15. SQL Intro

Github repo

Data persistancy

When you reset your app and you lose all your data? That ends this week. You'll also be learning how to become a full stack developer. You dealt with the backend, frontend and today is going to be that database layer which you will learn how to use sql later with LightBNB This will make you a fullstack dev.

What are tables?

They consist of rows and columns Columns - represent descriptive elements, such as what type of information the next few rows of that column will have

Name Price Calories Big Mac $3.99 500 Fries $1.99 300

Rows = records Columns = fields

This will get transformed into an array of objects. But we'll get more into that next week. Today we are going to be strickly talking about sql

Relational Databases

Instead of a colleciton of tables, - We can have it so that all tables are related to each other in one way or another Its the reason why data science has been on the rise - all of our tables have structure

SQL vs NoSQL (Mongo) - Theres no schema, or design of the database. Every user will have a field for user name or database

We wont have something talk directly to it, we need a middle man This upcomming week we are going to have our express server communicate with RDBMS

RDBMS

- Relational Database Management System We are going to talk to it, and its going to give the information back to us

express <--postgres + tcp--> rdbms We talk with the RDBMS and the RDBMS talks with the database

Primary Keys

- In order to reference a particular record in a table, each one is given a unique identifier we call a **Primary Key**
- Other tables can then make reference to a particular record in another table by storing the Primary Keys value
- We call a Primary Key stored in another table a **Foreign Key**
- It is through this Primary Key/Foreign Key relationship that our tables are *related* to one another

SELECT

- The **SELECT** clause queries the database and returns records that match the query
- Always accompanied by the **FROM** keyword which indicates which table we'd like to query
- SELECT takes a list of field names as an argument
- Every SQL command ends in a semicolon (;), that's how we tell the application that we are finished entering our query

```
-- basic SELECT query
SELECT username, email FROM users;

-- the asterisk (*) can be used as a wildcard to return all fields in the
table
SELECT * FROM users;

-- it is customary to put each SQL clause or keyword on a separate line
for readability
SELECT username, email
FROM users;
```

Filtering and Ordering

- We use `WHERE` to filter our results
- If the record satisfies the `WHERE` criteria (eg. before a certain date, greater than a certain amount), it is included in the query results
- NOTE: using the `WHERE` clause can filter your records down to zero (ie. no records satisfy the filter criteria)

```sql
SELECT *
FROM table_one
-- return only records where date_due is before the current date
WHERE date_due < NOW();
```

- Order your results with the `ORDER BY` clause
- We specify the field that we want to sort by and the sort direction
- Sort direction is either ascending (`ASC`) or descending (`DESC`)
- NOTE: the default sort direction is ascending (`ASC`) so you don't need to specify it

```sql
SELECT *
FROM table_one
ORDER BY field_one;

-- or in descending order
ORDER BY field_one DESC;
```

`JOIN`

- We connect tables together using **JOIN**s
- The tables are joined together using the primary key and foreign key
- There are various types of joins:
  - `INNER JOIN` : The default. Return only records that have matching records in the other table
  - `LEFT JOIN` : Return all records from the "left" table and only those from the other table that match
  - `RIGHT JOIN` : The same as a *LEFT JOIN*, but from the *RIGHT* instead
  - `FULL OUTER JOIN` : Return all records from both tables

```sql
-- basic INNER JOIN
SELECT *
FROM table_one
INNER JOIN table_two
ON table_one.id = table_two.table_one_id;

-- since it is the default, you don't have to specify "INNER"
SELECT *
FROM table_one
JOIN table_two
ON table_one.id = table_two.table_one_id;
```

Grouping Records

- Records that contain the same values (eg. **students** with the same `cohort_id`) can be *grouped* together using the `GROUP BY` clause
- If the records contain any unique values, they will not be grouped together

```
SELECT cohort_id, COUNT(cohort_id) AS num_students
FROM students
GROUP BY cohort_id;
```

Aggregation Functions

- Aggregation functions give us meta data about our records (eg. count responses, average player score, get minimum value)
- Some aggregation functions:

Function	Purpose	Example Usage
`COUNT`	Return the number of records grouped together	`COUNT(*) AS num_users`
`SUM`	Add the values of the specified field together	`SUM(player_score) AS total_score`
`MIN`	Return the minimum value from the field	`MIN(player_score) AS lowest_score`
`MAX`	Return the maximum value	`MAX(player_score) AS high_score`
`AVG`	Return the average value	`AVG(player_score) AS average_score`

`LIMIT` and `OFFSET`

- We can limit the amount of records returned from a query using `LIMIT`
- `LIMIT` accepts an *integer* as an argument

```
SELECT *
FROM table_one
-- only return 50 records
LIMIT 50;
```

- NOTE: `LIMIT` runs **after** `ORDER BY` (ie. sort your records then specify how many to return)

```
SELECT *
FROM table_one
-- order by a field(s)
ORDER BY field_name DESC
-- return the top 10
LIMIT 10;
```

- We can skip any number of records using `OFFSET`
- Like `LIMIT`, `OFFSET` accepts an *integer* as an argument

```
SELECT *
FROM table_one
-- skip the first 10 records
OFFSET 10;
```

- `OFFSET` and `LIMIT` work hand-in-hand to create pagination

```
SELECT *
FROM table_one
-- skip the first 20 records, return the next 10
LIMIT 10 OFFSET 20;

-- you can specify these in any order
OFFSET 20 LIMIT 10;
```

SELECT Challenges

For the first 6 queries, we'll be using the `users` table.

users	
PK	id
	first_name
	last_name
	email
	age
	country
	payment_due_date

1. List total number of users

```
SELECT COUNT(*) AS num_users
FROM users;
```

1. List users over the age of 18

```
SELECT *
FROM users
WHERE age > 18;
```

1. List users who are over the age of 18 and have the last name 'Barrows'

```
SELECT *
FROM users
WHERE age > 18 AND last_name = 'Barrows';
```

1. List users over the age of 18 who live in Canada sorted by age from oldest to youngest and then last name alphabetically

```
SELECT *
FROM users
WHERE age > 18 AND country = 'Canada'
ORDER BY age DESC, last_name;
```

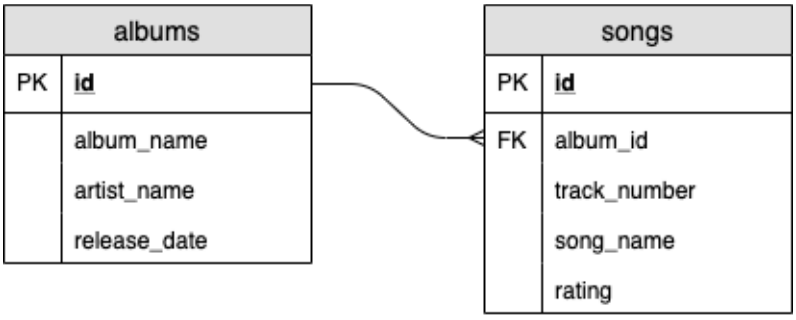
1. List users who live in Canada and whose accounts are overdue

```
SELECT *
FROM users
WHERE country = 'Canada' AND payment_due_date < '01/10/23';

-- using the NOW function instead
SELECT *, NOW() -- check out these comments
-- adding the FROM clause
FROM users
WHERE country = 'Canada' AND payment_due_date < NOW();

-- using the CURRENT_DATE constant
SELECT *, CURRENT_DATE
FROM users
WHERE country = 'Canada' AND payment_due_date < CURRENT_DATE;
```

1. List all the countries users live in; don't repeat any countries

```sql
SELECT DISTINCT country
FROM users
ORDER BY country;
```

For the rest of the queries, we'll be using the `albums` and `songs` tables.



1. List all albums along with their songs

```sql
SELECT *
FROM songs
JOIN albums ON albums.id = album_id;
```

1. List all albums along with how many songs each album has

```sql
SELECT albums.*, COUNT(songs.id) AS num_songs
FROM albums
JOIN songs ON albums.id = album_id
GROUP BY albums.id;
```

1. Enhance previous query to only include albums that have more than 10 songs

```sql
SELECT albums.*, COUNT(songs.id) AS num_songs
FROM albums
JOIN songs ON albums.id = album_id
GROUP BY albums.id
HAVING COUNT(songs.id) > 10;
```

1. List ALL albums in the database, along with their songs if any

```sql
SELECT *
FROM albums
LEFT JOIN songs ON albums.id = album_id;
```

1. List albums along with average song rating

```
SELECT albums.*, ROUND(AVG(rating) * 1000) / 1000 AS avg_rating
FROM albums
JOIN songs ON albums.id = album_id
GROUP BY albums.id
ORDER BY avg_rating DESC;
```

1. List albums and songs with rating higher than album average

```
SELECT AVG(rating) FROM songs WHERE album_id = 1;

SELECT albums.*, songs.*
FROM albums
JOIN songs ON albums.id = songs.album_id
WHERE rating > 3.54;

SELECT albums.*, songs.*
FROM albums
JOIN songs ON albums.id = songs.album_id
WHERE rating > (SELECT AVG(rating) FROM songs WHERE album_id =
albums.id);
```

Useful Links

- Top 10 Most Popular RDBMSs
- Another Ranking of DBMSs
- SELECT Queries Order of Execution
- SQL Joins Visualizer
- Datatypes
- Common psql commands

View on Compass | Leave Feedback