Structured Query Language

KJM

November 29, 2024

Contents

1 Manipulation

In this first section, ways to alter or add data to a table will be introduced. This will differ from the next section, where we will then discover ways to, once given our table in final form, extract insights from it.

All data stored in a relational database is of a certain data type. 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

1.1 CREATE

CREATE statements allow us to create a new table in the database. You can use the CREATE statement anytime you want to create a new table from scratch. The statement below creates a new table named celebs.

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

1.2 INSERT

The INSERT statement inserts a new row into a table.

We can use the INSERT statement when you want to add new rows. The statement below enters a record for Justin Bieber into the celebs table.

```
INSERT INTO celebs (id, name, age)
VALUES (1, 'Justin Bieber', 29);
```

1.3 ALTER

The ALTER TABLE statement adds a new column to a table. You can use this command when you want to add columns to a table. The statement below adds a new column 'twitter_handle' to the celebs table.

```
ALTER TABLE celebs
ADD COLUMN twitter_handle TEXT;
```

Note that after adding this, all rows will have a NULL value

1.4 UPDATE

The UPDATE statement edits a row in a table. You can use the UPDATE statement when you want to change existing records. The statement below updates the record with an id value of 4 to have the 'twitter_handle' @taylorswift13.

```
UPDATE celebs
SET twitter_handle = '@taylorswift13'
WHERE id = 4;
```

1.5 DELETE

The DELETE FROM statement deletes one or more rows from a table. You can use the statement when you want to delete existing rows. The statement below deletes all rows in the celebs table with no twitter_handle:

```
DELETE FROM celebs
WHERE twitter_handle IS NULL;
```

1.6 Constraints

When creating a new table, we must specify column names and then their data types. We can additionally specify some extra information about a given column. For example, we could impose that upon insertion a given column will not accept NULL values, or that if we do not specify a value for a given column then it is assigned some default value.

```
CREATE TABLE celebs (
   id INTEGER PRIMARY KEY,
   name TEXT UNIQUE,
   date_of_birth TEXT NOT NULL,
   date_of_death TEXT DEFAULT 'Not␣Applicable'
);
```

2 Queries

Now we will introduce different SQL commands to query a single table in a database. This means we will be using SQL to extract information and insights from a given table (querying), rather than changing the table itself like the previous section (manipulating).

We will be querying a database with one table named movies.

2.1 SELECT and FROM

SELECT statements are used to fetch data from a database. After the SELECT clause, we then list which columns from the data we went. If we want all columns we put an asterisk after SELECT.

At some point after a SELECT clause, we put a FROM statement in order to specify which table we are querying. In the example below, we want data from the id and age columns.

```
SELECT id, age
FROM celebs ;
```

2.1.1 AS

AS is a keyword in SQL that allows you to rename a column or table using an alias. When selecting a column, we use the name as provided by the table we are working with. Then we can rename it to something which may be more convenient or easy to work with.

```
SELECT name AS 'Titles'
FROM movies;
```

Perhaps one may find the column name 'name' too vague and when drawing insights we want to re-label this column with a more explanatory name like 'Titles'.

2.1.2 DISTINCT

When we are examining data in a table, it can be helpful to know what distinct values exist in a particular column. DISTINCT is used to return unique values in the output. It filters out all duplicate values in the specified column(s). For instance,

```
SELECT DISTINCT genre
FROM movies;
```

2.2 WHERE

We can restrict our query results using the WHERE clause in order to obtain only the information that meets a certain condition. Following this format, the statement below filters the result set to only include top rated movies (IMDb ratings greater than 8):

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

The > is an operator. Operators create a condition that can be evaluated as either true or false. Comparison operators used with the WHERE clause are:

- != not equal to
- = equal to
- > greater than
- < less than
- >= greater than or equal to
- <= less than or equal to

2.2.1 LIKE

LIKE can be a useful operator when you want to compare similar values. There are two movies in the movies table with similar titles, 'Se7en' and 'Seven'. How could we select all movies that start with 'Se' and end with 'en' and have exactly one character in the middle?

```
SELECT *
FROM movies
WHERE name LIKE 'Se_en';
```

The _ means you can substitute any individual character here without breaking the pattern. The names Seven and Se7en both match this pattern.

The percentage sign % is another wildcard character that can be used with LIKE. This statement below filters the result set to only include movies with names that begin with the letter 'A':

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

Notice, if we wanted a movie that started with 'The', then we would need to write LIKE 'The %' instead of 'The%'. This is because in the latter, we could return results that begin with 'There' for example.

We can also use this to find names of movies that end with the letter 'a'. This is done by instead writing LIKE '%a'. So we can see that the % sign is used to represent any sequence of characters, integers, white space, etc.

Furthermore, we could find any movie with the sequence 'man' in its name. This is done by writing '%man%'. LIKE is not case sensitive. 'Batman' and 'Man of Steel' will both appear in the result of the query above.

2.2.2 IS NULL

To filter for all movies with an IMDb rating:

```
SELECT *
FROM movies
WHERE imdb_rating IS NOT NULL;
```

2.2.3 BETWEEN

The BETWEEN operator is used in a WHERE clause to filter the result set within a certain range. It accepts two values that are either numbers, text or dates. For example, this statement filters the result set to only include movies with years from 1990 up to, and *including* 1999.

```
SELECT *
FROM movies
WHERE year BETWEEN 1990 AND 1999;
```

When the values are text, BETWEEN filters the result set for within the alphabetical range. In this statement, BETWEEN filters the result set to only include movies with names that begin with the letter 'A' up to, but not including ones that begin with 'J'.

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

However, if a movie has a name of simply 'J', it would actually match. This is because BETWEEN goes up to the second value — up to 'J'. So the movie named 'J' would be included in the result set but not 'Jaws'.

2.2.4 AND, OR

Sometimes we 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.

```sql
SELECT *
FROM movies
WHERE (year BETWEEN 1970 AND 1979)
  AND (imdb_rating > 8);
```

2.3 ORDER BY

It is often useful to list the data in our result set in a particular order. We can sort the results using ORDER BY, either alphabetically or numerically. For example, if we want to select all of the well-received movies, sorted from highest to lowest by their year:

```sql
SELECT *
FROM movies
WHERE imdb_rating > 8
ORDER BY year DESC;
```

2.3.1 DESC

DESC is a keyword used in ORDER BY to sort the results in descending order (high to low or Z-A). ASC is a keyword used in ORDER BY to sort the results in ascending order (low to high or A-Z).

2.4 CASE

2.4.1 SELECT

A CASE statement in SQL allows us to implement conditional logic, similar to "if-then" statements in programming. Typically used in a SELECT statement, it helps create a new column where the output is determined based on the conditions we define. Each condition is evaluated in order, and the corresponding result is returned when a condition is met. Consider the following example.

```sql
SELECT name,
 CASE
  WHEN imdb_rating > 8 THEN 'Fantastic'
  WHEN imdb_rating > 6 THEN 'Poorly␣Received'
  ELSE 'Avoid␣at␣All␣Costs'
 END AS 'Review'
FROM movies;
```

We will have an output with two columns, name and Review, where the name column is the same as that of our original table, with the corresponding Review column containing values based on the imdb_rating of each movie.

2.4.2 ORDER BY*

look into this.

3 Aggregate Functions

SQL Queries don't just access raw data, they can also perform calculations on the raw data to answer specific data questions. Calculations performed on multiple rows of a table are called aggregates.

Now we will be working with a table named fake_apps.

3.1 Common Functions

3.1.1 COUNT

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.

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

Here, we want to count every row, so we pass * as an argument inside the parenthesis.

3.1.2 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;
```

This adds all values in the downloads column.

3.1.3 MAX and MIN

The MAX() and MIN() functions return the highest and lowest values in a column, respectively.

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

3.1.4 AVG

SQL uses the AVG() function to quickly calculate the average value of a particular column.

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

3.1.5 ROUND

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. The 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.

Note that if we have found, say, an average of some column, this numerical value can be used as the first argument of ROUND(). This is because when we enter SELECT AVG(downloads) FROM fake_apps, we are actually returning a table with a column named AVG(downloads) and a singular row which is its value. So we would write:

```
SELECT ROUND(AVG(downloads), 2),
FROM fake_apps;
```

3.2 GROUP BY

Very often, we will want to calculate an aggregate for data with certain characteristics. For instance, we might want to know the mean number of downloads for each app category. We can use GROUP BY to do this in a single step.

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

GROUP BY is a clause in SQL that is used with aggregate functions. It is used in collaboration with the SELECT statement to arrange identical data into groups.

The GROUP BY statement comes after any WHERE statements, but before ORDER BY or LIMIT.

Sometimes, we want to GROUP BY a calculation done on a column. For instance, we might want to know how many movies have IMDb ratings that round to 1, 2, 3, 4, 5. We could do this using the following syntax:

```
SELECT ROUND(imdb_rating),
   COUNT(name)
FROM movies
GROUP BY 1
ORDER BY 1;
```

SQL lets us use column reference(s) in our GROUP BY that will make our lives easier. 1 is the first column selected, 2 is the second column selected, 3 is the third column selected, and so on.

3.2.1 HAVING

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 very similar to WHERE. In fact, all types of WHERE clauses covered thus far can be used with HAVING. We can use the following for the problem.

```
SELECT year,
   genre,
   COUNT(name)
FROM movies
GROUP BY 1, 2
HAVING COUNT(name) > 10;
```

4 Multiple Tables

In order to efficiently store data, we often spread related information across multiple tables. For example, imagine that we're running a magazine company where users can have different types of subscriptions to different products. Different subscriptions might have many different properties. Each customer would also have lots of associated information. We could have one table with all of the following information:

- order_id
- customer_id
- customer_name
- customer_address
- subscription_id
- subscription_description
- subscription_monthly_price
- subscription_length
- purchase_date

However, a lot of this information would be repeated. If the same customer has multiple subscriptions, that customer's name and address will be reported multiple times. If the same subscription type is ordered by multiple customers, then the subscription price and subscription description will be repeated. This will make our table big and unmanageable. So instead, we can split our data into three tables:

i **orders** would contain just the information necessary to describe what was ordered: (order_id, customer_id, subscription_id, purchase_date)

ii **subscriptions** would contain the information to describe each type of subscription: (subscription_id, description, price_per_month, subscription_length)

iii **customers** would contain the information for each customer: (customer_id, customer_name, address)

Here, we will cover SQL commands that will help us work with data that is stored in multiple tables.

4.1 Primary Key vs Foreign Key

The three tables: orders, subscriptions, and customers each have a column that uniquely identifies each row of that table: order_id for orders, subscription_id for subscriptions, and customer_id for customers. These special columns are called primary keys.

Primary keys have a few requirements:

- None of the values can be NULL.
- Each value must be unique (i.e., you can't have two customers with the same customer_id in the customers table).
- A table can not have more than one primary key column.

Note that customer_id (the primary key for customers) and subscription_id (the primary key for subscriptions) both appear in this. When the primary key for one table appears in a different table, it is called a foreign key. So customer_id is a primary key when it appears in customers, but a foreign key when it appears in orders.

In this example, our primary keys all had somewhat descriptive names. Generally, the primary key will just be called id. Foreign keys will have more descriptive names. Why is this important? The most common types of joins will be joining a foreign key from one table with the primary key from another table. For instance, when we join orders and customers, we join on customer_id, which is a foreign key in orders and the primary key in customers.

4.2 JOIN

By default, the type of join performed is an inner join. Additionally, unless we specify which columns we exactly want, the column names from the JOIN table will simply be appended on to the right of the FROM table.

```
SELECT *
FROM orders
JOIN customers
ON orders.customer_id = customers.customer_id;
```

We break down this command as follows:

1. The first line selects all columns from our combined table. If we only want to select certain columns, we can specify which ones we want (see below).
2. The second line specifies the first table that we want to look in, orders
3. The third line uses JOIN to say that we want to combine information from orders with customers.
4. The fourth line tells us how to combine the two tables. We want to match orders table's customer_id column with customers table's customer_id column.

Because column names are often repeated across multiple tables, we use the syntax table_name.column_name to be sure that our requests for columns are unambiguous. In our example, we use this 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: Instead of selecting all the columns using *, if we only wanted to select orders table's order_id column and customers table's customer_name column, we could use the following query:

```
SELECT orders.order_id,
   customers.customer_name
FROM orders
JOIN customers
  ON orders.customer_id = customers.customer_id;
```

4.2.1 CROSS JOIN

To do a left join, simply write LEFT JOIN instead of JOIN. Note the left table will be the one coming from the FROM command. Consider the example where we find all customers that either have the newspaper or both the newspaper and online paper. This is done by the left join below.

```
SELECT *
FROM newspaper
LEFT JOIN online
ON newspaper.id = online.id
WHERE online.id IS NULL
;
```

If we wanted to find out newspaper customers that did not have the online paper, then all the column names coming from online table that have been appended will be null. Thus we can find the them by searching for these null values.

4.2.2 CROSS JOIN

Cross joining is simply joining each row of one table with each row of the other. If the first table has n rows and the second has m, the therefore get $n * m$ rows. So we clearly will not need an ON statement.

While this same useless, it can be exploited to solve certain problems. Suppose we wanted to know how many users were subscribed during each month of the year. For each month (1, 2, 3) we would need to know if a user was subscribed. We do this by the following steps:

- We have a another table, months, where the only column, month, contains numbers 1 to 12
- We cross join newspaper with months

- We then only keep the rows of this cross join where newspaper.start_month <= months.month and newspaper.end_month >= months.month
- We then aggregrate by month the number of subscribers

In code:

```
SELECT month, COUNT(*)
FROM newspaper
CROSS JOIN months
WHERE (start_month <= month) and (end_month >= month)
GROUP BY month
```

4.2.3 UNION

To perform an outer join, we use the UNION command, which combines the results of two or more SELECT statements and returns a single result set. It automatically removes any duplicate rows, ensuring that each row is unique.

```
SELECT *
FROM newspaper
UNION
SELECT *
FROM online;
```

There are some rules for appending data in this way in SQL:

- Tables must have the same number of columns.
- The columns must have the same data types in the same order as the first table.

The columns do not necessarily have to have the same column names, or even represent the same exact thing. For example, if table 1 contained weight and age, and table 2 contained height and age, these would be appendable since the number of columns are the same and have the same data types (REAL then INTEGER) in the same order.

Note the result set will use the column names from the first SELECT statement.

4.2.4 WITH

Often times, we want to combine two tables, but one of the tables is the result of another calculation. For instance, they might want to know how many magazines each customer subscribes to. We do this in steps:

- First, we find the number of subscriptions a given customer has, using the orders table
- Using this new table, we then want to know the name corresponding to each customer_id. We thus do a JOIN with the customers table ON customer_id.

```
WITH previous_query AS (
  SELECT customer_id,
   COUNT(subscription_id) AS 'subscriptions'
  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
```