DEVELHOPE

#codeforimpact

Introduction to SQL

SQL and databases

The acronym **SQL** stands for **Structured Query Language**.

A **query** is a request for specific information from a database.

A **database (DB)** is an organised collection of data: for example, SQL-based databases organize data into tables, rows, columns, and indexes in order to facilitate the information management and retrieval.

SQL is a language that is used for storing, manipulating and retrieving data in databases.

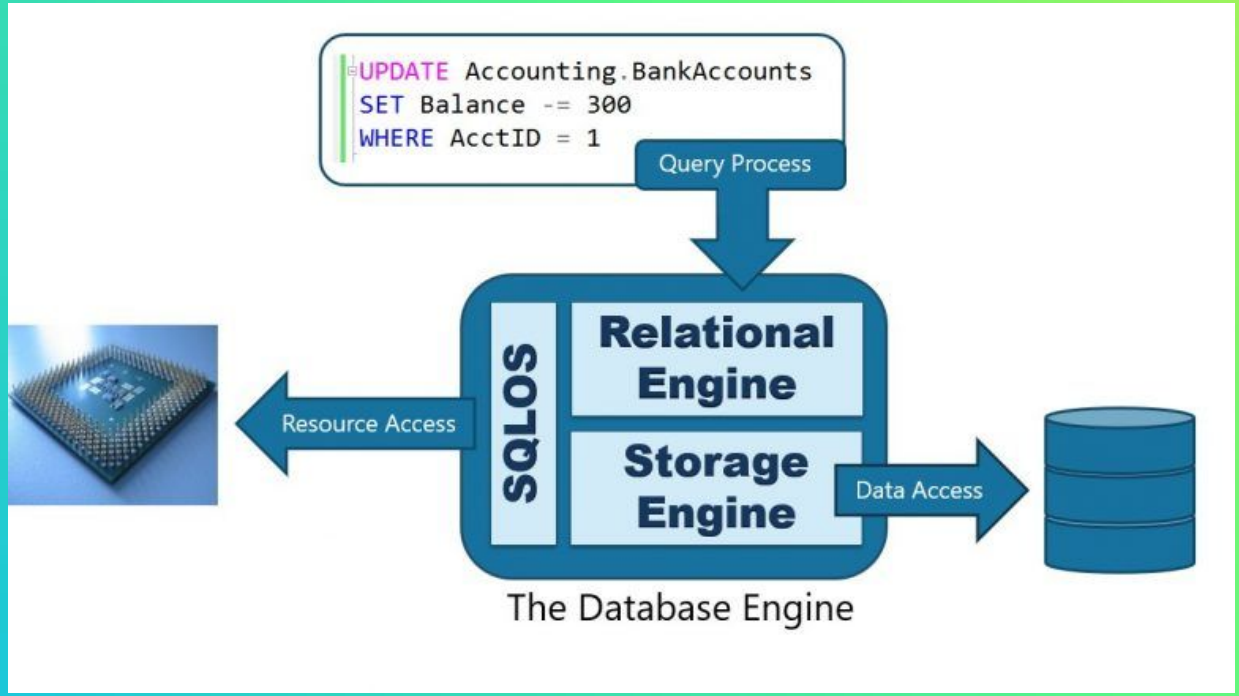
There are a lot of **different types of databases**, but some of them implement SQL as language.

So, the knowledge of SQL is very important for executing **CRUD** (**Create**, **Read**, **Update**, **Delete**) operations on databases.

DEVELHOPE

#codeforimpact

How a Database works



DEVELHOPE

#codeforimpact

Types of databases

Types of Databases

By data location:

- **Centralized database**: information store in 1 location, typical in big organisations like universities
- **Distributed database**: information is stored across multiple locations
- **Cloud database**: a virtual environment that executes over the cloud computing platform
- **End-user database**: primarily used by a single person, e.g. a spreadsheet on your local computer

By data structure:

- **Graph database**: the connections between data is stored and it's important like the original data
- **NoSQL database**: the hierarchical structure is similar to a file folder system and the stored data is non-relational (that means it isn't structured). NoSQL DBs are suitable for a better scalability
- **Object-oriented database**: data is represented and stored as objects, similarly to OOP principles
- **Relational database**: a **RDB** (*Relational Database*) is based on the *relational data model*, which stores data in the form of rows and columns (that together form a table). RDBs focus on the integrity of data.

Relational Databases (RDBs)

Considering that the main goal is to learn SQL, we will focus mostly on Relational Databases (e.g. *MySQL*, *PostgreSQL*) because SQL is the language *behind them*.

The **relational data model** is based on the following concepts:

- **Data Domain**: collection of values that a data element can contain
- **Attributes**: they are the basic storage units (also called **columns** or **fields**)
- **Records**: they contain fields that are related (also called **rows** or **tuple**)
- **Relation**: a relation is basically a table structure. Each table has 2 properties:
  - Rows (the records)
  - Columns (the attributes)
- **Relation Key**: each record (row) has an attribute or a combination of attributes that can identify the tuple in a unique way

DEVELHOPE

#codeforimpact

Relational Data Model - an example

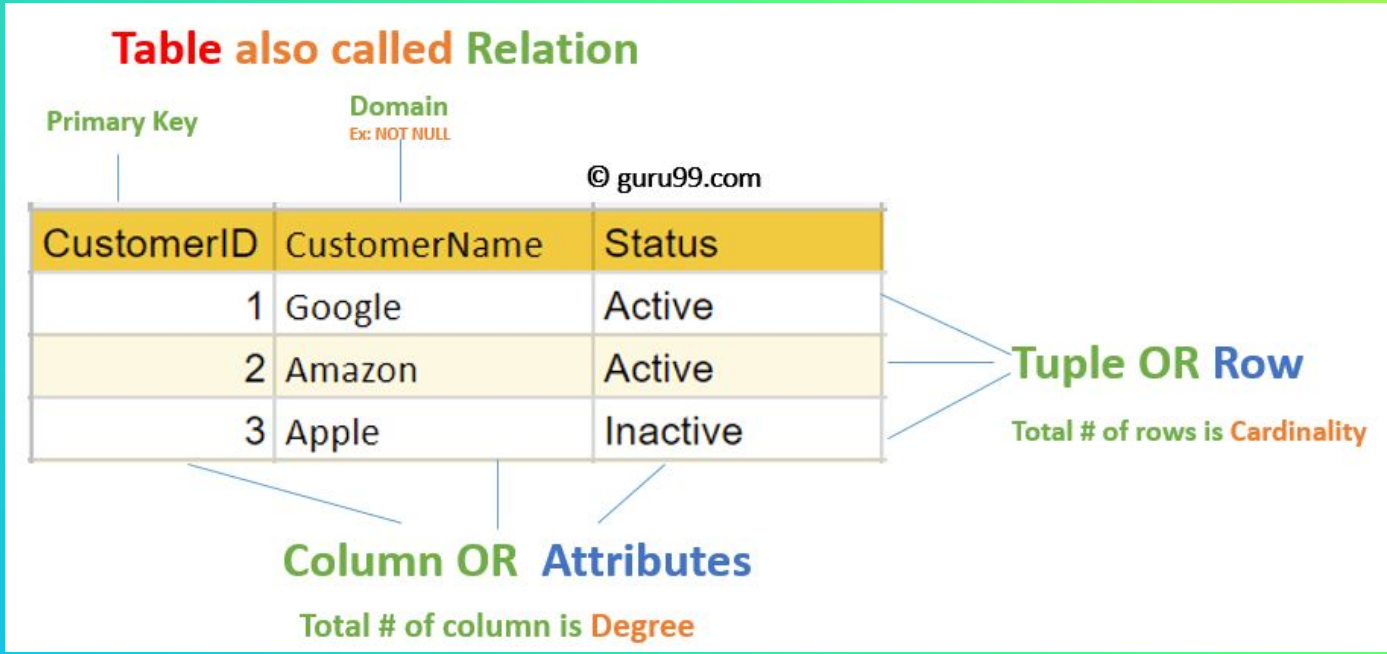


Image source

Relational Integrity Constraints and Keys

In order to be considered valid, each relation must respect some conditions called *Relational Integrity Constraints*:

- **Key (or Entity) Constraints** - a tuple has to be identified uniquely at least by a *minimal subset of attributes* (this subset is called **key**). If there are more *minimal subsets of attributes*, they are called **candidate keys**. A candidate key that is used to identify a record uniquely in a table is called **primary key**. Key Constraints impose that:
  - A key attribute (or attributes) can't have NULL values
  - In a relation, two tuples cannot have the same values for key attributes
- **Domain Constraints** - attributes can have just a specific set of values
- **Referential Integrity Constraints** - a **foreign key** is a set of attributes that refers to the primary key of another table. The *referential integrity rule* imposes that each foreign key value can be just in 1 of the 2 following states:
  - The foreign key value refers to a primary key value of another table in the RDB
  - Null value (if allowed by the data owner)

Types of keys

In the previous slide we mentioned different concepts about keys. Here is a list of all possible keys:

- **Super Key**: a group of single or multiple keys that **identifies rows** in a table
- **Candidate Key**: a set of one or more columns that can **identify a record uniquely** in a table
- **Primary Key**: a **candidate key** that has been **chosen to identify a record uniquely** in a table
- **Unique Key**: similar to the Primary Key, but it allows one null value in the column
- **Alternate Key**: a possible alternative to the Primary Key
- **Foreign Key**: a reference from Table A to the primary key of another Table B.
  - **The foreign key creates a relationship between 2 or more tables**

DEVELHOPE

#codeforimpact

Types of keys - an example

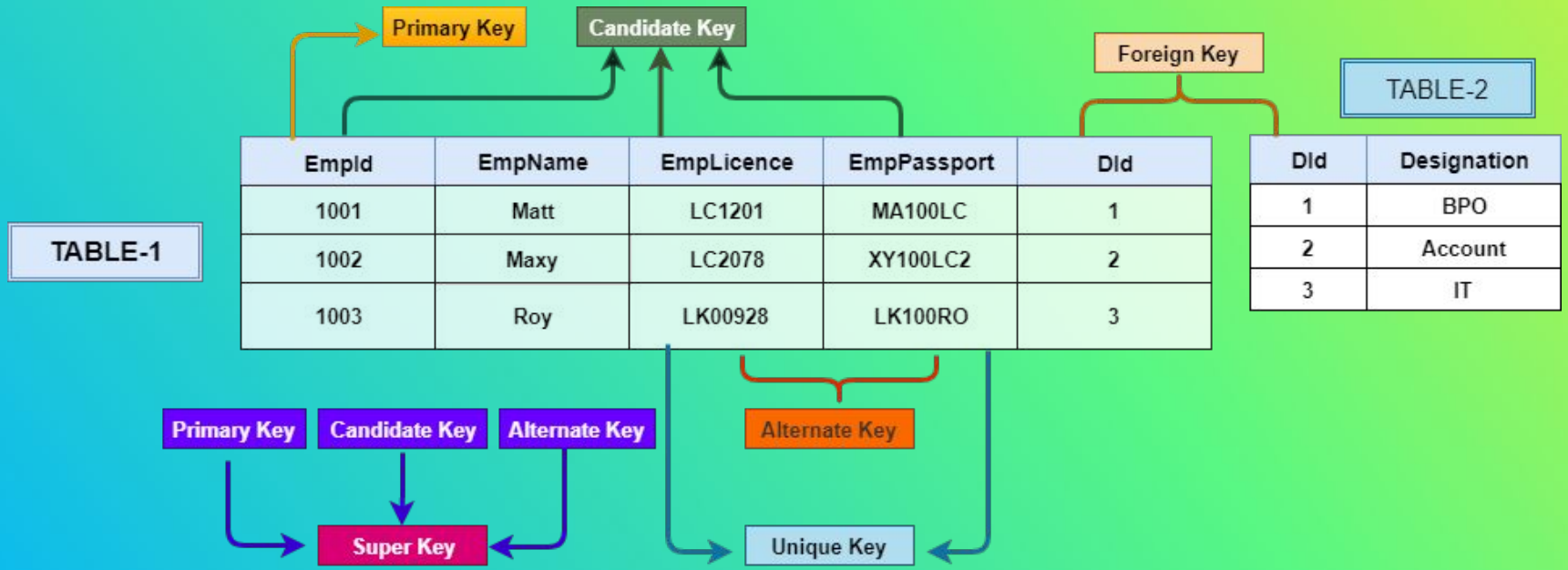


Image source

ACID properties for relational model data

A *database transaction* is a group of tasks that are executed by a **DBMS** (*Database Management System*).

Each database transaction has to observe the ACID properties:

- **Atomicity**: each transaction has to be treaded as an atomic unit. It means that either all the operations (within a transaction) are completed or none. So, it's not possible to have a partially completed database transaction

- **Consistency**: after the transaction execution the database must be remain in a correct state as before
- **Isolation**: each transaction execution is isolated and it doesn't affect any other transaction

- **Durability**: once the transaction is committed, data changes become permanent

DEVELHOPE

#codeforimpact

Entity Relationship Diagram (ERD)

When designing a new database, using some kind of conceptual visualisation tools is really helpful.

One of these tools is the ***Entity Relationship Diagram*** (*ERD*) or **ER model**, that is based on 3 basic concepts:

- **Entities** - objects that can have data stored about them → they correspond to the **tables**
- **Attributes** - properties of an Entity
- **Relationship** between entities (as alredy said, a foreign key creates a relationship between 2 or more tables)

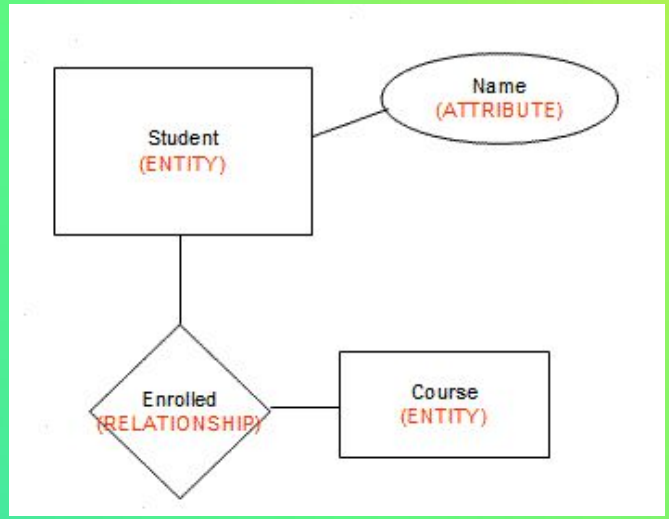


Image source

DEVELHOPE

#codeforimpact

Entity Relationship Diagram (ERD) - an example

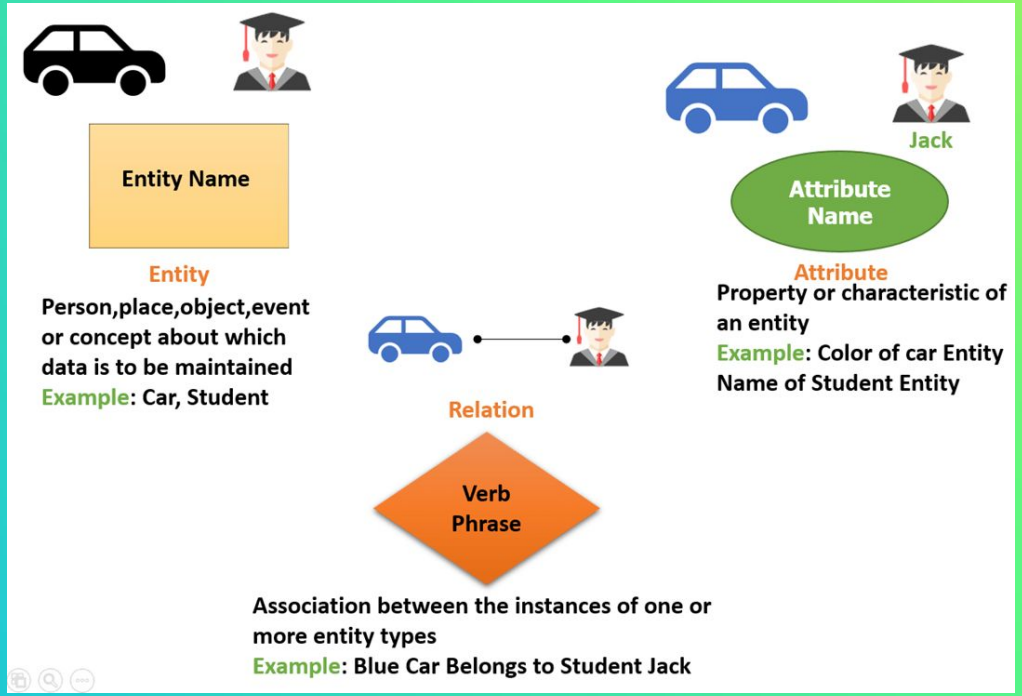


Image source

ER model: types of relationships

ER model's **relationships** can be grouped into 4 main types according to the **relationship cardinality**:

- **One-to-One**
  - One instance of entity A can be associated with at most one instance of entity B
- **One-to-Many**
  - One instance of entity A can be associated with multiple instances of entity B
- **Many-to-One**
  - Multiple instances of entity A can be associated with at most one instance of entity B
- **Many-to-Many**
  - Multiple instances of entity A can be associated with multiple instances of entity B

DEVELHOPE

#codeforimpact

ER model: types of relationships - an example

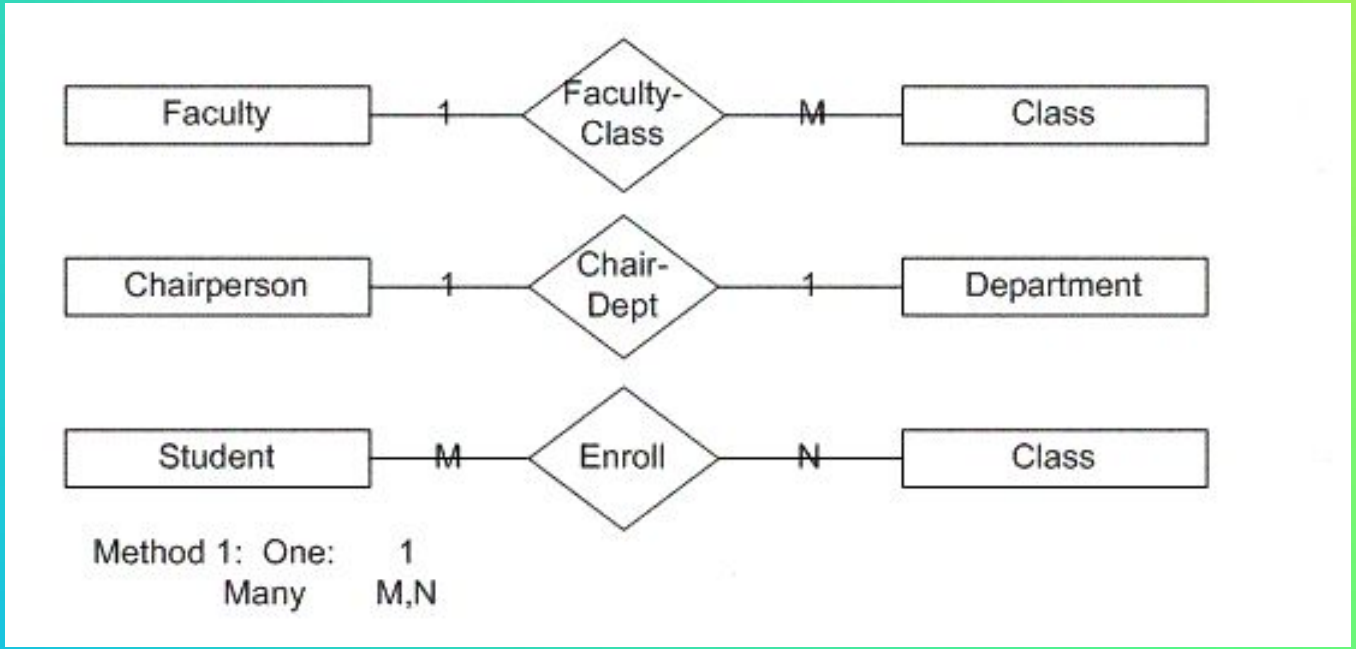


Image source

SQL: database commands

- Basic database creation: `CREATE DATABASE dbnamehere;`
- Create a database if not already existing: `CREATE DATABASE IF NOT EXISTS dbnamehere;`
- `Rename a database: ALTER DATABASE dbnamehere RENAME TO newdbnamehere;`
- Delete a database: `DROP DATABASE dbnamehere;`
- Backup:

```
BACKUP DATABASE dbnamehere;
TO DISK = “filepath”;
```

- Show available databases: `SHOW DATABASES;`

#codeforimpact

SQL: table commands

Create a table

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

Delete a table

```
DROP TABLE table_name;
```

Update data in the table

```
UPDATE table_name
SET column_name = value
WHERE [condition_here];
```

Add a column to table

```
ALTER TABLE table_name
ADD new_column datatype;
```

Remove a column from table

```
ALTER TABLE table_name
DROP COLUMN column_name;
```

DEVELHOPE

#codeforimpact

SQL: SELECT

Basic SELECT

```
SELECT column_name
FROM table_name;
```

SELECT all columns

```
SELECT * FROM table_name;
```

Select with no duplicates

```
SELECT DISTINCT
column_name
FROM table_name;
```

Select with filter

```
SELECT column_name
FROM table_name
WHERE column_name operator value;
```

Select with a numeric limit

```
SELECT column_name
FROM table_name
LIMIT number;
```

DEVELHOPE

#codeforimpact

SQL: Logical Operators

ALL - True if all of the subquery values meet the condition

```
SELECT ProductName
FROM Products
WHERE ProductID = ALL (SELECT ProductID FROM Orders WHERE Quantity = 10);
```

AND - TRUE if all the conditions separated by AND areTRUE

```
SELECT * FROM Customers
WHERE City = "Rome" AND Country = "IT";
```

SQL: Logical Operators

ANY - TRUE if any (at least one) of the subquery values meet the condition

```
SELECT * FROM Products
WHERE Price > ANY (SELECT Price FROM Products WHERE Price > 50);
```

BETWEEN - TRUE if the operand is within the range of comparisons

```
SELECT * FROM Products
WHERE Price BETWEEN 40 AND 80;
```

SQL: Logical Operators

EXISTS - TRUE if the subquery returns one or more records

```
SELECT SupplierName
FROM Suppliers
WHERE EXISTS (SELECT ProductName FROM Products WHERE Products.suppId =
Suppliers.id AND Price < 50);
```

IN - TRUE if the operand is equal to one of a list of expressions

```
SELECT * FROM Customers
WHERE City IN ('Rome','Berlin');
```

SQL: Logical Operators

LIKE - TRUE if the operand matches a pattern

```
SELECT * FROM Customers
WHERE City LIKE '%in%';
```

NOT - Displays a record if the condition(s) is NOT TRUE

```
SELECT * FROM Customers
WHERE City NOT LIKE '%in%';
```

SQL: Logical Operators

OR - TRUE if any of the conditions separated by OR is TRUE

```
SELECT * FROM Customers
WHERE City = "Rome" OR Country = "Italy";
```

SOME - TRUE if any of the subquery values meet the condition

```
SELECT * FROM Products
WHERE Price > SOME (SELECT Price FROM Products WHERE Price > 70);
```

DEVELHOPE

#codeforimpact

SQL: aggregate functions

MAX()/MIN() - Returns the largest/smallest value in a column

```
SELECT MAX|MIN(column_name)
FROM table_name;
```

SUM() - Returns the total sum of a numeric column

```
SELECT SUM(age)
FROM students;
```

AVG() - The average value for a numeric column

```
SELECT AVG(column_name)
FROM table_name;
```

COUNT() - Counts the number of rows where the column is not NULL

```
SELECT COUNT(column_name)
FROM table_name;
```

SQL: other commands

AS - Rename a column or table using an alias

```
SELECT column_name AS 'Alias'
FROM table_name;
```

AVG() - The average value for a numeric column

```
SELECT AVG(column_name)
FROM table_name;
```

COUNT() - Counts the number of rows where the column is not NULL

```
SELECT COUNT(column_name)
FROM table_name;
```

DEVELHOPE

#codeforimpact

SQL: other commands

DELETE - Delete rows from a table

```
DELETE FROM table_name
WHERE column_name = value;
```

GROUP BY - Aggregate functions

```
SELECT surname, AVG(age)
FROM customers
GROUP BY surname;
```

HAVING() - It's like a WHERE for aggregate functions

```
SELECT column_name, COUNT(*)
FROM table_name
GROUP BY column_name
HAVING COUNT(*) > value;
```

IS NULL - Test for empty values

```
SELECT column_name
FROM table_name
WHERE column_name IS NULL;
```

SQL: other commands

ORDER BY - sort the result set by a particular column either alphabetically or numerically

```
SELECT column_name
FROM table_name
ORDER BY column_name ASC | DESC;
```

TRUNCATE - removes all data entries from a table in a database, but keeps the table and structure in place

```
TRUNCATE TABLE students;
```

INSERT - Add a new row to the table

```
INSERT INTO table_name (column_1, column_2, column_3)
VALUES (value_1, 'value_2', value_3);
```

DEVELHOPE

#codeforimpact

SQL: JOIN - a visual overview

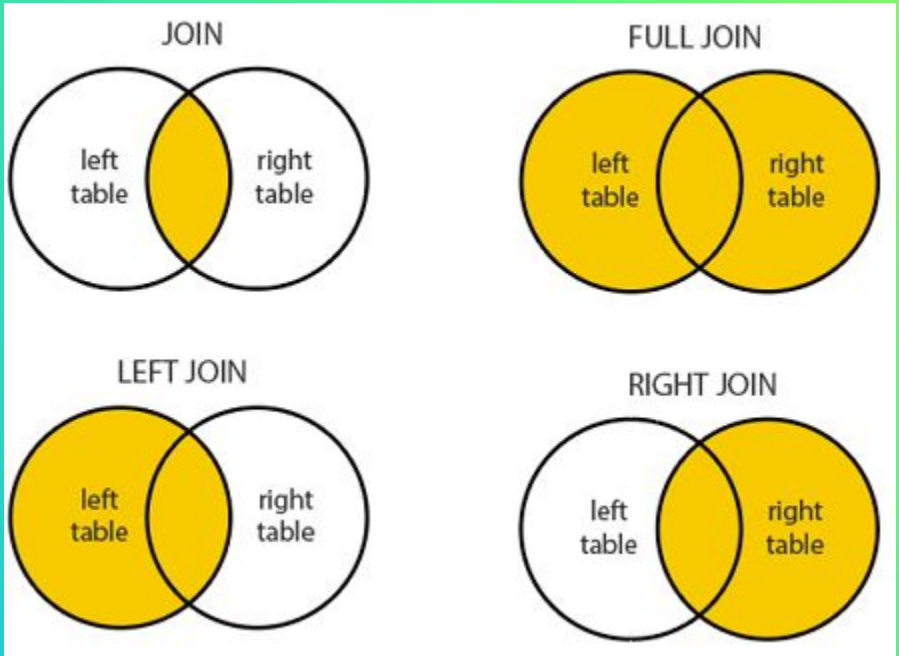


Image source

SQL: INNER JOIN (a kind of AND)

Records that have matching values in both tables

```
SELECT name
FROM customers
INNER JOIN orders
ON customers.customer_id = orders.customer_id;
```

SQL: LEFT JOIN

Records from the left table that match records in the right table

```
SELECT name
FROM customers
LEFT JOIN orders
ON customers.customer_id = orders.customer_id;
```

SQL: RIGHT JOIN

Records from the right table that match records in the left table

```
SELECT name
FROM customers
RIGHT JOIN orders
ON customers.customer_id = orders.customer_id;
```

SQL: FULL JOIN (a kind of OR)

Records that have a match in the left or right table

```
SELECT name
FROM customers
FULL OUTER JOIN orders
ON customers.customer_id = orders.customer_id;
```

DEVELHOPE

#codeforimpact

SQL: UNION

UNION - Combines the result-set (**distinct values**)
Of two or more SELECT statements

```
SELECT Postcode FROM Customers
UNION
SELECT Postcode FROM Suppliers
ORDER BY Postcode;
```

UNION ALL - Combines the result-set (**duplicates values**)
of two or more SELECT statements

```
SELECT Postcode FROM Customers
UNION ALL
SELECT Postcode FROM Suppliers
ORDER BY Postcode;
```

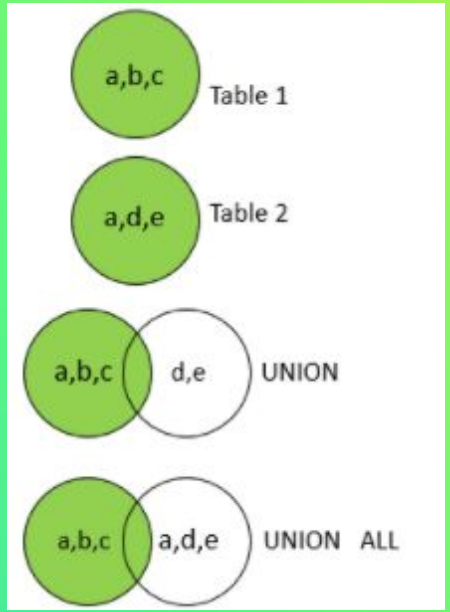


Image source

DEVELHOPE

#codeforimpact

Introduction to SQL