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 twosingle 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.