

Manipulation

- `CREATE TABLE` creates a new table.
- `INSERT INTO` adds a new row to a table.
- `SELECT` queries data from a table.
- `ALTER TABLE` changes an existing table.
- `UPDATE` edits a row in a table.
- `DELETE FROM` deletes rows from a table.

```
SELECT * FROM celebs;
```

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

```
INSERT INTO celebs (id, name, age)  
VALUES (1, 'Justin Bieber', 22);
```

```
INSERT INTO celebs (id, name, age)  
VALUES (2, 'Beyonce Knowles', 33);
```

```
INSERT INTO celebs (id, name, age)  
VALUES (3, 'Jeremy Lin', 26);
```

```
INSERT INTO celebs (id, name, age)  
VALUES (4, 'Taylor Swift', 26);
```

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

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

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

```
CREATE TABLE awards (  
    id INTEGER PRIMARY KEY,  
    recipient TEXT NOT NULL,  
    award_name TEXT DEFAULT 'Grammy'  
);
```

Queries

- `SELECT` is the clause we use every time we want to query information from a database.
- `AS` renames a column or table.
- `DISTINCT` return unique values.
- `WHERE` is a popular command that lets you filter the results of the query based on conditions that you specify.
- `LIKE` and `BETWEEN` are special operators.
- `AND` and `OR` combines multiple conditions.
- `ORDER BY` sorts the result.
- `LIMIT` specifies the maximum number of rows that the query will return.
- `CASE` creates different outputs.

```
SELECT name, genre, year
FROM movies;
```

```
SELECT imdb_rating AS 'IMDb'
FROM movies;
```

```
SELECT DISTINCT year
FROM movies;
```

```
SELECT *
FROM movies
WHERE year > 2014;
```

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

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

```
SELECT name
FROM movies
WHERE imdb_rating IS NULL;
```

```
SELECT *
FROM movies
WHERE year BETWEEN 1970 AND 1979;
```

```
SELECT *
FROM movies
WHERE year < 1985
AND genre = 'horror';
```

```
SELECT *
FROM movies
WHERE genre = 'romance'
      OR genre = 'comedy';
```

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

```
SELECT name, year, imdb_rating
FROM movies
ORDER BY imdb_rating DESC;
```

Combining your knowledge of `LIMIT` and `ORDER BY`, write a query that returns the top 3 highest rated movies. Select all the columns.

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

Select the name column and use a CASE statement to create the second column that is:

'Chill' if genre = 'romance'

'Chill' if genre = 'comedy'

'Intense' in all other cases

Optional: Rename the whole CASE statement to 'Mood' using AS.

```
SELECT name,
CASE
  WHEN genre = 'romance' THEN 'Chill'
  WHEN genre = 'comedy' THEN 'Chill'
  ELSE 'Intense'
END AS 'Mood'
FROM movies;
```

Aggregate Functions

- `COUNT()`: count the number of rows
- `SUM()`: the sum of the values in a column
- `MAX()/MIN()`: the largest/smallest value
- `AVG()`: the average of the values in a column
- `ROUND()`: round the values in the column
- `GROUP BY` is a clause used with aggregate functions to combine data from one or more columns.
- `HAVING` limit the results of a query based on an aggregate property.

```
SELECT *  
FROM fake_apps;
```

```
SELECT COUNT(*)  
FROM fake_apps  
WHERE price = 0.0;
```

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

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

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

```
SELECT name, ROUND (AVG(price), 2)  
FROM fake_apps;
```

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

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

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

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

```
SELECT price, ROUND(AVG(downloads)), COUNT(*)
FROM fake_apps
GROUP BY price
HAVING COUNT(*) > 10;
```

Multiple Tables

- **JOIN** will combine rows from different tables if the join condition is true.
- **LEFT JOIN** will return every row in the *left* table, and if the join condition is not met, **NULL** values are used to fill in the columns from the *right* table.
- *Primary key* is a column that serves a unique identifier for the rows in the table.
- *Foreign key* is a column that contains the primary key to another table.
- **CROSS JOIN** lets us combine all rows of one table with all rows of another table.
- **UNION** stacks one dataset on top of another.
- **WITH** allows us to define one or more temporary tables that can be used in the final query.

```
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 description = 'Fashion Magazine';
```

```
SELECT COUNT(*)
FROM newspaper;

SELECT COUNT(*)
FROM online;
```

```
SELECT COUNT(*)  
FROM newspaper  
JOIN online  
ON newspaper.id = online.id;
```

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

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

```
SELECT *  
FROM classes  
JOIN students  
ON classes.id = students.class_id;
```

```
SELECT COUNT(*)  
FROM newspaper  
WHERE start_month <= 3  
AND end_month >= 3;
```

```
SELECT *  
FROM newspaper  
CROSS JOIN months;
```

```
SELECT *  
FROM newspaper  
CROSS JOIN months  
WHERE start_month <= month  
AND end_month >= month;
```

```
SELECT month, COUNT(*) AS 'subscribers'  
FROM newspaper  
CROSS JOIN months  
WHERE start_month <= month  
AND end_month >= month  
GROUP BY month;
```

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

```
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;
```