

# Manipulation Key Concepts

## Column Constraints

```
CREATE TABLE student (  
id INTEGER PRIMARY KEY,  
name TEXT UNIQUE,  
grade INTEGER NOT NULL,  
age INTEGER DEFAULT 10  
);
```

Column constraints are the rules applied to the values of individual columns:

- **PRIMARY KEY** constraint can be used to uniquely identify the row.
- **UNIQUE** columns have a different value for every row.
- **NOT NULL** columns must have a value.
- **DEFAULT** assigns a default value for the column when no value is specified.

There can be only one **PRIMARY KEY** column per table and multiple **UNIQUE** columns.

## CREATE TABLE Statement

```
CREATE TABLE table_name (  
column1 datatype,  
column2 datatype,  
column3 datatype  
);
```

The **CREATE TABLE** statement is used to create a new table in a database. It allows one to specify the name of the table and the name of each column in the table.

## INSERT Statement

```
-- Insert into columns in order:  
INSERT INTO table_name  
VALUES (value1, value2, value3);  
  
-- Insert into columns by name:  
INSERT INTO table_name (column1, column2, column3)  
VALUES (value1, value2, value3);
```

The **INSERT INTO** statement is used to add a new record (row) to a table. It has two forms as shown in the code block:

- Insert values based on the order of the columns in the table.
- Define the columns to insert values into.

## ALTER TABLE Statement

```
-- Syntax:  
ALTER TABLE table_name  
ADD column_name datatype;  
  
-- Example:  
ALTER TABLE employees  
ADD first_name TEXT;
```

The `ALTER TABLE` statement is used to modify the columns of an existing table. When combined with the `ADD COLUMN` clause, it is used to add a new column to a table.

## DELETE Statement

```
DELETE FROM table_name  
WHERE some_column = some_value;
```

The `DELETE` statement is used to delete records (rows) in a table. This statement does not delete the whole table.

Inside, the `WHERE` clause specifies which record or records that should be deleted. If the `WHERE` clause is omitted, all records will be deleted.

## UPDATE Statement

```
UPDATE table_name  
SET column1 = value1, column2 = value2  
WHERE some_column = some_value;
```

The `UPDATE` statement is used to edit records (rows) in a table. It usually includes a `SET` clause that indicates the column to edit and a `WHERE` clause for specifying which record(s) should be updated.

# Queries Key Concepts

## AND Operator

```
SELECT model  
FROM cars  
WHERE color = 'blue'  
AND year > 2014;
```

The **AND** operator allows multiple conditions to be combined. Records must match both conditions that are joined by **AND** to be included in the result set. The example query will match any car that is blue and made after 2014.

## **AS Clause**

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

Columns or tables in SQL can be *aliased* using the **AS** clause. This allows columns or tables to be specifically renamed in the returned result set. The given query will return a result set with the column for **name** renamed to **movie\_title**.

## **OR Operator**

```
SELECT name
FROM customers
WHERE state = "ca"
OR state = "ny";
```

The **OR** operator allows multiple conditions to be combined. Records matching either condition joined by the **OR** are included in the result set. The given query will match customers whose state is either **ca** or **ny**.

## **% Wildcard**

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

The **%** wildcard can be used in a **LIKE** operator pattern to match zero or more unspecified character(s). The example query will match any movie that begins with **The**, followed by zero or more of any characters.

## **SELECT Statement**

```
SELECT *
FROM movies;
```

The **SELECT \*** statement returns all columns from the provided table in the result set. The given query will fetch all columns and records (rows) from the **movies** table.

## Wildcard

```
SELECT name
FROM movies
WHERE name
LIKE '_ove';
```

The `_` wildcard can be used in a `LIKE` operator pattern to match any single unspecified character. The given query will match any movie which begins with a single character, followed by `ove`.

## ORDER BY Clause

```
SELECT *
FROM contacts
ORDER BY birth_date DESC;
```

The `ORDER BY` clause can be used to sort the result set by a particular column either alphabetically or numerically. It can be ordered in ascending (default) or descending order with `ASC/DESC`. In the example, all the rows of the `contacts` table will be ordered by the `birth_date` column in descending order.

## LIKE Operator

```
SELECT name
FROM movies
WHERE name
LIKE 'Star%';
```

The `LIKE` operator can be used inside of a `WHERE` clause to match a specified pattern. The given query will match any movie that begins with `Star` in its title.

## DISTINCT Clause

```
SELECT DISTINCT city
FROM contact_details;
```

Unique values of a column can be selected using a `DISTINCT` query. For a table `contact_details` having five rows in which the `city` column contains Chicago, Madison, Boston, Madison, and Denver, the given query would return:

- Chicago
- Madison
- Boston
- Denver

## BETWEEN Operator

```
SELECT *  
FROM movies  
WHERE year  
BETWEEN 1980 AND 1990;
```

The **BETWEEN** operator can be used to filter by a *range* of values. The range of values can be text, numbers or date data. The given query will match any movie made between the years 1980 and 1990, inclusive.

## LIMIT Clause

```
SELECT *  
FROM movies  
LIMIT 5;
```

The **LIMIT** clause is used to narrow, or *limit*, a result set to the specified number of rows. The given query will limit the result set to 5 rows.

## NULL Values

```
SELECT address  
FROM records  
WHERE address IS NOT NULL;
```

Column values in SQL records can be **NULL**, or have no value. These records can be matched (or not matched) using the **IS NULL** and **IS NOT NULL** operators in combination with the **WHERE** clause. The given query will match all addresses where the address has a value or is not **NULL**.

## WHERE Clause

```
SELECT title  
FROM library  
WHERE pub_year = 2017;
```

The **WHERE** clause is used to filter records (rows) that match a certain condition. The given query will select all records where the **pub\_year** equals **2017**.

# Aggregate Functions Key Concepts

## Column References

```
SELECT COUNT(*)  
AS 'total_movies', rating  
FROM movies  
GROUP BY 2  
ORDER BY 1;
```

The `GROUP BY` and `ORDER BY` clauses can reference the selected columns by number in which they appear in the `SELECT` statement. The example query will count the number of movies per rating, and will:

- `GROUP BY` column 2 (`rating`)
- `ORDER BY` column 1 (`total_movies`)

## `SUM()` Aggregate Function

```
SELECT SUM(salary)  
FROM salary_disbursement;
```

The `SUM()` aggregate function takes the name of a column as an argument and returns the sum of all the value in that column.

## `MAX()` Aggregate Function

```
SELECT MAX(amount)  
FROM transactions;
```

The `MAX()` aggregate function in SQL takes the name of a column as an argument and returns the largest value in a column. The given query will return the largest value from the `amount` column.

## `COUNT()` Aggregate Function

```
SELECT COUNT(*)  
FROM employees  
WHERE experience < 5;
```

The `COUNT()` aggregate function in SQL returns the total number of rows that match the specified criteria. For instance, to find the total number of employees who have less than 5 years of experience, the given query can be used.

**Note:** A column name of the table can also be used instead of \*. Unlike COUNT(\*), this variation COUNT(column) will *not* count NULL values in that column.

## GROUP BY Clause

```
SELECT rating, COUNT(*)  
FROM movies  
GROUP BY rating;
```

The GROUP BY clause will group records in a result set by identical values in one or more columns. It is often used in combination with aggregate functions to query information of similar records. The GROUP BY clause can come after FROM or WHERE but must come before any ORDER BY or LIMIT clause.

The given query will count the number of movies per rating.

## MIN() Aggregate Function

```
SELECT MIN(amount)  
FROM transactions;
```

The MIN() aggregate function in SQL returns the smallest value in a column. For instance, to find the smallest value of the amount column from the table named transactions, the given query can be used.

## AVG() Aggregate Function

```
SELECT AVG(salary)  
FROM employees  
WHERE experience < 5;
```

The AVG() aggregate function returns the average value in a column. For instance, to find the average salary for the employees who have less than 5 years of experience, the given query can be used.

## HAVING Clause

```
SELECT year, COUNT(*)  
FROM movies  
GROUP BY year  
HAVING COUNT(*) > 5;
```

The HAVING clause is used to further filter the result set groups provided by the GROUP BY clause. HAVING is often used with aggregate functions to filter the result set groups based on an aggregate property. The

given query will select only the records (rows) from only years where more than 5 movies were released per year.

### ROUND() Function

```
SELECT year, ROUND(AVG(rating), 2)
FROM movies
WHERE year = 2015;
```

The `ROUND()` function will round a number value to a specified number of places. It takes two arguments: a number, and a number of decimal places. It can be combined with other aggregate functions, as shown in the given query. This query will calculate the average rating of movies from 2015, rounding to 2 decimal places.

## Aggregate Functions in SQL

Aggregate functions perform a calculation on a set of values and return a single value:

- `COUNT()`
- `SUM()`
- `MAX()`
- `MIN()`
- `AVG()`

# Multiple Tables Key Concepts

## Outer Join

```
SELECT column_name(s)
FROM table1
LEFT JOIN table2
ON table1.column_name = table2.column_name;
```

An outer join will combine rows from different tables even if the join condition is not met. In a `LEFT JOIN`, 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.

### WITH Clause

```
WITH temporary_movies AS ( SELECT * FROM movies )
SELECT *
FROM temporary_movies
WHERE year BETWEEN 2000 AND 2020;
```



The **WITH** clause stores the result of a query in a temporary table (**temporary\_movies**) using an alias. Multiple temporary tables can be defined with one instance of the **WITH** keyword.

## **UNION Clause**

```
SELECT name
FROM first_names
UNION
SELECT name
FROM last_names
```

The **UNION** clause is used to combine results that appear from multiple **SELECT** statements and filter duplicates.

For example, given a **first\_names** table with a column **name** containing rows of data "James" and "Hermione", and a **last\_names** table with a column **name** containing rows of data "James", "Hermione" and "Cassidy", the result of this query would contain three **names**: "Cassidy", "James", and "Hermione".

## **CROSS JOIN Clause**

```
SELECT shirts.shirt_color, pants.pants_color
FROM shirts
CROSS JOIN pants;
```

The **CROSS JOIN** clause is used to combine each row from one table with each row from another in the result set. This **JOIN** is helpful for creating all possible combinations for the records (rows) in two tables.

The given query will select the **shirt\_color** and **pants\_color** columns from the result set, which will contain all combinations of combining the rows in the **shirts** and **pants** tables. If there are 3 different shirt colors in the **shirts** table and 5 different pants colors in the **pants** table then the result set will contain  $3 \times 5 = 15$  rows.

## **Foreign Key in SQL**

A *foreign key* is a reference in one table's records to the *primary key* from another table. To maintain multiple records for a specific row, the use of *foreign key* plays a vital role. For instance, to track all the orders of a specific customer, the table **order** (illustrated at the bottom of the image) can contain a *foreign key*.

## **Primary Key in SQL**

A *primary key* column in a SQL table is used to uniquely identify each record in that table. A primary key cannot be **NULL**. In the example, **customer\_id** is the primary key. The same value cannot re-occur in a *primary key* column. Primary keys are often used in **JOIN** operations.

## Inner Join

```
SELECT *  
FROM books  
JOIN authors  
ON books.author_id = authors.id;
```

The `JOIN` clause allows for the return of results from more than one table by joining them together with other results based on common column values specified using an `ON` clause. `INNER JOIN` is the default `JOIN` and it will only return results matching the condition specified by `ON`.