SQL

[Summary]

Souvik Ghosh

Introduction to Database

It is important to understand what a database is before we start learning SQL. After all, SQL is used to communicate with a database.

What is a database?

A database—Database Management System (DBMS)—is a collection of data organized and stored in a structured format.

And there are two common types of databases:

1. Non-Relational
2. Relational (where SQL is used)

Let's explore each of these databases in brief.

Non-Relational Database

In a non-relational database, records are stored in key-value pairs. For example,

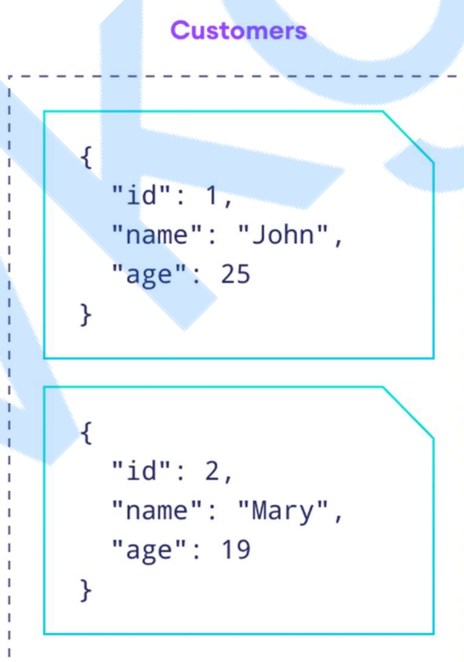

Figure: Non-Relational DBMS

Here, `Customers` is a container inside a non-relational database. The database may contain many containers like `Customers`.

Inside the `Customers` container, we have stored information of two customers.

For Customer1

Key	Value
id	1
name	John
age	25

For Customer2

Key	Value
id	2
name	Mary
age	19

Next, we will learn about relational databases.

Relational Database

In a relational database, data is stored in a tabular format. For example,

Customers

id	name	age
1	John	25
2	Mary	19

Figure: Relational Database

Here, `Customers` is a table inside a relational database. The database may contain other tables like `Customers`.

The first row of the table is fields or attributes.

Relation in Relational Databases

In a relational database, two or more tables may be related based on a common field. Hence, the term "relational". For example,

Customers

id	name	age
1	John	25
2	Mary	19

Products

customer_id	name
2	honey
1	milk
1	eggs
2	milk

Figure: Relation Between Tables

Here, `Customers` and `Products` are related through `id` (field of Customers) and `customer_id` (field of Products).

If we look at the `Customers` table, we know that

- customer with `id` 1 is `John`
- customer with `id` 2 is `Mary`

And if we look at the `Products` table, we know that

- customer with `id` 1 (`John`) bought `milk` and `eggs`
- customer with `id` 2 (`Mary`) bought `honey` and `milk`

Role of SQL

SQL is used to interact with and manage data stored in relational databases. This includes:

- retrieving data from a database table
- inserting data in a database table
- updating data in a database table
- deleting data in a database table
- creating new tables in a database
- and many more

Now that we know about relational databases, we are ready to learn SQL.

Select Records

SQL SELECT

The `SELECT` statement is used to retrieve data from a table.

Example 1

```
-- select all the employees with all the columns
SELECT *
FROM Employees;
```

Example 2

```
-- select all the employees with first_name,
-- last_name and department columns
SELECT first_name, last_name, department
FROM Employees;
```

The WHERE Clause

The `WHERE` clause in a `SELECT` statement allows us to select rows that meet the specified conditions.

Example 1

```
-- select employees whose age is 25
SELECT *
FROM Employees
WHERE age = 25;
```

Example 2

```
-- select employees whose
-- department is 'Finance'
SELECT *
FROM Employees
WHERE department = 'Finance';
```

Remember, textual data (strings) such as `'Finance'` should be enclosed inside quotation marks.

Example 3

```sql
-- select employees whose
-- age is greater than 25
SELECT *
FROM Employees
WHERE age > 25;
```

Example 4

```sql
-- select employees whose
-- age is greater than or equal to 25
SELECT *
FROM Employees
WHERE age >= 25;
```

Example 5

```sql
-- select employees whose
-- age is less than 25
SELECT *
FROM Employees
WHERE age < 25;
```

Example 6

```sql
-- select employees whose
-- age is less than or equal to 25
SELECT *
FROM Employees
WHERE age <= 25;
```

Example 7

```sql
-- select employees
whose
-- age is not equal to
25
SELECT *
FROM Employees
WHERE age <> 25;
```

AND, OR and NOT Operators

Example 1

The `AND` operator selects a row if all conditions separated by `AND` are TRUE.

```
-- select employees whose
-- age is greater than 23
-- and department is 'Sales'
SELECT *
FROM Employees
WHERE age > 23 AND department = 'Sales';
```

Example 2

The `OR` operator selects a row if any of the conditions separated by `OR` is TRUE.

```
-- select employees whose
-- age is greater than 26
-- or department is 'Sales'
SELECT *
FROM Employees
WHERE age > 26 OR department = 'Sales';
```

Example 3

The `NOT` operator selects a row if the condition is FALSE.

```
-- select employees whose
-- department is not 'Sales'
SELECT *
FROM Employees
WHERE NOT department = 'Sales';
```

DISTINCT Clause

The `DISTINCT` clause in SQL returns only the distinct (unique) values.

```
-- select distinct departments
SELECT DISTINCT department
FROM Employees;
```

IN and BETWEEN Operators

Example 1

The `IN` operator is used with the `WHERE` clause to match values in a list.

```
-- select rows if department is
-- either Sales or Operations
SELECT *
FROM Employees
WHERE department IN ('Sales', 'Operations');
```

The above code is equivalent to

```
SELECT *
FROM Employees
WHERE department = 'Sales' OR department = 'Operations';
```

Example 2

The `BETWEEN` operator is used with the `WHERE` clause to select values within a range. For example,

```
-- select rows if age is
-- between 25 and 27
SELECT *
FROM Employees
WHERE age BETWEEN 25 AND 27;
```

The `BETWEEN` operator is inclusive. In the above query, it selects ages 25 and 27 as well.

ORDER BY

The `ORDER BY` clause is used to sort the results in either ascending or descending order.

Example 1

```
-- order rows in ascending order by age
SELECT *
FROM Employees
ORDER BY age ASC;
```

Example 2

The `ORDER BY` clause sorts rows in ascending order by default. It is not necessary to explicitly use the `ASC` keyword.

```
-- order rows in ascending order by department
SELECT *
FROM Employees
ORDER BY department;
```

Example 3

We use the `DESC` keyword to sort the output in descending order.

```
-- order rows in descending order by age
SELECT *
FROM Employees
ORDER BY age DESC;
```

The LIMIT Clause

The `LIMIT` clause is used in a `SELECT` statement to specify the number of rows to return. For example,

```
-- select the first three rows
SELECT *
FROM Products
LIMIT 3;
```

Aggregate Functions

MIN() and MAX()

The `MIN()` function returns the minimum value in a column.

```
-- select the minimum age from the age column
SELECT MIN(age)
FROM Employees;
```

The `MAX()` function returns the maximum value in a column.

```
-- select the maximum age from the age column
SELECT MAX(age)
FROM Employees;
```

In the case of strings, `MIN()` and `MAX()` return data based on alphabetical order (dictionary order).

```
-- select the first_name of the employee
-- that comes last alphabetically
SELECT MAX(first_name)
FROM Employees;
```

COUNT()

The `COUNT()` function returns the count of rows that meet a certain criteria.

```
-- count the number of employees
SELECT COUNT(*)
FROM Employees;
```

If we count the number of rows based on a column, it ignores the count of all the NULL (empty) values.

```
-- count the number of employees
-- based on the last_name column
SELECT COUNT(last_name)
FROM Employees;
```

The `COUNT()` function can also be used to find the number of distinct values in a column.

```
-- select the count of distinct countries
SELECT COUNT(DISTINCT country)
FROM Customers;
```

SUM() and AVG()

The `SUM()` function is used to calculate the total sum of a numeric column.

```
-- return the total sum of salaries of all the employees
SELECT SUM(salary)
FROM Employees;
```

The SQL `AVG()` function is used to calculate the average of numeric values in a column.

```
-- return the average age of customers
SELECT AVG(age)
FROM Customers;
```

AS Keyword

The `AS` keyword is used to give a temporary name to columns in the output.

```
-- select the minimum age among employees who work in the Finance department
-- the column name in the output will be finance_min_age
SELECT MIN(age) AS finance_min_age
FROM Employees
WHERE department = 'Finance';
```

More on Aggregate Functions

Aggregate functions, such as `MIN()`, `MAX()`, `COUNT()`, `SUM()` and `AVG()`, are useful for solving real-world problems. We recommend that you become comfortable using these functions before continuing the course.

These functions are also commonly used in the `GROUP BY` clause and subqueries, which we will cover next.

Filter Records

GROUP BY

The `GROUP BY` clause is used to group rows based on values in a column. For example,

```
-- group average salary by departments
SELECT department, AVG(salary) AS average_salary
FROM Employees
GROUP BY department;
```

Here, we have grouped the output by the `department` column and computed the average salary of each department.

> 💡
> Note: The `GROUP BY` column is almost always used in conjunction with aggregate functions such as `COUNT()`, `AVG()`, `MIN()`, etc.

GROUP BY Multiple Columns

The `GROUP BY` clause can also group rows by multiple columns.

```
-- group data by customer_name
-- and then by product
SELECT customer_name, product, SUM(price * quantity) AS total_amount
FROM Sales
GROUP BY customer_name, product;
```

Sorting GROUP BY Result

Once we group rows using the `GROUP BY` clause, it is possible to order them in ascending or descending order using the `ORDER BY` clause.

```
-- order the results by different departments
-- get the average salary of each department
-- order the results from highest average salary to
lowest
SELECT department, AVG(salary)
FROM Employees
GROUP BY department
ORDER BY AVG(salary) DESC;
```

HAVING

The `HAVING` clause is used to filter the results of a `SELECT` statement after the `GROUP BY` clause has been applied.

```
-- group employees by departments
-- display records if the average salary of a department is more than 4000
SELECT department, AVG(salary) AS average_salary
FROM Employees
GROUP BY department
HAVING AVG(salary) > 4000;
```

LIKE and NOT LIKE

The `LIKE` operator is used to select rows that match a specified pattern. For example,

```
-- select customers whose countries start with the letter 'U'
SELECT *
FROM Customers
WHERE country LIKE 'U%';
```

Here, `'U%'` indicates any string that begins with `'U'`. The `%` wildcard indicates that `'U'` can be followed by zero or more characters.

NOT LIKE

The `NOT LIKE` operator is the opposite of the `LIKE` operator. The `NOT LIKE` operator is used to select rows that don't match a specified pattern in a column. For example,

```
-- select customers whose countries don't end with the letter 'A'
SELECT *
FROM Customers
WHERE country NOT LIKE '%A';
```

Wildcards

A wildcard character is used to represent one or more characters. There are two commonly used wildcard characters.

- `_` - represents a single character
- `%` - represents zero or more characters

Wildcard characters are commonly used with the `LIKE` and `NOT LIKE` operators:

```sql
-- select customers whose country names contain only three letters
SELECT *
FROM Customers
WHERE country LIKE '___';
```

CASE

In SQL, the `CASE` statement allows us to perform different actions based on different conditions.

For example, the SQL query below creates a new column named `age_group` in the output. The contents of this new column will be:

- `'18-19'` if `age` is between 18 and 19 (both inclusive)
- `'20-25'` if `age` is between 20 and 25 (both inclusive)
- `'26-30'` if `age` is between 26 and 30 (both inclusive)
- `'other'` if `age` is less than 18 or more than 30

```sql
SELECT *,
CASE
WHEN age >= 18 AND age <= 19 THEN '18-19'
WHEN age >= 20 AND age <= 25 THEN '20-25'
WHEN age >= 26 AND age <= 30 THEN '26-30'
ELSE 'other'
END AS age_group
FROM Customers;
```

JOINs

SQL JOINs are used to combine and select data from multiple tables based on a common column.

There are four commonly used JOIN statements:

- INNER JOIN
- LEFT JOIN
- RIGHT JOIN
- FULL JOIN

INNER JOIN

An INNER JOIN returns only the common rows between two or more tables.

```
-- joins the Customers and Orders tables on the customer_id column
-- selects rows if the customer_id column of the Customers and Orders tables match
SELECT Customers.name, Customers.city, Orders.product
FROM Customers
INNER JOIN Orders
ON Customers.customer_id = Orders.customer_id;
```

We can use aliases in an `INNER JOIN` to make our SQL code more readable.

```
SELECT c.name, c.city, o.product
FROM Customers AS c
INNER JOIN Orders AS o
ON c.customer_id = o.customer_id;
```

We can use the `JOIN` or `INNER JOIN` keyword to join more than two tables.

```
-- join Customers and Orders on the customer_id column
-- join Products and Orders on the product_id column
-- then select the name of the customer, the product they bought, and the price of the product
SELECT c.name, p.product, p.amount
FROM Customers c
JOIN Orders o
ON o.customer_id = c.customer_id
JOIN Products p
ON o.product_id = p.product_id;
```

LEFT JOIN

A `LEFT JOIN` joins two or more tables and selects all the rows from the left table and any matching rows from the right table.

```
-- left join the Customers and Orders tables
-- and select rows from the Customers table and common rows from the Orders table
SELECT *
FROM Customers c
LEFT JOIN Orders o
ON c.customer_id = o.customer_id;
```

RIGHT JOIN

A `RIGHT JOIN` joins two or more tables and selects all the rows from the right table and any matching rows from the left table.

```
-- right join the Customers and Orders tables
-- select all the rows from the Orders table and common rows from the
Customers table

SELECT *
FROM Customers c
RIGHT JOIN Orders o
ON c.customer_id = o.customer_id;
```

FULL JOIN

A `FULL JOIN` combines the results of both a `LEFT JOIN` and a `RIGHT JOIN`. It returns all rows from both tables, whether there is a match in the other table or not.

```
-- full join the math_grades and history_grades tables
-- and select the student_name, math_grade and history_grade

SELECT m.student_name, m.math_grade, h.student_name, h.history_grade
FROM math_grades m
FULL JOIN history_grades h
ON m.student_id = h.student_id;
```

Subquery

In SQL, a `SELECT` statement may contain another SQL statement, known as a subquery or nested query. For example,

```
-- select all the customers with the lowest age
SELECT *
FROM Customers
WHERE age = (
SELECT MIN(age)
FROM Customers
);
```

EXISTS

The `EXISTS` operator returns `TRUE` if the subquery returns one or more rows, and `FALSE` if the subquery does not return any rows.

If the result of the subquery is `TRUE`, the output of the outer query is displayed. However, if the result of the subquery is `FALSE`, the output of the outer query is omitted.

```
-- inner query returns a row if an employee is associated with the name column in the Departments table
-- if the inner query returns one or more rows, the EXISTS clause returns TRUE
-- the outer query returns the name of departments if the EXISTS clause returns TRUE
SELECT Departments.name
FROM Departments
WHERE EXISTS (
SELECT *
FROM Employees
WHERE Employees.department_name = Departments.name
);
```

Insert, Update & Delete

INSERT INTO

The `INSERT INTO` statement inserts new rows into a table.

```
-- insert a row
INSERT INTO
Students (id, name, age)
VALUES
(8, 'Jules', 22);
```

UPDATE

The `UPDATE` statement is used to update existing data in a database table. For example,

```
-- update the first_name column of all customers to Johnny
UPDATE Customers
SET first_name = 'Johnny';
```

Note: If we omit the `WHERE` clause, all the rows in the table will be updated. Therefore, it is important to use the `UPDATE` statement carefully to avoid accidental data loss.

DELETE FROM

The `DELETE FROM` statement is used to delete rows from a database table. For example,

```
-- delete rows if the customer's country is 'UK'
DELETE FROM Customers
WHERE country = 'UK';
```

Working With Tables

Data Types

The data type of a column specifies what type of value a column can hold. For example,

Data Type	Description
INTEGER	For storing integer values.
INT	For storing integer values similar to INTEGER.
VARCHAR(size)	For storing text (string) data. The size specifies the maximum number of characters that can be stored.
TEXT	For storing large pieces of text (string) data.
DATE	For storing data in the YYYY-MM-DD format.

CREATE TABLE

The CREATE TABLE statement is used to create a new table in the database. For example,

```
CREATE TABLE Companies (
id int,
name varchar(100),
email varchar(50),
phone varchar(10)
);
```

ALTER TABLE

The ALTER TABLE statement is used to modify an existing table. The ALTER TABLE can be used to

- rename a column
- add a new column
- modify a column
- delete a column
- rename a table

Rename a Column

```sql
ALTER TABLE Products
RENAME COLUMN price TO amount;
```

Add a New Column

```sql
ALTER TABLE Products
ADD COLUMN date_added DATE;
```

Modify a Column

```sql
-- change a column's data type
-- SQLite doesn't support modifying column
ALTER TABLE Products
MODIFY COLUMN name TEXT;
```

Delete Column

```sql
ALTER TABLE Products
DROP COLUMN quantity;
```

Rename Table

```sql
ALTER TABLE Products
RENAME TO ComputerAccessories;
```

Note: The ALTER TABLE syntax may differ among different database systems.

DROP TABLE

The DROP TABLE statement is used to delete tables from our database. For example,

```sql
DROP TABLE Customers;
```

Constraints

Constraints are additional rules that we can apply to table columns. For example, if you add a `NOT NULL` constraint to a column, you cannot insert an empty value (`NULL`) into that column.

```
-- create table
-- the id column has the NOT NULL constraint
CREATE TABLE Customers (
id INT NOT NULL,
name VARCHAR(50),
email VARCHAR(50)
);
-- insert data without providing value for the id column
-- results in an error because of the constraint
INSERT INTO
Customers(name, email)
VALUES
('Jack', 'jack@example.com');
```

UNIQUE Constraint

The `UNIQUE` constraint prevents the insertion of duplicate values into the table. For example,

```
-- create table
-- the id column has the UNIQUE constraint
CREATE TABLE Customers (
id INTEGER UNIQUE,
name VARCHAR(50),
email VARCHAR(50)
);
-- insert a row
INSERT INTO Customers(id, name, email)
VALUES (1, 'Jack', 'jack@example.com');
-- insert another row
-- results in an error because the id column already contains
1
INSERT INTO Customers(id, name, email)
VALUES (1, 'Anita', 'anita@example.com');
```

PRIMARY KEY

The PRIMARY KEY constraint is a combination of NOT NULL and UNIQUE constraints. The PRIMARY KEY constraint ensures that each row in a table is uniquely identifiable.

```
-- create a table
-- the id column has the PRIMARY KEY constraint
CREATE TABLE Customers (
id INTEGER PRIMARY KEY,
name VARCHAR(50),
email VARCHAR(50)
);

-- insert a row
INSERT INTO Customers(id, name, email)
VALUES (1, 'Jack', 'jack@example.com');
-- insert another row
-- results in an error because the id column already contains 1
INSERT INTO Customers(id, name, email)
VALUES (1, 'Anita', 'anita@example.com');
```

The AUTOINCREMENT Keyword

If we add the AUTOINCREMENT keyword to a column (of INTEGER data type), the value of the column is incremented automatically every time a new row is inserted into the table.

The AUTOINCREMENT keyword is often used with the PRIMARY KEY constraint to ensure that we do not insert NULL or duplicate values in a column. For example,

```
-- create table
-- the id column has the PRIMARY KEY constraint
-- the id column also has the AUTOINCREMENT feature
CREATE TABLE Customers (
id INTEGER PRIMARY KEY AUTOINCREMENT,
name VARCHAR(50),
email VARCHAR(50)
);
-- insert without passing value to the id field
INSERT INTO Customers(name, email)
VALUES ('Jack', 'jack@example.com');

-- insert another row
INSERT INTO Customers(name, email)
VALUES ('Anita', 'anita@example.com');
```

FOREIGN KEY

The FOREIGN KEY constraint is used when two tables are related through a column.

```
CREATE TABLE Customers (
id INTEGER PRIMARY KEY,
name VARCHAR(100),
age INTEGER
);
-- add foreign key to the customer_id column
-- the foreign key references the id column of the Customers table
CREATE TABLE Products (
customer_id INTEGER ,
name VARCHAR(100),
FOREIGN KEY (customer_id)
REFERENCES Customers(id)
);
```

Notice the following statement in the code above:

```
FOREIGN KEY (customer_id) REFERENCES Customers(id)
```

This statement makes customer_id the FOREIGN KEY, which references the id column of the Customers table.

The FOREIGN KEY constraint ensures that the value in the customer_id column of the Products table must be a value from the id column of the Customers table.

Additional Topics

SQL Views

A view is a virtual table based on the output of a `SELECT` statement. For example,

```
-- create a view named customer_demographics
CREATE VIEW customer_demographics AS
SELECT age, country
FROM Customers;
```

When you run the code, a view named `customer_demographics` is created.

Your database system will store the view until it is deleted. And you can perform select queries in a view, similar to a database table.

```
-- select all the rows from customer_demographics
SELECT *
FROM customer_demographics
```

Working of Views

Here is how a view works:

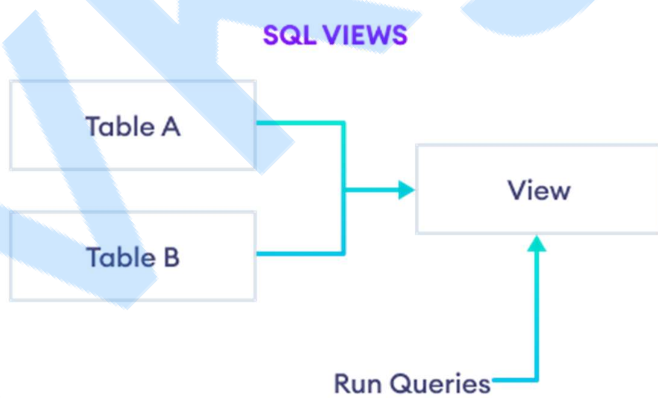

Figure: Working of an SQL View

Here, a view is created using two tables.

Once a view is created, we can run queries on the view. But you might be wondering, why create a view in the first place? Aren't tables enough?

Let's answer this question next.

Why Create Views?

The main reason to create views is to simplify our SQL query.

Imagine that you have a complex query that involving multiple tables and joins. In such cases, you can create a view that represents that query based on underlying tables, and then write code on the view instead. For example,

```
-- create a view by joining two tables
CREATE VIEW Customers_Orders AS
SELECT Customers.name, Customers.city, Orders.product
FROM Customers
INNER JOIN Orders
ON Customers.customer_id = Orders.customer_id;
```

When you run the code, a view is created.

name	city	product
John	New York	Computer
Robert	Los Angeles	Phone
John	New York	Tablet
David	Chicago	Phone
David	Chicago	Computer

Figure: A View Based on Two Tables

Now you can use this view to write queries. For example

```
-- select data from the view (which was created by joining two tables)
-- sort data by the city column
SELECT *
FROM Customers_Orders
ORDER BY city DESC;
```

As we can see, creating a view made writing complex queries easier.

SQL Commands

SQL can perform many tasks such as

- retrieving data
- creating tables
- inserting and changing rows
- deleting tables and so on

Due to the wide range of tasks SQL can perform, it can be overwhelming to remember all the commands.

To make it easier to understand different SQL commands, we can categorize SQL commands into several sublanguages, which we will explore in this lesson.

SQL Commands Categorization

SQL commands are mainly categorized into five sublanguages:

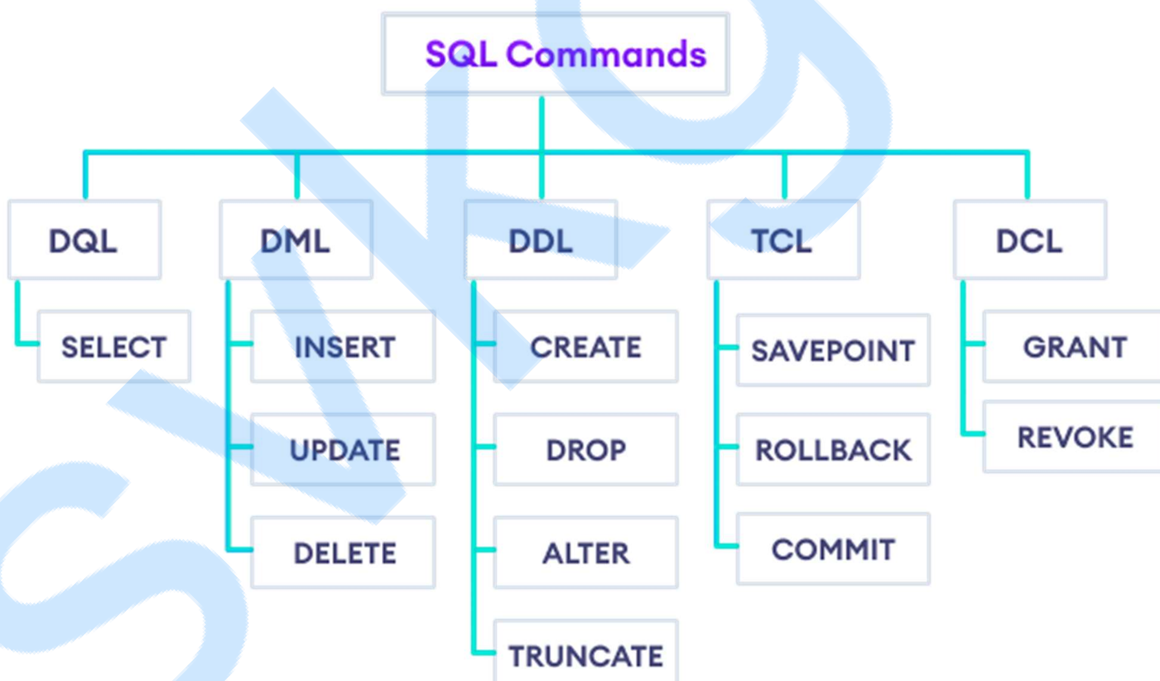

Figure: SQL Commands Categorization

Let's learn about each of these sublanguages in brief.

Data Query Language (DQL)

The Data Query Language (DQL) is responsible for retrieving data from a database table.

Commonly used DQL commands include:

- `SELECT` - to select data from a database table

Data Manipulation Language (DML)

The Data Manipulation Language (DML) is responsible for adding, updating and deleting data from a table.

Commonly used DML commands include:

- `INSERT` - to insert rows into a database table
- `UPDATE` - to edit data from a table
- `DELETE` - to delete rows from a table

Data Definition Language (DDL)

The Data Definition Language (DDL) is responsible for defining and deleting database tables.

Commonly used DDL commands include:

- `CREATE TABLE` - to create a database table
- `ALTER TABLE` - modify the structure of a table
- `DROP TABLE` - delete a table
- `CREATE DATABASE` - to create a database

We have already covered other DDL commands except `CREATE DATABASE`.

The `CREATE DATABASE` statement is used to create a new database. For example,

```
-- create database named test
CREATE DATABASE test;
```

Once a database is created, we can create tables inside it.

Data Control Language (DCL)

The Data Control Language (DCL) is responsible for managing permissions for different users.

In all major database systems, you can create different database users with different permissions.

For instance, you may want to grant all permissions to data administrators, but restrict the permissions of data analysts to only select data from certain tables.

Commonly used DCL commands include:

- `GRANT` - to grant permission to a database user
- `REVOKE` - removes previously granted permissions from a user

1. Example: GRANT

```
GRANT SELECT ON Customers, Products TO 'vincent';
```

The above statement grants the user with username `'vincent'` permission to perform the `SELECT` operation on the `Customers` and `Products` tables.

2. Example: GRANT

```
GRANT SELECT, INSERT ON * TO 'jules';
```

The above statement grants the user with username `'jules'` both `SELECT` and `INSERT` permissions on all tables in the database.

3. Example: REVOKE

```
REVOKE INSERT ON ALL TABLES FROM 'jules';
```

The above statement removes the `INSERT` permission on all tables for the user with the username `'jules'`.

Transaction Control Language (TCL)

The TCL commands allow us to save changes made in a database as well as rollback to our previous save point if needed.

Some commonly used TCL commands include:

- SAVEPOINT - for temporarily storing changes performed by DML (INSERT, UPDATE and DELETE statements)
- ROLLBALL - for reverting to SAVEPOINT
- COMMIT - for permanently storing changes performed by DML (INSERT, UPDATE and DELETE statements)

Recommendation: We suggest you look into DCL and TCL once you are comfortable with other commands.

Summary

- Data Query Language (DQL) - SELECT
- Data Manipulation Language (DML) - INSERT, UPDATE, DELETE
- Data Definition Language (DDL) - CREATE TABLE, CREATE DATABASE, ALTER TABLE, DROP TABLE
- Data Control Language (DCL) - GRANT, REVOKE
- Transaction Control Language (TCL) - SAVEPOINT, ROLLBACK, COMMIT