**Definitions**

* Structured Query Language is a programming language designed to manage data stored in relational databases.
* The statements covered in this Codecademy course, use SQLite Relational Database Management System (RDBMS).
* A *relation database* is a database that organizes information into one or more tables.
* A *table* is sometimes referred to as *relations*.
* Some of the most common data types are:
  + *Integer*, a positive or negative whole number
  + *Text*, a text string
  + *Date*, the date formatted as YYYY-MM-DD
  + *Real*, a decimal value
* A *statement* is text that the database recognizes as a valid command. Statements always end in a semi-colon. The structure of SQL statements vary. The number of lines used do not matter. A statement can be written all on one line, or split up across multiple lines if it makes it easier to read.
* *Clauses* perform specific tasks in SQL. By convention, clauses are written in capital letters. Clauses can also be referred to as commands.
* A *parameter* is a list of columns, data types, or values that are passed to a clause as an argument. Here, the parameter is a list of column names and the associated data type.
* One of the core purposes of the SQL language is to retrieve information stored in a database. This is commonly referred to as *querying*. Queries allow us to communicate with the database by asking questions and having the result set return data relevant to the question.
* Filtering the results of a query is an important skill in SQL.
* > is an *operator*. Operators create a condition that can be evaluated as either true or false. Common operators used with the WHERE clause are:
  + = equals
  + != not equals
  + > greater than
  + < less than
  + >= greater than or equal to
  + <= less than or equal to
* *Aggregate functions* compute a single result from a set of input values. For instance, when we need the sum or average of a particular column, we can use aggregate functions to quickly compute it for us.

**STATEMENTS**

* **SELECT \* FROM celebs;**
  + celebs is the table.
  + This statement queries the entire table celebs because we have used the \* character.
  + SELECT statements always return a new table called the *result set*.
* **SELECT name FROM celebs;**
  + This statement fetches all entries under the name column from the celebs table.
* **SELECT name, imdb\_rating FROM movies;**
  + This statement fetches the name and IMDb rating of every movie in the database.
* **SELECT DISTINCT genre FROM movies;**
  + SELECT DISTINCT is used to return unique values in the result set. It filters out all duplicate values. Here, the result set lists each genre in the movies table exactly once.
* **SELECT \* FROM movies WHERE imdb\_rating > 8;**
  + This statement filters the result set to only include movies with IMDb ratings greater than 8.
  + Here, only rows with a value greater than 8 in the imdb\_rating column will be returned in the result set.
* **SELECT \* FROM movies WHERE name LIKE ‘Se\_en’;**
  + LIKE can be a useful operator when you want to compare similar values. Here we are comparing two movies with the same name but are spelled differently. For example, we can see the results Se7en and Seven in the result set.
  + name LIKE Se\_en is a condition evaluating the name column for a specific pattern.
  + % is another wildcard character that can be used with LIKE. See below.
* **SELECT \* FROM movies WHERE name LIKE ‘a%’;**
  + This statement returns a result set in which all movie names starting with the letter A.
* **SELECT \* FROM movies WHERE name LIKE ‘%a”;**
  + This statement matches all movies that end with the letter A.
* **SELECT \* FROM movies WHERE name LIKE ‘%man%’;**
  + This statement returns a result set in which all movie names have the string “man”.
* **SELECT \* FROM movies WHERE name BETWEEN ‘A’ AND ‘J’;**
  + This statement returns all movie names that start with the letters A through J.
* **SELECT \* FROM movies WHERE name BETWEEN 1990 AND 2000 AND genre = ‘comedy’;**
  + This statement returns all comedy movies that were released between the years 1990 through 2000.
  + The values can also be dates.
* **SELECT \* FROM movies WHERE genre = ‘comedy’ OR year < 1980;**
* **SELECT \* FROM movies ORDER BY imdb\_rating DESC LIMIT 3;**
  + ORDER BY is a clause that indicates you want to sort the result set by a particular column either alphabetically or numerically.
  + imdb\_rating is the name of the column that will be sorted.
  + DESC is a keyword in SQL that is used with ORDER BY to sort the results in descending order. Here, it sorts all of the movies from highest to lowest by their IMDb rating.
  + It is also possible to sort the results in ascending order. ASC is a keyword in SQL that is used with ORDER BY to sort the results in ascending order.
  + LIMIT is a clause that lets you specify the maximum number of rows the result set will have. Here, we specify that the result set can not have more than three rows.
* **SELECT COUNT(\*) FROM fake\_apps;**
  + Count how many apps are in the database.
  + The fastest way to calculate the number of rows in a table is to use the COUNT( ) function.
  + COUNT( ) is a function that takes the name of a column as an argument and counts the number of rows where the column is not NULL. Here, we want to count every row so we pass \* as an argument.
* **SELECT COUNT(\*) FROM fake\_apps WHERE price = 0;**
  + Return the total number of free apps in the table.
* **SELECT price, COUNT(\*) FROM fake\_apps GROUP BY price;**
  + Count the number of apps at each price.
  + GROUP BY is a clause in SQL that is only used with aggregate functions. It is used in collaboration with the SELECT statement to arrange identical data into groups.
  + Here, our aggregate function is COUNT( ) and we are passing price as an argument to GROUP BY. SQL will count the total number of apps for each price in the table.
  + It is usually helpful to SELECT the column you pass as an argument to GROUP BY. Here we select price and COUNT(\*). You can see that the result set is organized into two columns making it easy to see the number of apps at each price.
* **CREATE TABLE celebs (id INTEGER, name TEXT, age INTEGER);**
  + The clause is CREATE TABLE.
  + id is the first column in the table. It stores values of data type INTEGER. We can say similarly for the rest of the parameters.
* **INSERT INTO celebs (id, name, age) VALUES (1, ‘Justin Bieber’, 21);**
  + Adds a row to the table.
* **UPDATE celebs**

**SET age = 22**

**WHERE id = 1;**

* + UPDATE is a clause that edits a row in the specified table.
  + SET is a clause that indicates the column to edit.
    - age is the name of the column that is going to be updated.
    - 22 is the new value that is going to be inserted into the age column.
  + WHERE is a clause that indicates which row(s) to update with the new column value. Here the row with a 1 in the id column is the row that will have the age updated to 22.
* **ALTER TABLE celebs ADD COLUMN twitter\_handle TEXT;**
  + ALTER TABLE is a clause that lets you make the specified changes.
  + celebs is the name of the table that is being changed.
  + ADD COLUMN is a clause that lets you add a new column to a table.
  + twitter\_handle is the name of the new column being added.
  + TEXT is the data type for the new column.
  + NULL is a special value in SQL that represents missing or unknown data. Here, the rows that existed before the column was added have NULL values for twitter\_handle.
* **DELETE FROM celebs WHERE twitter\_handle is NULL;**
  + Delete all of the rows that have a NULL value in the twitter column. This will delete the entire row!
  + DELETE FROM is a clause that lets you delete rows from a table.
  + IS NULL is a condition in SQL that returns true when the value is NULL and false otherwise.

**AGGREGATE FUNCTIONS**

* The fastest way to calculate how many rows are in a table is to use the COUNT( ) function. COUNT( ) is a function that takes the name of a column as an argument and counts the number of non-empty values in that column. Here, we want to count every row, so we pass \* as an argument inside the parentheses.
  + SELECT COUNT(\*) FROM table\_name;
* SQL makes it easy to add all values in a particular column using SUM( ). SUM( ) is a function that takes the name of a column as an argument and returns the sum of all the values in that column.
  + SELECT SUM(downloads) FROM fake\_apps;
* The MAX( ) and MIN( ) functions return the highest and lowest values in a column, respectively.
  + SELECT MAX(downloads) FROM fake\_apps;
* SQL uses the AVG( ) function to quickly calculate the average value of a particular column. The AVG( ) function works by taking a column name as an argument and returns the average value for that column.
  + SELECT AVG(downloads) FROM fake\_apps;
* By default, SQL tries to be as precise as possible without rounding. We can make the result table easier to read using the ROUND( ) function. ROUND( ) function takes two arguments inside the parenthesis: a column name, and an integer. It rounds the values in the column to the number of decimal places specified by the integer.
  + SELECT ROUND(price, 0) FROM fake\_apps;
  + SELECT name, ROUND(price, 0) FROM fake\_apps;
    - Returns the name column and a rounded price column.
  + SELECT ROUND(AVG(price), 2) FROM fake\_apps;
    - Rounds the average price of an app to 2 decimal places.
* For instance, we might want to know the mean IMDb ratings for all movies each year. So, we can use GROUP BY to do this in a single step! GROUP BY is a clause in SQL that is used with aggregate functions. It is used in collaborating with the SELECT statement to arrange identical data into *groups*. The GROUP BY statement comes after any WHERE statements, but not before ORDER BY or LIMIT.
  + SELECT year, AVG(imdb\_rating) FROM movies GROUP BY year ORDER BY year;
  + SELECT price, COUNT(\*) FROM fake\_apps GROUP BY price;
    - Lists all groupings of prices (price values that are identical), and shows the number of occurrences for each price value.
  + SELECT price, COUNT(\*) FROM fake\_apps WHERE downloads > 20000 GROUP BY price;
    - Count the total number of apps that has been downloaded more than 20,000 times, at each price.
  + SELECT category, SUM(downloads) FROM fake\_apps GROUP BY category;
    - Calculates the total number of downloads for each category.
  + SELECT ROUND(imdb\_rating), COUNT(name) FROM movies GROUP BY ROUND(imdb\_rating) ORDER BY ROUND(imdb\_rating);
    - Find out how many movies have IMDb ratings that round to 1, 2, 3, 4, or 5.
  + SELECT ROUND(imdb\_rating), COUNT(name) FROM movies GROUP BY 1 ORDER BY 1;
    - Equivalent to the statement above, the 1 refers to the first column in our SELECT statement, ROUND(imdb\_rating).
  + SELECT category, price, AVG(downloads) FROM fake\_apps GROUP BY 1, 2;
    - Find the average number of downloads for each category and each price value.
* In addition to being able to group data using GROUP BY, SQL also allows you to filter which groups to include and which to exclude. For instance, imagine that we want to see how many movies of different genres were produced each year, but we only care about years and genres with at least 10 movies. We can’t use WHERE here because we don’t want to filter the rows; we want to filter groups. This is where HAVING comes in. HAVING is similar to WHERE. In fact, all types of WHERE clauses you learned about thus far can be used with HAVING. When we want to limit the results of a query based on values of the individual rows, use WHERE. When we want to limit the results of a query based on an aggregate property, use HAVING. HAVING statement always comes after GROUP BY, but before ORDER BY and LIMIT.
  + SELECT year, genre, COUNT(name) FROM movies GROUP BY 1, 2 HAVING COUNT(name) > 10;
  + SELECT price, ROUND(AVG(downloads)) FROM fake\_apps GROUP BY price HAVING COUNT(price) > 9;
    - Restrict the query to prices where the total number of apps at that price point is greater than 9.

**MULTIPLE TABLES**

* Combining tables manually is time-consuming. Luckily, SQL gives us an easy sequence for this: it’s called a JOIN. If we want to combine orders and customers, we would type:
  + SELECT \* FROM orders JOIN customers ON orders.customers\_id = customers.customer\_id;
    - We want to match customer\_id from orders with customer\_id from customers.
* Because column names are often repeated across multiple tables, we use the syntax table\_name.column\_name to be sure that our requests are unambiguous. In our example, we use the syntax in the ON statement, but we will also use it in the SELECT or any other statement where we refer to column names. For example, if we only wanted to select the order\_id from orders and the customer name from customers, we could use the following query:
  + SELECT orders.order\_id, customers.customer\_name FROM orders JOIN customers ON orders.customer\_id = customers.customer\_id;
* Join orders and subscriptions and select all columns. Make sure to join on subscription\_id.
  + SELECT \* FROM orders JOIN subscriptions ON orders.subscription\_id = subscriptions.subscription\_id;
  + SELECT \* FROM orders JOIN subscriptions ON orders.subscription\_id = subscriptions.subscription\_id WHERE subscriptions.description = ‘Fashion Magazine’;
    - Adds a second query after the first one that only selects rows from the join where description is equal to ‘Fashion Magazine’.
* When we perform a simple JOIN (often called an inner join) our result only includes rows that match on our ON condition. Therefore, the non-matching values are not included.
* Count the number of subscribers who get have both a newspaper and online subscription.
  + SELECT COUNT(\*) FROM newspaper JOIN online WHERE newspaper.id = online.id;
* What if we want to combine two tables and keep some of the un-matched rows? SQL lets us do this through a command called LEFT JOIN. A *left join* will keep all rows from the first table, regardless of whether there is a matching row in the second table.
  + SELECT \* FROM newspaper LEFT JOIN online ON newspaper.id = online.id;
    - Suppose we want to know how many users subscribe to the print newspaper, but not to the online. Select all columns.
  + SELECT \* FROM newspaper LEFT JOIN online ON newspaper.id = online.id WHERE online.id IS NULL;
    - Select all rows where there was no corresponding row from the online table (all users who do not subscribe to the online edition from the newspaper table).
* There can never be more than two primary key columns in a table, and each of the values under the primary key column cannot be NULL.
  + SELECT \* FROM classes JOIN students ON classes.id = students.class\_id;
    - The name of the foreign key does not have to match the name of the primary key.
* Sometimes, we just want to combine all rows of one table with all rows of another table. For instance, if we had a table of shirts that described different shirts we own, and another table called pants that described different pants that we owned, we might want to know all possible combinations of shirts and pants to create outfits. Our code might look like this:
  + SELECT shirts.shirt\_color, pants.pant\_color FROM shirts CROSS JOIN pants;
    - Notice that cross joins don’t require an ON statement. You’re not really joining on any columns.
  + Eventually, we’ll use a cross join to help us, but first, let’s try a simpler problem. Let’s start by counting the number of customers who were subscribed to the newspaper during March.
    - SELECT COUNT(\*) FROM newspaper WHERE start\_month < 3 AND end\_month > 3;
  + SELECT \* FROM newspaper CROSS JOIN months;
    - Select all columns from the cross join of newspaper and months.
  + SELECT \* FROM newspaper CROSS JOIN months WHERE months.month > newspaper.start\_month AND months.month < newspaper.end\_month;
    - Create a third query to select all months where a user was subscribed.
  + SELECT month, COUNT(\*) AS subscribers FROM newspaper CROSS JOIN months WHERE months.month > newspaper.start\_month AND months.month < newspaper.end\_month GROUP BY month;
    - Create a final query where you aggregate over each month, and return a table showing each month with subscribers and its number of subscribers.
* Sometimes, we want to stack one dataset on top of the other. SQL has strict rules for appending data. Tables must have the same # of columns. The columns must have the same data types in the same order as the first table.
  + SELECT \* FROM newspaper UNION SELECT \* FROM online;
    - Combines both tables together into one big table with both sets of data.
* For instance, our marketing department may want to know how many magazines each customer subscribes to.
  + SELECT customer\_id, COUNT(subscription\_id) AS subscriptions FROM orders GROUP BY customer\_id;
    - The query is good, but a customer\_id isn’t terribly useful for our marketing department, they probably want to know the customer’s name.
    - We want to be able to join the results of this query with our customers table, which will tell us the name of each customer. We can do this by using a WITH clause.
  + WITH previous\_results AS (  
     SELECT customer\_id,  
     COUNT(subscription\_id) AS subscription  
     FROM orders  
     GROUP BY customer\_id  
    )  
    SELECT customers.customer\_name, previous\_query.subscriptions   
    FROM previous query  
    JOIN customers  
     ON previous\_query.customer\_id = customers.customer\_id;
    - The WITH statement allows us to perform a separate query (such as aggregating customer’s subscriptions) and then joining those results with another table to gather the customer name and subscription count.
    - AS is a keyword that introduces a variable-like name.