Lesson 3-Accessing Data Part 2

Lesson 3 Concepts

1. We can eliminate duplicate rows in our result set.
2. We can have multiple conditional clauses to refine our results.
3. We can use imprecise values to find all data that matches a pattern.
4. We can provide a set of values or range of values to search for.
5. We can put a hard limit the number of rows in our result set.
6. We can define the sort order of our result set.

We Can Eliminate Duplicate Rows in Our Result Set

When querying to find categories, it is unhelpful to have a result set that repeats the categories over and over. You may have only six categories, but a data set with 1000 rows. To eliminate these unnecessary duplicates, we can use the **DISTINCT** keyword;

```
SELECT DISTINCT category FROM stock_items;
```

Now, the above query will return only one of each of the requested data, in this case, each unique value in the category column. This can be useful when creating a menu or dropdown list of categories. Another example would be if you had a blog with multiple authors, you would only want to list each author once on your website, not for every article that they had posted.

We Can Have Multiple Conditional Clauses to Refine Our Results

SQL has logical operators (like we saw in JavaScript) that allow us to add logical AND/OR/NOT provisions to our **WHERE** clauses, and therefore, multiple conditions. Consider:

```
SELECT * FROM stock_items WHERE category = "Canine";

SELECT * FROM stock_items WHERE category = "Canine" AND
price >= 15;
```

In the second query above, we have added an additional criteria to our search with the **AND** keyword. With **AND**, both of the conditions need to be true. So, "Get all the rows from the stock_items table with a category of "Canine" that *also* have a price value that is less than or equal to 15." The first query returns seven rows; the second, more refined query, returns five rows.

With the logical **OR**, only one of the conditions needs to be true:

```
SELECT * FROM employees WHERE role = "Sales" OR role =
"Stock";
```

In the statement above, we are increasing our result set with the **OR** keyword to include multiple values.

The logical **NOT** acts to exclude values, in a similar fashion to the != or <> (not equal to) operators. The difference is that **NOT** can be used with other clauses and keywords. For the majority of cases, however, use != to exclude values. The logical **XOR** excludes where neither case is true *or* if *both* cases are true. It is not a common enough use case that I would include it here except in the interests of completion of the topic.

You can have multiple AND/OR/NOT combinations in your WHERE clause:

```
SELECT * FROM `stock_items` WHERE category = "Canine"
OR category = "Feline" AND NOT inventory = 0;
```

Another useful conditional is, IS NULL. Though you might not use it to retrieve content for your website, it is very useful for “sanitizing” your data by searching for null values in your database:

```
SELECT * FROM stock_items WHERE price IS NULL;
```

We Can Use Imprecise Values to Find All Data That Matches a Pattern

If a grocery store wanted to query for all of their soups, they would have to write queries for “tomato soup”, “mushroom soup”, “chicken soup” et cetera. See the pattern? The text strings all end with “soup”. The **LIKE** keyword allows us to search for matches that are close without having to specify each exact value. There are two special characters that act as wildcard placeholders in our pattern: the underscore (**_**), and the percentage sign (**%**). The underscore matches any single character (like in a game of hangman), and the **%** matches any number of characters, including zero characters. So “_hick” would match “**c**hick” and “**t**hick”; “%hick%” would match the preceding and, “Chicken”, “hickory” “Schick razors”… To find all of our soups:

```
SELECT * FROM canned_goods WHERE item LIKE “%soup”;
```

This will get any item from the canned_goods column that starts with “…soup”. If you are familiar with regular expressions, there is an **RLIKE** keyword that lets you provide a regular expression to match. MySQL uses POSIX-style syntax for regular expressions.

We Can Provide a Set of Values or Range Of Values to Search For

With the **IN** and **BETWEEN** keywords, we can provide parameters to our **WHERE** clause for retrieving ranges of values. The **IN** keyword is followed by parentheses where we can specify multiple comma-separated values to search for:

```
SELECT * FROM stock_items WHERE category IN ("Canine",
"Feline", "Murine");
```

This query checks the category column for all of the values passed into the parentheses.

The **BETWEEN** keyword lets us specify an inclusive range to search for. The data can be numbers, text (alphabetical), or dates.

```
SELECT * FROM stock_items WHERE price BETWEEN 20 AND
100;
```

We Can Put a Hard Limit the Number of Rows in Our Result Set

In some case, we have crafted the perfect query, but we don’t want to see all of the rows at once: blogs often have a side bar that show the three most recent posts; ecommerce sites let you pick how many items that you want to view at once; social media sites show a limited number of posts until you scroll down. At the bottom of the non-scrolling pages are often “Next” buttons or page numbers for the grouped results. This is referred to as “pagination” – breaking your result set into “pages” of results. We can accomplish this with the **LIMIT** keyword. It puts the brakes on the result set after it finds the number of records that you specify.

```
SELECT * FROM stock_items LIMIT 5;
```

No matter how many results match the first part of the statement, the result set will only have a maximum of five results.

If we provide two values, the first number is the “offset” starting row (first result row is 0), and the second number is the limiter. For the third page of our 10-items-per-page results (items 30-40) would be:

```
SELECT * FROM stock_items LIMIT 29, 10;
```

NOTE: The **LIMIT** clause, if you use one, must always be the final clause of your query statement.

We Can Define the Sort Order of Our Result Set

Typically, your result set is ordered by the order of rows in the database. I say, “typically”, because this is what usually happens, but there is no guarantee that it will do so. To ensure the sort order of your result set, we can specify the column(s) that we want the results ordered by with the **ORDER BY** clause added to your statement. So, do you want your employees listed by *last* name, or *first* name? Are your hotel bookings listed in *chronological* or *reverse* chronological order?

```
SELECT * FROM stock_items ORDER BY category;
```

Without **ORDER BY**, our **SELECT** statement would return the above in the order that they appear in the database. With **ORDER BY**, the results will be grouped by category alphabetically ascending (A-Z). We can also specify multiple columns in which to order our results, with each column separated by a comma. The following query will order the results by category alphabetically, then by price ascending from lowest to highest:

```
SELECT * FROM stock_items ORDER BY category, price;
```

If we want a reverse ordering, we add the **DESC** keyword (for descending) to each column that we wish to be ordered in reverse:

```
SELECT * FROM stock_items ORDER BY category DESC, price
DESC;
```