create the email_results.SH file that I showed, change the permissions on it and try running it. Just be sure to replace the example email address with your own. I can't guarantee that the mail command will actually get the email to your inbox. It depends on the configuration of your network and on your email provider. When I tested it, it worked for me, but the email ended up in my spam folder. You would have to do a bunch of complicated configuration to get it to work reliably and that's beyond the scope of this video.

So as that example demonstrated, you can include Impala shell commands in a shell script simply by entering the commands into the script file in the same way you would enter them at the command prompt and it works the same way for Beeline. Just make sure you invoke Beeline or Impala shell in non-interactive mode in shell scripts. Refer back to the videos in the previous weeks honors lesson to review the arguments that you need to use to do that. Shell scripting is a big topic. I could say much more about it but that's beyond the scope of this course. If you're already familiar with shell scripting, you should have no trouble incorporating Beeline or Impala shell commands into shell scripts. If you want to learn more about shell scripting, there are lots of good online tutorials and books.

For example, there are some books from the publisher O'Reilly on this topic. When you include Beeline or Impala shell commands in shell scripts, you'll often want to use other commands to process the result using various Linux utilities. Some of them the more commonly used ones are said OK and grep and there are some books about these two. If you would like to get more practice invoking Beeline or Impala shell and shell scripts, see the reading following this video. In the reading, there's an optional ungraded challenge exercise where you can write a shell script to query the crayons table to return the hexadecimal code for a specific color and then use the results to change the background color on the desktop of the VM. This is meant to be a difficult to exercise and you might need to learn more about shell scripting to complete it.

(Optional Exercise) Change VM Desktop Color 30 min

Important: ***Read the entire exercise before starting!***

In this exercise, you will write a shell script that prompts the user to enter a crayon color, queries the hexadecimal code for that color from the crayons table in the wax database, and uses that hexadecimal code to set the VM's desktop background to that color.

Begin with this partially completed shell script. The first line (after the **#!/bin/bash**) prompts the user to enter the name of a color, and the last line sets the color of the desktop background using a hexadecimal color code:

```
#!/bin/bash read -p "Enter the name of a crayon color: " COLOR
⋮
gconftool-2 -t str -s /desktop/gnome/background/primary_color "#$HEX"
```

Your task is to fill in the missing lines, to:

Invoke Beeline or Impala Shell to query the crayons table, using the environment variable named **COLOR** in the **WHERE** clause of the **SELECT** statement, and return the hexadecimal color code.

Assign the returned six-character hexadecimal color code to the environment variable named **HEX**.

This is intended to be a difficult exercise. You might need to learn some more about shell scripting to complete it. There is more than one correct solution. Start with a basic solution, then later, if you like, you can improve your script to gracefully handle edge cases and to avoid possible errors. (For example, what if the user enters a color name that’s not in the crayons table?) Click here for solutions and how to return to the original color.

Video: LectureQuerying Hive and Impala in Scripts and Applications
In the previous video, I described how you can write shell scripts that invoke Beeline or Impala shell and I mentioned that you can run these scripts from other scripts or programs or applications. However, that is not the only way to integrate it program with Hive or Impala. There are some other programmatic interfaces you can use. The three you should know about our ODBC, JDBC, and Apache Thrift. These are three interfaces standards that were designed to make it easy and efficient to integrate scripts and applications written in any language with Sequel engines and other services. One major benefit of using these interfaces to query Hive or Impala is portability. The drivers or libraries that are required to use them can be installed on virtually any computer. To use Beeline or Impala shell on the other hand, you need a local installation of Beeline or Impala shell, and that's impractical or impossible on many systems. I'm not going to get into the details about how to use these interfaces, they vary depending on what kind of system you're running and what language your script or application is written in. But I will show one example just to illustrate the concept. Here are a few lines of Python code that use the Apache Thrift interface to connect to Impala and run a query. The code fetches the results into a list to object and then prints the rows to the screen. This code uses a Python package called Impala. You can run this code for yourself on the VM. Open the terminal and use the command python in all lowercase to start a Python session, then enter and run the code. Be sure to include the leading spaces on the final line. Indentation has significance in Python code. This is just a simple example but you could integrate this kind of data retrieval code into any script or application to accomplish any kind of task. In fact, many of the applications that data analysts use like BI and analytics software, have the ability to query Hive and Impala and it's almost always built using one of these interfaces most often ODBC or JDBC.
Quiz: Week 3 Honors Quiz

WEEK 4

LEARNING OBJECTIVES

- Aggregate data in a dataset to answer analytic questions
- Aggregate data on groups within a dataset to answer analytic questions
- Evaluate and defend choices on when to group and aggregate data over specific columns
- Write and run a SELECT statement that filters data using aggregated values
- (Honors) Identify differences between versions of Hive, Impala, and Hue, using documentation or experimentation

INTRO

Welcome to Week 4 of Analyzing Big Data with SQL. So far in this course, you have learned about three of the clauses that you can use in a Select statement. The Select clause, which specifies what columns should be returned in your query result. The From clause, which specifies where the data you're querying should come from. The Where clause, which filters the rows of data based on some conditions. In this week of the course, you'll learn about two more Select clauses. Group BY, and Having. These clauses enable you to answer questions about aggregates of the data. But before you learn how to use these clauses, you need to understand what aggregation is.

So I'll begin with a brief introduction to this concept. Then I'll describe some common forms of aggregation and show how you can compute each one using a built-in function in SQL. You'll see that the syntax for using these functions, is mostly the same as for the built-in functions you learned about in previous weeks, but the way they work and what they return is different. We'll also return this week to the topic of missing values, and you'll see how SQL engines handle those in aggregations. When you're working with large scale data, grouping and aggregation are indispensable because they allow you to summarize all the rows in a table, with a result set that has just a few rows. So at the end of this week, you'll have the skills necessary to produce succinct summaries of enormous data sets.

AGGREGATE OPERATIONS

Introduction to Aggregation

Aggregation is the act of taking multiple values and reducing them down to a single value. Two of the simplest forms of aggregation, are counting and adding. Everyone understands these but just for completeness, I'll show an example of each using the employees table. Counting, simply means tallying up the roads figuring out how many rows there are. It's easy to see that this table has five rows. You do not need to look at any of the values in the rows to count them. Adding or summing is also very simple but it does require you to look at the values in the rows. For example, to compute the sum or total of all the salary values, you add up all the values in the salary column.

The sum of these is $185,403. The result of an aggregation is called an aggregate. So in these examples 5 and 185,403 are both aggregates counting and adding are not the only forms of aggregation. Here are a few others. Computing the average of the values in a column. The average is the sum of all the values divided by the number of values. In this example the average salary is $37,080.60. Finding the minimum value in a Column. The minimum salary is $25,784. Finding the maximum value in a column. The maximum salary is $54,523. All of these are types of aggregation. In the next video you'll learn about the built-in functions in SQL for computing all of these.

COMMON AGGREGATE FUNCTIONS

SQL provides built-in functions for performing common aggregations. These are called aggregate functions. In this video, I'll introduce the most important aggregate functions and I'll describe what arguments they expect. SQL's aggregate function for counting rows in a table is COUNT. To count rows, you do not need to know what's in the rows. So the count function does not require that you specify what values to aggregate.

In SQL, there's a special syntax for the count function. Its COUNT(*). You use an asterisk, a star, as the argument, and this returns the number of rows. This star syntax is used only for the COUNT function, not with other aggregate functions. You'll see in a later video that you can specify different arguments to the COUNT function but for now, just use COUNT(*). The aggregate function for adding is SUM. SUM adds up the values in a particular column. When you use the sum function in SQL, you need to specify what values to add up. So for example, you could use SUM(salary) to add up the values in the column named Salary. You'll see that for all the aggregate functions besides COUNT, you need to specify an argument like this, so the aggregate function knows what values to aggregate.

The aggregate function for computing the average or mean of a column is AVG. For example to calculate the average salary, you would use AVG(salary). The aggregate function for computing the minimum, the lowest value in a column, is MIN. So to calculate the minimum salary, you would use MIN(salary), and finally, the aggregate function for computing the maximum, the highest value in a column is MAX. So to calculate the maximum salary you would use MAX(salary). These five aggregate functions are all provided by all the major SQL engines. There are some other aggregate functions beyond these five but they're less commonly used and they're beyond the scope of this course. The names of aggregate functions are typically case insensitive, but by convention, we set them all in capital letters. This is different from the names of other built-in functions which we set in all lowercase letters. These different capitalization conventions can help you to distinguish the aggregate functions from other functions in this course.

COUNT(*) AND SUM(1)

Some data analysts use the expression **SUM(1)** instead of **COUNT(*)**. These two aggregate expressions do the same thing: they count the number of rows in a table.

This is because when you use a *scalar argument* (in this case, **1**) to an aggregate function (in this case, **SUM**), then the aggregate function aggregates that same value over all the rows.

For example, here is the **toys** table in the **toy** database:

id	name	price	maker_id
21	Lite-Brite	14.47	105
22	Mr. Potato Head	11.50	105
23	Etch A Sketch	29.99	106

Imagine executing **SELECT SUM(price) FROM toys;** You can think of this as running through the rows in the table, and for each row, add the value in **price** to a running total. So you would get 14.47, then 14.47 + 11.50, then 14.47 + 11.50 + 29.99.

If instead you execute **SELECT SUM(1) FROM toys;** the result would be like substituting the value **1** for each of those prices. Instead of 14.47 + 11.50 + 29.99, you would have 1 + 1 + 1. That is, each row contributes 1 to the sum. This is the same as counting the rows.

USING AGGREGATE FUNCTIONS IN THE SELECT STATEMENT

The most basic way to use aggregate functions in SQL query, is to use them in the select list. Here's an example to demonstrate this. Recall that the aggregate function for counting is count, and the syntax for using this function to count rows is count star. If you want to use the count function to count the number of rows in the employees table, then here's how you would do it. Select count star, from employees. This returns a result set with just one row, and one column containing the number five. There are five rows in the employees table and the count function aggregated over all of them, reducing these five rows down to a single value, the number 5. You can use a column alias to give a name to the column that's returned by this statement. After count star, add as num underscore rows. So now the result column is named num rows. The example I just showed use the aggregate function count. Here's an example that uses sum. Select sum salary from employees. This returns the sum of all the values in the salary column. They add up to 185,403.

Again, you could add an as expression after sum salary to give an alias to the result column. So count and sum are aggregate functions, and when you use an aggregate function in a select clause, like in these examples, it makes an aggregate expression. So count star and sum salary, those are aggregate expressions. You can include two or more aggregate expressions in the select list, like in this example, select min salary as lowest salary comma, max salary as highest salary from employees. This returns a result set with just one row, but with two columns, one for each of the aggregate expression. You can see the lowest salary in the employees table is 25,784 and the highest salary is 54,523. Those examples all used very simple aggregate expressions. They all had just a single aggregate function, with no other functions or operators. But you can use more complex aggregate expressions.

For example, this query returns the range or spread of the salaries in the employees table. The aggregate expression is max salary minus min salary. Here's another example of a more complex aggregate expression. This one answers the question, if there's a 6.2 percent payroll tax for each employee, then what's the total tax for all employees rounded to the nearest dollar? The aggregate expression to compute this is, round sum salary times 0.062. You can also use an expression as an argument to an aggregate function. The argument does not need to just be a column reference, it could be any valid expression. In this example, the expression round salary times 0.062, is used as the argument to the sum function. This is just another way to answer the question from the previous example, although the round function is applied differently here, and this might make the result slightly different. In each of the examples I just showed, the expression aggregates over all the rows, and returns a single row. That's what makes these expressions aggregate expressions. They can combine values from multiple rows, aggregating them together. These are different from the expressions you learned about earlier in the course, which operate independently on the values in each row. Those are called non-aggregate expressions or scalar expressions. They return one value per row. When you write SQL queries, you need to be careful about mixing aggregate and scalar operations. You can use aggregate and scalar operations together in an expression like in some of the statements I showed.

For example, you can take an average, then round it. Average is an aggregate operation and rounding is a scalar operation, but they can be combined together like this to form a valid aggregate expression. You can also use scalar arithmetic operators together with aggregate functions to form a valid aggregate expression like in this example, where you take a numeric column, multiply it by a literal number, then find the sum. But aggregate and scalar operations can form invalid expressions in some cases.

For example, you might try to use an expression like salary minus average salary to try to compute the difference between each individual employees salary and the average salary of all the employees. The right side of this expression aggregates, but the left side does not. As a result, this expression is invalid and will throw an error with most SQL engines. There is actually a valid way to compute the difference between each individual employee salary and the average salary of all the employees, but this is not it, and that's a more advanced topic that's beyond the scope of this course. Also, you cannot use scalar and aggregate expressions together in a select list.

For example, this query has two items in the select list. The first is just the column reference first name, that evaluates as a scalar expression. It returns a value for each row in the employees table. The second is the aggregate expression, sum salary. That returns just one row that aggregates all the salary values in the employees table. You cannot use both of these types of expressions together in one select list. Most SQL engines will throw an error if you try. When you use aggregate expressions in the select list, you can also use a where clause. You learned about the where clause in the previous week of the course, and you can continue to use it in this week of the course and beyond.

For example, to answer the question, how many employees make more than $30,000, you could use count star in the select list, together with the where clause, where salary greater than 30,000. This returns three. However, do not try to use aggregate expressions in the where clause. The where clause always processes individual rows, so you cannot use it to examine aggregates. All the aggregation examples I showed in this lesson returned a results set with only one row. They all aggregated the whole table down to a single row.

But what if you wanted to aggregate subsets of the table and return a result that gives separate aggregates for the different subsets? This is what the group by clause is for. You'll see how to use that clause in the next lesson. Interpreting Aggregates: Populations and Samples

THE LEAST AND GREATEST FUNCTIONS

Two built-in functions that often cause confusion are
the **least** and **greatest** functions. These are often confused with **MIN** and **MAX**. **MIN** and **MAX** are aggregate functions. They return the minimum or maximum value within a column.

least and **greatest** are *non*-aggregate functions. They return the smallest or largest of the arguments that are passed to them.
For example, the query:

SELECT MAX(red), MAX(green), MAX(blue) FROM crayons;
aggregates the full crayons table (which has 120 rows) down to a result set with just one row. The three columns in the result set give the largest value of **red**, the largest value of **green**, and the largest value of **blue** that exist in the full table.
Whereas the query:
SELECT greatest(red, green, blue) FROM crayons;
returns a result with 120 rows. Each row in the result set gives the largest of the three RGB values (**red**, **green**, **blue**) that make up each crayon color.
The **least** and **greatest** functions are available in many (but not all) SQL engines. One unusual aspect of the **least** and **greatest** functions is that they can take a very large number of arguments. Recall that there are a few other functions like this (including **concat**,**concat_ws**, and **coalesce**).

THE GROUP BY CLAUSE

All the examples of aggregation that I showed in the previous lesson, returned to result set with only one row. They all aggregated the whole table down to a single row. But what if you want to aggregate subsets of the table and return a result that gives aggregates for each of the subsets? That is what the GROUP BY clause is for. It splits a table into groups of rows so that the aggregates can be computed for each group. I'll demonstrate this with an example.

First, recall from the previous lesson, that this is the SELECT statement that you would use to count the number of rows in the employees table. SELECT COUNT(*) from employees. There's just one expression in the SELECT list, so the results that has just one column and it's an aggregate expression, so it aggregates the five rows of the employees table down to one row in the result set. This query answers the question: How many employees are there in total?

Each row represents one employee, so counting the total number of rows, gives you the answer, five. But what if you wanted to answer the question, how many employees are there in each office? There's a column named office_id whose values identify which office each employee works in. The values are coded as letters A, B, C, and E, representing four different offices. So instead of counting the total number of rows, you want to count the number of rows that have each unique office_id.

In effect, what you want to do is split up the employees table into four separate tables. Each representing the employees in one particular office. Then count the number of rows in each of those four separate tables. To do this, you add a GROUP By clause to the SELECT statement. The GROUP BY clause specifies which column to use to split up the table into groups. It comes after the FROM clause. In this example, to split up the employees table by office, you use "GROUP BY office_id." When you run this query, it returns a result set with four rows. One row for each unique value of office_id. Like before, the result set has just one column which now gives the count of the number of rows that have each unique office_id. But with just this one column, you cannot tell which count corresponds to which office_id. So you can add office_id to the select list and this makes the query return two columns. The first gives the office_id and the second gives the count of rows that have that office_id. This now answers the question we started with, how many employees are there in each office? There's one employee in office C, one in office E, two in office B, and one in office A.

You can optionally use a column alias to control the names of the columns in the result set just like you can with any SQL query. So for example, you could add "AS num_employees" after count star. So this example demonstrates the basic syntax of a SELECT statement with a GROUP BY clause and the shape of the result set it returns. Recall that the order of the rows in a result set is arbitrary. So these rows might be in a different order for you, but their counts in the second column will still match up with the corresponding office_ids in the first column, the same way they do here. If you're familiar with spreadsheets or other software that can produce summary tables or pivot tables, you might be more accustomed to seeing the results presented like this with the rows and columns transposed so each group, each office is represented as a column.

But SQL engines do not return results sets this way. SQL queries with a GROUP BY clause always return one row per group. The columns of the result set are specified by the select list, just like with any SQL query. You can use the GROUP BY clause together with the WHERE clause in a SELECT statement. The WHERE clause, if you use it, always comes before the GROUP BY clause. It filters the individual rows of the table before they're grouped and aggregated. For example, say you wanted to know how many employees with a salary of more than $35,000 are there in each office. To answer this, you would use the same query as before, but with a WHERE clause added. The WHERE clause filters the rows of the employees table before they're grouped and aggregated.

So the results that only includes the offices where there's at least one employee with a salary greater than $35,000. You can see from the results that there are two employees with a salary greater than 35,000 in office B and one in office E. The absence of the other offices A and C, in this result set, indicates that there are no employees with a salary greater than 35,000 in those offices. In other words, all the employees in the other offices have salary less than or equal to 35,000.

CHOOSING AN AGGREGATE FUNCTION AND GROUPING COLUMN

When you write a SQL query that uses aggregation and grouping, you need to choose which aggregation function to use and which column to group by. Sometimes these choices are straightforward, but other times they're not so obvious. In this video, I will pose some different questions and show SQL queries that answer them using different aggregate functions and different grouping columns. These examples use the inventory table in the [inaudible] database. Each row in the inventory table represents a particular game that's in stock at a particular shop.

There is more than one copy of each game at each shop, and the number of copies is given in the QTY or quantity column. Here's the first question. How many different games does each shop have in stock? Since each row in the inventory table represents a different game in a different shop, you can answer this question by counting rows using the count function. Since it's asking at each shop, you need to group by the shop column. So to answer this question, you would use the query SELECT shop, count (star) FROM inventory GROUP BY shop. The result shows that there are two different games in stock in the dicey shop and three in the board 'Em shop. Now, say you wanted to answer a slightly different question.

How many total games are in stock at each shop? In other words, how many total copies of the games are in stock at each shop? To answer this you need to use the aggregate function sum to add up the values in the quantity column, and again GROUP BY shop. So the query is SELECT shop, sum (quantity) FROM inventory GROUP BY shop. The result shows that the dicey shop has 10 total games in stock and the board 'Em shop has 18. With questions like these, language can be ambiguous.

If someone asks how many games? Do they mean how many different games or how many total copies of the games? When you're working as a data analyst, if you're asked an ambiguous question like this, you should ask for clarification or make some reasonable assumptions and then clearly communicate the assumptions when you share your results. Here's another example. Say you want to answer the question, how many total copies of each game are in stock? In this question the grouping of interests is not shops, it's games. So to answer this question you would use a select statement like this.

SELECT game, sum (quantity) as total quantity FROM inventory GROUP BY game. The previous examples used the shop column in the group by clause and also in the first position in the select list. But this example uses the game column there. The results that shows that there are 18 copies of monopoly, three of clue, four of candy land, and three of risk. But once again, questions like this can be ambiguous. Was the intent really to consider the combined inventory of both shops? That's what this query does. Maybe the intent was to count the copies of each game separately for each shop.

In the inventory table, each row represents one particular game in one particular shop. So to count how many copies of each game are in stock at each shop, you actually don't need to use grouping an aggregation at all, you can just answer the question simply by looking at the quantity values in the rows of the table. So whenever you're writing a query to answer a question about some grouping of the data in a table, always consider whether the rows of the table already represent the grouping you're looking for.

GROUPING EXPRESSIONS

The simplest kind of GROUP BY clause, consists of, the key words GROUP BY, followed by, a column reference. For example, if you are querying the games table in the fun data base, you could use a GROUP BY clause like GROUP BY min_age or GROUP BY max_players. But this is not the only form of a group by clause, in this video you'll learn about what else you can do in this clause. Besides a column reference you can also use an expression in the GROUP BY clause. For example say you want to define two groups in the game's table. The group of games that cost less than $10 and the group that cost $10 or more.

There is no existing column in the games table that defines these two groups, but you can write an expression that uses the list price column to define them. The simplest way to do this is with the Boolean expression list_price<10 this expression returns a true or false value for each row or a one or a zero value with some SQL engines. So all the games are grouped into these two categories. Another way to do this is to use the if function in the GROUP BY clause. If list_price<10, low price, high price. This also groups all the games into two categories but here the groups are defined by the character strings, low price and high price. And you could also do this with a case expression. That would give you the flexibility to define more than two categories if you wanted to. .When you use a grouping expression like in these examples, you generally need to use the expression in the GROUP BY clause and also in the Select list.

If you don't include the grouping expression in the select list, then you can't tell which row in the result set corresponds to which group. So in this example, to count how many games are in each of these two price categories, you would use the SELECT statement, SELECT list_price <10, COUNT (*) FROM games GROUP BY list_price > 10. The result set shows that there are 2 low-price games and 3 higher-price games. I used the simple Boolean expression in this example but you could use the the if function or a case expression. Whichever way you wanted to write the expression, you would just repeat it in both the Select list and in the GROUP BY clause.

But repeating an expression like this in two places can be cumbersome, so some SQL engines offer a shortcut. With some SQL engines, you can specify the grouping expression in the SELECT list. Give it an alias and then use that alias in the GROUP BY clause. This way you do not need to repeat the grouping expression twice. This is especially useful when the expression is long or complex. This shortcut works with Impala, MySQL, and post PostgreSQL, but not with Hive, and not with many other SQL engines. If you're using some other engine, try it to see if it works. The reason this does not work with Hive and others Is that SQL engines generally process the group by clause before they compute the expressions in the SELECT list. Recall that there is a similar limitation with the where clause.

But the developers of Impala, MySQL, and PostgreSQL implemented a work around to allow column aliases in the group by clause even though it goes against the usual order in which the clauses are processed. With Hive and the other SQL engines that do not support this shortcut, you can still use an alias to give a name to the grouping column in the result set but you cannot use that alias in the group by clause. All the examples I've presented so far had only one column reference or expression in the GROUP BY clause, but you can use more than one. After the keywords GROUP BY, you can specify a list of column references or expressions separated by commas. This is called the GROUP BY list. Here's an example with a group by list that has two items. This statement groups by min_age and max_players. It returns the counts of the number of rows in the groups defined by these two columns.

When you specify two or more items in the GROUP BY list, the sequel engine splits up the data into one group for each combination of the values that occur in these grouping column. It splits the data into groups by the first column specified in the GROUP BY clause. And then splits those groups further by the next column specified and so on. Finally, it computes the specified aggregates on those groups. In this example every game in the games table has one of these four combinations of min_age and max_players. You can see in the results set that two games have min_age 8 and max_players 6, so the count in this row is 2. The other three games all have unique combinations of min_age and max_players, so those counts are all 1. This example uses column references in the GROUP BY list. But the items the list can be color reference, expression and column areas if the SQL engine support them in the group by clause. You can include any name of these. Using example that uses a column reference and an expression. If you're using a SQL engine that allow aliases in the GROUP BY clause, then you could rewrite the statement this way to avoid repeating the expression. These examples used two items in the GROUP BY list, but you can use three or more, and of course, you can use other aggregate functions besides count. And you can include multiple aggregate expressions in the SELECT list.

GROUPING AND AGGREGATION, TOGETHER AND SEPARATELY

Grouping and aggregation go hand in hand. In practice, they're mostly used together. But you can use an aggregate expression without a GROUP BY clause, and it's also possible to use a GROUP BY clause without an aggregate expression. In this video, I'll describe the rules governing how you can use each one without the other. You can't use aggregation without grouping.

You can use an aggregate expression in the select list in a statement that has no GROUP BY clause. When you do this, the SQL engine aggregates the whole table down to one row in the results set. In effect, when you use an aggregate expression with no GROUP BY clause, there is an implicit GROUP BY clause that creates one group for the whole table. You can also use two or more aggregate expressions in the select list with no GROUP BY clause. The result will still have just one row, but it will have multiple columns, one for each of the expressions in the select list. Now what about grouping without aggregation? You might imagine that if you ran a query like select star from games, GROUP BY min age, that maybe it would return a set of multiple result tables, each with one unique value of min age.

But SQL does not work that way. A query result can have only one table in it. When you use a GROUP BY clause, the select list must consist only of aggregate expressions, the expressions used in the GROUP BY list, and literal values. Here are some examples to demonstrate what you can do. Here's the statement with a GROUP BY clause but no aggregate expression. The select list has just one item which is the grouping column, min age. This statement returns three rows, one for each unique value of min age, and just one column, giving those values of min age. There's nothing else in the select list so that's all it returns.

You could run a statement like this just to return the unique values of min age, but it's typically better to do this using SELECT DISTINCT. Like this, SELECT DISTINCT min age from games. That statement expresses the intent of your query more clearly. It's better to use the GROUP BY clause only when you're actually going to compute aggregates like in this example. The select list here consists of min age, the grouping column, and max list price, an aggregate expression. This is the classic form of a simple GROUP BY query. Here's an example that also has a literal value in the select list. When you use a literal value in the select list in a statement with a GROUP BY clause, that literal value is just repeated in each row of the result set. This is often not useful, but there are some cases where you might want to do it like in this example.

This statement groups the games table into three groups based on min age, and returns the average list price, the tax rate, and the average list price with tax for the games in these three groups. The tax rate is a constant value of 21 percent. It does not vary between the groups. So, it's specified here as a literal value, 0.21. In this case, it is helpful to display this literal value in the results set even though it's just the same value repeated in each row. So, to review, when you use a GROUP BY clause, the select list must consist only of aggregate expressions, the expressions used in the GROUP BY list, and literal values. Although you can use a GROUP BY clause with no aggregate expressions in the select list, it's better to use SELECT DISTINCT for that instead. Unfortunately, not all SQL engines enforce these rules for what's allowed in a select statement when you use a GROUP BY clause. MySQL for example.

MySQL is a fine relational database system I've used it a lot, but it has one really atrocious behavior in my opinion. It allows you to use non aggregate expressions in the select list in a query that has a GROUP BY clause. For example, with MySQL, you can run a query like select star from games grouped by min age. Most SQL engines will throw an error if you try to run a query like this but not MySQL. What MySQL does is it splits the games table into three groups based on min age, and then it picks one arbitrary row from each group to include in the results set. If that seems strange to you, I agree.

Similarly, MySQL will let you run this query which has in the select list min age, the column that's in the GROUP BY clause, and list price, a column that's not in the clause GROUP BY with no aggregation performed on it. In the results at the min age column gives the three unique values of min age, but the list price column just gives prices picked from arbitrary rows in the corresponding groups. To me, SQL engines should not behave this way. There are some logical reasons for the behavior but overall I think it causes much more confusion than it's worth, and it's best to avoid writing queries like this. More about Grouping and Aggregation

NULL Values in Grouping and Aggregation

In this video, you'll learn how SQL engines handle NULL values when they group and aggregate data. But first, recall how NULL values are handled in queries that do not group or aggregate. Here are some examples using the inventory table in the fun database. In the inventory table, there are a couple of NULL values. One in the aisle column and one in the price column. When you run a query that returns NULL values in the results set, they are represented by the keyword NULL.

Although the way they're displayed might vary depending on what software you're using to display the results. When you use an expression with arithmetic operators or with most non aggregate functions, when an operand or argument is NULL, the expression evaluates to NULL. For example, the NULL value in the price column multiplied by 1.21 and rounded to two digits after the decimal returns NULL. There are some special conditional functions that take a NULL argument and return a non-NULL value, but those are a special case. When you use any of the standard comparison operators in an expression, then any value compared to NULL yields NULL.

So when the NULL value in the price column is compared to 10 in an inequality expression, that returns NULL. Finally, recall that when you use a WHERE clause, rows in which the Boolean expression in the WHERE clause evaluates to NULL are omitted from the results just like the rows where it evaluates to false. So that's a review of how NULL values are handled in queries that do not use grouping or aggregation. Now let's consider how SQL engines handle NULL values in aggregation. Here's a simple example that uses the aggregate function avg to compute the average of the values in the price column for all the rows in the inventory table.

Although one of the price values is NULL, this query does not return NULL, It returns 21.99. That's because aggregate functions in SQL ignore NULL values. Instead of returning a NULL result when some of the values being aggregated are NULL, SQL engines just ignore the NULL values and use only the non-NULL values to compute aggregates. Some other languages and systems work differently, but this is what SQL engines do. The only case in which an aggregate function will return a NULL, is when there are no non-NULL values to aggregate. For example, if you use a WHERE clause to filter out all the games except Candy Land, then take the average price, the query returns NULL because there are no rows with game equals Candy Land that have a non-NULL price.

If you add a GROUP BY clause to the query, the same principles apply. Within each group, NULL values are ignored and the aggregates are computed using the non-NULL values. So although there are three games in the Board 'Em shop, the average price of a game in that shop, $30, is calculated using just the two non-NULL prices, $25 and $35. But if you GROUP BY game instead of by shop, then there is one game, Candy Land, that has no non-NULL prices in the data. So in that case since there are no non-NULL prices in that group, the average price for the group is NULL. These examples all use the AVG function, but the same principles apply for SUM, MIN, and MAX. Finally, let's consider how SQL engines handle NULL values in grouping columns. I'll use the inventory table to demonstrate this. Notice that there is a NULL value in the aisle column. Say we GROUP BY that aisle column. Then, in the results set there is a row representing the group in which aisle is NULL. So when there are NULL values in a column that you use in the GROUP BY clause, a NULL group is created and it includes all the rows in which that column value is NULL.

WHY AGGREGATE EXPRESSIONS IGNORE NULL VALUES

The previous video described how aggregate expressions handle **NULL** values differently than non-aggregate (scalar) expressions. This reading describes the reasons for this difference, and warns about how it can cause misinterpretations. For a scalar expression, it would be misleading to report anything except **NULL** in individual row values containing **NULL**s in the operands or arguments of the expression. (Review the lesson, “Working with Missing Values,” in Week 3 of this course for more about why this is so.)

However, for aggregate expressions, if **NULL**s were not ignored, then just one **NULL** value in a large group of rows would cause the query to return a **NULL** result as the aggregate for the whole group. By ignoring the **NULL** values, aggregate expressions are able to return meaningful results even when there are **NULL** values in the groups.

But sometimes this behavior can lead to misinterpretations, especially with sparse data. For example, if you compute the average of a column in a table with ten million rows, but only three of those rows have a non-**NULL** value in the column you’re averaging, then the query would return a non-**NULL** average in the result.

This might mislead you into thinking that this average provides meaningful information about all ten million rows, when it reality the average comes from only three rows, and there is probably no reason to believe it is representative of all ten million rows.

Therefore, it is important to explicitlycheckfor **NULL** values and handle them in your queries, instead of just relying on aggregate expressions to ignore them. One way to do this is to use an aggregate expression like:
SUM(*column* IS NOT NULL)
to return the number of rows in which *column* is non-**NULL**. In this expression, ***column*** **IS NOT NULL** evaluates to **true** (**1**) or **false** (**0**) for each row, and the **SUM** function adds these 1s and 0s up and returns the number of rows in which ***column*** **IS NOT NULL**.
For example, when you run the following query, the second column in the result tells you exactly how many non-**NULL** values were used to compute each of the averages in the third column:
SELECT shop, SUM(price IS NOT NULL), AVG(price) FROM inventory GROUP BY shop;

shop	SUM(price IS NOT NULL)	AVG(price)
Dicey	2	13.99
Board 'Em	2	30.00

The next video describes another way to check for **NULL** values in aggregates.

THE COUNT FUNCTION

Ah, greetings. I am The Count. Do you know? Oh! Look, a table with rows. I will count the rows. One, one row. Two, two rows. Three, three rows. Four, four rows. Five, five rows! Okay. That's enough of that. So as you saw, the inventory table in the fun database has five rows. Recall that you can return the number of rows in a table by using the COUNT function with an asterisk, a star, as the argument. So COUNT(*) FROM inventory returns five. You can also use a GROUP BY clause. If you do, then COUNT(*) returns the number of rows in each group. For example, this query uses COUNT(*) and GROUP BY shop, and the result tells you that there are two games in the Dicey shop and three games in the Board 'Em shop.

But there is another way to use the COUNT function. Instead of using * as the argument, you can specify a column reference as the argument. When you do this, the COUNT function does something different: it returns the number of rows in which that column has a non-NULL value. For example, if I run the query, SELECT COUNT(price) FROM inventory, then the result is not five, it's four because there are four non-NULL values in the price column. If you include a GROUP BY clause, then the result is not two and three like before, it's two and two; there are two games in the Dicey shop with a non-NULL price and two games in the Board 'Em shop with a non-NULL price.

So when you use a column reference or it could be an expression as the argument to the COUNT function, then the COUNT function does not count the missing values in that column, it ignores them. To understand why this is, remember that this is what the other aggregate functions do, the functions like SUM and AVG, they ignore NULL values. The COUNT function was designed to be consistent with these other aggregate functions except in the case where you use COUNT(*). So the general rule is that aggregate functions ignore NULL values, and the one exception to that rule is when you use COUNT(*).

The COUNT function has another useful feature. You can use it to count the number of distinct values, unique values, in a column. To do this, you use the keyword DISTINCT inside the parentheses after COUNT. For example, to count the number of unique values in the aisle column in the inventory table, you would run the query SELECT COUNT(DISTINCT aisle) FROM inventory. This returns three, which tells you that there are three unique non-NULL values in the aisle column. NULL values are not counted regardless of whether or not you use the DISTINCT keyword. With some SQL engines, you can specify more than one column reference or expression after the DISTINCT keyword in the COUNT function.

This returns the number of unique combinations of the specified columns that exist in the data. This works in Hive, Impala, and MySQL, but not in PostgreSQL. Also, with some SQL engines, you can use more than one COUNT DISTINCT in a SELECT list, like in this example which uses the crayons table. Here, the COUNT function is used three separate times in the SELECT list and the DISTINCT keyword is included in all three. So the result set has three columns giving the number of unique values in the red column, the number of unique values in the green column, and the number of unique values in the blue column. But with some other SQL engines including Impala, you are limited to only one COUNT DISTINCT per SELECT list. Impala will throw an error if you try to run this query. In the examples I just presented, the DISTINCT keyword is used with the COUNT function inside the parentheses after COUNT. But recall that this is not the only place where you can use the DISTINCT keyword, you can also use it just after the SELECT keyword without any aggregate function. That returns the unique rows of the table instead of returning the count of how many unique values or combinations of values there are.

In SQL, the opposite of DISTINCT is ALL. In fact, you can use the ALL keyword in both of these places where you can use the DISTINCT keyword. But using the ALL keyword does nothing because in both of these cases the default behavior when you use no keyword is the same as what you get when you explicitly use the ALL keyword. You can actually use the DISTINCT keyword with other aggregate functions besides COUNT, but there is not usually any good reason to do that. For example, there is usually no good reason to calculate the sum or the average of all the unique values in a column. You can do this, but typically the result does not answer any practical question. If you're using the MIN or MAX aggregate functions, you should not use the DISTINCT keyword because the minimum or maximum of the unique values is always the same as the minimum or maximum of all the values.

So the COUNT function is unique among the common aggregate functions because it's the only one that is often used with the DISTINCT keyword. The COUNT function is also the only one of these common aggregate functions that you'll often see used with character string columns. You can't find the sum or average of a character string column, those aggregate functions are for numeric columns. Although you can find the min or max of a string column according to their lexicographical order, it's not so common for data analysts to need to do that. But finding the count of the values in a string column is something data analysts often need to do.

Tips for Applying Grouping and Aggregation

Shortcuts for Grouping

This reading describes two techniques you can use to save time and make your SQL queries more concise when you're using the GROUP BY clause.
For example, consider this query from the "Tips for Applying Grouping and Aggregation" video:

```
SELECT MIN(dep_time), MAX(dep_time), COUNT(*)
   FROM flights
   GROUP BY CASE WHEN dep_time IS NULL then 'missing'
            WHEN dep_time < 500 then 'night'
            WHEN dep_time < 1200 THEN 'morning'
            WHEN dep_time < 1700 THEN 'afternoon'
            WHEN dep_time < 2200 THEN 'evening'
            ELSE 'night'
      END;
```

Note that this query doesn't actually include the grouping column in the output:
Results

```
min(dep_time) max(dep_time) count(*)
1200          1659          18366410
500           1159          24513240
NULL          NULL          961944
1             2400          2067458
1700          2159          15483770
```

Instead it uses MIN(dep_time) and MAX(dep_time) as a way to indicate which of these time bins reach row in the result set represents. This results in a curious row that appears to encompass all values from 1 to 2400; this is actually the group defined by WHEN dep_time < 500 then 'night' (including values from 0 to 499) and the ELSE 'night' (including values from 2200 to 2400).
To include the grouping column in the output with Hive and some other SQL engines, you would have to do this:

```
SELECT CASE WHEN dep_time IS NULL then 'missing'
         WHEN dep_time < 500 then 'night'
         WHEN dep_time < 1200 THEN 'morning'
         WHEN dep_time < 1700 THEN 'afternoon'
         WHEN dep_time < 2200 THEN 'evening'
         ELSE 'night'
      END AS dep_time_category,
      COUNT(*)
```

```
  FROM flights
  GROUP BY CASE WHEN dep_time IS NULL then 'missing'
        WHEN dep_time < 500 then 'night'
        WHEN dep_time < 1200 THEN 'morning'
        WHEN dep_time < 1700 THEN 'afternoon'
        WHEN dep_time < 2200 THEN 'evening'
        ELSE 'night'
      END;
```

Results

dep_time_category	count(*)
afternoon	18366410
morning	24513240
missing	961944
night	2067458
evening	15483770

That gives a result that's more easily understood, but it's a long, repetitive query!
Using an Alias in the SELECT List
With Impala, MySQL, and PostgreSQL, you can use an alias in the SELECT list and then refer to it in the GROUP BY clause. That is, you can use this query instead:

```
SELECT CASE WHEN dep_time IS NULL then 'missing'
        WHEN dep_time < 500 then 'night'
        WHEN dep_time < 1200 THEN 'morning'
        WHEN dep_time < 1700 THEN 'afternoon'
        WHEN dep_time < 2200 THEN 'evening'
        ELSE 'night'
      END AS dep_time_category,
      COUNT(*)
  FROM flights
  GROUP BY dep_time_category;
```

This produces the same results, but in a more concise query.
Using Positional References
Another way to do this with Impala, MySQL, PostgreSQL, and newer versions of Hive (but not in older versions of Hive and not in some other SQL engines) is to use an integer (1, 2, and so on) as the grouping expression, and the engine will use the corresponding column in the SELECT list as the grouping column. If you

use GROUP BY 3, then the third column you specify in your SELECT list will be the grouping column.
This means you could also use this query to get the same results for the departure time category:

```
SELECT CASE WHEN dep_time IS NULL then 'missing'
         WHEN dep_time < 500 then 'night'
         WHEN dep_time < 1200 THEN 'morning'
         WHEN dep_time < 1700 THEN 'afternoon'
         WHEN dep_time < 2200 THEN 'evening'
         ELSE 'night'
      END AS dep_time_category,
      COUNT(*)
   FROM flights
   GROUP BY 1;
```

Since dep_time_category is the first column in the SELECT list, GROUP BY 1 directs the SQL engine to group by that column.

NOTE: In general, this shortcut method is less preferable, because it's harder to see what your query does, and it could cause trouble if you changed your SELECT list but forgot to change your ORDER BY clause.
Another Example
The tables in the fly database are not available to the MySQL and PostgreSQL engines in the VM. If you want to test this on the VM using either of those databases, you can try these queries:
Using an alias:

```
SELECT CASE
         WHEN price <= 10 THEN 'inexpensive'
         WHEN price > 10 THEN 'expensive'
         ELSE 'unknown'
      END AS price_category,
      COUNT(*)
   FROM inventory
   GROUP BY price_category;
```

Using positional reference:

```
SELECT CASE
         WHEN price <= 10 THEN 'inexpensive'
         WHEN price > 10 THEN 'expensive'
```

```
        ELSE 'unknown'
      END AS price_category,
      COUNT(*)
    FROM inventory
    GROUP BY 1;
```

How Grouping and Aggregation Can Mislead

Care must be taken when grouping. It's possible to produce misleading results, or even results that seem contradictory.

In the **fly.flights** table, if you compare average on-time performance, **AVG(arr_delay)**, over all flights for the carriers Virgin America (carrier code **VX**) and SkyWest Airlines Inc. (carrier code **OO**), then SkyWest has a better average delay (approximately 5.7 minutes) than Virgin (approximately 6.5 minutes). You might conclude, then, that Virgin is worse than SkyWest in terms of delays, and when given a choice for a particular trip between two cities, choose SkyWest.

However, Virgin would actually be a slightly better choice in that case (and if arrival delay is your only criterion)! If you limit the data to the airports where both airlines have flights, then Virgin looks slightly better than SkyWest (9.5 minutes for Virgin and 9.7 minutes for SkyWest). It might not be a problem with Virgin being worse than SkyWest, then, in terms of delays. Instead, the problem could be with the airports where they operate. The airports where Virgin operates overall have worse delays than the airports where SkyWest operates, so Virgin's average on-time performance over ***all*** flights looks worse than SkyWest.

(Note: The queries for these comparisons require some techniques you haven't learned yet, but they are included at the end of this reading in case you want to try them.)

In this case, the airports is a ***confounding variable***—an underlying variable that affects each of the other variables that you are examining. The airports themselves can be a source of delay (San Francisco often has delays on account of fog, for example), and the carriers work with different airports. This underlying variable makes a difference, so a good comparison needs to accommodate that variable.

Another example is Simpson's Paradox, in which grouping in one way can provide one conclusion for every single group, but when taken as a whole, the ***opposite*** conclusion is reached. This is often because of a significant difference in sample size for the groups. For example, a study of kidney stone treatment found that while one treatment appeared to be more effective for both small stones and for large stones, when you looked at all the cases together, the other treatment appeared to be more effective. (See the Wikipedia page, Simpson's paradox, for this and other examples. "Simpson's Paradox: How to Prove Opposite Arguments with the Same Dataset" also explains this phenomenon well.)

	Treatment A	**Treatment B**
Small stones	***Group 1*** 93% (81/87)	***Group 2*** 87% (234/270)
Large stones	***Group 3*** 73% (192/263)	***Group 4*** 69% (55/80)
Both	78% (273/350)	83% (289/350)

Notice that treatment A was provided about three times as much for large stones as for small stones, while treatment B was provided about three times as much for small stones as for large stones. The severity (size) of the stones is the confounding variable.

To avoid making conclusions from misleading data, you might:

- Use different levels of aggregation, including the highest aggregation and no aggregation at all, if possible, when looking at the data;
- Try different ways to group your data, to be sure your choice of groups isn't causing an effect that disappears or reverses for different choices; and
- Include counts in your results so you can see when one group has a significantly different number of contributions than others.

QUERIES USED TO COMPARE CARRIERS

These queries use some techniques you haven't learned yet, but they are included here in case you want to try them.

Comparing Virgin to SkyWest, all flights:

```
SELECT carrier, AVG(arr_delay), COUNT(arr_delay)
   FROM flights WHERE carrier='VX' OR carrier='OO'
   GROUP BY carrier ORDER BY avg(arr_delay);
```

```
carrier   avg(arr_delay)    count(arr_delay)
OO        5.685446716780021 5926697
VX        6.484657383617123 367408
```

Comparing Virgin to SkyWest, identical origins and destinations:

```
SELECT f.carrier, avg(f.arr_delay), count(f.arr_delay)
   FROM flights f
   JOIN
      (SELECT DISTINCT origin, dest FROM flights WHERE carrier='VX') vx
         ON (f.origin=vx.origin AND f.dest=vx.dest)
   JOIN
      (SELECT DISTINCT origin, dest FROM flights WHERE carrier='OO') oo
         ON (f.origin=oo.origin AND f.dest=oo.dest)
   WHERE carrier='VX' OR carrier='OO'
   GROUP BY f.carrier ORDER BY avg(f.arr_delay);
```

Note that this query uses the JOIN keyword to combine tables; you'll learn about this in Week 6 of this course. It also uses ***subqueries*** to isolate the origin/destination pairs for each carrier; you'll learn about subqueries if you continue to the fourth course in this specialization.

```
carrier   avg(f.arr_delay)  count(f.arr_delay)
OO        9.72568710815216  231914
```

carrier	avg(f.arr_delay)	count(f.arr_delay)
VX	9.546630527151848	180043

THE HAVING CLAUSE

Two of the techniques you've learned about so far in this course are filtering data using the WHERE clause and grouping and aggregating data using the GROUP BY clause and aggregate expressions. Equipped with your knowledge of these two techniques, you might try to use an aggregate expression in the WHERE clause to filter the results by an aggregate column. However, this does not work. Here's an example to demonstrate this. First, this query which is valid, it does work., it groups the inventory table by shop and returns the name of each shop and the sum of the prices of all the copies of the game in each shop. That is, the sum of the price of each game times the quantity in stock. So the results set has two rows representing the two shops. The second column in the results set gives the sum of price times quantity for the games in each shop. In other words, the total retail value of the games in each shop.

Now, say you wanted to filter these results to return only the shops that have an inventory with a total retail value greater than $300. You might try to do this by adding the clause where sum of price times quantity, greater than 300. But this query will fail in all major SQL engines. This query fails because the WHERE clause can only filter individual rows of data. You cannot use the WHERE clause to filter based on aggregates of the data. Also, SQL engines process the WHERE clause before the GROUP BY clause.

So when the WHERE clause is processed, the SQL engine doesn't yet know what groups the data has been split into, because that splitting hasn't happened yet. So there is a strict rule in SQL that you cannot use aggregate expressions in the WHERE clause. So how can you write a select statement that filters based on aggregates? Since you cannot use the WHERE clause to do this, SQL has a different Clause for this. The HAVING clause. You learn how to use this clause next. Filtering on Aggregates

DISCUSSION PROMPT: THE ANALYTIC JOURNEY

Most of the problems you get in this course have clear solutions—you run the query, answer the question, and you’re done. In real life, it's rarely so clear or simple. You might find a strange result and wonder what's causing it, then write and execute many more queries to try to find the root of the strange result. Data analysis is often more about exploring data than getting simple reports.

In the final In-Video Question of the “The HAVING Clause” video, you should have seen a calculation for average flight speed of two flights from MCO to JAX that couldn't possibly be correct. It gave the average speed as 1251.25 miles per hour, which is well over Mach 1 (the speed of sound) and getting close to Mach 2 (twice the speed of sound).

Dig into this mystery. Try to solve it on your own, and then come back here to see how others approached it. Contribute your own approach, thoughts, and results. If the mystery hasn't yet been solved, collaborate with your peers! Ask questions and get ideas on what to look for or what to try.
Participation is optional

WORKING WITH DIFFERENT VERSIONS OF HUE, HIVE, AND IMPALA

In the videos in this honor's lesson, you'll learn some tips for working in some different versions of Hive, Impala, and Hue. When you're working as a data analyst in the real world, it's important to be able to adapt to different versions of these tools. The VM that you've been using for this course has specific versions of Hive and Impala and Hue installed on it. But a company or organization you're working for might use different version. Also, if you're interested in taking the Cloudera Certified Associate Data Analyst certification exam, the exact versions of Hive, Impala and Hue that you'll need to use to complete that exam might not match the versions on this course VM.

So you'll need to be prepared to deal with different versions. In this video, I'll discuss Hive and Impala, then in the next video, I'll talk about Hue. Over time, additional capabilities have been added to Hive and Impala, and some of the default behaviors have changed. So if you're using a new or unfamiliar instance of Hive and Impala, or if there might have been a version update on the instance you're using, it's good to check what the version is. To see exactly what version of hive or Impala you're using, run the SQL statement, SELECT version();. Version is a special built-in function that returns a character string containing version information. When you run SELECT version, that returns a result with a single row and a single column containing that character string. The most important part to look for is the first set of numbers that appears in that string.

For example, 2.10.0 for Impala, or 1.1.0 for Hive. If you're using a version of Hive or Impala that was distributed by Cloudera, then you'll also see a Cloudera platform version number after cdh. In both of the examples shown here, their Cloudera platform version is 5.13.0. After those numbers, you might also see some build information but you can usually ignore that. The examples here show the output when you use this version function in Hive and Impala, but you can use the version function with many other SQL engines to including MySQL and PostgreSQL. Once you know what version of Hive or Impala you're using, the best way to get detailed information about that version is to review the documentation. For Hive, you can find the documentation by going to hive.apache.org and clicking the link for language manual.

For Impala, you can go to impala.apache.org and click the link for documentation. However for Impala, if you're using a version that was distributed by Cloudera, it's easier to use the Impala documentation that's hosted on Cloudera's website. To access that, follow the provided link. I'll first show the Hive LanguageManual and demonstrate how you can find version specific information there. The Hive LanguageManual is structured as a wiki that members of the hive developer and user communities can contribute to. From the main LanguageManual page, you can click to access subpages. Under Data Retrieval Queries, I'll click the link for Select.

And here, you can see that there are many details about the syntax of the select statement in HiveQL, which is the name for Hive's dialect of SQL. Interspersed throughout this content, you will see references to changes that occurred in different versions of Hive. For example, under the heading ALL and DISTINCT Clauses, there is a note that says, Hive supports SELECT DISTINCT star starting in release 1.1.0. You might recall that a SELECT DISTINCT star query returned the distinct full rows in a table. Hive versions earlier than 1.1.0 did not support this. The version on the course VM that you've been using is 1.1.0 or later so it does support this. Often, you can get additional information about a feature that was added to Hive by clicking a link included in the note.

This takes you to the Apache Hive issue tracking system. There is often lots of technical information included here that's beyond the scope of this course, but it can be helpful to read the title and description fields. And to check the fix version, which tells you what version of Hive first had this feature. One page in the Hive LanguageManual that is especially useful to consult for version information is the Operators and User-Defined Functions or UDFs page. The title of this page is a bit confusing, with Hive when people use the term User-Defined Function or UDF, this often encompasses built-in functions. On this page, there are sections listing the different types of operators and built-in functions available in Hive.

For example, there's a section listing Hive's conditional functions. I'll click the link to go to that section. In the description field for some of these functions, you'll see notes indicating the version of Hive in which the function was first included. For example, the nullif function was added in Hive version 2.3.0. So if the version of Hive you're using is 2.3.0 or higher, then the nullif function is available. Otherwise, it's not available. If you're using a version of Hive that was distributed by Cloudera, there are some cases where the Cloudera engineers make a new feature or function available early.

So it's a good idea to test the feature or the function yourself on the version of Hive you're using to verify that it's consistent with what the hive documentation says. Now, I'll show how to use the Impala documentation that's hosted on Cloudera's website. There are many different sections of Cloudera's Impala documentation. But I'll focus here on the Impala sequel language reference which you can get to by following the provided link. From this main page, you can click to access subpages. I'll click to go to the page for built-in function. Here you can see further subpages for the different categories of built-in functions.

I'll click to go to the Impala Conditional Functions page. Impala and Hive have many of the same built-in functions, but there are some differences. So the least of functions here does not exactly match the list I showed in the Hive documentation. In the descriptions of many of the functions here, there are notes indicating what version of Impala the function was added in. For example, the nullif function was added to Impala in version 1.3.0. In some of the notes, you'll also see a Cloudera platform version number which begins with CDH. That's because recent versions of Impala are bundled with specific versions of the Cloudera platform. And it's sometimes more convenient to use this Cloudera platform version number.

If you know which version of the Cloudera platform you're using, in other words which CDH version you're using, then there is a trick that can help you navigate the Impala documentation. In the URL for all of the Impala documentation pages, you should see the word latest. This means that you're viewing the documentation for the most recent version of the Cloudera platform. You can see the specific version number this corresponds to at the top of the page. In the URL, you can replace the word latest with a specific CDH version number to see the Impala documentation for that specific version of the Cloudera platform.

But you need to format the version number a certain way. You use dashes instead of dots and change the final digit to an x. For example, if you're using version 5.13.0 of the Cloudera platform, then you can replace latest with 5-13-x. After making that change to the URL, I'll press Enter. And now, I'm viewing a different version of the Impala documentation that specifically describes the version of Impala that's bundled with the 5.13 versions of the Cloudera platform. I recommend using this trick whenever you're using Impala on a specific version of the Cloudera platform that's not the latest version.

By using the Hive and Impala documentation in the ways I described in this video, you should be able to resolve most questions about what features are available in what versions of Hive and Impala. As you browse the documentation, you'll notice there is a lot of information there about topics that you have not yet learned about in this course. If you see something in the documentation that you don't understand, it's okay to just ignore it for now. We'll cover many of these topics later in this course and in the other course as they're part of this specialization.

UNDERSTANDING HIVE AND IMPALA VERSION DIFFERENCES

In the videos in this honor's lesson, you'll learn some tips for working in some different versions of Hive, Impala, and Hue. When you're working as a data analyst in the real world, it's important to be able to adapt to different versions of these tools. The VM that you've been using for this course has specific versions of Hive and Impala and Hue installed on it. But a company or organization you're working for might use different version. Also, if you're interested in taking the Cloudera Certified Associate Data Analyst certification exam, the exact versions of

Hive, Impala and Hue that you'll need to use to complete that exam might not match the versions on this course VM.

So you'll need to be prepared to deal with different versions. In this video, I'll discuss Hive and Impala, then in the next video, I'll talk about Hue. Over time, additional capabilities have been added to Hive and Impala, and some of the default behaviors have changed. So if you're using a new or unfamiliar instance of Hive and Impala, or if there might have been a version update on the instance you're using, it's good to check what the version is. To see exactly what version of hive or Impala you're using, run the SQL statement, SELECT version();. Version is a special built-in function that returns a character string containing version information. When you run SELECT version, that returns a result with a single row and a single column containing that character string. The most important part to look for is the first set of numbers that appears in that string. For example, 2.10.0 for Impala, or 1.1.0 for Hive. If you're using a version of Hive or Impala that was distributed by Cloudera, then you'll also see a Cloudera platform version number after cdh. In both of the examples shown here, their Cloudera platform version is 5.13.0. After those numbers, you might also see some build information but you can usually ignore that. The examples here show the output when you use this version function in Hive and Impala, but you can use the version function with many other SQL engines to including MySQL and PostgreSQL.

Once you know what version of Hive or Impala you're using, the best way to get detailed information about that version is to review the documentation. For Hive, you can find the documentation by going to hive.apache.org and clicking the link for language manual. For Impala, you can go to impala.apache.org and click the link for documentation. However for Impala, if you're using a version that was distributed by Cloudera, it's easier to use the Impala documentation that's hosted on Cloudera's website. To access that, follow the provided link.

I'll first show the Hive LanguageManual and demonstrate how you can find version specific information there. The Hive LanguageManual is structured as a wiki that members of the hive developer and user communities can contribute to. From the main LanguageManual page, you can click to access subpages. Under Data Retrieval Queries, I'll click the link for Select. And here, you can see that there are many details about the syntax of the select statement in HiveQL, which is the name for Hive's dialect of SQL. Interspersed throughout this content, you will see references to changes that occurred in different versions of Hive. For example, under the heading ALL and DISTINCT Clauses, there is a note that says, Hive supports SELECT DISTINCT star starting in release 1.1.0.

You might recall that a SELECT DISTINCT star query returned the distinct full rows in a table. Hive versions earlier than 1.1.0 did not support this. The version on the course VM that you've been using is 1.1.0 or later so it does support this. Often, you can get additional information about a feature that was added to Hive by clicking a link included in the note. This takes you to the Apache Hive issue tracking system. There is often lots of technical information included here that's beyond the scope of this course, but it can be helpful to read the title and description fields. And to check the fix version, which tells you what version of Hive first had this feature. One page in the Hive LanguageManual that is especially useful to consult for version information is the Operators and User-Defined Functions or UDFs page.

The title of this page is a bit confusing, with Hive when people use the term User-Defined Function or UDF, this often encompasses built-in functions. On this page, there are sections listing the different types of operators and built-in functions available in Hive. For example, there's a section listing Hive's conditional functions. I'll click the link to go to that section. In the description field for some of these functions, you'll see notes indicating the version of Hive in which the function was first included. For example, the nullif function was added in Hive version 2.3.0. So if the version of Hive you're using is 2.3.0 or higher, then the nullif function is available. Otherwise, it's not available. If you're using a version of Hive that was distributed by Cloudera, there are some cases where the Cloudera engineers make a new feature or function available early.

So it's a good idea to test the feature or the function yourself on the version of Hive you're using to verify that it's consistent with what the hive documentation says. Now, I'll show how to use the Impala documentation that's hosted on Cloudera's website. There are many different sections of Cloudera's Impala documentation. But I'll focus here on the Impala sequel language reference which you can get to by following the provided link. From this main page, you can click to access subpages.

I'll click to go to the page for built-in function. Here you can see further subpages for the different categories of built-in functions. I'll click to go to the Impala Conditional Functions page. Impala and Hive have many of the same built-in functions, but there are some differences. So the least of functions here does not exactly match the list I showed in the Hive documentation. In the descriptions of many of the functions here, there are notes indicating what version of Impala the function was added in.

For example, the nullif function was added to Impala in version 1.3.0. In some of the notes, you'll also see a Cloudera platform version number which begins with CDH. That's because recent versions of Impala are bundled with specific versions of the Cloudera platform. And it's sometimes more convenient to use this Cloudera platform version number. If you know which version of the Cloudera platform you're using, in other words which CDH version you're using, then there is a trick that can help you navigate the Impala documentation. In the URL for all of the Impala documentation pages, you should see the word latest.

This means that you're viewing the documentation for the most recent version of the Cloudera platform. You can see the specific version number this corresponds to at the top of the page. In the URL, you can replace the word latest with a specific CDH version number to see the Impala documentation for that specific version of the Cloudera platform. But you need to format the version number a certain way. You use dashes instead of dots and change the final digit to an x. For example, if you're using version 5.13.0 of the Cloudera platform, then you can replace latest with 5-13-x. After making that change to the URL, I'll press Enter. And now, I'm viewing a different version of the Impala documentation that specifically describes the version of Impala that's bundled with the 5.13 versions of the Cloudera platform.

I recommend using this trick whenever you're using Impala on a specific version of the Cloudera platform that's not the latest version. By using the Hive and Impala documentation in the ways I described in this video, you should be able to resolve most questions about what features are available in what versions of Hive and Impala. As you browse the documentation, you'll notice there is a lot of information there about topics that you have not yet learned about in this course. If you see something in the documentation that you don't understand, it's okay to just ignore it for now. We'll cover many of these topics later in this course and in the other course as they're part of this specialization.

Understanding Hue Version Differences

[BLANK AUDIO] In this brief video, I'll help orient you to some Version Differences you might encounter when using Hue. On the VM for this course, when you open Hue in the web browser, the user interface that you see is the Hue version 4 user interface. This version 4 interface was introduced fairly recently. So in other environments, you might still encounter the older Hue version 3 interface. The Hue 3 interface looks a little different, and some of the menus and links are in different places. For example, to access the Table Browser in Hue 3, you open the Data Browsers menu in the top bar, and click Metastore Tables.

This takes you to the Metastore Manager, which is essentially the same as the Table Browser in Hue 4. And to access the Query Editors in Hue 3, you open the Query Editors menu in the top bar, and click Hive or Impala. In the Query Editors, things work mostly the same as in Hue 4. You can use the assist panel on the left side to browse databases and tables, you can use the active database selector on the upper right.

And you can write and run queries, and view the results. All just like you can in Hue 4. So if you encounter the Hue version 3 interface, try not to get disoriented. Remember that it enables the same actions that you've been doing in the Hue 4 interface, browsing tables and querying with Hive and Impala. Once you familiarize yourself with its different appearance and the different locations of the menus and links, you should have no trouble using it.

WEEK 5

Learning Objectives

- Sort results using ORDER BY in circumstances for which sorted data is needed
- Apply the LIMIT clause in appropriate circumstances
- Identify which parts of the SELECT statement are processed before others
- (Honors) Use documentation as a reference to get details on available capabilities

The ORDER BY Clause

When you're writing a select statement and you want the rows of your results set to be in a specific order, you use the Order By Clause. The order by clause takes the result from all the earlier clauses, select, from, where, and to group by, and it arranges those rows, sorts them in a specific order before returning them to you. I'll demonstrate this with an example but first, recall that when you run a select statement with a distributed SQL engine, if the statement does not have an order by clause, then the order of the rows in the results set is arbitrary and unpredictable.

You could run the exact same query twice on the same data and you could get the rows in a different order each time. On the VM for this course, you will probably not experience this unpredictability of row order to its full extent. On the VM, the distributed SQL engines and the tables they're querying are not actually distributed across multiple computers. They're all just on one computer and this takes away much of the randomness that causes the rows to get shuffled around. Especially, when you're querying the very small tables like this one. So you might forget and start to expect that the rows will always come back in the same order each time. Don't let that happen.

Remember, row order is arbitrary and unpredictable. Unless you explicitly tell the SQL engine to return the rows in a specific order, there's no way to know for sure what order they'll come back in. If you want the rows of your results set to be in a specific order, the way to tell the SQL engine that is to use an order by clause. For example, to return the rows of the games table ordered by the ID column, you would add the clause order by ID to the end of the select statement. Then the rows of the results that are guaranteed to be in this specific order, ID one, two, three, four, five from top to bottom. You can use any column in the order by clause. For example, you can order by list price.

This arranges the rows of the result set according to the values in the list price column with the least expensive game in the top row and the most expensive game in the bottom row, or you can order by max players. But when you do that, notice that there are some rows that have the same max player's values. Both Scrabble and Candy Land have max players four and both monopoly and clue have max players six. When there are ties like this in the column that you order by, then within the sets of rows where those values are tied, the order of those rows is arbitrary. So you might get Scrabble first then Candy Land or Candy Land first then Scrabble, and the same with Monopoly and Clue, they could be in either order. You can include more than one column reference in an order by clause. Here's an example to demonstrate that.

The order by clause in this example has two column references. First, max players and second, list price. So the results set is ordered first by max players then by list price. What this means is that when there are ties in the first sorting column, then the ties are broken using the values in the second column. If you look at the first two rows of the results that they both have the same value of max players, four. So the order of these heroes is determined by list price. So you can specify one or two or even more than two column references in the order by clause. If you have two or more, then you separate them by commas and this comma separated list in the order by clause is called the order by list. You can use the order by clause together in a select statement with other clauses where group by and having, but it must come after those clauses. Here's an example to demonstrate the order by clause with some of these other clauses.

This select statement queries the inventory table, filters out the rows that have a null in the price column, groups by Shop and then returns the name of each shop and the sum of the quantity of games in stock at that shop. Finally, it orders by the shop column. The shop column is a character string column not a numeric column like in the other examples. So what the order by clause does is it arranges the rows of the results that in alphabetical order by shop. Since B comes before D in the alphabet, Board 'Em is first and Dicey is second.

INTRODUCTION TO THE ORDER BY CLAUSE

Controlling Sort Order

When you use an ORDER BY clause in a query, the SQL engine returns the rows in order by the column or columns that you specify. The order they're returned in by default is ascending order. Ascending means from smallest to largest. Smallest at the top, largest at the bottom. Or if you're ordering by a string column, then ascending means in alphabetical order from A to Z. Here's an example using a numeric column. When you run the query, SELECT * FROM games ORDER BY list_price, then in the result set, the least expensive game is in the first row at the top and the most expensive game is in the last row at the bottom. But sometimes you'll want the rows to be arranged in the opposite order, with the largest values at the top and the smallest at the bottom. This is called descending order.

For a string column, descending order is reversed alphabetical order, from Z to A. In SQL, you can arrange rows in descending order by using the keyword DESC in the ORDER BY clause. This keyword goes after the column reference. So, in this example, it's ORDER BY list_price DESC. When I use this DESC keyword, I'll speak it as descending. That's how most people say it. So, for this example, I would say order by list price descending, but the actual keyword is DESC. You can also use the keyword ASC for ascending. But since ascending is the default sort order, this keyword has no effect. So ORDER BY list_price ASC does the same thing as just ORDER BY list_price. But sometimes it's helpful to include this ASC keyword just to make the sort order abundantly clear. When you use multiple column references in an ORDER BY clause, you specify ascending or descending order separately for each one. Each keyword ASC or DESC only applies to the one column reference it's used with.

The query in this example arranges the results in descending order by max_players, then in ascending order by list_price. So the rows with the largest values of max_players are at the top and then within the sets of rows in which max_players are tied, the smallest values of list_price come first. And of course, remember that you could leave off the ASC keyword here, and you would get the same result because the default sort order for each column is ascending. But in this kind of case, where you're sorting on multiple columns, some ascending, some descending, it's helpful to include the ASC keyword for clarity. There is an important point I want to clarify about sort order when you're using HUE. In the HUE query editors, if you run a query that returns a result with more than 100 rows, only 100 rows are initially returned and displayed. The demonstrate this, I'll run the query SELECT * FROM flights. The flights table has tens of millions of rows, but when I run this query, HUE only displays 100 rows. If I scroll down to the bottom of the page, HUE will load and display 100 more rows. I could scroll down again to load more rows. When I scroll back up to the top, and move the cursor over one of the column headers, you can see that there are grayed out up and down arrow icons visible there.

If I click the header once, HUE sorts the displayed results by the values in that column in ascending order. If I click a second time, it sorts in descending order, and if I click a third time, it returns the results to the original order, with no sorting applied. What's happening here is that HUE is sorting only the portion of the results that are displayed. HUE is not sorting the full result set. If you mistakenly think that HUE is sorting the full result set, you could easily misinterpret the data. For example, when you sort in ascending order, you might mistakenly think that the value shown in the top row is the minimum value in this column for the whole data set. And when you sort in descending order, you might mistakenly think that the value shown in the top row is the maximum value. But you would be wrong because these are only the minimum and maximum values within this arbitrary subset of the rows that's displayed in HUE.

So I recommend not using this sort feature in HUE unless you are sure that the results that are displayed are the entire result set, not just a subset. This issue is not unique to HUE. There some other SQL clients and BI applications that have table viewer interfaces like this with similar controls that might mislead you in a similar way. So whenever you're using software that displays a subset of your query results, keep in mind that the controls or functions for sorting the rows might only be sorting that subset, not the full result set.

Ordering Expressions

Recall that the list of column references that comes after the keywords ORDER BY is called the ORDER BY list. This list can have just one column reference, or it can have two or more column references, separated by commas. The ORDER BY list can also have expressions. Here's an example with an ORDER BY list that consists of one expression. This expression is a little bit complicated but what it does is it computes the saturation of the colors in the crayons table. Saturation is a decimal number between 0 and 1, representing how intensely colorful a color is. White and black have a saturation of 0, the flashiest colors have a saturation close to 1.

So this expression computes the saturation of each color in the crayons table. And since the expression is used in the ORDER BY clause, the SQL engine arranges by rows of the result set according to saturation. It arranges then in descending order with the most saturated colors at the top. However, the result set from this query does not include a column showing the saturation values. That's because the expression that computes saturation is used only in the ORDER BY clause, not in the SELECT list. Fortunately, most SQL engines allow you to use an expression in the SELECT list, give it a column alias, and then use that column alias in the ORDER BY clause. So here I modified the query.

I took the expression for computing saturation, I moved it into the SELECT list after all the other columns and I gave it the alias saturation. And then I specified ORDER BY saturation descending. The results of this query is arranged in descending order by saturation. And it includes the saturation values in the rightmost column. You can see the crayon color with the highest saturation value is Electric Lime. These examples used just one expression, or column alias, in the ORDER BY list. But you can use two or more, separated by commas. You can use any mix of column references, expressions, and column aliases. And you can specify ascending or descending order, separately, for each one.

Ordering by String Columns

You can control the sort order of SQL query results using the ORDER BY clause. When sorting on a numeric column, the resulting order typically makes intuitive sense, but when sorting on a string column, you might be surprised by the resulting order. This is especially true when the strings include numbers, or a mix of numbers and letters or other characters within a value.

Unfortunately, there isn't a simple explanation to tell you how SQL will sort your results, because it depends on what ***collation*** you are using.

A DBMS uses a ***collating sequence,*** or ***collation,*** to determine the order in which characters are sorted. The collation defines the order of precedence for every character in your character set. Your character set depends on the language that you're using—European languages (a Latin character set), Hebrew (the Hebrew alphabet), or Chinese (ideographs), for example. The collation also determines case sensitivity (is 'A' < 'a'?), accent sensitivity (is 'A' < 'À' ?), width sensitivity (for multibyte or Unicode characters), and other factors such as linguistic practices. The SQL standard doesn't define particular collations and character sets, so each DBMS uses its own sorting strategy and default collation… Search your DBMS documentation for ***collation*** or ***sort order***. (1)

Collations have different options associated with them, and many can be customized depending on the system you are using. For English, ***case sensitivity*** is a major one to consider—should "A" and "a" be considered the same character for the purposes of ordering? Others include ***accent sensitivity*** (for example, should "a" and "á" be considered the same), ***Kana sensitivity*** (which distinguishes between the two types of Japanese characters), and ***script order*** (for example, which should be ordered first: Hebrew, Greek, or Cyrillic). See "Customization" (2) and "Collation" (3) for more examples of these and other options.

When using Unicode—an industry standard that assigns a number to each character or symbol— SQL will most likely follow the Unicode ordering to distinguish the order of two characters, while taking customizations into account. Non-Unicode data may have a different order:

When you use a SQL collation you might see different results for comparisons of the same characters, depending on the underlying data type. For example, if you are using the SQL collation "SQL_Latin1_General_CP1_CI_AS", the non-Unicode string 'a-c' is less than the string 'ab' because the hyphen ("-") is sorted as a separate character that comes before "b". However, if you convert these strings to Unicode and you perform the same comparison, the Unicode string N'a-c' is considered to be greater than N'ab' because the Unicode sorting rules use a "word sort" that ignores the hyphen. (4)

When it comes to numbers represented within strings, you must remember than string sorting is done on a character-by-character basis. For example:

'42' < '71'	This compares only the first characters: '4'<'7'. The order is now established and any other remaining characters can be ignored.
'42' < '45'	The first characters are the same, '4' = '4', so the sort then compares the next characters, '2'<'5'. So '42' < '45'.
'42' < '7'	Although numerically 42 > 7, the sort compares the first characters, '4' and '7'. Since '4' < '7', the order is established and any other remaining characters are ignored. For this string sort, '42' < '7'.

You can sometimes find ways to customize the sort, when necessary. For example, "Use SQL Server to Sort Alphanumeric Values" (5) provides a method, usable with Microsoft SQL Server, to sort values with a mixture of letters and numerals that would consider '7' < '42'.

Spaces, especially leading spaces, often cause confusion as well. The space character is typically considered to come before any number or letter, and some punctuation as well. Again, sort order is done character by character. For example:

'no one' < 'nobody'	The first characters are equivalent, 'n' = 'n', so the sort would move to the second characters. These are also equivalent, 'o' = 'o', so the sort moves to the third characters. These are ' ' and 'b', and ' ' < 'b', so 'no one' < 'nobody'.

' start' < 'begin'	Notice that the first character in the string on the left is a space. While 'begin' < 'start' because 'b' < 's', these string sort as ' start' < 'begin' because ' ' < 'b'.

For more detail on these points, see the referenced articles.
(1) Fehily, Chris. ***SQL VIsual QuickStart Guide, 3rd Edition.*** Retrieved from http://www.peachpit.com/articles/article.aspx?p=1276352&seqNum=4 on May 25, 2018.
(2) Unicode® Technical Standard #10: Unicode Collation Algorithm. Retrieved from http://unicode.org/reports/tr10/#Customization on May 25, 2018.
(3) Collation and Unicode Support. Retrieved from https://docs.microsoft.com/en-us/sql/relational-databases/collations/collation-and-unicode-support?view=sql-server-2017#Collation_Defn on May 25, 2018.
(4) Comparing SQL collations to Windows collations. Retrieved from https://support.microsoft.com/en-us/help/322112/comparing-sql-collations-to-windows-collations on May 25, 2018.
(5) Use SQL Server to Sort Alphanumeric Values. Retrieved from https://www.essentialsql.com/use-sql-server-to-sort-alphanumeric-values/ on May 25, 2018.

Missing Values in Ordered Results

Recall that when you use an order by clause to sort your results in a descending order by sum column, then the rows with the lowest values in that column will be at the top of the results. If you sort in descending order, then the rows with the highest values will be at the top. As a data analyst, you'll often use the order by clause for exactly this purpose, to get the lowest or highest values at the very top of your result. This is what you do to identify the worst, or the best, or the smallest, or the biggest of whatever your data represents. But there is something they can prevent this from working as expected, and that is when the column that you're ordering by contains null values. Different SQL engines handle null values in ordering columns in different ways. In Impala and PostgreSQL, nulls sort as if they are higher than any non null value.

In other words, these SQL engines put the nulls at the bottom of the results when you sort in ascending order, and at the top when you sort in descending order. So for example, the query shown here sorts the results from the inventory table by the price column in ascending order, that's the default sort order. When you run this query in Impala or PostgreSQL, the row with the null price is at the bottom of the results. But if you add the DESC keyword to sort in descending order by price, then the row with the null price is at the top of the results. That's how Impala and PostgreSQL work, but Hive and MySQL do exactly the opposite.

In Hive and MySQL, nulls sort as if they are lower than any non null value. In other words, these SQL engines put the nulls at the top when you sort in ascending order, and at the bottom when you sort in descending order. So if you sort by price in ascending order with Hive or MySQL, the row with the null price is at the top of the results, and if you sort in descending order, then the null prices at the bottom. So if there are null values in the column or columns you're ordering by, you should always remember to consider where the SQL engine you're using will put them in the results. With some SQL engines, there is a way to explicitly control how the order by clause handles null values.

You can do this by using the keywords nulls first or nulls last in the order by clause. I'll use an example to demonstrate how this works. Recall that Impala and PostgreSQL put the nulls at the bottom of the results when you sort in ascending order. So if you sort the inventory table by price in ascending order, then the row with the null price is at the bottom. To make that row with the null price appear at the top of your results set, you can add the keywords NULLS FIRST after the column reference in the order by clause. So it's order by price, NULLS FIRST.

When you use multiple column references in an order by clause, you specify the keywords nulls first or nulls last separately for each column. These keywords come after the ASC or DESC keywords if you use them. So for example, this query uses the clause ORDER BY aisle descending NULLS LAST, price ascending NULLS FIRST. With this example Impala and PostgreSQL would sort the results differently if you had left off the nulls last or the nulls first. Note that regardless of whether you sort in ascending or descending order, the keywords nulls first and nulls last will do exactly what they sound like they do. They'll put the nulls first or last in the ordering. Using the nulls first and nulls last keywords won't necessarily change the order of your results. For example, with Impala or PostgreSQL using nulls first after a descending will have no effect because the SQL engines already put the nulls first by default when you sort in descending order.

Not all SQL engines support these keywords nulls first and nulls last. Impala and PostgreDSQL do, but MySQL does not and only newer versions of Hive do. You can try it out with the SQL engine that you're using to see if it works. In the SQL engines like MySQL and older versions of Hive that do not support this syntax, you can use a trick to achieve the same result. The trick is to order by the Boolean expression, column name is null, then order by the column itself. For example, Hive and MySQL both put the row with the null price first in the top row when you sort by price in ascending order.

But you can make them put it last in the bottom row by ordering first by price is null ascending, then ordering by price. Then the result has the null price in the bottom row. This works because price is null is a column of Boolean values, true and false values. When you sort Boolean values in ascending order, false comes before true. So the rows in which prices null is true and up at the bottom. Another option when you're working with data that has null values in the ordering columns is simply to remove the rows that contain those null values using the where clause or the having clause. In many cases, that's the easiest thing to do, but you should always be careful about removing rows with null values, because you could end up censoring meaningful parts of the data in ways that could dramatically changed the outcome of your results or your analysis. So anytime you're tempted to just remove the nulls from a dataset, always stop and think first about whether this could make your results incomplete or misleading.

Using ORDER BY with Hive and Impala

In this video, I'll describe a limitation that you need to know about when using the ORDER BY clause with Hive, a shortcut for specifying the ORDER BY clause with Impala, and some general advice for using ORDER BY with engines like Hive and

Impala. First the Hive limitation. With Hive when you want to order by a specific column, that column must be included in the result set. So for example, the query shown here gives an error when you try to run it with Hive, because it's ordering by list price, but list price is not one of the columns that's specified in the select list. To resolve this error, you need to either add the list price column to the select list or use select star. So all the columns including list price are returned. Both of these queries run without error in Hive.

You'll also encounter this limitation if you try to use an expression in the ORDER BY clause that has one or more column references in it. Like in this example with order by quantity times price. This gives the same type of error if you try to run it in Hive. Once again, to resolve the error, you'll need to include those columns quantity and price in the select list or you select star. As long as you do this, the select list doesn't need to include the expression itself. Or with expressions, another option is to use the expression in the select list give it a column alias and then use that column alias in the ORDER BY clause. Because recall that most SQL engines allow you to use a column alias in the ORDER BY clause. With Hive that method works and it gets around this limitation. This limitation only affects Hive not Impala and not most other SQL engines.

Now I'll describe a shortcut for specifying the ORDER BY clause with Impala. With Impala, you can specify which columns to order by using position numbers. These are integers starting from one that refer to columns in the results set by their position. Here's an example to demonstrate this; in this query, the price column is the third column in the select list, shop is one, game is two, and price is three. So two order by the price column, you can specify order by three. This is a convenient shortcut that can save you from typing the full column name in the ORDER BY clause.

As I said this works in Impala and it works in some other SQL engines too, it might work with Hive if the instance of Hive you're using is configured a certain way or if you're using a newer version of Hive, but you should not count on it working in Hive. You can try this out to see if it works with the SQL engine you're using, but keep in mind that if it doesn't work you won't necessarily see an error. Instead the SQL engine will probably act like you're asking it to sort the results by a scalar number repeated in every row. So it will just return the results in arbitrary order. In general, I don't recommend using the shortcut method because it makes it harder to see what your query does and it could cause trouble if you changed your select list but then forgot to change your ORDER BY clause. Finally, some advice for using ORDER BY with engines like Hive and Impala. When you're working with very large datasets, sorting them it uses a lot of computing power.

Sorting is a notoriously difficult operation to optimize especially for distributed systems. So you should only sort your results when you need to. If you're accustomed to using relational database systems with small to medium sized data, you might be in the habit of using ORDER BY when it's not strictly necessary. Because in that situation it doesn't introduce as much of a performance penalty. But you should break that habit when you're using distributed SQL engines. Sorting happens after all the other operations that you learned about. After filtering, after grouping, and aggregation. So one good strategy is to use those prior operations to reduce the number of rows in the data as much as possible before using ORDER BY. That will help your ORDER BY queries to use fewer compute resources and finish faster.

THE LIMIT CLAUSE

The final clause that's covered in this course and the clause that comes last in the select statement is the optional limit clause. The purpose of the limit clause is to set a maximum number of rows for the query to return. Here's an example. The flights table in the fly database contains tens of millions of rows, but say you want to only return five of those rows. To do this, you would run the query, select * FROM flights LIMIT 5, then the result will have five rows and no more. Now you're probably wondering which five rows do you get. What you get is just five arbitrary rows. If you ran this same query again on the same data, you might get five different rows, or it might be the same five. There's no guarantee.

The exception to this is when you also use an orderBy clause, you'll learn about that in an upcoming video. Here's another example. This one is a more complicated query. It finds the carriers, the airlines that have at least 5,000 flights with an air time of seven hours or longer and the limit clause limits the results to 10 rows. But notice that the results set has only four rows. That's because in this case, the result of all the other clauses returns fewer than the specified limit of 10. When this happens the limit clause has no effect.

So you should never assume that you'll get exactly as many rows as your limit clause specifies, it can be fewer but it will never be more. The limit clause must come after all the other clauses, and it's applied after all the other clauses. So in this example it does not affect which of the individual rows of the flight table are included in the aggregations, it's applied after the filtering, after the grouping, after the aggregation, after everything else at the very end and it only affects how many of the result rows are returned. The number that comes after the limit keyword generally needs to be a non negative literal integers.

So any whole number, zero or higher. Some SQL engines will let you use an expression after the limit keyword, but only a constant numeric expression, not an expression with column references or column aliases. Some SQL interfaces like Hue automatically limit the number of rows that a query can return. Recall that the Query Editors in Hue display up to 100 rows initially, and they let you load more rows by scrolling down. That makes it unnecessary to use the limit clause unless you want to use a limit smaller than 100.

In Hue, if you do use the limit clause and you specify a number higher than 100, then Hue will still display only 100 rows at first, and you'll need to scroll down to display anything beyond the 100th row. However, if you continue scrolling down, it will not load any more than the number of rows you specified in the limit clause. But you can't always rely on your SQL client to automatically limit how many rows are displayed. With some software if you run a query that returns millions or billions of rows, the software will attempt to download them all, which could put a big load on your SQL engine, saturate your network, use up lots of memory, and take a long time to finish. By using the limit clause, you can prevent this from happening.

WHEN TO USE THE LIMIT CLAUSE

In this video, I'll describe when it's a good idea to use the LIMIT clause, and when you could get yourself in trouble by using it for the wrong purpose. One very common use of the LIMIT clause is when you want to return just a few rows from a table to get a sense of what the data in the table looks like and what values are in the different columns. The LIMIT clause is great for this. You use it to return a few arbitrary rows and then inspect them. Often, that will give you ideas for how you might analyze the data or what queries to run next. For example, one of the first things you might do when working with the flights table is to return 5 or maybe 10 or 20 rows, and take a look at them. By doing this, you can quickly ascertain some things about the data.

For example, you can see how the year, month, and day are represented. Four-digit years, and integer month and day values. You can see that the time columns appear to be formatted as the hour in 24-hour time, followed by the minutes. From there, you might run some follow-up queries to find out things like, what's the earliest and the latest year represented in the table, and are there any missing values in the time columns? Another common use of the LIMIT clause is when you've written a select statement but you don't know how many rows it will return, and you want to avoid returning an enormous number of rows. This is a very common situation.

Working as a data analyst, it's not unusual to realize you have no idea whether a query that you just wrote will return 100 rows or 100,000 rows. So if you find yourself in this situation, often the safest and easiest thing to do is to put a LIMIT clause at the end of the query to limit the result set to a number of rows that's small enough that it won't cause any problems. Then after you run the query, you can check whether the number of rows it returned reached that limit that you set. If it did, then you could modify the query to reduce the number of rows, for instance, by using more restrictive filtering conditions or by using different grouping columns. Of course, another option is to find out exactly how many rows will be returned by writing and running a separate count query, but that requires writing and running a separate query.

So often, it's just quicker and easier to add a LIMIT clause to the query you already have. Here's an example of this. Say, you're using a BI or Data Visualization Tool, and you want to use it to draw a map showing all the routes represented in the flights table, all the origin and destination pairs with lines connecting them, and you want the lines to vary in thickness based on how many flights flew on that route. Many BI and Data Visualization Tools can draw maps just like that. To get the data you would need to draw a map like that, you would run a query like this. It groups the flights table by origin and destination, and in each row of the result set, it returns an origin, a destination, and the number of flights that had that origin and destination.

The trouble is, we don't know how many rows this query will return. If it were to return more than about 1,000 rows, the map might take a really long time to render, and it would be so crowded with lines that you couldn't interpret it. A quick and easy way to avoid this problem is to add LIMIT 1000 to the query, then you know your map will not take too long to render and it will not be crowded with many thousands of lines. After running this query with the LIMIT 1000, you could see in the BI or Data Visualization Tool that it did in fact return 1000 rows. It hit the limit. So then you could experiment with ways to filter the data to avoid hitting this limit of 1000 rows.

One way to achieve this would be to return only the routes that had many flights. In other words, omit the less frequently flown routes from the results. With a little bit of experimentation, you could find out that filtering out the routes that had fewer than 20,000 flights gives a result with fewer than 1,000 rows. A result with fewer than 1,000 rows means this query is no longer hitting the limit, so then you could remove the LIMIT clause. In addition to limiting the number of rows that are returned, the LIMIT clause can also reduce the amount of compute resources that the SQL engine needs to process your query.

So in addition to speeding up your queries, it could also reduce the load you're putting on the SQL engine making other people's queries run faster too. Recall that when the LIMIT clause limits the number of rows that are returned, it picks the rows arbitrarily. There's no guarantee about which rows you'll get. You might think this means that you can use the limit clause to get a random sample of rows, but you should not use the limit clause if what you want is a random sample. With a distributed query engine like Hive or Impala, there are a variety of unpredictable factors that affect which rows are returned by the LIMIT clause, factors like the processor, memory, and network usage of the different computers on the cluster, and how the data is distributed across the cluster. So which rows are returned is not predictable, but it's also not random.

This is an important distinction. Unpredictable does not mean random. Here's an analogy to help you understand this. Imagine you have a brand new deck of cards and you just took it out of the box, so the cards are in predetermined order, arranged by suit, then by rank. Then imagine you shuffle this deck just one time. One shuffle is not enough to make the cards be in really random order. After just one shuffle, they're not in a totally predictable order but they're definitely not in totally random order either. This is a lot like what happens with rows in a distributed SQL engine. They get shuffled a little bit so they're not in a predictable order but they're not thoroughly shuffled, so you should never consider them to be in random order. So the subset of rows that's returned when you use a LIMIT clause is not a random sample. It's taken from the top of a pile that's not in random order. It might be called a sample of the data, but it's definitely not a random sample. Because it's not a random sample, you should never interpret the results from a select statement with a LIMIT clause as being representative of the full results like the way that a real random sample of the rows would be.

USING LIMIT WITH ORDER BY

The LIMIT clause is especially useful when it's used together with the ORDER BY clause. Recall that the ORDER BY clause arranges the rows of the result set in order by some column or columns. This means that the rows at the top of the result set represent the greatest or the least, or the best or the worst of all the records, as measured by the values in that column or columns. In data analysis, it's common to want to return just a few of these most extreme cases to identify them and take some action. For example, who are the one hundred highest spending customers? So we can send them a loyalty reward?

Or who are the ten lowest performing sales people? So we can put them on an improvement plan? Using ORDER BY together with LIMIT, lets you return a specified number of the most extreme records. Here's an example of this. Here's a query that returns the ten routes in the flights table that have the longest average air time. A route is a combination of origin and destination. So this query groups the flights table by origin and destination. And returns the avg_air_time for each origin destination pair. The order by clause arranges the results in descending order by avg_air_time with the nulls last at the bottom. The nulls last is important here if you're using Impala.

Without it, you would just see a bunch of nulls at the top of the results set. The nulls last puts the nose at the bottom where they're out of the way. Finally, this query has limit 10, to return the top 10 origin destination pairs. This kind of query that uses a LIMIT clause, together with an ORDER BY clause, is called a top-n query. Because it returns the top-n results where n is some number, like ten in this example. Or it might be called a bottom-N query depending on what row order is used. Looking at the results from running this query, you can see that all of these ten longest flights are between H and L, which is Honolulu, Hawaii and cities in the Eastern United States.

It looks like the ones going to Honolulu from the east to the west, take longer on average than the ones going the other way. This is because the jet stream winds in the upper atmosphere go from west to east. And it looks like the two longest routes at the top of the results set are the routes from the New York City airports, JFK and EWR to Honolulu. These are the only ones that have average flight times exceeding 600 minutes or 10 hours. In the results set, noticed that I also included a column named count_air_time.

This is computed using the expression count of airtime, so it shows how many flights on each route have a non missing value in the airtime column. It's a good idea to include a count column like this, when ever you're computing averages or other aggregates. It helps you to understand how confident you should be in those aggregates. For example, looking at the first two rows of this result set, you can see that the average in the first row was calculated using almost 2,000 values of air time. And the average in the second row was calculated using more than 3,500 values of air time.

These large count values give me confidence in these averages. But if there were a row here where average_air_time was very high, but count_air_time was very low, then I would want to check whether that was caused by erroneous values in the air_time column. When you use top n queries you need to watch out for ties in the column or columns that you're sorting by. Here's an example to explain why this is important. Imagine you're the manager of a company's sales department and you want to reward your top three highest performing sales people with a trip to a luxury resort. The performance metric you're using is last year's total sales in dollars.

So to find your top three performers, you would run a top three query like this. For each salesperson the query computes their name and their total sales. And the result set is returned in descending order by total sales, and it's limited to three rows. And here's the result. It shows that the three top performers are Ambrosio, Lujza and Sabahattin in that order. So they are the three sales people who get to go on the trip. But here's the problem, if you change limit three to limit five and run the query again, you can see that there's another sales person, Val, who has exactly the same total sales as Sabahattin.

They both have the same value, $320,000. The query with the limit three arbitrarily returned Sabahattin not Val, because when there's a tie in the ordering column, the order of the tied rows is arbitrary. So it was purely by chance that it returned Sabahattin and not Val. Imagine you ran the query with the limit three, and based on the result of that query, you sent Sabahattin on the luxury trip but not Val. I think Val would be pretty angry. Believe it or not people actually make this type of error. Even if the values were an exact tie, say Val's total sales were a few hundred dollars less then Sobahattin's.

It would still seem unfair to deny Val a reward based on that. Important distinctions should not be made on the basis of insignificant differences. So when you're running a top n query, if there's going to be some action taken based on whether a row is in the top n or not, then it's best to return some extra rows, and check for ties or near ties, in the values that are around that nth row. If there are ties or near ties there, then look at some of the rows above or below and try to find a threshold, a cutoff point that seems fair. In this example, by returning five rows instead of three, you're able to notice the tie between Sabahattin and Val.

And you can also see that the next best performer, Virginia, has a significantly lower total sales number. Based on that, you might decide to send both Sabahattin and Val on the trip. Or to send neither of them. Or you might find some other appropriate metric to break the tie between them.

Using LIMIT for Pagination

Recall that the LIMIT clause is used for returning up to a certain number of result rows, like up to 100 rows. But what if you want to do something like return 100 rows, then return the next 100 rows, then the next 100 rows, and so on? This is called pagination or paging because you can think of it like returning one page of results at a time. Many SQL engines allow you to paginate query results by specifying a row limit and a row offset. So for example, to paginate your results into pages of 100 rows each, you would first specify a limit of 100 and an offset of zero. That's the first page.

Then you would specify a limit of 100 and an offset of 100 to get the second page. Then a limit of 100 and an offset of 200 to get the third page and so on. So by running a sequence of multiple queries with different offsets, you can return the rows of the result set split up into multiple pages. So you already know how to specify the limit by using the limit clause. But how do you specify the offset? Well, different SQL engines support different ways of specifying it. With Impala and PostgresQL, it's specified using the offset keyword which comes after the limit. But Hive uses a different syntax. With Hive, there's no offset keyword. Instead you specify two numbers after the limit keyword, separated by a comma. The offset comes first, then a comma, then the limit. This only works with newer versions of Hive. MySQL supports both of these syntaxes. Some other SQL engines use other syntaxes.

For example, some use the keyword skip instead of offset. Check the documentation for the SQL engine you're using to see which syntax it supports. In some of the SQL engines that use the offset keyword like Impala, you can actually use it without the limit keyword. So you can specify only an offset and no limit. So you could think of offset as a separate clause, but it's pretty unusual to use offset without limit. So usually I just think of offset as an optional part of the limit clause. There's something important you need to remember when you use limit and offset for pagination. First, recall that when you run a select statement using a distributed SQL engine, if the statement does not have an ORDER BY clause, then the order of the rows in the results set is arbitrary and unpredictable. You could run the exact same query twice and get the rows in a different order each time.

What this means for pagination is that if you were to run a sequence of multiple queries with different offsets to paginate your results, each of those queries could shuffle all the rows differently before applying the limit and offset. So the query to get the second page of results might return some of the same rows that were returned by the query that you ran to get the first page and some of the rows might not be on any of the pages. So across the pages of the results, there could be duplicate rows and missing rows.

That's no good. Because of this, it's essential that you use an ORDER BY clause to arrange the rows in deterministic order whenever you use limit and offset for pagination. When I say deterministic order, I mean that the order of the rows is not at all or arbitrary or unpredictable. Every row is arranged in predictable order according to the ORDER BY clause. With the rows of the full results at in deterministic order, then the limit and offset clause can split it up into nice clean pages of results, where every row appears on exactly one page. Some SQL engines like Impala try to enforce this by throwing an error if you try to use limit and offset without ORDER BY. But some other SQL engines will allow you to leave off the ORDER BY clause.

So it's up to you to remember that you need to use it. To ensure that you don't have missing or repeated rows in paginated results, you need to ORDER BY a column that has a unique value in every row, or by multiple columns that have a unique combination of values in every row. That way, there are no ties in the values you're ordering by because recall that if there are ties, then the order of the tied rows is arbitrary not deterministic, and we need to make the row order deterministic. Often, there is a single column like a unique ID or unique timestamp column that has a unique value in every row. If so, you can ORDER BY that column.

But sometimes, there's not a single column like this. For example, the flights table on the VM does not have any one column with a unique value in every row. But if you ORDER BY year, month, day, origin, destination, carrier, and flight, then the rows will be in deterministic order and you can use limit and offset to get cleanly paginated results. I think there are still a few rows there that are in indeterminate order due to missing or erroneous values. So you might also want to filter those out. Also, keep in mind when you are using the ORDER BY clause in a query that paginate the results, that the purpose of the ORDER BY clause in this case is not necessarily to return the rows in a meaningful order, like it is in a top end query.

Often, the point is just to get the rows in some deterministic order, so the pagination works correctly. Any order might be fine, just so long as it's deterministic. This method of paginating results by running a sequence of separate queries with different offsets, it works, but it's inefficient. It requires running multiple queries which can put a lot of stress on the SQL engine. It's especially inefficient if you run many queries that each return a small number of rows. For performance, it's better to run fewer queries that each return a larger number of rows. There are more efficient ways to implement pagination. But to use them, you would need to use a programming language to write an application that communicates with the SQL engine through one of the interface standards like ODBC or JDBC or Thrift.

The details of that are beyond the scope of this course. Review of the SELECT Statement In this course, you've learned about all seven of the clauses that you can use in a select statement. The select clause, which specifies what columns should be returned in your query result. The from clause, which specifies where the data you're querying should come from. The where clause, which filters the individual rows of data based on some conditions. The group by clause, which splits data into groups, and the related topic of aggregation which reduces each group down to a single row. The having clause, which filters the data based on aggregates.

The order by clause which sorts or arranges the results of a query, and finally the limit clause which controls how many rows a query can return and can also be used to paginate query results if you specify an offset. The order in which I have taught these seven clauses in this course matches their correct order within a select statement. Select, from, where, group by, having, order by and limit in that order. There are some minor exceptions. But in general, you must use the clauses in that order or you'll get an error from the SQL engine. But as you know, a query does not need to include all of these clauses. The only clause that's strictly required by all SQL engines is the select clause.

Some engines also require the from clause, but all the others are optional. So you can pick and choose which clauses to use depending on the task at hand. This makes select statements very flexible, and you've seen in this course how you can use different combinations of these clauses to answer very different kinds of questions. So that's the order in which you must use the clauses in a select statement. But the order in which the SQL engine executes these clauses when it runs a select statement is slightly different. Specifically, SQL engines execute the select clause not at the beginning, but later after the from, where, group by and having clauses. So what a SQL engine does when it runs a select statement is, first, it executes the from clause, which tells it which table the data should come from.

It reads the data from that table, and if there's a where clause it uses the conditions specified there to filter the individual rows of data as it reads them in. Then once it's done reading the data in, if there's a group by clause, the SQL engine uses the grouping columns specified there to split the data into groups. Next, if there's a having clause, it computes the aggregate expressions there and uses those to filter the groups. Only then does the select list get executed to create the columns that will be returned in the results set. Next, if there's an order by clause, the SQL engine uses the columns specified there to arrange the rows of the result set. Finally, if there's a limit clause, the SQL engine uses that to specify the maximum number of rows that can be returned.

So that's the order in which SQL engines execute the clauses of a select statement in general. However, different SQL engines have their own minor variations on this. For example, a common variation that many SQL engines use is that they partially process the select list earlier to identify the column aliases that are defined there. That way those aliases can be used in the group by and having clauses. But regardless of minor exceptions like that, remembering this execution sequence will help you to understand how a SQL engine takes a select statement and turns it into a result set. Review In this course, you've learned about all seven of the clauses that you can use in a select statement. The select clause, which specifies what columns should be returned in your query result. The from clause, which specifies where the data you're querying should come from. The where clause, which filters the individual rows of data based on some conditions.

The group by clause, which splits data into groups, and the related topic of aggregation which reduces each group down to a single row. The having clause, which filters the data based on aggregates. The order by clause which sorts or arranges the results of a query, and finally the limit clause which controls how many rows a query can return and can also be used to paginate query results if you specify an offset. The order in which I have taught these seven clauses in this course matches their correct order within a select statement. Select, from, where, group by, having, order by and limit in that order. There are some minor exceptions. But in general, you must use the clauses in that order or you'll get an error from the SQL engine. But as you know, a query does not need to include all of these clauses.

The only clause that's strictly required by all SQL engines is the select clause. Some engines also require the from clause, but all the others are optional. So you can pick and choose which clauses to use depending on the task at hand. This makes select statements very flexible, and you've seen in this course how you can use different combinations of these clauses to answer very different kinds of questions. So that's the order in which you must use the clauses in a select statement. But the order in which the SQL engine executes these clauses when it runs a select statement is slightly different.

Specifically, SQL engines execute the select clause not at the beginning, but later after the from, where, group by and having clauses. So what a SQL engine does when it runs a select statement is, first, it executes the from clause, which tells it which table the data should come from. It reads the data from that table, and if there's a where clause it uses the conditions specified there to filter the individual rows of data as it reads them in. Then once it's done reading the data in, if there's a group by clause, the SQL engine uses the grouping columns specified there to split the data into groups. Next, if there's a having clause, it computes the aggregate expressions there and uses those to filter the groups.

Only then does the select list get executed to create the columns that will be returned in the results set. Next, if there's an order by clause, the SQL engine uses the columns specified there to arrange the rows of the result set. Finally, if there's a limit clause, the SQL engine uses that to specify the maximum number of rows that can be returned. So that's the order in which SQL engines execute the clauses of a select statement in general. However, different SQL engines have their own minor variations on this. For example, a common variation that many SQL engines use is that they partially process the select list earlier to identify the column aliases that are defined there. That way those aliases can be used in the group by and having clauses. But regardless of minor exceptions like that, remembering this execution sequence will help you to understand how a SQL engine takes a select statement and turns it into a result set.

WEEK 6

Learning Objectives

- Differentiate between using UNION and using JOIN
- Write and run SELECT statements using UNION and UNION ALL
- Write SELECT statements using JOIN, handling non-matching records appropriately for the task
- (Honors) Write and run SELECT statements using advanced JOIN clauses, including cross joins, non-equijoins, left semi-joins, and null-safe joins

Combining Query Results with the UNION Operator

The union operator in SQL combines two results sets into one. It takes the rows returned by one select statement and the rows returned by another select statement, and it stacks them together. It combines them vertically. There are two variations of the union operator: union all and union distinct. I'll first to demonstrate union all. Here is the simple select statement that returns the id and name columns from the games table in the fun database. You can see it returns five rows. Here's a simple select statement that returns the id and name columns from the toys table in the toy database.

You can see that this one returns three rows. To return one result set containing the five rows from the games table and the three rows from the toys table for a total of eight rows, you can combine these two queries into one using union all. The results set shown here has the games on the top, and the toys on the bottom, but in general the order of the rows in a result set of a union is arbitrary. Just like with any unordered result set. So you will not necessarily see the rows from the first query on the top and the second query on the bottom, they could be the other way or they could be shuffled together.

Here's another example of union all. Say you wanted to return the country codes for the countries in the customers table, and for the countries in the offices table. Both of these tables are in the default database. To do that, you could run the query SELECT COUNTRY FROM customers UNION ALL SELECT country FROM offices. This is a query that you might run in the real world to see all the countries where you're doing business.

In the results that you can see this returns eight rows. But notice that one country code appears twice in the results set US. Because there's a customer in the United States and an office in the United States US shows up twice. To eliminate this, duplicate result row. You could use Union Distinct, instead of Union All. When you use Union Distinct, then after the SQL engine combines the results of the two queries, it also omits any duplicate rows from the result set. So every row in the results set will be unique. Most SQL engines support both Union All and Union Distinct. This includes Impala, MySQL, Postgres QL, and newer versions of HIV. However, older versions of HIV do not support Union Distinct. They only support Union All. Check the documentation or run a simple test to see whether the SQL engine you're using supports Union Distinct. So these keywords all and distinct are used to modify the behavior of the union operator. You might be curious what happens if you use the union operator alone without one of these keywords. Oddly Union by itself is the same as Union Distinct.

This might make sense if you understand set theory in mathematics, but otherwise it probably seems backwards. I recommend including the distinct keyword whenever you want the duplicate rows removed from the results set. Including the distinct keyword makes the purpose of the query more explicit. Unfortunately, there are some SQL engines that will perform a Union Distinct. If you use the union operator alone, but they will not allow you to explicitly use the distinct keyword. So again, if you're using some other SQL engine, check the documentation or run a test to see whether the union operator alone with no modifiers returns the distinct rows of the combined result set.

The select statements on both sides of the union operator should have the same schema. In other words they should have the same number of columns and the pairs of corresponding columns should have the same names, and the same datatypes or at least the same high level categories of data types like both numeric or both character string. When you write a query that includes a union, you should use explicit type conversion, and column aliases to make this happen. Here are a couple of examples to demonstrate this. Say you wanted to union together the name and list price columns from the games table in the fun database with the name and price columns from the toys table in the toy database. Both of these pairs of columns, name, and name, list price, and price have the same datatypes. But list price and price are different names.

So to avoid any problems, you should use a column alias in one or both of the select statements to give these two columns the same name. In this example, I added as price to the first select statement so they're both named price. Say you wanted to return the distinct values of year that occur in either the flights table in the fly database or in the games table in the fund database. Both of these tables have columns named Year. But in the flights table, year is an integer column, whereas in the games table it's a string column. So in this example, I explicitly casted that string column to an integer column, and I also added as year to keep it named Year.

Some SQL engines will tolerate the names or datatypes being different, but to make your queries safe and portable, I recommend making the names and data types the same using explicit casting and the as keyword like in these examples. Don't rely on the SQL engine to do it for you. You might notice if you're browsing the documentation or searching for help that Union is sometimes referred to as a Clause. I think it's better to think of it as an operator whose operands on the two sides are select statements. Also, the examples I showed in this video were all very simple. The select statements on the two sides of the union had only select and from clauses, but you can use more complex select statements with union. They can include some of the other clauses.

MISSING OR TRUNCATED VALUES FROM TYPE CONVERSION

In queries that use UNION (and in other types of queries), you will sometimes need to use ***explicit type conversion*** (also called ***explicit casting***) to convert a column (or a scalar value) from one data type to another. In most SQL dialects, this is done using the cast function. However, you need to be careful about a couple of things when using cast.

Review: Explicit Type Conversion
As you might recall, you can cast any numeric column to a character string column, like this:

SELECT cast(list_price AS STRING) FROM games;

If you have a character string column whose values represent numbers, then you can cast it to a numeric column, like this:
SELECT cast(year AS INT) FROM games;

Refer back to the video “Data Type Conversion” in Week 2 of this course if you need more of a refresher on the basics of data type conversion.

Type Conversion Can Return Missing Values

Under some circumstances, the **cast** function will return missing (**NULL**) values. A common situation in which this happens is when you have a character string column whose values do ***not*** represent numbers, and you try to convert it to a numeric column.

For example, this query attempts to convert the character string values in the **name** column (values like **Monopoly** and **Scrabble**) to integer values:
SELECT cast(name AS INT) FROM games;

When you run this query with Hive or Impala, it returns a column of **NULL** values, because there is no way to cast these character string values to known integer values.

However, some other SQL engines have different ways of handling situations like this. If you use MySQL to cast a character string column as a numeric column, it returns ***zeros*** for the values that do not represent numbers (not **NULL**s like Hive and Impala). And PostgreSQL throws an error if you attempt to cast a character string that does not represent a number as a number. Also note that different SQL engines have different data types, so the data type name you use after the **AS** keyword in the **cast** function varies depending on the engine. For example, in MySQL you should use **SIGNED INT** instead of **INT**, and in MySQL and PostgreSQL you should use **CHAR** instead of **STRING**.
Type Conversion Can Return Truncated Values

Under some circumstances, the cast function will return truncated (cut off) values. A common situation in which this happens is when you convert decimal number values to integer values.
For example, this query converts the decimal numbers in the list_price column (which have two digits after the decimal) to integer values:

```
SELECT cast(list_price AS INT) FROM games;
```

When you run this query with Impala or Hive, it truncates (cuts off) the decimal point and the two digits after it. For example: 19.99 becomes 19.
However, in this situation, some other SQL engines ***round*** instead of truncating. When you run a query like this with MySQL or PostgreSQL, it rounds each decimal number value to the nearest integer value. For example: 19.99 becomes 20.

USING ORDER BY AND LIMIT WITH UNION

The SELECT statements on both sides of a UNION operator, can use any of the clauses that you've learned about in this course, with two exceptions; the ORDER BY, and LIMIT clauses. I'll discuss the ORDER BY clause first. The way that the ORDER BY clause works when you use the UNION operator, differs depending on what SQL engine you're using. Here are some examples to demonstrate this. Say you wanted to return the names and prices of all the games, from the games table, and all the toys from the toys table. To do that, you would run a query like this; SELECT name, list_price AS price FROM games UNION ALL SELECT name, price FROM toys.

On the VM, you can run this query in MySQL or PostgreSQL as it is. For Hive or Impala, you would need to add the database names before the table names; fun.games and toy.toys, and in the results set, you can see the names and prices of all five games and three toys. Recall that the row order is arbitrary, the rows might be in a different order for you. Now, what if you wanted to return these rows in a specific order? Say in ascending order by price; the lowest price at the top, the highest price at the bottom? Well, with some SQL engines, including MySQL and PostgreSQL, you can do that by adding an ORDER BY clause at the end of the whole query.

Notice that all the games and toys are now arranged in ascending order by price. This ORDER BY clause does not get only applied to the result of the second statement, it gets applied to the full results of the UNION. So it arranges all the toys and the games in order by price. You can see Candy Land is the least expensive, and Etch A Sketch and Risk are tied for the most expensive, so the order of these last two rows is arbitrary. Again, this method works with some SQL engines but not with others.

This does not work with Impala, and it does not work with Hive, or at least not until the very recent version of Hive. If you try this with Hive or Impala, you will not get an error, in Impala depending on what client you're using to run the query, you might see a warning, but it will still return a results set. However, the rows of the result set will not be arranged in order by the price column. Also, you might be tempted to try something like this. Adding an ORDER BY clause, to both of the two SELECT statements on both sides of the UNION operator. But this will also not return a result set that's arranged in order by the price column.

Depending on what SQL engine you're using, it might throw an error, or it might just ignore the two ORDER BY clauses and return the results in arbitrary order, or it might actually arrange each of the two intermediate results sets in the desired order, but then when the UNION operator shuffles the two intermediate results sets together, that row order might be lost. So using an ORDER BY clause in both of these SELECT statements in a UNION, is generally not a technique that you should use with any SQL engine. With some SQL engines, including Impala and some versions of Hive, the only way to return the full results of a UNION arranged in order, is to use something called a subquery.

Subqueries are a more advanced topic that's beyond the scope of this course, they're covered in a later course, in this specialization. The way that the LIMIT clause works when you use the UNION operator, is similar with some SQL engines including MySQL and PostgreSQL. You can put one LIMIT clause, at the end of the whole query, and that will limit the number of rows returned in the full results set. But with others SQL engines including Impala and some versions of Hive, this does not work. You can use two separate LIMIT clauses in the statements on either side of the UNION, to limit how many rows each of these can return, but there is no way to limit the total number of rows in the full results set, except by using subqueries, which you'll learn about in a later course in this specialization.

So when you're using the UNION operator, remember that it's safe to use most of the clauses you've learned about, in the two SELECT statements on either side of the Union. You can use the SELECT, FROM, WHERE, GROUP BY, and HAVING clauses, but be careful about using the ORDER BY and LIMIT clauses. Check the documentation for this specific SQL engineer you're using, and run some simple tests to make sure you understand how it will interpret the ORDER BY and LIMIT clauses in UNION queries.

USING UNION TO COMBINE THREE OR MORE RESULTS

You can use **UNION ALL** or **UNION DISTINCT** to combine three or more query results into a single result set. To do this, simply add another **UNION** operator after the final **SELECT** statement and add another **SELECT** statement after it. For example, the following query uses three **SELECT** statements, combined with two **UNION ALL** operators:

```
 SELECT color, 'red' AS component, red AS value
    FROM crayons
    WHERE color = 'Mauvelous'
UNION ALL
SELECT color, 'green' AS component, green AS value
    FROM crayons
    WHERE color = 'Mauvelous'
UNION ALL
SELECT color, 'blue' AS component, blue AS value
    FROM crayons
    WHERE color = 'Mauvelous';
```

This query returns the three component values (**red**, **green**, **blue**) of the color named **Mauvelous**, in three separate rows.
Be sure to use a semicolon only at the very end.

When using three or more **UNION** operators in one query, it’s a good idea to make them all **UNION ALL** or all **UNION DISTINCT**. Mixing the two different types of **UNION** operators in a single query is likely to cause confusion.

The rules that apply when using a **UNION** to combine two results also apply in the case of three or more results:

The **SELECT** statements should have the same number of columns and the sets of corresponding columns should have the same names and the same high-level categories of data types. Use explicit casting and column aliases to ensure this. You can use the **SELECT**, **FROM**, **WHERE**, **GROUP BY**, and **HAVING** clauses in each **SELECT** statement, but be careful about using the **ORDER BY** and **LIMIT** clauses. Check the documentation for the specific SQL engine you're using, and run some simple tests to make sure you understand how it will interpret the **ORDER BY** and **LIMIT** clauses in **UNION** queries.

INTRODUCTION TO JOINS

A join in SQL combines data from two related tables into one result set. In the simplest sense, a join takes columns from one table and columns from another table, and it merges them together, it combines them horizontally. But a join does not to just throw together the columns from the two tables, it also matches the rows from the two tables. When you write a query in SQL that joins two tables, you specify what the relationship between these two tables is, and the SQL engine uses that relationship to match the rows.

For example, the toys table and the makers table both in the toy database are related by the fact that each toy is made by a specific maker. The column named maker ID in the toys table, refers to the column named ID in the makers table. So for example, in the toys table, you can see that light bright has maker ID 105. In the makers table, that id 105 represents the company Hasbro. So when you write a query that joins these two tables, you specify that the maker ID column in the toys table corresponds to the ID column in the makers table, and the SQL engine uses the matching values in these corresponding columns to match the rows when it combines the two tables together.

The reason that joins are so important and so widely used in SQL is that related data are often stored in separate tables. Here for example, the name and city of the maker of each toy is not included in the toys table. Instead, that information is stored in the separate makers table. This principle of storing related data in separate tables is an important element of what's called normalized design. If you completed the first course in this specialization, you might recall some of the advantages of storing data this way. Although it's advantageous to store data in separate tables, it's often necessary to join the data from those separate tables together in order to analyze it and answer questions about it.

For example, say you needed to answer the question, which toys are made by Ohio Art Company? It's impossible to answer this by using only one of these two tables, you need both tables. The makers table tells you that Ohio Art Company has ID 106, and then the toys table tells you that the only toy with that maker ID one 106 is Etch-a-Sketch. You could answer a question like this by running a sequence of two queries, but by using a join, you can answer it with just one query. In the next video, I'll introduce the syntax of join queries. Join Syntax Earlier in this course, you learned that the FROM clause is the clause you used to specify where the data should come from.

That was true when you were querying data from single tables, and it's also true when you use a JOIN to combine data from two tables. In the case of a single table, you use a single table reference after the FROM keyword. But to join data from two tables, you use two table references separated by the keyword JOIN. The table references can be simple table names like in this example, toys and makers, or they can be in the form, database name.table name. The two tables you're joining can be in the same database, or in two different databases, or schemas. If they are in different databases, then you'll need to use database name.table name.

So that's how you specify which two tables to join, but you also need to specify what the relationship between the two tables is so that the SQL engine can match the rows. To do this, you use the ON keyword followed by an expression that specifies the relationship between the tables. This comes right after the table names, it's part of the FROM clause. In this example, it's on toys.maker_id equals makers.id. Looking at the toys and makers tables, recall that the maker id column in the toys table corresponds to the id column in the makers table. So this expression after the ON keyword, tells the SQL engine to use the matching values in these corresponding columns to match the rows when it joins the two tables together. This expression after the ON keyword is known as a join condition, and it's typically in this form, a reference to a column in one table, the equals sign, and a reference to the corresponding column in the other table.

The column names are prefaced by the names of the tables they come from, toys.maker_id and makers.id. This way the SQL engine knows which table each of the corresponding columns comes from. The maker_id column comes from the toys table, and the id column comes from the makers table. These corresponding columns in a join are sometimes called join columns or join key columns. If you run this query, you get this result. The information about each toy is returned along with information about the company that makes it. Because the select list is a select star, the query returns all the columns from both tables. But looking at the results set, you can see there are a few problems. There are two columns named name, one is from the toys table and one is from the makers table, and also there are two columns named id.

Also, the column named maker_id and the second column name id contain the same information. So you do not need both of these columns. To solve these problems, you need to replace the star after the select keyword with an explicit list of the columns to return. But because you're querying data from two tables, you need to write this list differently than you would if you are querying from just one table. The columns id and a name each need to be prefaced with a table name so the SQL engine knows whether you want the one from the toys table, or the one from the makers table. So in the list, you can see toys.id, toys.name, and makers.name.

For each of these, I also included a column alias with the as keyword to control the names of the resulting columns, toys.id as id, toys.name as toy, and makers.name as maker. These column aliases make the results easier to understand. For the remaining columns in this select list, price, maker_id, and city, there is no ambiguity about which table there should come from, there's only one price column, it's from the toy table. There's only one maker id column, it's from the makers table, and there's only one city column and it's from the makers table. So for those three columns, you do not need to qualify each column name with a table name, just the bare column name is sufficient. However, some people prefer to qualify all the column names with table names just to avoid any possibility of ambiguity. This is a good practice with join queries, especially if you're writing queries that we run again in the future or built into an application. So that's the syntax of a basic join query in SQL.

But there's one more thing, qualifying column names with table names like this can get awfully repetitive especially if the table names are long. Fortunately, there is a shortcut that you can use to avoid typing the table names over and over again. You can give an alias to each of the table names, and use these table aliases instead of the full table names. Table aliases are usually chosen to be very short often just a single character. A common choice is the first letter of the table name if they're different letters. In this example, it's toys as t and makers as m. The AS keyword is optional and it's common to omit it.

Once you have given these aliases to the tables, you can use them in the select list and in the join condition after the ON keyword. So in both of these places, you can replace toys with t and makers with m. Notice how that makes the query much more concise. Using table aliases is optional, but in the real world, you'll find that they are used in almost every join query. The example join query I showed in this video used only the SELECT clause and the FROM clause, but you can use all of the other clauses with a join query, there are no exceptions to this. The aliases that you give to the tables in the FROM clause, you can use those in all the other clauses to resolve any ambiguity about which columns are from which tables.

INNER JOINS

So you've seen what a basic SQL join query looks like. In the from clause there are two table references separated by the keyword join. After that there's the on keyword followed by the join condition and table aliases are used to resolve ambiguity about which table each column comes from while keeping the query concise. Here I've omitted the optional as keyword before each of the table aliases. So it's just toys t and maker is m. The result of a join query like this can contain columns from both of the tables and the rows are matched according to the join condition. In this example, the first column comes from the toys table and the second column comes from the makers table and each toy is in the same row as its maker. However, as you might have noticed the maker Mattel is in the makers table but does not appear in the result of this join.

The reason for this as you might have inferred is that none of the toys in the toys table are made by Mattel. Mattel is represented by maker id 107 and none of the toys have maker id 107. This is how joins work by default in SQL. For a row to be returned in the join result, it must match a row in the other table, rows without a match are not returned. This default type of join has a name, it's called an inner join. To understand why this is called an inner join, it helps to visualize the values in the columns we're joining on. In this Venn diagram the circle on the left side represents the unique values in the maker id column in the toys table, and the circle on the right side represents the unique values of the id column in the makers table, these are the columns we're joining on.

Two of the values 105 and 106 are in both circles those values exist in both tables. However one of these values 107 is only in the right circle, that value exists only in the makers table. In an inner join, the result set only includes the rows that have values in the inner region of this diagram, the region where the two circles overlap, rows that have values in either of the outer regions are excluded from the results of an inner join. So, when you use the keyword join between the two table names like in this example, what the SQL engine performs is an inner join. But you can also explicitly specify that you want an inner join by using the keyword inner before join.

This has exactly the same result. It does not matter if you include the inner keyword or leave it off. I usually prefer to include it. I think it's better to be explicit. With an inner join the order of the tables in the from clause does not matter. You could have toys on the left and makers on the right or you could have makers on the left and toys on the right, the result will be the same. The only case in which the order of the tables in the from clause does matter is if you use select star then the order of the columns in the results set will depend on which table comes first in the from clause. In this example where we joined the toys and makers tables an inner join is appropriate.

The purpose of this join query is to show more information about the maker of each toy. For each toy in the toys table there is a corresponding maker in the makers table and since there are no toys made by Mattel it seems appropriate that Mattel is excluded from the results. In this and many other situations an inner join gives you exactly the result you're looking for. But in some other situations, the way that an inner join excludes the non-matching rows can be problematic. Here's an example. This query is slightly more complex. It joins the same two tables makers and toys, but this time it groups by maker and it uses the count function to return the number of toys made by each maker.

The trouble is the result set totally excludes Mattel. What I would like in this case is for the result set to include Mattel with a count of zero, but since this is an inner join and there's no row in the toys table with maker id 107, Mattel is simply excluded. Depending on how this result set was used the absence of Mattel could be misleading, it could cause an oversight or a misinterpretation. This is just one example of a case where inner joins do not return the result you're looking for. In the next video, you'll learn how to solve this problem by using outer joins.

OUTER JOINS

In this video, I'll describe a different class of joins called outer joins. To demonstrate outer joins, I'll use the employee table and the offices table both in the default database. These tables represent five different employees and four different offices. The relationship between these two tables represents which employees work at which offices. The columns with the corresponding values that can be matched to join the tables together are named office_id in both of the tables. So in this example, I'll refer to office_id as the join column. There are two important things to notice in these tables. First, there is one employee, Val, who has an office id that does not exist in the offices table. Val has office_id e, but there is no office with office_id e.

Second, there is one office, the Singapore office that has no employees. The Singapore office has office_id d, but none of the employees have office_id e. Aside from these two rows, all the other rows in both of these tables have office ID values that do exist in the other table. It helps to use a Venn diagram to visualize this. You can see that the office_ids a, b, and c are in the inner region. They exist in both tables. But office_id e, that's Val's office_id is found only in the employees table, the one on the left. Office_id d for the Singapore office, that's found only in the offices table, the one on the right.

Before I talk about outer joins, first recall what an inner join does. It returns only the rows with join column values in the inner region. In this example office_ids a, b, and c are in this inner region, those values exist in both tables. So when the SQL engine combines these two tables, it identifies ther corresponding rows by matching their office id values, and it merges the rows according to these matches. If it's an inner join, it returns only the rows that have matches. The rows that don't have matches, that's Val with office_id e and the Singapore office with office_id d, those are not returned by an inner join.

Outer joins handle non-matching rows differently. There are three types of outer joins, and each one handles non-matching rows in a particular way. In a left outer join, if there are rows in the left table with a join column value that does not exist in the right table, it returns them anyway. So in this example, a left outer join will include the employee Val in the result. Even though, Val's office_id e does not match any of the office_ids in the offices table. In a right outer join, if there are rows in the right table with a join column value that does not exist in the left table, it returns them anyway.

So a right outer join will include the Singapore office in the result, even though it's office_id d does not match any of the office_ids in the employees table. Finally, in a full outer join, if there are rows in either of the tables with joint column values that don't exist in the other table, it returns them anyway. Now let's look at the SQL syntax for these three types of outer joins and see what exactly they return. The syntax is the same as for an inner join, except that you replace inner with a left outer, right outer, or full outer. Here's a left outer join query, combining these two tables, employees and offices. Notice in the form clause that employees is the table on the left side of the keywords left outer join, and offices is the table on the right. Notice that the results that includes Val, even though there's no match between Val and any of the offices. In a left outer join, non-matching rows from the left table are returned.

In the result row representing Val, notice how the city value is null. The city column comes from the offices table but since there's no office for Val, all the values that would have come from the offices table are null in this row. The office id for Val also shows up as null in the result. That's because I used o.office_id in the select list. If I change this to e.office_id, then that value will come from the employees table so it's not null. In outer joins, you should always consider which table the join key columns in the results set are coming from. Forgetting about this can cause confusion. Next is the right outer join. The syntax is the same, except it's right instead of left. Notice this time that the Singapore office is included in the result set even though none of the employees work there.

In a right outer join, non-matching rows from the right table are returned. In the result row representing Singapore, the columns that come from the employees table like employee id and first name are null since there's no employee that matches this office. Office id is also null because I used e.office_id in the select list. If I change this to o.office_id, then it's not null. It shows the value from the offices table. In left and right outer joins, the order of the table references in the from clause does matter. Because the one and only difference between these two types of outer joins is, which table has its non-matching rows included in the result? Is that the one on the left side or the one on the right side?

That's the only difference. In fact, right outer joins are very rarely used. Most people prefer to always use a left outer joins and just list the table with the non-matching rows that you want to return on the left side in the from clause. The third and final type of outer join is the full outer join. Again, the syntax is the same except you use the keyword full. Notice how the rows representing the employee Val and the office in Singapore are included in the results set. In a full outer join, non-matching rows from both tables are returned. By far, the most common type of outer join you'll see in the real-world is the left outer join. When you're joining two tables together, typically one of them is the main table.

The one that represents the items or the units that you're analyzing. Often, you'll want all the rows from that main table to be included in the result irrespective of whether they have matches in the other table. The most common thing to do in that case is specify the main table on the left side of the join and use a left outer join. For example, here's a join query that answers the question how many employees work in each city? The main table in this join is the offices table because the unit of analysis in this question is city, and the city values are found in the offices table. So the offices table is on the left side of the join.

If you used an inner join here, the row representing Singapore would be excluded from the result. Using a left outer join includes it, and the aggregate expression account e.employee_id returns zero in the row for Singapore. Depending on how our results set like this was used, it could be really important to include the non-matching row. Even though it's counted zero. Excluding it could cause an oversight or a misinterpretation. Another common use of outer joins is to identify and return only the non-matching or unmatched rows.

You can do this by adding a where clause to filter the joined results at to include only the rows that have a null value indicating that there was no match. Depending on how you write the Boolean expression in the where clause, you can return the unmatched rows from the left table or from the right table or from both tables. This is useful for identifying inconsistencies or anomalies in related tables. Like in the examples I showed here, maybe Val actually works in the Singapore office. So Val's office_id is really supposed to be d not e. Identifying that inconsistency could enable you to get it fixed. Some SQL engines do not support all three types of outer joins. Mysql supports left and right but not full. Some others only support left. Hive, Impala, and Postgres QL do support all three. Also, many SQL engines allow you to leave off the outer keywords. So you can just write left join, or right join, or full join. I prefer to include the outer keyword just to be fully explicit about what kind of join it is.

ALTERNATIVE JOIN SYNTAX

This reading describes alternative ways of expressing joins in SQL. We do not recommend using the techniques described in this reading, but you should familiarize yourself with them so you can read and understand SQL queries that use them.
SQL-92-Style Joins and SQL-89-Style Joins
In the video lectures describing joins in SQL, the following join syntax is used:

```
SELECT ...
  FROM toys JOIN makers
    ON toys.maker_id = makers.id;
```

Notice the JOIN keyword between the table names, and the ON keyword followed by the join condition. This is called a ***SQL-92-style join***, or ***explicit join syntax***, and it is usually considered to be the best syntax to use for joins in SQL. However, many SQL engines also support another join syntax, called the ***SQL-89-style join***, or ***implicit join syntax***. In this syntax, you use a comma-separated list of table names in the FROM clause, and you specify the join condition in the WHERE clause:

```
SELECT ...
  FROM toys, makers
    WHERE toys.maker_id = makers.id;
```

With most SQL engines, this join query returns exactly the same result as the previous one.
With both join styles, you can use table aliases (t and m in this example):

```
SELECT ...
  FROM toys AS t JOIN makers AS m
    ON t.maker_id = m.id;
SELECT ...
  FROM toys AS t, makers AS m
  WHERE t.maker_id = m.id;
```

With both styles, the AS keyword before each table alias is optional.
When you use a SQL-89-style join, the SQL engine always performs an ***inner*** join. With this syntax, there is no way to specify any other type of join. If you want to use one of the other types of joins (left outer, right outer, full outer), then you must use a SQL-92-style join. Because of this limitation, and because the SQL-89-style join syntax makes it harder to understand the intent of the query, we recommend using SQL-92-style joins.

Unqualified Column References in Join Condition

In the join condition that comes after the ON keyword in a join query, the references to the corresponding columns are typically ***qualified*** with table names or table aliases. For example, when joining the toys table (alias t) and makers table (alias m), the join condition is specified as:

```
ON t.maker_id = m.id
```

However, in the case where a bare column name unambiguously identifies a column, most SQL engines allow you to use a bare column name. For example, since there is no column named maker_id in the makers table, the table alias t is not required in this join condition. So you could specify the join condition as:

```
ON maker_id = m.id
```

But because there are columns named id in both tables, the table alias m is required in this join condition. If you omit the table alias m, then the SQL engine will throw an error indicating that the column reference id is ambiguous.
In join conditions, we recommend ***always*** qualifying column names with table names or table aliases, whether or not they are strictly required. Doing this makes your queries safer and clearer.

In some join queries, the names of the two corresponding columns in the join condition are identical. For example, in this query, the corresponding columns in the employees and offices table are both named office_id:

```
SELECT …
  FROM employees e JOIN offices o
    ON e.office_id = o.office_id;
```

When the corresponding columns in the join condition have identical names, some SQL engines allow you to use a shorthand notation to specify the join condition. Instead of using the ON keyword and specifying the condition as an equality expression, you use the USING keyword and specify the common join key column name in parentheses after USING:

```
SELECT …
  FROM employees e JOIN offices o
    USING (office_id);
```

Natural Joins

When the corresponding columns in the join condition have identical names, some SQL engines will allow you to omit the join condition, and will automatically join the tables on all the pairs of columns that have identical names in the left and right tables. To make a SQL engine do this, you need to specify the keyword NATURAL before the other join keywords. For example:

```
SELECT …
   FROM employees e NATURAL JOIN offices o;
```

MySQL and PostgreSQL support natural joins, but Hive and Impala do not. In the SQL engines that support it, you can use the keyword NATURAL with any type of join; for example: NATURAL LEFT OUTER JOIN or NATURAL INNER JOIN.

Omitting Join Conditions

What happens if you attempt to perform a join without specifying the join condition, and you do ***not*** specify NATURAL before the join keywords?
For example, you might run a query like this:

```
SELECT *
   FROM toys JOIN makers;
```

Notice that no join condition is specified. With some SQL engines (including PostgreSQL), this throws an error. But with other SQL engines (including Impala, Hive, and MySQL) this performs what's called a ***cross join***. In a cross join, the SQL engine iterates through each row in the table on the left side and combines it with every row in the table on the right side. So the result set includes every possible combination of the rows in the left table and the rows in the right table. The number of rows in the result set is the product (multiplication) of the number of rows in the left table and the number of rows in the right table (in this example, 3 x 3 = 9):

```
id  name            price  maker_id id   name               city
21 Lite-Brite       14.47 105       105 Hasbro              Pawtucket, RI
21 Lite-Brite       14.47 105       106 Ohio Art Company Bryan, OH
21 Lite-Brite       14.47 105       107 Mattel              Segundo, CA
22 Mr. Potato Head 11.50 105        105 Hasbro              Pawtucket, RI
22 Mr. Potato Head 11.50 105        106 Ohio Art Company Bryan, OH
22 Mr. Potato Head 11.50 105        107 Mattel              Segundo, CA
23 Etch A Sketch    29.99 106       105 Hasbro              Pawtucket, RI
23 Etch A Sketch    29.99 106       106 Ohio Art Company Bryan, OH
23 Etch A Sketch    29.99 106       107 Mattel              Segundo, CA
```

id	name	price	maker_id	id	name	city

In most cases, the result of a cross join is meaningless. The rows of the result contain values with no correspondence. If you don't realize that you have performed a cross join, you might be misled by the results. In addition, when performed on large tables, a cross join can return a dangerously large number of rows.

There are some specific cases when cross joins are useful, and in most SQL dialects, you can explicitly specify **CROSS JOIN** in your SQL statement to make it clear that you are performing a cross join. This is discussed in a video in the upcoming honors lesson.

So unless you intend to perform a cross join, and you understand the risks of this and how to interpret the output, we recommend specifying the join condition in every join query.

JOINING THREE OR MORE TABLES

When you're working as a data analyst, you will often need to use join queries to combine *three or more* tables. To do this, you use the same syntax as with two tables, but with more **JOIN**s added at the end of the **FROM** clause. Each **JOIN** should have its own **ON** keyword and join condition.
For example, to join the customers, orders, and employees tables, you could use this query:

```
SELECT c.name AS customer_name,
     o.total AS order_total,
     e.first_name AS employee_name
  FROM customers c
    JOIN orders o ON c.cust_id = o.cust_id
    JOIN employees e ON o.empl_id = e.empl_id;
```

Notice how the final two lines of this query have the same structure: the **JOIN** keyword, a table reference, a table alias, the **ON** keyword, and a join condition. You can add arbitrarily many lines like this to the **FROM** clause, to join together arbitrarily many tables.

The result the above query is:

customer_name	**order_total**	**employee_name**
Arfa	28.54	Sabahattin
Brendon	48.69	Virginia
Brendon	-16.39	Virginia
Chiyo	24.78	Ambrosio

The arrangement of the rows in the result is arbitrary. Each result row represents an order, and gives the name of the customer who placed the order, the total amount of the order, and the employee who recorded the order. Because this is an inner join (the default type), all non-matching rows are excluded from the result. You can specify other types of joins, in any combination. For example, to include the order placed by the customer who is not in the customers table, change the first **JOIN** to **RIGHT OUTER JOIN**.

The sequence of the tables in a multi-table join does not matter, except with left and right outer joins, where it affects which table's non-matching rows are included. Each join condition can refer to join key columns in any of the tables mentioned earlier in the **FROM** clause.

Join queries are computationally expensive and can be slow, especially when you're joining three or more tables, or if you're working with very large data. One strategy that can be used to remedy this problem is to join the data in advance and store the pre-joined result in a separate table, which you can then query. (If you completed the first course in this specialization, you might recall this is one way of *denormalizing* a normalized database, to make it easier to run analytic queries.) This approach of pre-joining tables has some costs, but it can make analytic queries easier to write and faster to run. A later course in this specialization will go into more details about how you can do that.

HANDLING NULL VALUES IN JOIN KEY COLUMNS

When you join together two tables, if there are no values in the join key columns in both tables, the SQL engine will not match these nulls. It will not combine rows based on a null value in one table matching a null value in the other table. Here's an example to demonstrate this. These two tables are the same as the customers and orders tables on the VM, except here each one has an additional row added containing a null value in the cust_id column. So there's one customer record with a cost_id value of null, and there's one order record with a cost_id value of null. When you run a query to join these tables, the records with these null values in the join key column are not merged together in the result set. To understand why this is, look at the join condition, c.cust_id = o.cust_id.

That's the equality comparison that the SQL engine uses to identify matches when it performs the join. Recall that whenever one or both sides of an equality comparison are null, the comparison yields null. So when the SQL engine compares the null cost_id value in the one table to the null cost_id value in the other table, these two nulls do not yield a match. Null equals null is not true, it's null. So the SQL engine does not merge together these two rows with the null join key values. And since this is an inner join, neither of these unmatched rows is included in the result set. Typically this is exactly the behavior you would want and expect in a join. In an example like this, nulls in the joint key columns would probably mean unknown. And you would not want to merge together two records on the basis of them both having an unknown value in the join key column.

However, in some cases, you might want to match null values when doing a join. For example, suppose that the company that this data comes from allows customers to place an order anonymously. For example, to complete a transaction as a guest instead of creating an account or signing in. And suppose that the row in the customers table that has a null in the cust_id column represents all anonymous customers. And that orders by anonymous customers are stored in the orders table with a null in the cust_id column. Using null that way is a questionable choice, but suppose that someone else designed the database that way, and you're just trying to analyze the data. So in this case, perhaps you would want the join query to merge together these records with null in the join key column. You would want it to treat null as equal to null. To do this, you can change the join condition to use this special operator instead of the equality operator. You might recall this operator from earlier in the course. It's written as less than sign, equals sign, greater than sign, and it's shorthand for is not distinct from.

This operator is sometimes called the null safe equality operator. And this type of join is called a null safe join. When this operator compares two null values, it returns true instead of null. So with this join condition, the SQL engine merges together the two records that have null values in the join key column, and in includes the merged row in the join result. This type of join works with many SQL engines, including Hive, Impala, and MySQL. With PostgreSQL, you need to use the long form of this operator, is not distinct from, instead of the shorthand form.

NON-EQUIJOINS

In a typical join, the join conditions are expressed as one or more equality conditions. This is called an Equijoin. A Non-Equijoin is a join that uses some other comparison instead of equality. In a Non-Equijoin, you can use comparisons such as not equal, less than, or greater than in the join conditions. Here's an example: The employees table lists the salary for each employee, and the salary grades table lists the salary grades or levels and their associated minimum and maximum salaries. To find the salary grade for each employee, you could use a Non-Equijoin with inequality comparisons for the join conditions. For instance, in the results you can see Ambrosia Rojas has salary grade two, because his salary which is 25,784 is greater than or equal to the minimum salary for grade two which is 20,000 and less than or equal to the maximum salary for grade two which is 29,999. Impala, MySQL, PostgreSQL, and some other SQL engines support Non-Equijoin queries like this one. But with HIVE, only equality conditions are allowed in joins.

CROSS JOINS

Unlike other types of joins, a Cross Join does not try to match records based on the values and corresponding columns. Instead what a Cross Join does, is it returns every possible combination of the records from the tables. In other words a cross join generates the cross-product, also known as the Cartesian product of the datasets. For this reason, it's also known as a Cartesian Join. Be careful when doing Cross Joins, because they can generate huge amounts of data.

The number of rows a cross join generates, is equal to the product, the multiplication of the number of rows in each table. In many cases the result of a cross join is meaningless. The rows of the result set contain values that have no correspondence. However, in some cases generating every possible combination of values can be useful, as in this example. There's a table named card_rank with 13 rows, one row for each rank of a playing card in a standard deck of card, two through 10, and Jack, Queen, King and Ace. There's a table named card_suit, with four rows, four Clubs, Diamonds, Hearts and Spades. If you do a cross join on these two tables, you get a result set with 13 times four rows, that's 52 rows, and each row represents one card in a standard 52 card deck.

The best way to write a SQL statement that performs a cross join, is to explicitly specify cross join in your statement, like I did in this example. But many SQL engines will also perform a cross join if you simply omit the join condition. In other words, if you use only the join keyword, and no other keywords between the two table names, and you leave off the join condition, so there's no on keyword, then most SQL engines will perform a cross join. This feature is dangerous because it means that by simply forgetting to include the join condition in an inner join query, you can inadvertently caused the SQL engine to return an enormous number of rows. So remember to specify the join condition whenever you do not want to do a cross join. All of this also applies when you use the SQL 89 style Inner Join syntax, where you use a comma instead of the join keyword, and you put the join condition in the WHERE clause.

If you use that syntax, but you forget to include the join condition in the WHERE clause, then the SQL engine will perform a cross join. It can be helpful to understand an Inner Join as a cross join, followed by a filter. I'll use an example to show what I mean by this. I'll use the SQL 89 style Inner Join syntax in this example, because it demonstrates the point more clearly. This statement performs a cross join. It's written like an Inner Join, but with no join condition. So it returns every possible combination of toys and makers. There are three toys and three makers, so it returns three times three rows, nine rows. Most of these rows contain values that do not correspond to one another. In most of the rows, the maker id value from the toys table is not equal to the id value from the makers table. But if you add a WHERE clause to filter this Cross Join result, so it only contains the rows in which the join key column values are equal, then what you have is an Inner Join.

All the join key values in the results set match, and all the values in each row correspond. This is exactly the same result you get if you use the SQL 92 style Inner Join syntax. So an Inner Join is effectively the same as a Cross Join, followed by a filter. This equivalence is especially evident when you use the SQL 89 style syntax, where an Inner Join is literally written as a Cross Join followed by a filter. What a SQL engine does internally when it performs an Inner Join, is much more efficient than what would happen if it actually generated all possible combinations, and then filtered them. But the end result is equivalent.

LEFT SEMI-JOINS

Some SQL engines support a less common type of join called the Left Semi-Join. A Left Semi-Join is a special type of Inner Join that's used for its efficiency. It behaves more like a filter than a join because only the data from one of the tables is included in the result set. A Left Semi-Join returns only records from the table on the left for which there is a match in the table on the right. Here's an example. Say you want to use the data in the fly database on the VM to find out which aircraft models are used for very long flights.

For instance, flights with a distance of more than 4,000 nautical miles. There's a table named Planes that has information about the different aircraft, including the manufacturer and the model. But there's no information about the range, the maximum flying distance of the aircraft in that table. However, there is a distance column in the flights table that gives the distance of every flight in miles. These two tables, aircraft and flights can be joined using tail number as the join key column. So it's possible to answer this question by joining these two tables. But a regular Inner Join does more than you need in this case. You do not need to return any values from the flights table. You just need to use the values from the flights table to filter the matching rows in the planes table.

This is the kind of situation where it's possible and it's more efficient to use a Left Semi-Join. Here's the syntax of a Left Semi-Join. You preface the join keyword with left semi and then after the ON keyword, you specify both the join conditions and you can specify other criteria for filtering the table on the right side of the join. This is different from what's included after the ON keyword in other joins. In the other kinds of joins, only the join criteria are included there.

But in a Left Semi-Join, you can include filtering criteria there as well. The expression after the on keyword is the only place where you're allowed to refer to the columns from the table on the right. So in this example, if you attempted to use this filter condition in the where clause, the SQL engine would throw an error. Or if you attempted to include references to any of the columns from the flights table in the select list or in other clauses, the SQL engine would throw an error. The only place they're allowed is in an expression after the ON keyword. In cases where this limitation is acceptable, a Left Semi-Join can give you much better performance than an Inner Join.

The result of this query shows that the aircraft used for these long flights were all large Boeing and Airbus jets as you might expect, but there is one strange row in the results set listing, a Bombardier Jet which I don't think has a range this long. Maybe that plane made a refueling stop on those flights, I don't know. If you want it to return more details about those flights like the carrier, origin, and destination, then you would need to use an Inner Join instead. You cannot return those columns in a Left Semi-Join because they're from the table on the right side. Hive and Impala allow left semi joins, but many other SQL engines do not. Some SQL engines also allow Right Semi-Joins, which allows you to reverse the order of the table, so they return records from the table on the right that have matches in the table on the left. Some SQL engines are smart enough to process regular Inner Joins as efficient right or left semi joins when it's possible to do so.

SPECIFYING TWO OR MORE JOIN CONDITIONS

In all the examples of joins presented in this course, it was possible to join two tables together by specifying a *single* join condition after the **ON** keyword. For example, to join the toys and makers tables together, the join condition was:

toys.maker_id = makers.id

However, in the real world, to join some pairs of tables together, you will need to specify *two or more* join conditions after the **ON** keyword. For example, imagine you have a table containing historical daily weather conditions data for all United States airports, and this table includes columns named **year**, **month**, **day**, and **airport**. To join this weather table (alias **w**) with the flights table (alias **f**), you would need to specify an expression with four join conditions, like this:

```
ON f.year = w.year
    AND f.month = w.month
   AND f.day = w.day
   AND f.origin = w.airport
```

In this example, the four conditions are combined into a single expression using the **AND** operator, so the SQL engine checks for all four criteria to be true when it matches the rows from the flights table and the weather table.
Join conditions like this are called *multiple join conditions* or*compound join conditions*. It is common for joins to require conditions like this.

When the pairs of corresponding columns have identical names in the two tables, some SQL engines allow you to use the **USING** keyword to specify multiple join conditions. In the parentheses after the **USING** keyword, separate the column names with commas. For example, if you were joining together two tables using columns named **city** and **state** as the join key columns, you could use the following join syntax:

```
SELECT …
   FROM table1 JOIN table2
      USING (city, state);
```