SQL Cheatsheet

16 commits | **1** branch | **0** releases | **1** contributor

Branch: **master** | New pull request | Find file | Clone or download

Fetching latest commit…

README.md	Update README.md	Aug 24, 2017
SQL_commands.md	Update SQL_commands.md	May 21, 2017

README.md

SQL cheatsheet

This is note taken from the SQL course on Codecademy.

Manipulation

```
CREATE TABLE celebs (id INTEGER, name TEXT, age INTEGER);
```

This `CREATE` statement creates a new table in the database named `celebs`. You can use the `CREATE` statement anytime you want to create a new table from scratch.

1. `CREATE TABLE` is a clause that tells SQL you want to create a new table.
2. `celebs` is the name of the table.
3. `(id INTEGER, name TEXT, age INTEGER)` is a list of parameters defining each column in the table and its data type.

- `id` is the first column in the table. It stores values of data type `INTEGER`
- `name` is the second column in the table. It stores values of data type `TEXT`
- `age` is the third column in the table. It stores values of data type `INTEGER`

Add a row to the table. In the code editor type

```
INSERT INTO celebs (id, name, age) VALUES (1, 'Justin Bieber', 21);
```

To view the row you just created, under the `INSERT` statement type `SELECT * FROM celebs;`.

This `INSERT` statement inserts new rows into a table. You can use the `INSERT` statement when you want to add new records.

1. `INSERT INTO` is a clause that adds the specified row or rows.
2. `celebs` is the name of the table the row is added to.
3. `(id, name, age)` is a parameter identifying the columns that data will be inserted into.
4. `VALUES` is a clause that indicates the data being inserted. `(1, 'Justin Bieber', 21)` is a parameter identifying the values being inserted.

- `1` is an integer that will be inserted into the `id` column
- `'Justin Bieber'` is text that will be inserted into the `name` column
- `21` is an integer that will be inserted into the `age` column

```
SELECT name FROM celebs;
```

`SELECT` statements are used to fetch data from a database. Here, `SELECT` returns all data in the `name` column of the `celebs` table.

1. `SELECT` is a clause that indicates that the statement is a query. You will use `SELECT` every time you query data from a database.
2. `name` specifies the column to query data from.
3. `FROM celebs` specifies the name of the table to query data from. In this statement, data is queried from the`celebs` table.

You can also query data from all columns in a table with `SELECT`.

```
SELECT * FROM celebs;
```

`*` is a special wildcard character that we have been using. It allows you to select every column in a table without having to name each one individually. Here, the result set contains every column in the `celebs` table.

`SELECT` statements always return a new table called the*result set.*

```
UPDATE celebs
SET age = 22
WHERE id = 1;
```

The `UPDATE` statement edits a row in the table. You can use the `UPDATE` statement when you want to change existing records.

1. `UPDATE` is a clause that edits a row in the table.
2. `celebs` is the name of the table.
3. `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.

4. `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;
```

The `ALTER TABLE` statement added a new column to the table. You can use this command when you want to add columns to a table.

1. `ALTER TABLE` is a clause that lets you make the specified changes.
2. `celebs` is the name of the table that is being changed.
3. `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

4. `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;
```

The `DELETE FROM` statement deletes one or more rows from a table. You can use the statement when you want to delete existing records.

1. `DELETE FROM` is a clause that lets you delete rows from a table.
2. `celebs` is the name of the table we want to delete rows from.
3. `WHERE` is a clause that lets you select which rows you want to delete. Here we want to delete all of the rows where the twitter_handle column `IS NULL`.
4. `IS NULL` is a condition in SQL that returns true when the value is `NULL` and false otherwise.

Queries

Database Schema

movies220 rows	
id	INTEGER
name	TEXT
genre	TEXT
year	INTEGER
imdb_rating	REAL

```
SELECT name, imdb_rating FROM movies;
```

In Lesson 1 you learned that `SELECT` is used every time you want to query data from a database.

Multiple columns can be queried at once by separating column names with a comma. By specifying `name, imdb_rating`, the result set contains a `name` and `imdb_rating` column.

```
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.

1. `SELECT DISTINCT` specifies that the statement is going to be a query that returns unique values in the specified column(s)

2. `genre` is the name of the column to display in the result set.

3. `FROM movies` indicates the table name to query from.

Filtering the results of a query is an important skill in SQL. It is easier to see the different possible genres a movie can have after the data has been filtered, than to scan every row in the table.

The rest of this lesson will teach you different commands in SQL to filter the results of a query.

```
SELECT * FROM movies
  WHERE imdb_rating > 8;
```

This statement filters the result set to only include movies with IMDb ratings greater than 8. How does it work?

1. `WHERE` is a clause that indicates you want to filter the result set to include only rows where the following *condition* is true.

2. `imdb_rating > 8` is a condition that filters the result set. Here, only rows with a value greater than 8 in the `imdb_rating` column will be returned in the result set.

3. `>` 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

There are also some special operators that we will learn more about in the upcoming exercises.

```
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.

1. `LIKE` is a special operator used with the `WHERE` clause to search for a specific pattern in a column.

2. `name LIKE Se_en` is a condition evaluating the `name` column for a specific pattern.

3. `Se_en` represents a pattern with a *wildcard* character. The `_` means you can substitute any individual character here without breaking the pattern. The names `Seven` and `Se7en` both match this pattern.

`%` is another wildcard character that can be used with `LIKE`. We will learn more about `%` in the next exercise.

```
SELECT * FROM movies
WHERE name LIKE 'A%';
```

This statement filters the result set to only include movies with names that begin with the letter "A"

`%` is a wildcard character that matches zero or more missing letters in the pattern.

- `A%` matches all movies with names that begin with "A"
- `%a` matches all movies that end with "a"

```
SELECT * FROM movies WHERE name LIKE '%man%';
```

You can use `%` both before and after a pattern. Here, any movie that contains the word "man" in its name will be returned in the result set. Notice, that `LIKE` is not case sensitive. "Batman" and "Man Of Steel" both appear in the result set.

The `BETWEEN` operator is used to filter the result set within a certain range. The values can be numbers, text or dates.

```
SELECT * FROM movies
WHERE name BETWEEN 'A' AND 'J';
```

This statement filters the result set to only include movies with `name`s that begin with letters "A" up to but not including "J".

```
SELECT * FROM movies WHERE year BETWEEN 1990 AND 2000;
```

In this statement, the `BETWEEN` operator is being used to filter the result set to only include movies with `year` s between 1990 up to and including 2000.

```
  SELECT * FROM movies
  WHERE year BETWEEN 1990 and 2000
  AND genre = 'comedy';
```

Sometimes you want to combine multiple conditions in a `WHERE` clause to make the result set more specific and useful. One way of doing this is to use the `AND` operator.

1. `year BETWEEN 1990 and 2000` is the first condition in the `WHERE` clause.
2. `AND genre = 'comedy'` is the second condition in the `WHERE` clause.
3. `AND` is an operator that combines two conditions. Both conditions must be true for the row to be included in the result set. Here, we use the `AND` operator to only return movies made between 1990 and 2000 that are also comedies.

```
  SELECT * FROM movies
  WHERE genre = 'comedy'
  OR year < 1980;
```

The `OR` operator can also be used to combine more than one condition in a `WHERE` clause. The `OR` operator evaluates each condition separately and if any of the conditions are true then the row is added to the result set.

1. `WHERE genre = 'comedy'` is the first condition in the `WHERE` clause.
2. `OR year < 1980` is the second condition in the `WHERE` clause.
3. `OR` is an operator that filters the result set to only include rows where either condition is true. Here, we return movies that either have a genre of comedy or were released before 1980.

```
SELECT * FROM movies
ORDER BY imdb_rating DESC;
```

You can sort the results of your query using `ORDER BY`. Sorting the results often makes the data more useful and easier to analyze.

1. `ORDER BY` is a clause that indicates you want to sort the result set by a particular column either alphabetically or numerically.
2. `imdb_rating` is the name of the column that will be sorted.
3. `DESC` is a keyword in SQL that is used with `ORDER BY` to sort the results in*descending order* (high to low or Z-A). 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 (low to high or A-Z).

```
SELECT * FROM movies
ORDER BY imdb_rating DESC
LIMIT 3;
```

Sometimes even filtered results can return thousands of rows in large databases. In these situations it becomes important to cap the number of rows in a result set.

`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.

Aggregate Functions

Database Schema

fake_apps200 rows	
id	INTEGER
name	TEXT
category	TEXT
downloads	INTEGER
price	REAL

```
SELECT COUNT(*) FROM fake_apps;
```

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 price, COUNT(*) FROM fake_apps
GROUP BY price;
```

Aggregate functions are more useful when they organize data into groups.

`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.

Count the total number of apps at each price that have been downloaded more than 20,000 times.

```
SELECT price, COUNT(*) FROM fake_apps WHERE downloads > 20000 GROUP BY price;
```

```
SELECT SUM(downloads) FROM fake_apps;
```

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. Here, it adds all the values in the `downloads` column.

```
SELECT MAX(downloads) FROM fake_apps;
```

You can find the largest value in a column by using`MAX()`.

`MAX()` is a function that takes the name of a column as an argument and returns the largest value in that column. Here, we pass `downloads` as an argument so it will return the largest value in the`downloads` column.

Return the names of the most downloaded apps in each category.

```
SELECT name, category, MAX(downloads) FROM fake_apps GROUP BY category;
```

```
SELECT MIN(downloads) FROM fake_apps;
```

Similar to `MAX()`, SQL also makes it easy to return the smallest value in a column by using the`MIN()` function. `MIN()` is a function that takes the name of a column as an argument and returns the smallest value in that column. Here, we pass `downloads` as an argument so it will return the smallest value in the `downloads` column.

```
SELECT AVG(downloads) FROM fake_apps;
```

This statement returns the average number of downloads for an app in our database. 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 price, ROUND(AVG(downloads), 2) FROM fake_apps
GROUP BY price;
```

By default, SQL tries to be as precise as possible without rounding. We can make the result set easier to read using the `ROUND()` function.

`ROUND()` is a function that takes a column name and an integer as an argument. It rounds the values in the column to the number of decimal places specified by the integer. Here, we pass the column `AVG(downloads)` and `2` as arguments. SQL first calculates the average for each price and then rounds the result to two decimal places in the result set.

Round the average number of downloads to the nearest integer for each price.

```
SELECT price, ROUND(AVG(downloads)) FROM fake_apps
GROUP BY price;
```

Multiple Tables

So far we have learned how to build tables, write queries, and perform calculations using one table. In this lesson we will learn to query multiple tables that have relationships with each other.

Most of the time, data is distributed across multiple tables in the database. Imagine a database with two tables,`artists` and `albums`. An artist can produce many different albums, and an album is produced by an artist.

The data in these tables are related to each other. Through SQL, we can write queries that combine data from multiple tables that are related to one another. This is one of the most powerful features of relational databases.

Database Schema

albums14 rows	
id	INTEGER
name	TEXT
artist_id	INTEGER
year	INTEGER

artists0 rows	
id	INTEGER
name	TEXT

We have created a table named `albums` for you. Create a second table named `artists`.

```
CREATE TABLE artists(id INTEGER PRIMARY KEY, name TEXT);
```

```
CREATE TABLE artists(id INTEGER PRIMARY KEY, name TEXT)
```

Using the `CREATE TABLE` statement we added a `PRIMARY KEY` to the `id` column.

A **primary key** serves as a unique identifier for each row or record in a given table. The primary key is literally an`id` value for a record. We're going to use this value to connect `artists` to the `albums` they have produced.

By specifying that the `id` column is the `PRIMARY KEY`, SQL makes sure that:

- None of the values in this column are `NULL`
- Each value in this column is unique

A table can not have more than one `PRIMARY KEY` column.

```
SELECT * FROM albums WHERE artist_id = 3;
SELECT * FROM artists WHERE id = 3;
```

A *foreign key* is a column that contains the primary key of another table in the database. We use foreign keys and primary keys to connect rows in two different tables. One table's foreign key holds the value of another table's primary key. Unlike primary keys, foreign keys do not need to be unique and can be `NULL`.

Here, `artist_id` is a foreign key in the `albums` table. We can see that Michael Jackson has an `id` of 3 in the `artists` table. All of the albums by Michael Jackson also have a 3 in the `artist_id` column in the `albums` table.

This is how SQL is linking data between the two tables. The*relationship* between the `artists` table and the `albums` table is the `id` value of the artists.

```
SELECT albums.name, albums.year, artists.name FROM albums, artists
```

One way to query multiple tables is to write a `SELECT` statement with multiple table names separated by a comma. This is also known as a *cross join*. Here, `albums` and `artists` are the different tables we are querying.

When querying more than one table, column names need to be specified by `table_name.column_name`. Here, the result set includes the `name` and `year` columns from the `albums` table and the `name` column from the `artists` table.

Unfortunately the result of this cross join is not very useful. It combines every row of the `artists` table with every row of the `albums` table. It would be more useful to only combine the rows where the album was created by the artist.

```
SELECT * FROM albums JOIN artists ON albums.artist_id = artists.id;
```

In SQL, joins are used to combine rows from two or more tables. The most common type of join in SQL is an*inner join*.

An inner join will combine rows from different tables if the*join condition* is true. Let's look at the syntax to see how it works.

1. `SELECT *` specifies the columns our result set will have. Here, we want to include every column in both tables.
2. `FROM albums` specifies the first table we are querying.
3. `JOIN artists ON` specifies the type of join we are going to use as well as the name of the second table. Here, we want to do an inner join and the second table we want to query is `artists`.
4. `albums.artist_id = artists.id` is the join condition that describes how the two tables are related to each other. Here, SQL uses the foreign key column `artist_id` in the `albums` table to match it with exactly one row in the `artists` table with the same value in the `id` column. We know it will only match one row in the `artists` table because `id` is the `PRIMARY KEY` of `artists`.

```
SELECT * FROM albums LEFT JOIN artists ON albums.artist_id = artists.id;
```

Outer joins also combine rows from two or more tables, but unlike inner joins, they do not require the join condition to be met. Instead, every row in the *left* table is returned in the result set, and if the join condition is not met, then `NULL` values are used to fill in the columns from the *right* table.

The left table is simply the first table that appears in the statement. Here, the left table is `albums`. Likewise, the right table is the second table that appears. Here, `artists` is the right table.

```
SELECT
  albums.name AS 'Album',
  albums.year,
  artists.name AS 'Artist'
FROM
  albums
JOIN artists ON
  albums.artist_id = artists.id
WHERE
  albums.year > 1980;
```

`AS` is a keyword in SQL that allows you to rename a column or table using an*alias*. The new name can be anything you want as long as you put it inside of single quotes. Here we want to rename the `albums.name` column as `'Album'`, and the `artists.name` column as `'Artist'`.

It is important to note that the columns have not been renamed in either table. The aliases only appear in the result set.

Some other commands:

The `SUBSTR` functions return a portion of `char`, beginning at character `position`, `substring_length` characters long. `SUBSTR` calculates lengths using characters as defined by the input character set. `SUBSTRB` uses bytes instead of characters. `SUBSTRC` uses Unicode complete characters. `SUBSTR2` uses UCS2 code points. `SUBSTR4` uses UCS4 code points.

- If `position` is 0, then it is treated as 1.
- If `position` is positive, then Oracle Database counts from the beginning of `char` to find the first character.
- If `position` is negative, then Oracle counts backward from the end of `char`.
- If `substring_length` is omitted, then Oracle returns all characters to the end of `char`. If `substring_length` is less than 1, then Oracle returns null.
- Examples

  The following example returns several specified substrings of "ABCDEFG":

  ```
  SELECT SUBSTR('ABCDEFG',3,4) "Substring"
       FROM DUAL;

  Substring
  ---------
  CDEF

  SELECT SUBSTR('ABCDEFG',-5,4) "Substring"
       FROM DUAL;

  Substring
  ---------
  CDEF
  ```