# SQL FOR DATA SCIENCE

By Tajamul Khan

Data Blogs

Follow Tajamul Khan 

# Basic

- Database - Types
- DBMS vs RDBMS
- ER Diagram
- Relational Database Schema
- SQL - ACID Properties, Commands
- Data Types
- Constraints
- Errors in SQL
- Keys
- Normalisation - 1NF, 2NF, 3NF, BCNF
- Operators
- Clauses
- Alias
- Case Statement

# Practicals

W3 schools

## Easy Level

stratascratch

LeetCode

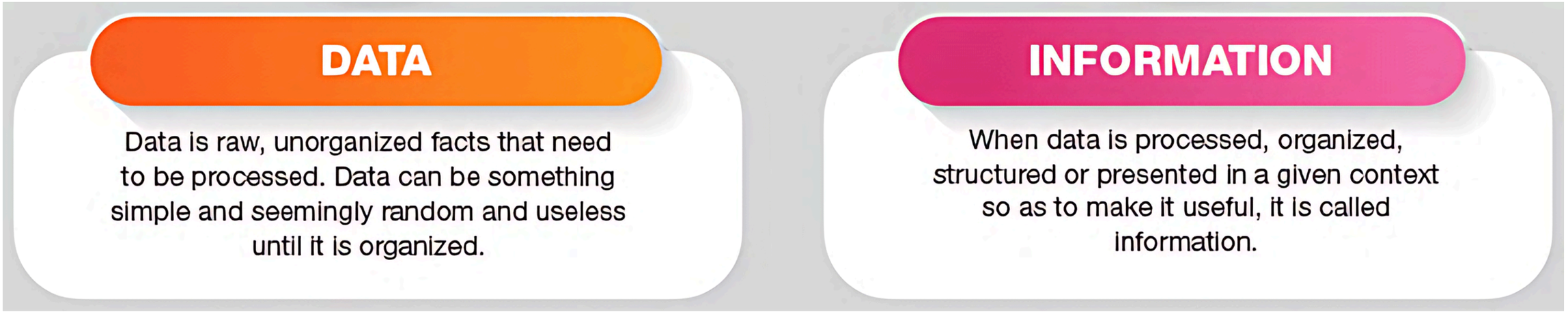
DataLemur

## Data

Data refers to raw, unprocessed facts

## Information

processed and organized data that is meaningful and useful.



# Key Concepts

## Data Redundancy

Same piece of data exists in multiple places in Database

## Data Quality

Data meets the applicable standards

Invalid / Wrong data Type

- Data Not Found
- No Rows Selected

| Emp ID | Name |
|---|---|
| 101 | Zac |
| Tom | 23 |

## Data Integrity

Data is valid and consistent. It is used to to restrict invalid data from entering the table. It can be achieved by:

- *Data Types*
- *Constraints*

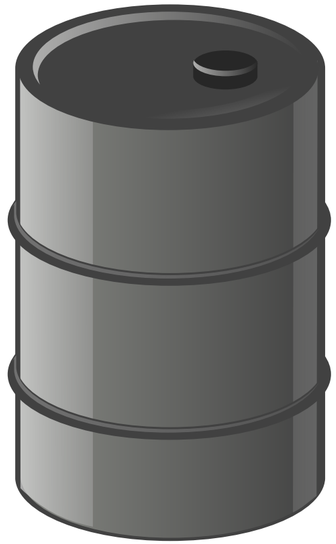
# Database

Data Container

It is a storage or a container in which we store and organise data

Databases help us with efficiently storing, accessing, and manipulating data

| Excel | Database |
|---|---|
| Easy to use—untrained person can work | Trained person can work |
| Data stored less data | Stores large amount of data |
| Good for one-time analysis, quick charts | Can automate tasks |
| No data integrity due to manual operation | High data integrity |
| Low search/filter capabilities | High search/filter capabilities |



# Evolution of Databases

Flat File Database

↓

Hierarchical Database

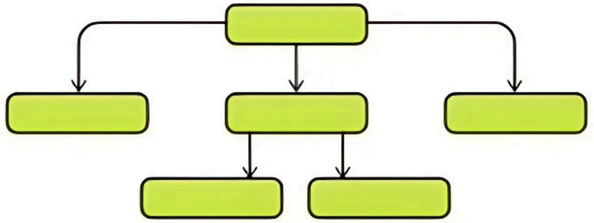
↓

Network Database

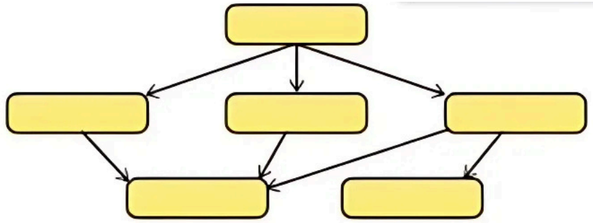
**2020**

| Relational Database | Non-Relational Database |
| --- | --- |
| Oracle | MongoDB |

# Type of Databases

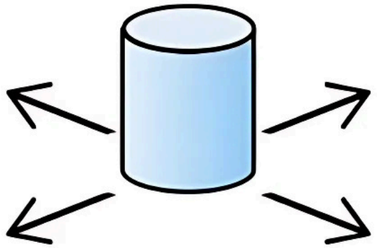


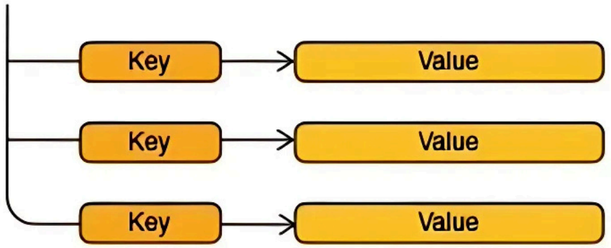
Hierarchical
Tree structure



Network
Graph structure

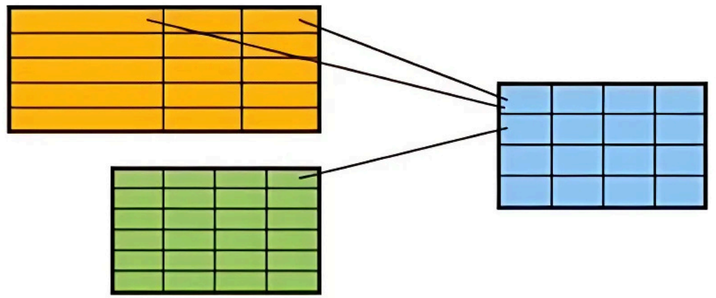
DBMS






NoSQL
Key-value pairs
Graph
Document

Non Relational



Relational
Tabular structure
The focus of this course

## Flat File Database 1960

The data & information is stored in the form of numerous physical files with no relation among each other.

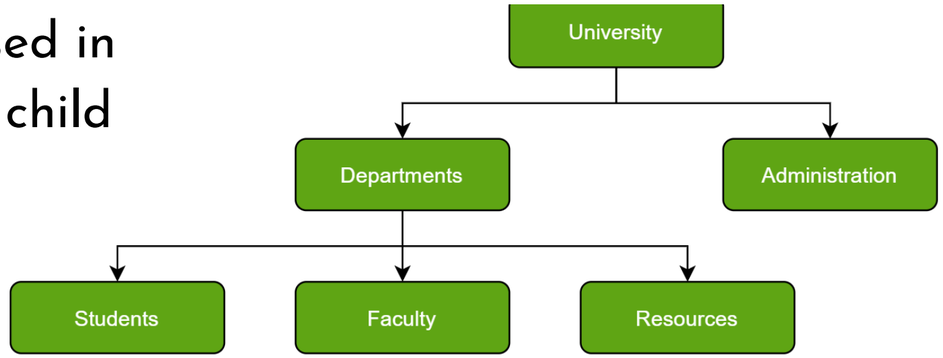
- Data Redundancy
- Time Consuming to search for data or information
- It is recommended only if data to be maintained is minimum

| Person | Height |
|---|---|
| Wendy | 5'6" |
| Michael | 5'9" |
| Rachael | 5'3" |
| Allen | 5'11" |

## Hierarchical Database

The data is stored in the form of tree like structure organised in parent-child relationship, parent record is allowed multiple child records.

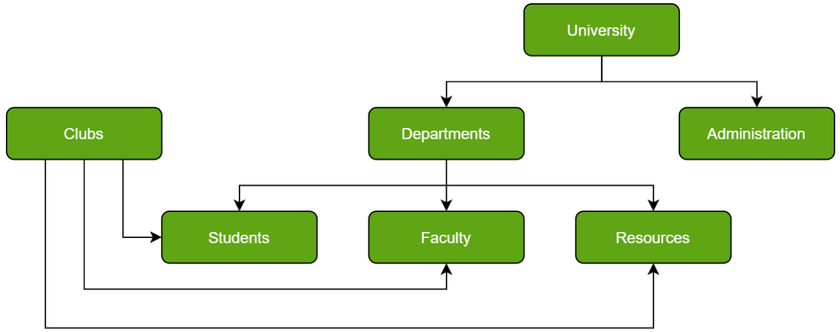
- Due to relationship data redundancy was reduced
- Searching was faster as compare to flat files



## Network Database

Just like hierarchical database, the data is stored in tree like structure except child can connect to multiple parent records

- Due to relationship, redundancy was reduced more and search was fast as compared to flat and hierarchical.
- Complex design, if one node fails entire model shuts down




## Non Relational Database

Big Data due to dynamic Schema

A non-relational database, often referred to as NoSQL, is a type of database that doesn't store data in tabular forms. Instead, it stores data as key-value pairs, documents (json), graphs, wide-columns, time series, search engine

- These databases are designed to handle large volumes of data efficiently and are highly scalable.

## Relational Database

E.F Codd in 1970

The Data is stored in the form of Tables. These tables are connected to each other with the help of primary and foreign key due to which the data duplicity was completely reduced.

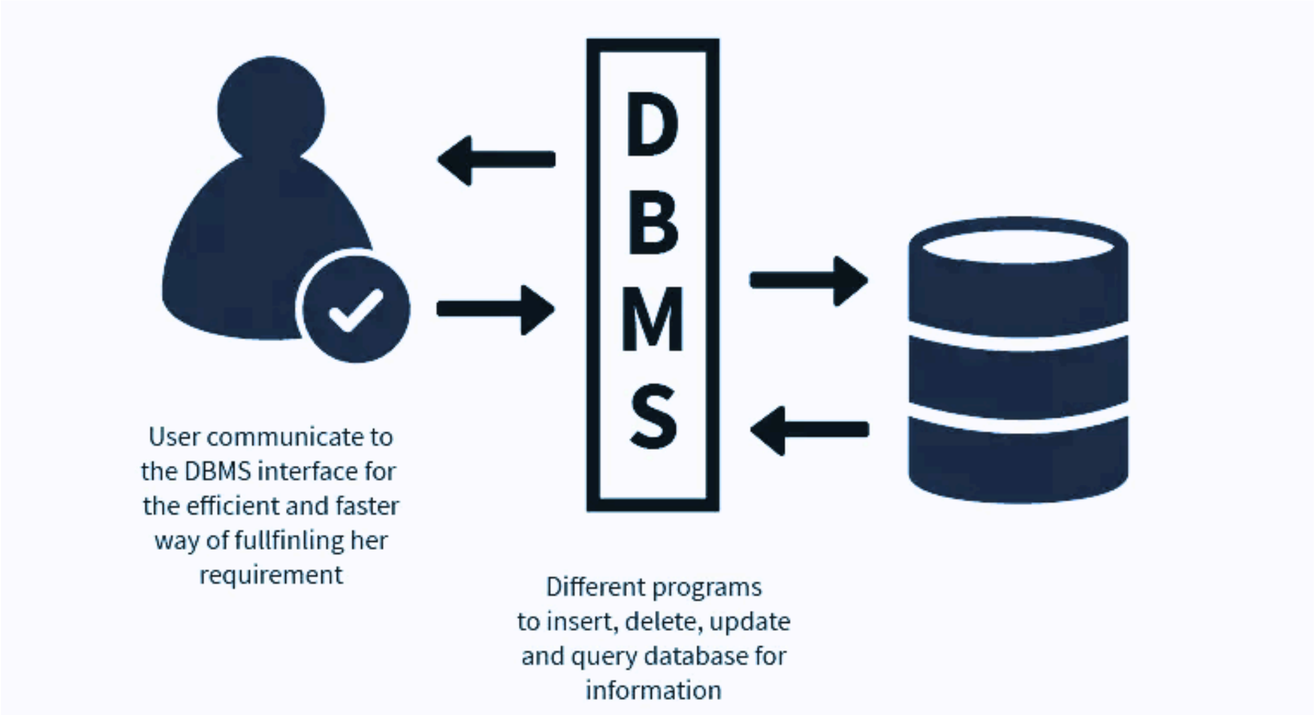
- This model is effective and efficient than all other Databases

# DBMS Data Base Management System

Software used to manage data base.

It is used to create, manage, and organizig data into a Database,

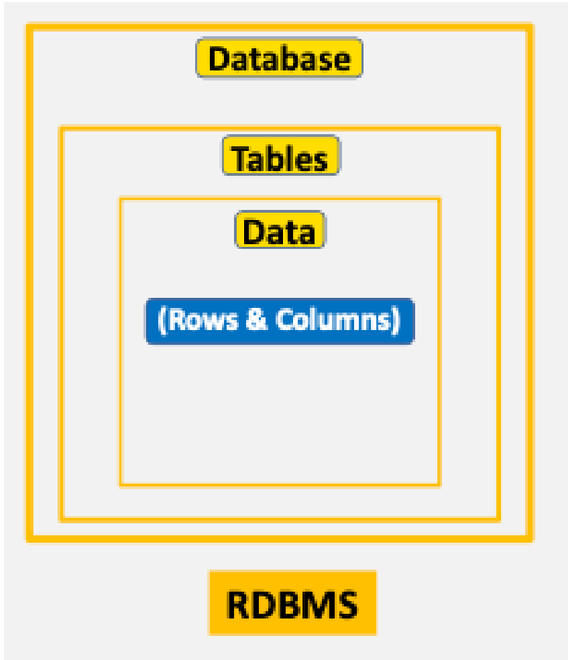
Data is stored in the form of files, no relations & Normalisation



# RDBMS Relational Data Base Management System

- Advanced version of DBMS allows to access the data more efficiently.
- Data is stored in the form of Tables unlike DBMS.
- Enhanced Security features, Good performance and also it can store huge volume of data into the Database

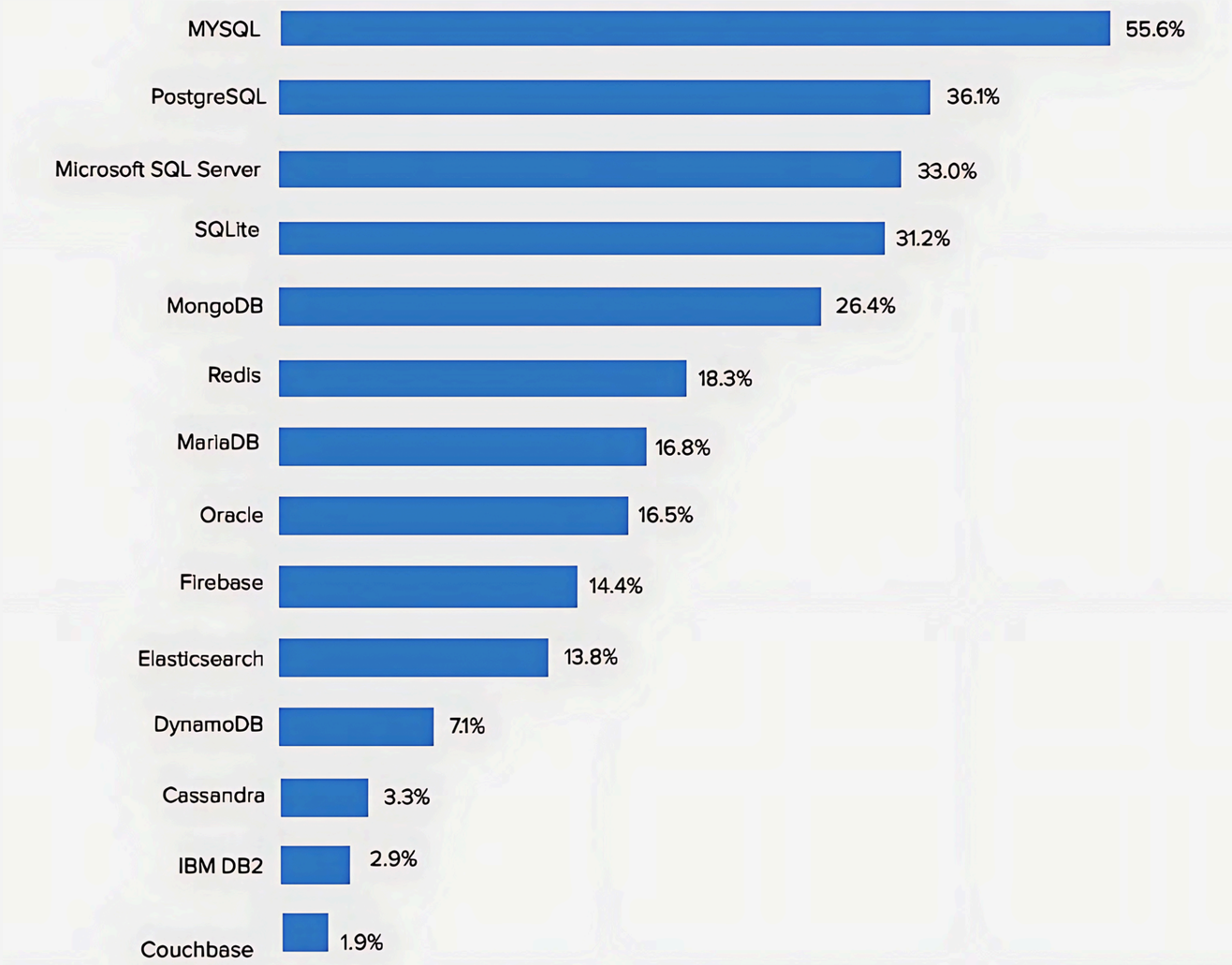
| DATABASE | IDE |
|---|---|
| PostgreSQL | PgAdmin |
| MySQL | MySQL Workbench |
| Microsoft SQL Server | SQL Server Management Studio (for Windows)<br>Azure Data Studio (for Mac) |
| Oracle | SQL Developer |



# DBMS vs RDBMS

| DBMS | RDBMS |
|---|---|
| DBMS stores data as file. | RDBMS stores data in tabular form. |
| No relationship between data. | Data is stored in the form of tables which are related to each other. |
| Normalization is not present. | Normalization is present. |
| DBMS does not support distributed database. | RDBMS supports distributed database. |
| It stores data in either a navigational or hierarchical form. | It uses a tabular structure where the headers are the column names, and the rows contain corresponding values. |
| It deals with small quantity of data. | It deals with large amount of data. |
| Data redundancy is common in this model. | Keys and indexes do not allow Data redundancy. |
| Security is less. | More security measures provided. |
| It supports single user. | It supports multiple users. |
| Data fetching is slower for the large amount of data. | Data fetching is fast because of relational approach. |

# RDBMS Popularity



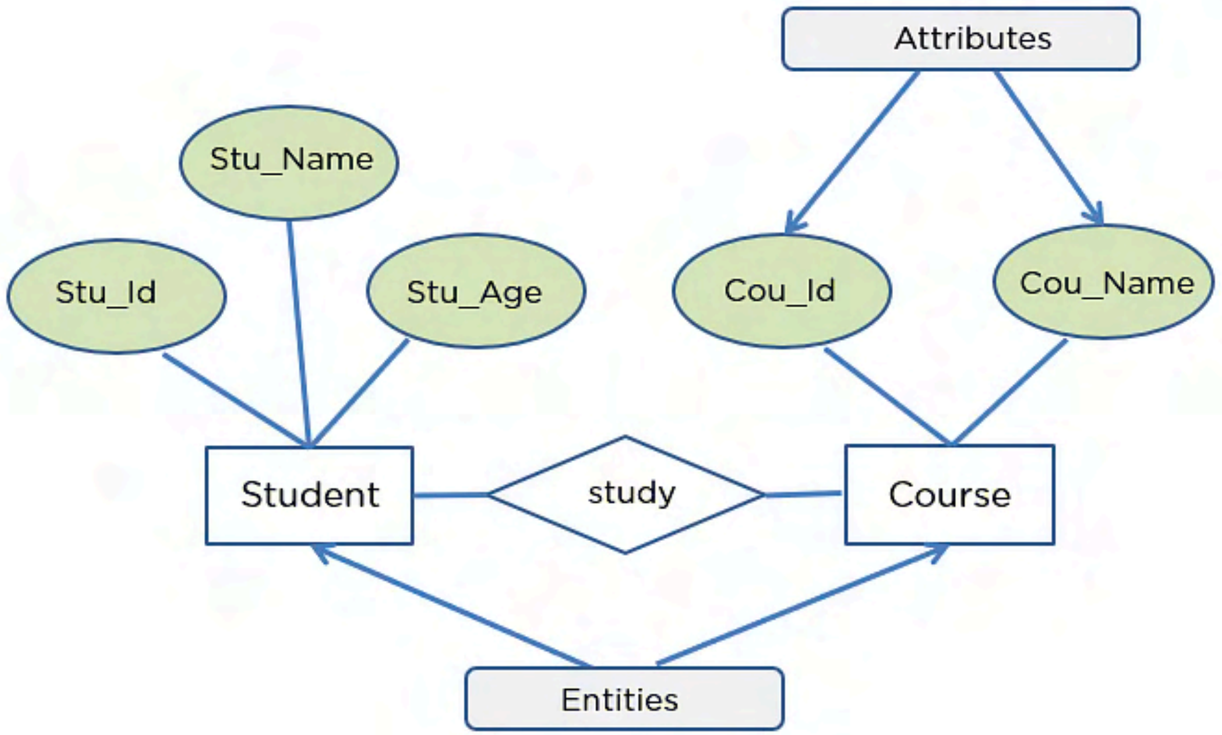
# SQL vs Non SQL Database

| Relational Database | Non-Relational Database |
|---|---|
| SQL database | NoSQL database |
| Data stored in tables | Data stored are either key-value pairs, document-based, graph databases, or wide-column stores |
| These databases have fixed or static or predefined schema | They have dynamic schema |
| Low performance with huge volumes of data | Easily work with huge volumes of data |
| Eg: PostgreSQL, MySQL, MS SQL Server | Eg: MongoDB, Cassandra, Hbase |

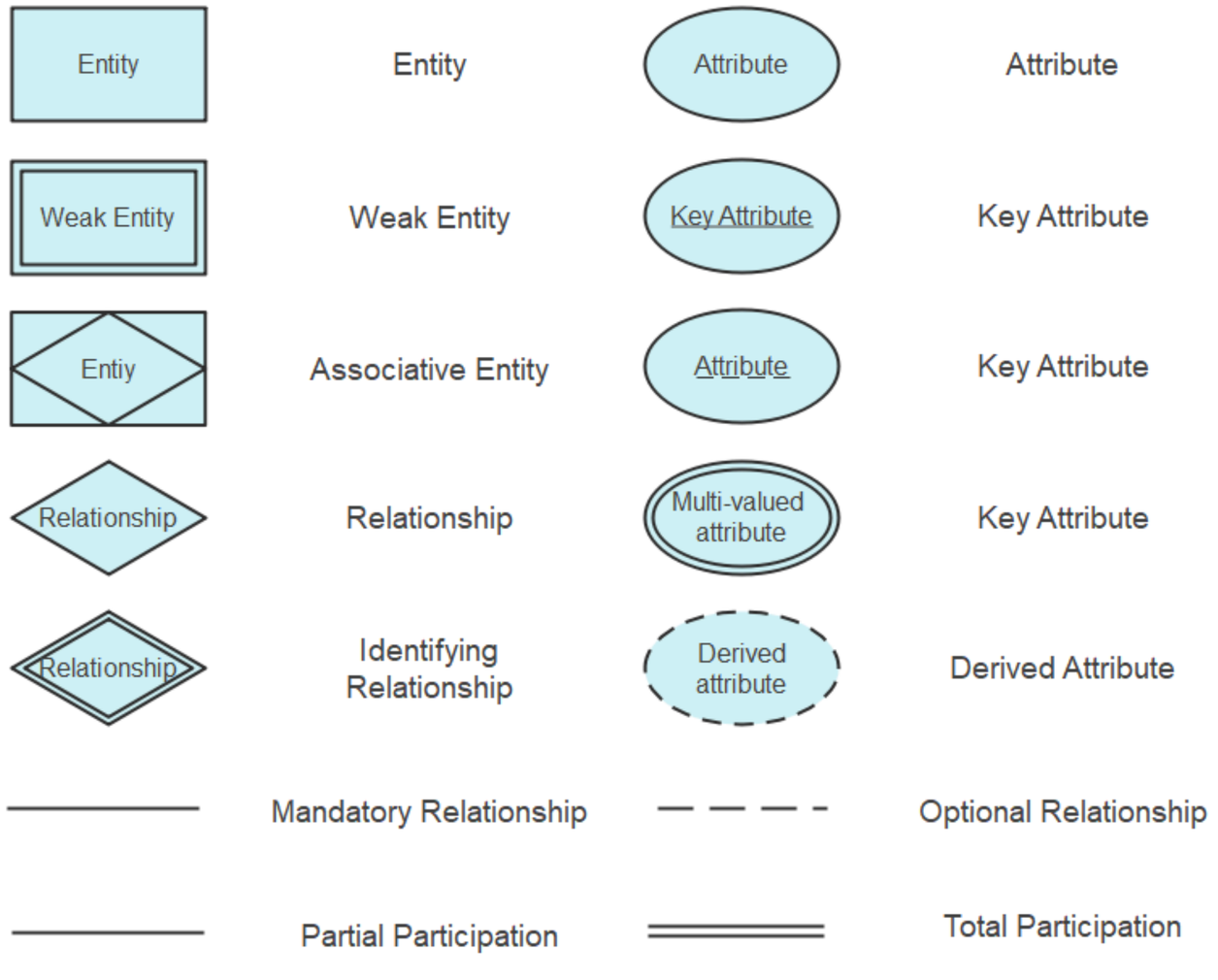
# ER Diagram

First step for designing Relational Database

- A diagram that represents relationships among entities in a database.
- An Entity Relationship Diagram (ER Diagram) **pictorially** explains the relationship between entities to be stored in a database

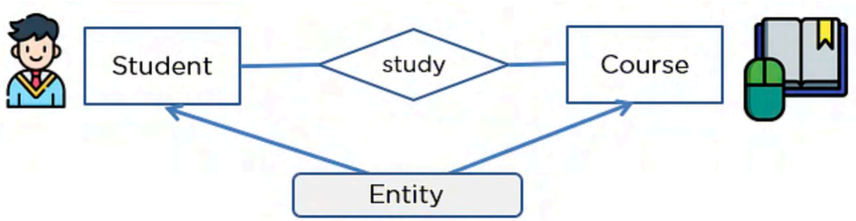


# Components of ER diagram



# Entity

- An entity may be any object, class, person or place.
- It is represented as **rectangles**.




independent

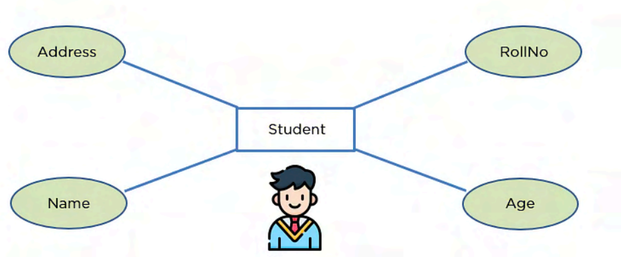


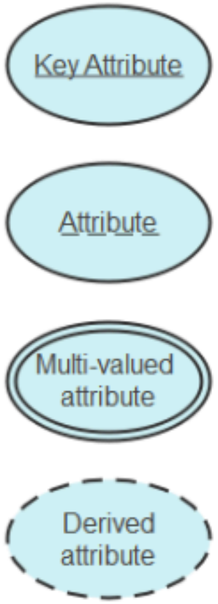
dependent

| Entity Type | Entity Name | Definition | Example |
|---|---|---|---|
| Strong | Product | Represents an independent entity that can exist on its own, such as a product with attributes like ProductID, Name, Price, and Category. | A product with ProductID P001, Name "Laptop", Price $1000, Category "Electronics" |
| Weak | Order | Represents an entity that depends on another entity for its existence and is identified by a combination of its own attributes and the primary key of the related entity. | An order with OrderID 123, Date "2024-03-30", linked to CustomerID C001 |

# Attribute

- The attribute is used to describe the property of an entity.
- Eclipse is used to represent an attribute.

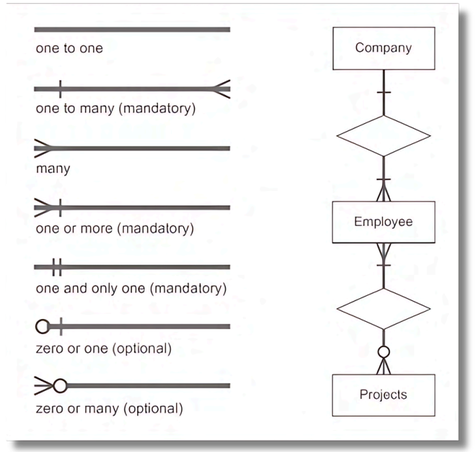



| Attribute Type | Attribute Name | Definition | Example |
|---|---|---|---|
| Key Attribute | StudentID | A unique identifier for a student within a system. | 101 |
| Composite Attribute | Address | An attribute that combines multiple sub-attributes. | Street: "123 Main St", City: "New York", Zip: "10001" |
| Multivalued Attribute | Phone Numbers | An attribute that can hold multiple values for a single entity. | Home: "123-456-7890", Mobile: "987-654-3210" |
| Derived Attribute | Age | An attribute whose value is derived from other attributes. | Calculated based on Date of Birth |

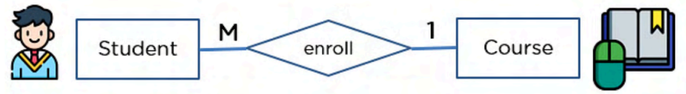
# Relationships

- A relationship is used to describe the relation between entities.
- Diamond is used to represent the relationship.

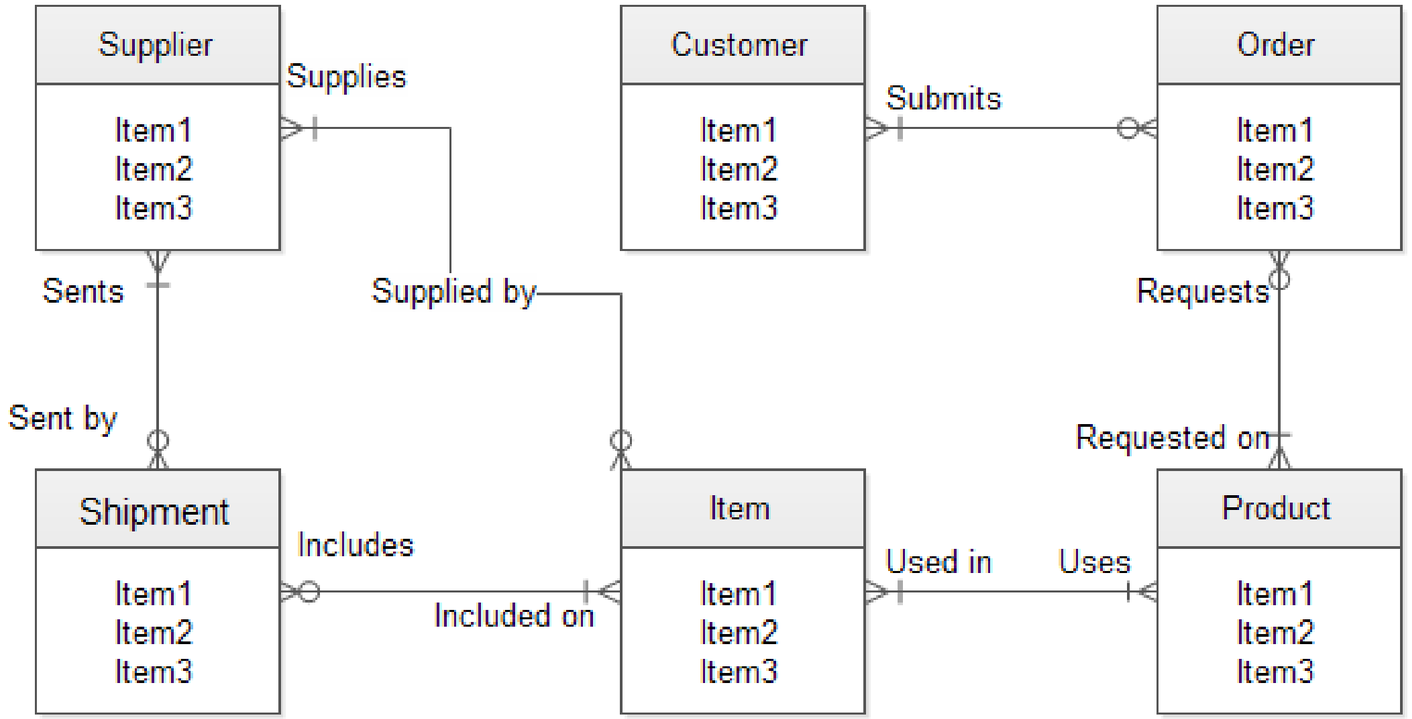


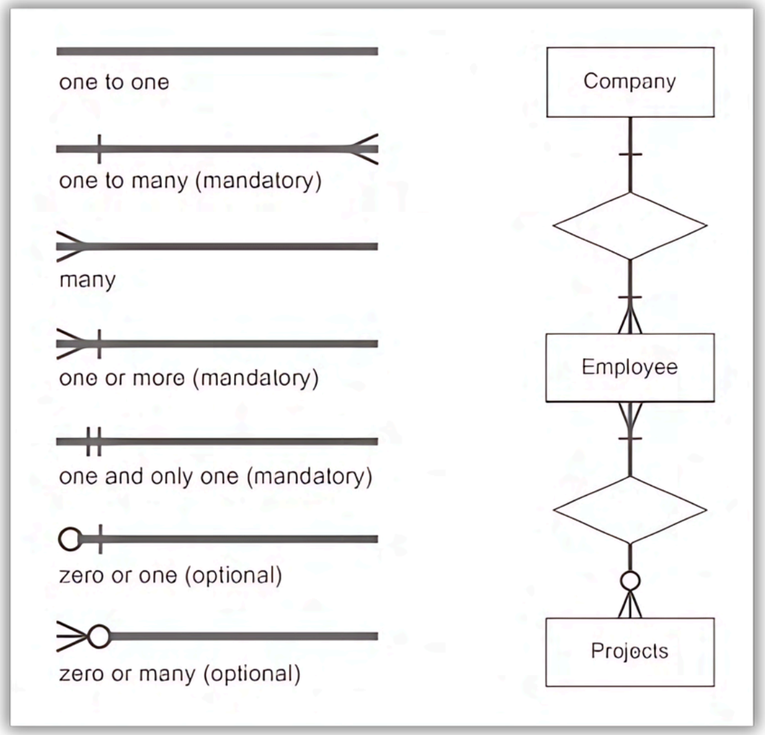

| Relationship Type | Description | Example |
|---|---|---|
| One-to-One | One instance of an entity is associated with exactly one instance of another entity. | One person has exactly one ID. |
| One-to-Many | One instance of an entity is associated with zero, one, or multiple instances of another entity. | One customer can place multiple orders. |
| Many-to-One | Many instances of an entity are associated with exactly one instance of another entity. | Many students can enroll in one course. |
| Many-to-Many | Many instances of an entity are associated with many instances of another entity. | Many employees can work on many projects, and vice versa. |



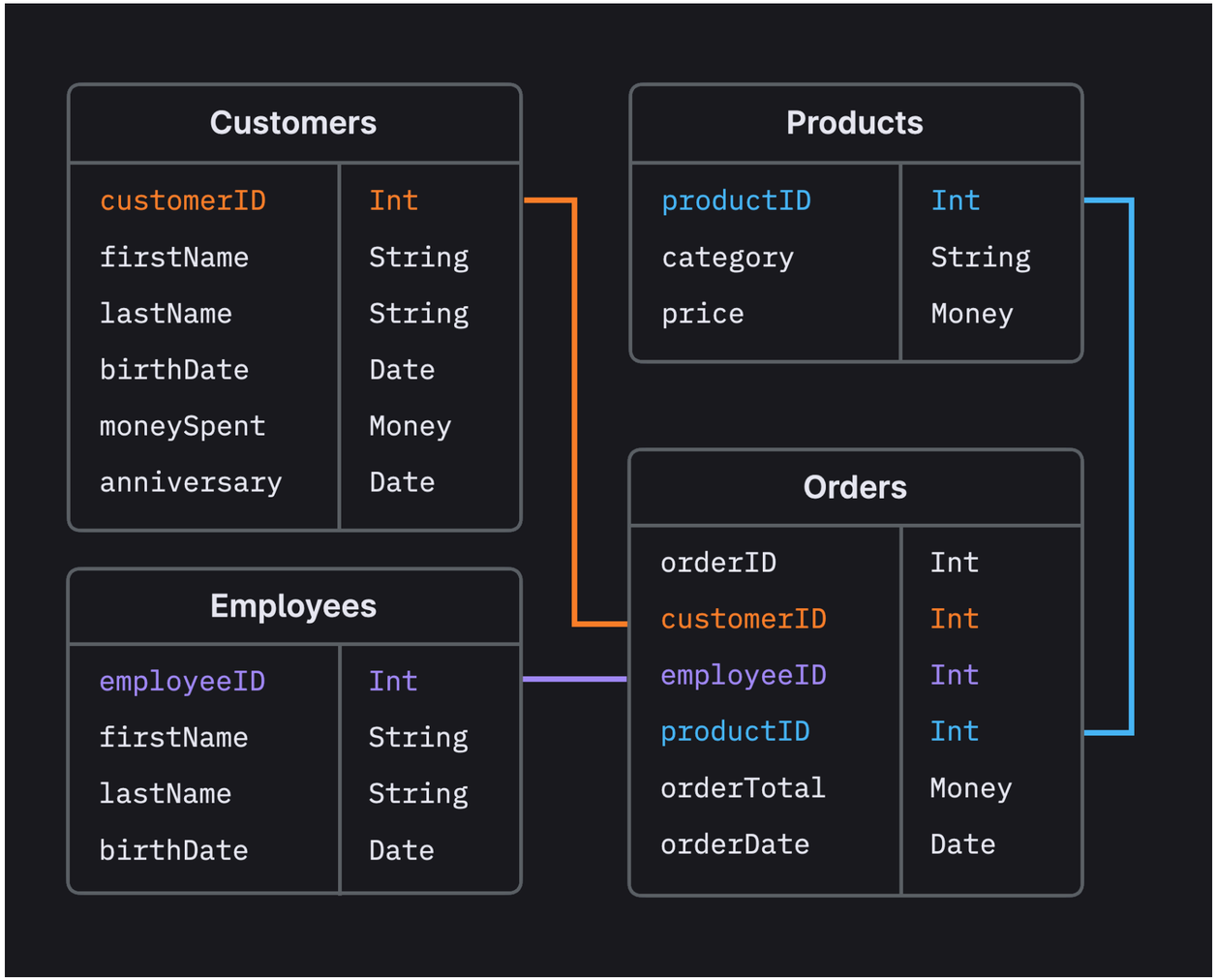



# ER Diagram Schema



# Relational Database Schema



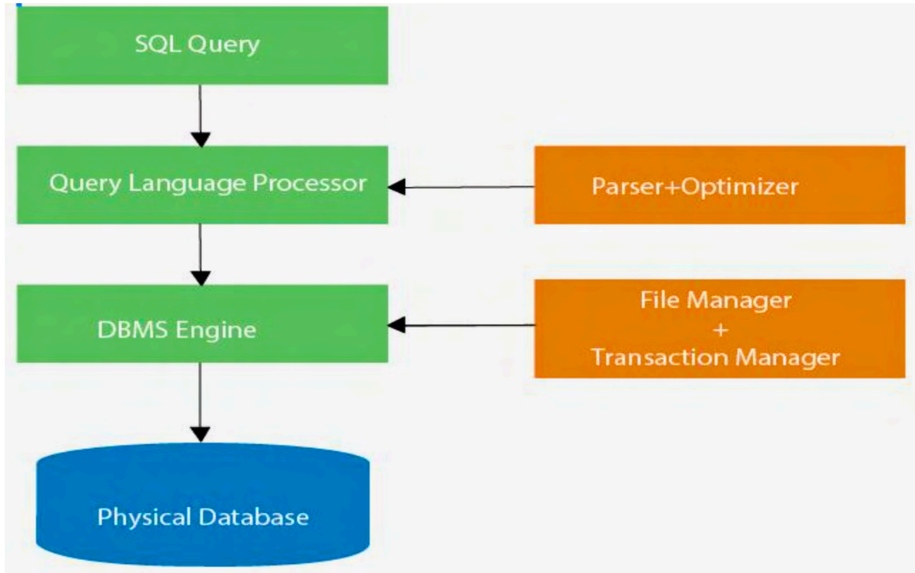
# SQL

Case Insensitive  Sequel  RDBMS

Structured Query Language is a programming language used to interact with database. SQL enables a user to create, read, update and delete relational databases, tables or rows

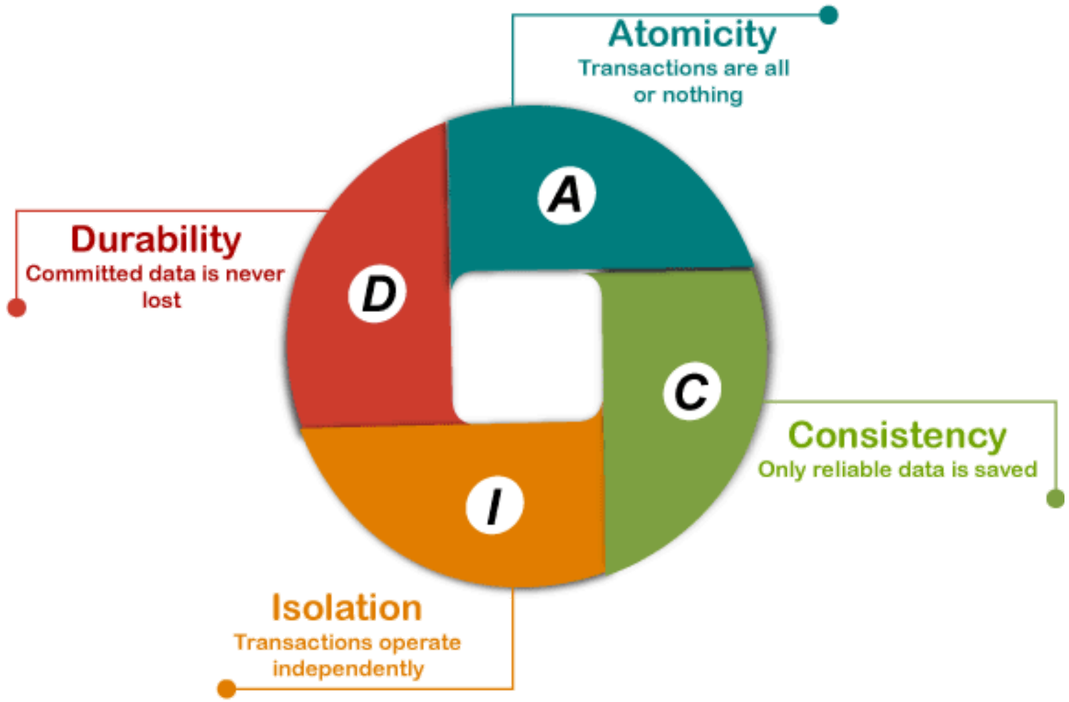
## Concept


## Working



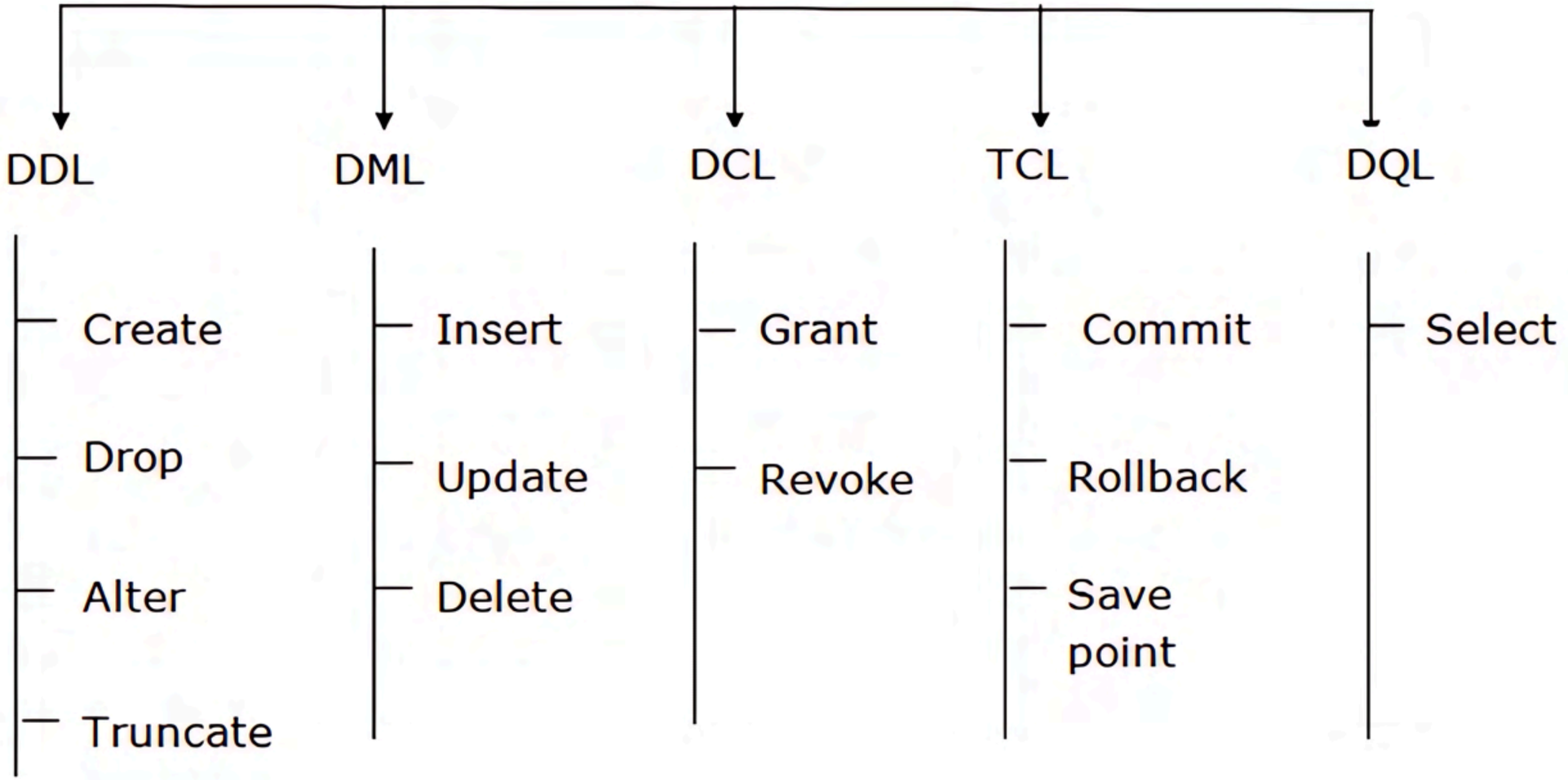
# ACID Properties

ensure reliable and secure processing of database transactions.




| Property | Definition | Key Points | Example |
|---|---|---|---|
| **Atomicity** | Ensures all operations within a transaction are completed successfully or none are. | - All-or-Nothing: The transaction is treated as a single unit.<br> - Rollback Mechanism: Rollback on failure. | **Bank Transfer:** If transferring money from Account A to Account B, both deduction from A and addition to B must succeed. If either fails, both operations are rolled back. |
| **Consistency** | Ensures that a transaction brings the database from one valid state to another, maintaining database invariants. | - Integrity Constraints: Must be maintained. <br> - Valid States: Database must be consistent before and after the transaction. | **Bank Transfer:** If Account A does not have enough balance, the transaction should fail, ensuring the database remains consistent. |
| **Isolation** | Ensures that operations of one transaction are isolated from those of other transactions. | - Concurrency Control: Multiple transactions do not affect each other. <br> - Isolation Levels: Control visibility of intermediate states. | **Concurrent Transactions:** If two transactions update the same account balance simultaneously, isolation ensures each transaction is unaware of the other until it is completed, preventing data corruption. |
| **Durability** | Ensures that once a transaction has been committed, it will remain so even in the event of a system failure. | - Persistence: Changes are permanently written. <br> - Recovery Mechanism: Use transaction logs for recovery. | **System Crash Recovery:** After transferring money and committing the transaction, the changes should be permanent. Even if the system crashes immediately after, the changes are recoverable and not lost. |

# SQL Commands



# DDL Data Definition Language

## Used to define structure of databases and their objects (CREATE, DROP, ALTER, RENAME, TRUNCATE)

- **CREATE TABLE:**
  - Used to create a new table in the database.
  - Specifies the table name, column names, data types, constraints, and more.
  - Example:
    CREATE TABLE employees (id INT PRIMARY KEY, name VARCHAR(50), salary DECIMAL(10, 2));
- **ALTER TABLE:**
  - Used to modify the structure of an existing table.
  - You can add, modify, or drop columns, constraints, and more.
  - Example: ALTER TABLE employees ADD COLUMN email VARCHAR(100);
- **DROP TABLE:**
  - Used to delete an existing table along with its data and structure.
  - Example: DROP TABLE employees;
- **DROP CONSTRAINT:**
  - Used to remove an existing constraint from a table.
  - Example: ALTER TABLE orders DROP CONSTRAINT fk_customer;
- **CREATE INDEX:**
  - Used to create an index on one or more columns in a table.
  - Improves query performance by enabling faster data retrieval.
  - Example: CREATE INDEX idx_employee_name ON employees (name);
- **CREATE CONSTRAINT:**
  - Used to define constraints that ensure data integrity.
  - Constraints include PRIMARY KEY, FOREIGN KEY, UNIQUE, NOT NULL, and CHECK.
  - Example: ALTER TABLE orders ADD CONSTRAINT fk_customer FOREIGN KEY (customer_id) REFERENCES customers(id);
- **DROP INDEX:**
  - Used to remove an existing index from a table.
  - Example: DROP INDEX idx_employee_name;
- **TRUNCATE TABLE:**
  - Used to delete the data inside a table, but not the table itself.
  - Syntax – TRUNCATE TABLE table_name

# DDL All table commands

```
/*Table Commands*/

Create Table Tableu ([Name] Varchar(50), ID int)  --Creating a Table

Insert into Tableu([Name],ID)
Values('Tajamul',1)   --Inserting Values into a Table

Alter Table Tableu Add Constraint XYZ unique(Tableu) --Adding Constraint into Column

Alter Table Tableu Add New_Column int --Adding New Column

Alter Table Tableu alter Column ID bigint --changing Data Type of a Column

Alter Table Tableu Drop Column New_Column --Dropping a Column

Drop Table Tableu --Dropping a Table

sp_RENAME 'Tableu','Tableu_New' --Renaming Table

sp_RENAME 'Tableu_New.ID', 'Super_ID' --Renaming Column

Select * from Tableu_New --Viewing Table Contents
```

We Can’t Add Null Constraint in A Filled Column but we can add Null Constraint in empty Column

```
Truncate Table #TB --Truncate Is Used to Delete All the Records from Table
```

# DML Data Manipulation Language

Used to manipulate data within database.
(INSERT, UPDATE, DELETE)

- **INSERT:**

- The INSERT statement adds new records to a table.
- Syntax: INSERT INTO table_name (column1, column2, ...) VALUES (value1, value2, ...);

- Example: INSERT INTO employees (first_name, last_name, salary) VALUES ('John', 'Doe', 50000);

- **UPDATE**:

- The UPDATE statement modifies existing records in a table.
- Syntax: UPDATE table_name SET column1 = value1, column2 = value2, ... WHERE condition;
- Example: UPDATE employees SET salary = 55000 WHERE first_name = 'John';

- **DELETE**:

- The DELETE statement removes records from a table.
- Syntax: DELETE FROM table_name WHERE condition;
- Example: DELETE FROM employees WHERE last_name = 'Doe';

# DCL Data Control Language

DCL is an important in ensuring database security by controlling access and permission. (GRANT, REVOKE)

**1. GRANT:**

The GRANT command is used to provide specific privileges or permissions to users or roles. Privileges can include the ability to perform various actions on tables, views, procedures, and other database objects.

Syntax:

```
GRANT privilege_type
ON object_name
TO user_or_role;
```

In this syntax:

- privilege_type refers to the specific privilege or permission being granted (e.g., SELECT, INSERT, UPDATE, DELETE).
- object_name is the name of the database object (e.g., table, view) to which the privilege is being granted.
- user_or_role is the name of the user or role that is being granted the privilege.

Example: Granting SELECT privilege on a table named "Employees" to a user named "Analyst":

```
GRANT SELECT ON Employees TO Analyst;
```

**2. REVOKE:**

The REVOKE command is used to remove or revoke specific privileges or permissions that have been previously granted to users or roles.

Syntax:

```
REVOKE privilege_type
ON object_name

FROM user_or_role;
```

In this syntax:

- privilege_type is the privilege or permission being revoked.
- object_name is the name of the database object from which the privilege is being revoked.
- user_or_role is the name of the user or role from which the privilege is being revoked.

Example: Revoking the SELECT privilege on the "Employees" table from the "Analyst" user:

```
REVOKE SELECT ON Employees FROM Analyst;
```

# TCL Transaction Control Language

A transaction in SQL is a sequence of one or more SQL statements that are executed as a single unit of work. TCL commands are used to initiate, execute, and terminate transactions. (COMMIT, ROLLBACK, SAVEPOINT)

**1. COMMIT:**

The COMMIT command is used to permanently save the changes made during a transaction.

It makes all the changes applied to the database since the last COMMIT or ROLLBACK command permanent.
Once a COMMIT is executed, the transaction is considered successful, and the changes are made permanent.

Example: Committing changes made during a transaction:

```
UPDATE Employees
SET Salary = Salary * 1.10
WHERE Department = 'Sales';

COMMIT;
```

**2. ROLLBACK:**

The ROLLBACK command is used to undo changes made during a transaction.
It reverts all the changes applied to the database since the transaction began.

ROLLBACK is typically used when an error occurs during the execution of a transaction, ensuring that the database remains in a consistent state.

Example: Rolling back changes due to an error during a transaction:

```
BEGIN;

UPDATE Inventory
SET Quantity = Quantity - 10
WHERE ProductID = 101;

-- An error occurs here

ROLLBACK;
```

**3. SAVEPOINT:**

The SAVEPOINT command creates a named point within a transaction, allowing you to set a point to which you can later ROLLBACK if needed.

SAVEPOINTs are useful when you want to undo part of a transaction while preserving other changes.

Syntax: SAVEPOINT savepoint_name;

Example: Using SAVEPOINT to create a point within a transaction:

```
BEGIN;
UPDATE Accounts
SET Balance = Balance - 100
WHERE AccountID = 123;

SAVEPOINT before_withdrawal;

UPDATE Accounts
SET Balance = Balance + 100
WHERE AccountID = 456;

-- An error occurs here

ROLLBACK TO before_withdrawal;

-- The first update is still applied

COMMIT;
```

# DQL

Data Query Language

used to retrieve data from databases
(SELECT)

- **SELECT:**

The SELECT statement is used to select data from a database.

Syntax: SELECT column1, column2, ... FROM table_name;

Here, column1, column2, ... are the field names of the table.

If you want to select all the fields available in the table, use the following syntax:
SELECT * FROM table_name;

Ex: SELECT CustomerName, City FROM Customers;

Select is used with where which uses Operators

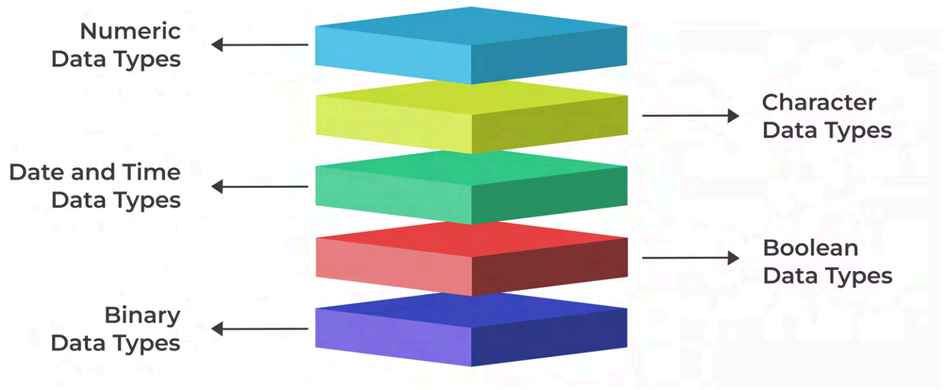
Where used to filter records

```
SELECT ID, NAME, SALARY
FROM CUSTOMERS
WHERE SALARY > 2000;
```

# Data Types

What type of data, a column can hold.

- Character
- Numeric
- Data and Time
- Boolean




## Character Data Type

| Data Type | Maximum Length |
|---|---|
| CHAR(n) | Fixed length, up to 255 characters |
| VARCHAR(n) | Variable length, up to 255 characters |
| TEXT | Variable length, up to 65,535 characters |

VARCHAR is memory efficient

| Attribute | UNICODE | NON-UNICODE |
|---|---|---|
| **Languages Supported** | All languages (including English) | Only English |
| **Character Size** | 1 Char = 2 Bytes | 1 Char = 1 Byte |
| **Fixed Length Data** | nChar (e.g., nChar(4000)) | Char (e.g., Char(8000)) |
| **Varying Length Data** | nVarchar (e.g., nVarchar(4000)) | Varchar (e.g., Varchar(8000)) |
| **Variable Length Data** | nVarchar (MAX) = Variable length up to 1GB | Varchar (MAX) = Variable length up to 2GB |

# Numeric Data Type

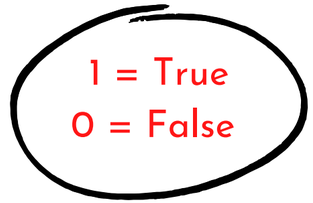
| Data Type | Length (Digits) |
|---|---|
| TINYINT | 3 digits (-128 to 127) |
| SMALLINT | 5 digits (-32,768 to 32,767) |
| INT | 10 digits (-2,147,483,648 to 2,147,483,647) |
| BIGINT | 19 digits (-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807) |
| DECIMAL(S,D) | Variable length, depends on precision and scale |

# Date Time Data Type

| Data Type | Definition | Format |
|---|---|---|
| TIME | Specific time of day without time zone info. | HH:MM |
| DATE | Date (year, month, day) without time zone info. | YYYY-MM-DD |
| YEAR | Year in 4-digit format. | YYYY |
| TIMESTAMP | Date and time without time zone info. | YYYY-MM-DD HH:MM |
| TIMESTAMPZ | Date and time with time zone info. | YYYY-MM-DD HH:MM ±HH |

# Boolean Data Type (Bit)

| Database | Boolean? | Use Instead |
|---|---|---|
| Oracle | No | NUMBER(1) |
| SQL Server | No | BIT |
| MySQL | No | BIT or TINYINT |
| PostgreSQL | Yes | |



# Float vs Decimal

Precision is the number of digits in a number. Scale is the number of digits to the right of the decimal point in a number. For example, the number 123.45 has a precision of 6 and a scale of 2.

## Float

it gives the approximate value of the stored number. Rounds up to

*FLOAT(precision)*

```
-- Create the table
CREATE TABLE STUDENT(
    Name VARCHAR(20),
    Result FLOAT(5)
);
```

## Decimal

it has the fixed number of digits after the decimal point.

*DECIMAL(precision, scale)*

```
-- Create the table
CREATE TABLE STUDENT(
    Name VARCHAR(20),
    Result DECIMAL(5, 2)
);
```

# Identity Column

Series in Post gre

Identity column of a table is a column whose value increases automatically. Identity column can be used to uniquely identify the rows in the table.

```
IDENTITY [( seed, increment)]


Seed: Starting value of a column.
      Default value is 1.

Increment: Incremental value that is
added to the identity value of the previous
row that was loaded. The default value 1.
```

**Example**

```
CREATE TABLE GEEK_6
(
[ID]    INT IDENTITY(1,1),
[NAME]  VARCHAR(50)
)
INSERT INTO GEEK_6 VALUES('Geek');
INSERT INTO GEEK_6 VALUES('Geeks');
INSERT INTO GEEK_6 VALUES('GeeksF');
INSERT INTO GEEK_6 VALUES('GeeksFo');
INSERT INTO GEEK_6 VALUES('GeeksFor');
INSERT INTO GEEK_6 VALUES('GeeksForG');
```

| | ID | NAME |
|---|---|---|
| 1 | 1 | Geek |
| 2 | 2 | Geeks |
| 3 | 3 | GeeksF |
| 4 | 4 | GeeksFo |
| 5 | 5 | GeeksFor |

**Note**

The 'ID' column of the table starts from 1 as the seed value provided is 1 and is incremented by 1 at each row.

# Constraints

Constraints are rules used to limit the type of data entering the columns. It ensures accuracy and reliability of the data

***Not Null***
Ensures that a column cannot have NULL value

***Default***
Provides a default value for a column when none is specified.

***Unique*** Can have NULL Value
Ensures that all rows in a column are different.

***Primary*** Can't have NULL Value
Uniquely identifies each row/record in a database table

***Foreign***
Ensures column can be referenced by Primary key

***Check***
Ensures that all the values in a column satisfies certain conditions.

```
Create Table Customers(
ID INT NOT NULL,
Salary DECIMAL(5,2) DEFAULT(5000),
FingerprintID INT UNIQUE,
PRIMARY KEY (ID),
FOREIGN KEY SID INT references ZIP(ID),
Age INT CHECK(AGE > 18)
```

# Errors in SQL

### Syntax Error

When SQL statements do not follow the correct syntax and structure of the language

### Semantic Error

Provides a default value for a column when none is specified.

### Constraint Voilation

When SQL violates one or more constrainsts on the database

### Datatype Error

When trying to insert non matching data type

### Transaction Error

Problem that occurs during the execution of a transaction. A transaction in SQL is a sequence of one or more SQL statements that are treated as a single unit of work

# Keys

"Key" refers to a column or set of columns in a table that uniquely identify each row within that table

- To uniquely identify a row
- To enforce Data integrity & Constraints
- To establish relationship between multiple tables in the database

## Types

- Primary Key
- Foreign Key
- Unique Key
- Composite Key
- Alternate Key
- Candidate
- Super Key

## Primary Key

- A primary key uniquely identifies each record in a table.
- It must contain unique values and cannot contain NULL values.
- Only one primary key is allowed per table.

```
CREATE TABLE employees (employee_id INT PRIMARY KEY);
```

## Foreign Key

- A foreign key establishes a link between two tables, by referencing the primary key in another.
- It ensures referential integrity by enforcing a relationship between the tables.

```
CREATE TABLE orders ( order_id INT PRIMARY KEY, customer_id INT,
FOREIGN KEY (customer_id) REFERENCES customers(customer_id) );
```

## Unique Key

- A unique key ensures that all values in a column or a group of columns are unique.
- Unlike primary keys, unique keys can contain NULL values (2 NULL are not same)

```
CREATE TABLE students ( student_id INT UNIQUE, ... );
```

## Composite Key

- A composite key consists of multiple columns that together uniquely identify a record in a table.
- It's useful when a single column cannot uniquely identify records, but a combination of columns can.

```
CREATE TABLE orders ( order_id INT, product_id INT,
PRIMARY KEY (order_id, product_id) );
```

## Alternate Key

- An alternate key is a candidate key that is not selected as the primary key.
- It could serve as a **unique** identifier if the primary key didn't exist.

```
CREATE TABLE students ( student_id INT PRIMARY KEY,
email VARCHAR(50) UNIQUE, ... );
```

## Candidate Key minimal

Minimal set of columns that uniquely identifies each row in a table

*The combination of EmployeeID and Email uniquely identifies each employee*

## Super Key not minimal

Set of columns that uniquely identifies each row in a table It may contain more columns than necessary for uniqueness.

*{EmployeeID, Email, SSN} also uniquely identifies each employee. This set is a super key because it goes beyond the minimal requirement for uniqueness.*

In summary, while both candidate keys and super keys ensure uniqueness in a table, candidate keys are minimal sets of columns fulfilling this requirement, while super keys can include more columns than necessary.

# Anomalies

Deficiency in database design which lead to data redundancy, and integrity.

| Anomaly Type | Definition | Example |
|---|---|---|
| **Insertion Anomaly** | You can't add new information to a table without including unrelated data. | Imagine a table to record student grades with columns for StudentID, Name, and Grade. If a new student joins but hasn't been graded yet, you can't add their information without leaving the Grade column empty or duplicating the student's information. |
| **Update Anomaly** | Changing information in a table leads to inconsistencies because related data is not updated uniformly. | Suppose you have a table recording student grades with columns for StudentID, Name, and Grade. If a student's name changes but the change is only made in some rows and not others, it leads to inconsistency in the data. |
| **Deletion Anomaly** | Removing data from a table unintentionally removes other related data. | Consider a table recording student grades with columns for StudentID, Name, and Grade. If you delete a row corresponding to a student's grade, you might unintentionally lose information about the student. |

# Normalisation

Normalisation helps improve database design by following NFs which ensures

- reduce redundancy,
- ensure data integrity.
- prevent anomalies,

## Types

- 1NF
- 2NF
- 3NF
- BCNF
- 4NF
- DKNF

| Normal Form | Description |
|---|---|
| 1NF | A relation is in 1NF if it contains an Atomic Value. |
| 2NF | A relation will be in 2NF if it is in 1NF and all non-key attributes are fully functional dependent on the primary key.No Partial Dependency |
| 3NF | A relation will be in 3NF if it is in 2NF and No Transition Dependency exists. |
| BCNF | A stronger definition of 3NF is known as Boyce Codd's normal form. No Functional Dependency |
| 4NF | A relation will be in 4NF if it is in Boyce Codd's normal form and has no multi-valued dependency. |
| 5NF | A relation is in 5NF. If it is in 4NF and does not contain any join dependency, joining should be lossless. |

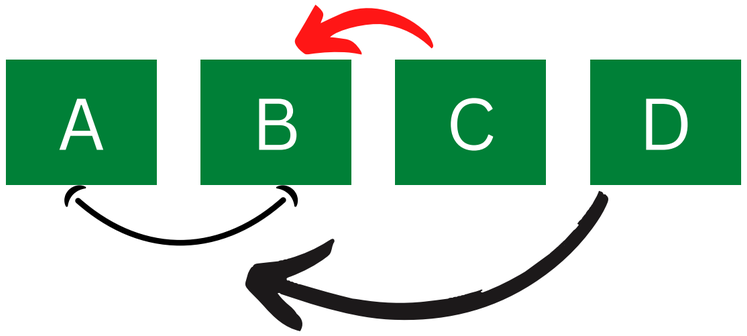
## 1NF Atomic Value

- In 1NF, all the rows in a column must have **atomic** values with consistent data types

| EmployeeName | Project |
|---|---|
| Alice | Project X |
| Bob | Project Y |
| Carol | Project X |

## 2NF No Partial Dependency

- In 2NF, the table must be in 1NF first.
- It means if a table has composite primary keys, each non-prime attribute should be dependent on the entire composite key, not just on part of it.



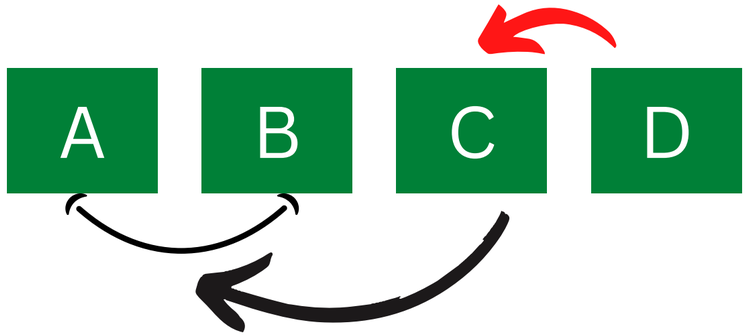

A B **PRIME ATTRIBUTE**
Composite Key

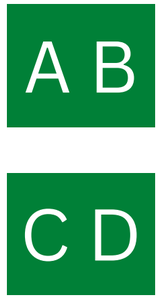
C D **NON PRIME ATTRIBUTE**

- Here D is dependent on canditate key
- But C is dependent on only B which is subset of canditate key and that is known as partial dependency

## 3NF No Transitive Dependency

- In 3NF, the table must be in 2NF first.
- Every non-prime attribute should be non-transitively dependent on the primary key.
- This means that no column should depend on another non-key column.




- Here c is dependent on canditate key
- But D is dependent on C which is also non-prime attribute and that is known as transitive dependency

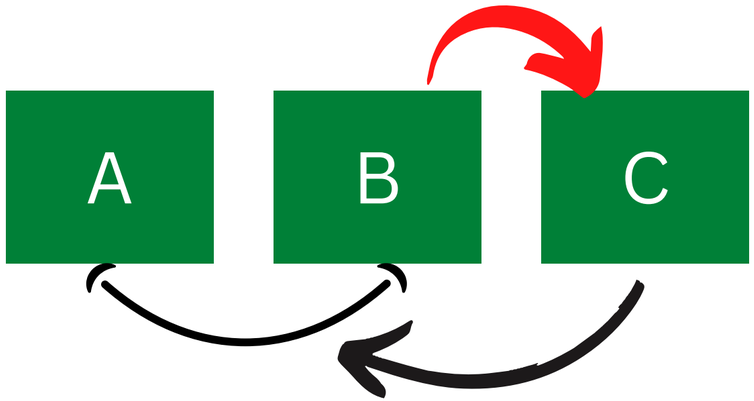
# BCNF

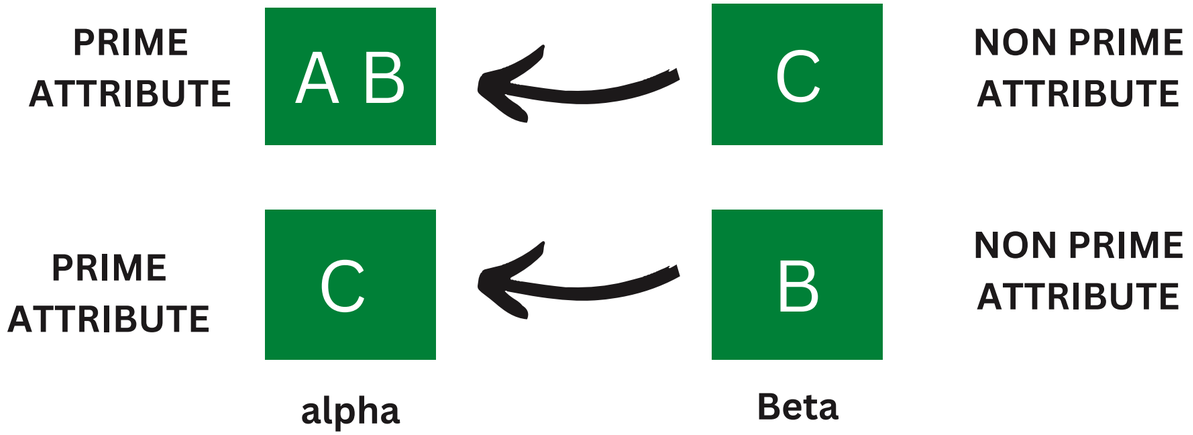
Every Non Prime is fully dependent on Candidate keys

No Functional Dependency

- Boyce Codd Normal form (BCNF) AKA 3.5NF
- BCNF is the advance version of 3NF. It is stricter than 3NF.
- A table is in BCNF if X depends on Y, Y is the super key of the table.

In simple words, BCNF means every piece of information in a table should only depend on the primary key. If it depends on anything else, it needs its own table. This keeps the database organized and prevents data duplication



## Advantages of Normalisation

- efficient database design
- reduce redundancy,
- ensure data integrity.
- prevent anomalies,

## Disdvantages of Normalisation

- The performance degrades when normalizing the relations to higher normal forms, i.e., 4NF, 5NF, DKNF.
- Time-consuming and difficult to normalize relations of a higher degree.
- Careless decomposition may lead to a bad database design, leading to serious problems.

# Normalisation vs Denormalisation

| Aspect | Normalization | Denormalization |
|---|---|---|
| **Definition** | Organizes data to minimize redundancy by splitting it into related tables. | Combines related data into fewer tables to reduce the need for joins. |
| **Data Redundancy** | Minimal redundancy, data is stored in separate, related tables. | Increased redundancy, with duplicate data across tables. |
| **Storage Efficiency** | Efficient storage, less data duplication. | Less efficient storage, more data duplication. |
| **Query Complexity** | More complex queries due to multiple joins. | Simpler queries with fewer joins needed. |
| **Read Performance** | Potentially slower read performance due to joins. | Faster read performance, as joins are minimized or eliminated. |
| **Write Performance** | Faster writes, fewer updates needed as data is not duplicated. | Slower writes, as duplicated data requires more updates. |
| **Data Integrity** | High data integrity, easier to enforce constraints and maintain accuracy. | Lower data integrity, more challenging to maintain consistency. |
| **Use Cases** | Transactional systems, where data integrity and write performance are crucial. | Analytical systems, reporting, read-heavy applications where performance is key. |

# Operators

The SQL reserved words and characters used with a WHERE clause in a SQL query

| Operator Group | Operators |
|---|---|
| Arithmetic Operators | +, -, *, /, % |
| Comparison Operators | =, <>, <>, >, <, >=, <= |
| Logical Operators | AND, OR, NOT |
| Concatenation Operator | + (SQL Server, MySQL) |
| Wildcard Operators | %, - |
| IN and NOT IN Operators | IN, NOT IN |
| BETWEEN Operator | BETWEEN |
| IS NULL and IS NOT NULL Operators | IS NULL, IS NOT NULL |

# Arithmetic Operator

| Operator | Description | SQL Query Example |
|---|---|---|
| + | Addition | `` `SELECT column1 + column2 AS sum_result FROM table_name;` `` |
| - | Subtraction | `` `SELECT column1 - column2 AS difference FROM table_name;` `` |
| * | Multiplication | `` `SELECT column1 * column2 AS product FROM table_name;` `` |
| / | Division | `` `SELECT column1 / column2 AS quotient FROM table_name;` `` |
| % | Modulus | `` `SELECT column1 % column2 AS remainder FROM table_name;` `` |

# Comparison Operator

| Operator | Description | SQL Query Example |
|---|---|---|
| = | Equal to | `` `SELECT * FROM table_name WHERE column1 = 10;` `` |
| <> | Not equal to | `` `SELECT * FROM table_name WHERE column1 <> 'value';` `` |
| != | Not equal to | `` `SELECT * FROM table_name WHERE column1 != 'value';` `` |
| > | Greater than | `` `SELECT * FROM table_name WHERE column1 > 10;` `` |
| < | Less than | `` `SELECT * FROM table_name WHERE column1 < 10;` `` |
| >= | Greater than or equal to | `` `SELECT * FROM table_name WHERE column1 >= 10;` `` |
| <= | Less than or equal to | `` `SELECT * FROM table_name WHERE column1 <= 10;` `` |

## Logical Operator

| Operator | Description | SQL Query Example |
|---|---|---|
| AND | Logical AND | `` `SELECT * FROM table_name WHERE column1 > 10 AND column2 < 20;` `` |
| OR | Logical OR | `` `SELECT * FROM table_name WHERE column1 = 'value' OR column2 = 'value';` `` |
| NOT | Logical NOT | `` `SELECT * FROM table_name WHERE NOT column1 = 'value';` `` |

## Concatenation Operator

| Operator | Description | SQL Query Example |
|---|---|---|
| + | Concatenation (SQL Server, MySQL) | `` `SELECT column1 + ' ' + column2 AS full_name FROM table_name;` `` |

# Wildcard Operator

| Operator | Description | SQL Query Example |
|---|---|---|
| % | Matches any string of zero or more characters | `SELECT * FROM table_name WHERE column1 LIKE 'J%';` |
| _ | Matches any single character | `SELECT * FROM table_name WHERE column1 LIKE '_0%';` |

## Using %

Match any string starting with 'A':

```sql
SELECT * FROM table_name WHERE column1 LIKE 'A%';
```

Match any string ending with 'ing':

```sql
SELECT * FROM table_name WHERE column1 LIKE '%ing';
```

Match any string containing 'at':

```sql
SELECT * FROM table_name WHERE column1 LIKE '%at%';
```

## Using _

Match any string starting with 'A' and having exactly 3 characters:

```sql
SELECT * FROM table_name WHERE column1 LIKE 'A__';
```

Match any string having 'a' as the second character:

```sql
SELECT * FROM table_name WHERE column1 LIKE '_a%';
```

Match any string having exactly 5 characters and starting with 'A':

```sql
SELECT * FROM table_name WHERE column1 LIKE 'A____';
```

## Membership Operator

| Operator | Description | SQL Query Example |
|---|---|---|
| IN | Matches any value in a specified list | `` `SELECT * FROM table_name WHERE column1 IN ('value1', 'value2', 'value3');` `` |
| NOT IN | Matches any value not in a specified list | `` `SELECT * FROM table_name WHERE column1 NOT IN ('value1', 'value2', 'value3');` `` |

## Between Operator

| Operator | Description | SQL Query Example |
|---|---|---|
| BETWEEN | Matches a range of values | `` `SELECT * FROM table_name WHERE column1 BETWEEN 10 AND 20;` `` |

## Is Null and Is Not Null Operator

| Operator | Description | SQL Query Example |
|---|---|---|
| IS NULL | Matches NULL values | `` `SELECT * FROM table_name WHERE column1 IS NULL;` `` |
| IS NOT NULL | Matches non-NULL values | `` `SELECT * FROM table_name WHERE column1 IS NOT NULL;` `` |

# SQL Clauses

The SQL clauses are foundational elements of a SQL Query

| Clause | Name | Definition | SQL Query Example |
|---|---|---|---|
| SELECT | Query | Retrieves data from one or more tables or expressions, optionally performing calculations or transformations. | `` `SELECT column1, column2 FROM table_name;` `` |
| FROM | Source | Specifies the table or tables from which data is retrieved in a SELECT statement. | `` `SELECT * FROM table_name;` `` |
| WHERE | Filter | Filters rows based on a specified condition in a SELECT statement. | `` `SELECT * FROM table_name WHERE column1 = 'value';` `` |
| GROUP BY | Group | Groups rows sharing a common value into summary rows in a SELECT statement. | `` `SELECT column1, COUNT(*) FROM table_name GROUP BY column1;` `` |
| HAVING | Group Filter | Filters groups of rows returned by a GROUP BY clause based on a specified condition. | `` `SELECT column1, COUNT(*) FROM table_name GROUP BY column1 HAVING COUNT(*) > 1;` `` |
| ORDER BY | Sort | Sorts the result set of a SELECT statement in ascending or descending order based on specified columns. | `` `SELECT * FROM table_name ORDER BY column1 ASC;` `` |
| LIMIT | Limit | Limits the number of rows returned by a SELECT statement (not directly supported in MS SQL Server). | `` `SELECT TOP 10 * FROM table_name;` `` |

| Clause | Name | Definition | SQL Query Example |
|---|---|---|---|
| OFFSET | Offset | Specifies the number of rows to skip before starting to return rows in a SELECT statement. | `` `SELECT * FROM table_name OFFSET 10 ROWS;` `` |
| FETCH | Fetch | Specifies the number of rows to return after skipping rows specified in the OFFSET clause. | `` `SELECT * FROM table_name ORDER BY column1 OFFSET 10 ROWS FETCH NEXT 10 ROWS ONLY;` `` |
| DISTINCT | Distinct | Filters duplicate rows from the result set of a SELECT statement. | `` `SELECT DISTINCT column1 FROM table_name;` `` |
| WITH | Common Table Expression | Defines a temporary named result set that can be used within the scope of a single SQL statement. | `` `WITH CTE AS (SELECT column1 FROM table_name) SELECT * FROM CTE;` `` |

# Alias

Aliases in SQL are used to provide temporary names

| Purpose | Query |
| --- | --- |
| Table Aliases | `SELECT e.employee_id, e.first_name, d.department_name FROM employees AS e JOIN departments AS d ON e.department_id = d.department_id;` |
| Column Aliases | `SELECT first_name AS "First Name", last_name AS "Last Name" FROM employees;` |
| Aliases for Derived Tables or Subqueries | `SELECT e.employee_id, e.first_name, e.salary, dept.avg_salary FROM employees AS e JOIN (SELECT department_id, AVG(salary) AS avg_salary FROM employees GROUP BY department_id) AS dept ON e.department_id = dept.department_id;` |
| Aliases for Aggregate Functions | `SELECT COUNT(*) AS total_employees FROM employees;` |

# Case Statement

A case statement is like an if-elif-else statement in programming, allowing different actions to be taken based on different conditions.

```
CASE
    WHEN condition1 THEN result1
    WHEN condition2 THEN result2
    ...
    WHEN conditionN THEN resultN
    ELSE default_result
END
```

- Efficient than If-Else
- Causes problems when dealing with NULL

## Example

```
SELECT customer_id,
CASE amount
WHEN 500 THEN 'Prime Customer'
WHEN 100 THEN 'Plus Customer'
ELSE 'Regular Customer'
END AS CustomerStatus
FROM payment
```

Data Blogs

Follow Tajamul Khan

# Intermediate

- Joins
- Set Operations
- Group By and Having clause
- Order of Execution
- Functions - Aggregate, Datetime, String
- Windows Function
- Sub-Query
- CTE table
- In-built functions
- Views
- Indexes
- Stored Procedures
- Triggers
- Temporary Tables

# Practicals

W3 schools

## Intermediate Level

stratascratch

LeetCode

DataLemur

# Joins

a join is an operation that joins rows from two or more tables based on a related column between them.

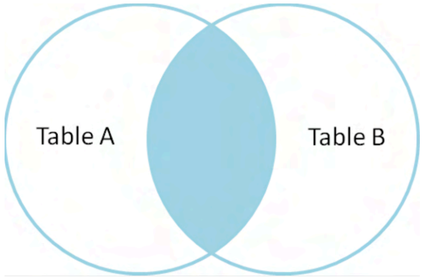
## Types

| Join Type | Description |
|---|---|
| INNER JOIN | Returns rows when there is at least one match in both tables. NULL values are not considered equal, so rows with NULL values from both tables will not be included in the result set. |
| LEFT JOIN (or LEFT OUTER JOIN) | Returns all rows from the left table and the matched rows from the right table. If there is no match, NULL values are returned for the missing values. |
| RIGHT JOIN (or RIGHT OUTER JOIN) | Returns all rows from the right table and the matched rows from the left table. If there is no match, NULL values are returned for the missing values. |
| FULL JOIN (or FULL OUTER JOIN) | Returns all rows from both tables. If there is no match, NULL values are returned for the missing values in the corresponding columns. |
| CROSS JOIN | Returns the Cartesian product of the two tables, combining every row of the first table with every row of the second table. |
| SELF JOIN | Joins a table to itself, typically used to compare rows within the same table based on specified conditions. |

## Inner Join

Returns only matching rows between both tables.

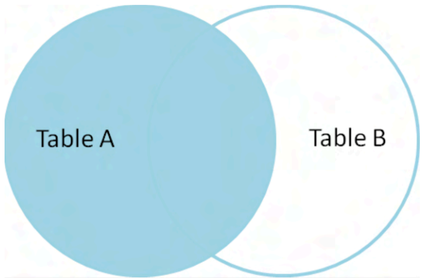
**Note:** NULL values are not considered equal, the rows with NULL values from both tables will not be matched with each other. Therefore, they will not be included in the result set of the INNER JOIN.




```
SELECT *
FROM customer AS c
INNER JOIN payment AS p
ON c.customer_id = p.customer_id
```

## Left Join

Returns all rows from the left table and the matched rows from the right table. If there is no match, NULL values are returned for the missing values.

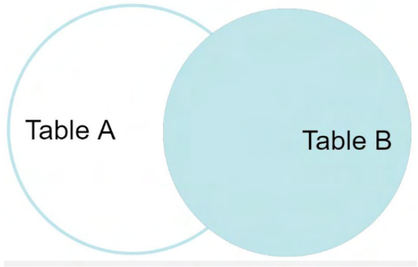



```
SELECT *
FROM customer AS c
LEFT JOIN payment AS p
ON c.customer_id = p.customer_id
```

We will not see null in Inner Join as Null can't be same

## Right Join

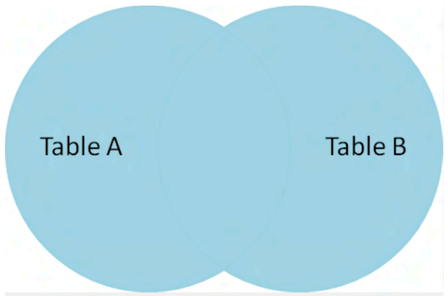
Returns all rows from the Right table and the matched rows from the Left table. If there is no match, NULL values are returned for the missing values.




```
SELECT *
FROM customer AS c
RIGHT JOIN payment AS p
ON c.customer_id = p.customer_id
```

## Full Join

Returns all records when there is a match in either left or right table. If there is no match, NULL values are returned for the missing values in the corresponding columns.

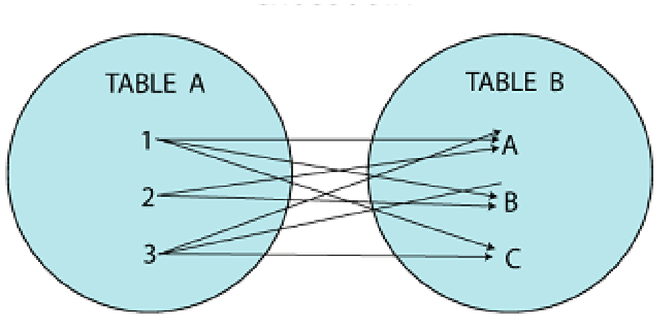



```
SELECT *
FROM customer AS c
FULL OUTER JOIN payment AS p
ON c.customer_id = p.customer_id
```

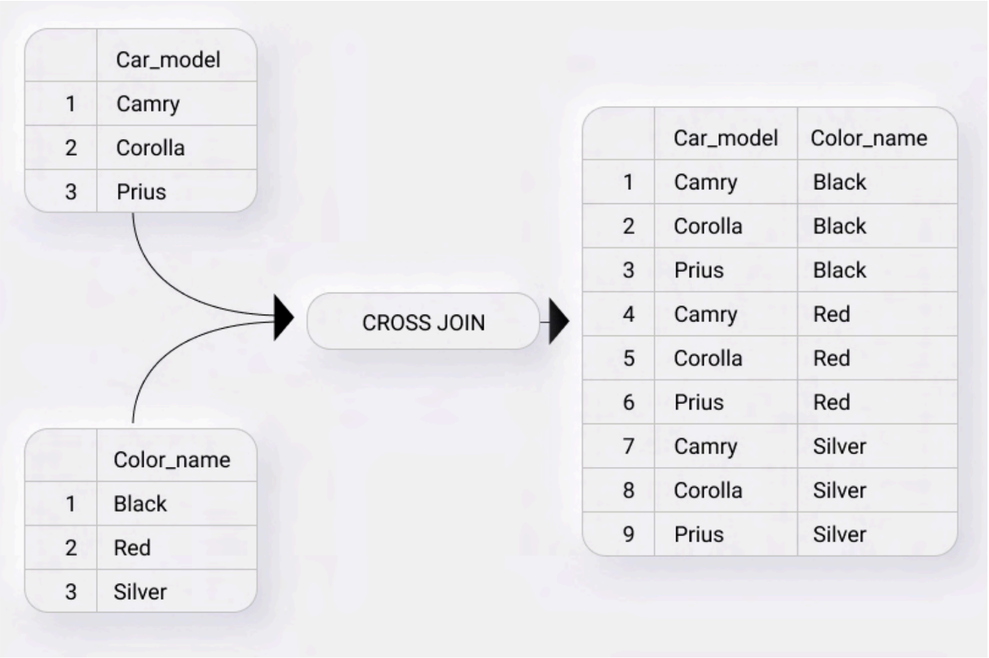
# Cross Join

It returns the Cartesian product of the two tables involved, meaning it combines every row of the first table with every row of the second table. In other words, it produces a result set where each row from the first table is paired with every row from the second table.

Use Case: Get all possible combinations




```
SELECT *
FROM customers
CROSS JOIN orders;
```



# Self Join

A join in which a table is joined to itself

Self Joins are powerful for comparing rows within the same table based on specified conditions.

When evaluating a hierarchy, the self join is utilized generally.

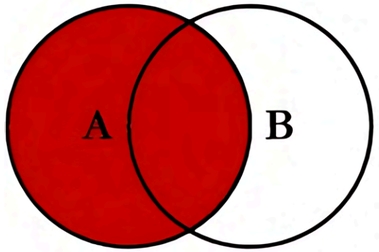
| Id | Full Name | Salary | Teamlead Id |
|---|---|---|---|
| 1 | Chirs Hemsworth | 200000 | 5 |
| 2 | Tom Holland | 250000 | 5 |
| 3 | Ben Affleck | 120000 | 1 |
| 4 | Christian Bale | 150000 | |
| 5 | Gal Gadot | 300000 | 4 |

```
SELECT
 member. Id,
 member.FullName,
 member.teamleadId,
 teamlead.FullName as
teamleadName
FROM members member
JOIN members teamlead
ON member.teamleadId =
teamlead.Id
```

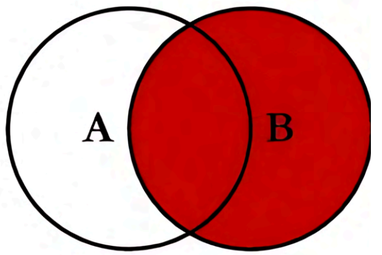
| Id | FullName | Teamlead Id | Teamlead Name |
|---|---|---|---|
| 1 | Chirs Hemsworth | 5 | Gal Galdot |
| 2 | Tom Holland | 5 | Gal Galdot |
| 3 | Ben Affleck | 1 | Chirs Hemsworth |
| 5 | Gal Galdot | 4 | Christian Bale |

# Joins Cheat Sheet

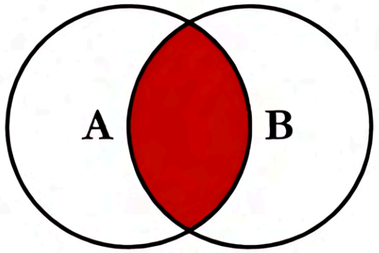
## SQL JOINS



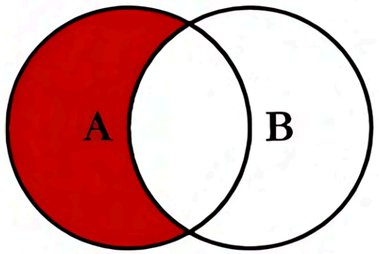

```
SELECT <select_list>
FROM TableA A
LEFT JOIN TableB B
ON A.Key = B.Key
```



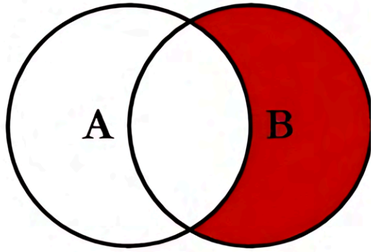

```
SELECT <select_list>
FROM TableA A
RIGHT JOIN TableB B
ON A.Key = B.Key
```



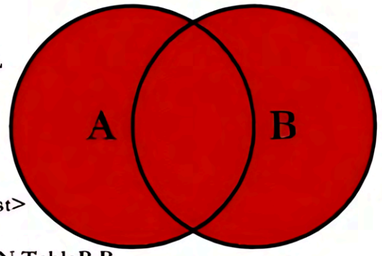

```
SELECT <select_list>
FROM TableA A
INNER JOIN TableB B
ON A.Key = B.Key
```



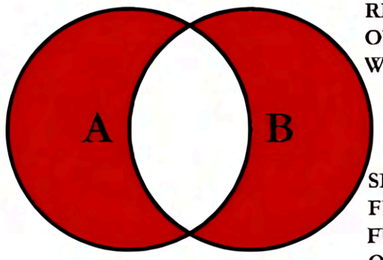

```
SELECT <select_list>
FROM TableA A
LEFT JOIN TableB B
ON A.Key = B.Key
WHERE B.Key IS NULL
```




```
SELECT <select_list>
FROM TableA A
RIGHT JOIN TableB B
ON A.Key = B.Key
WHERE A.Key IS NULL
```




```
SELECT <select_list>
FROM TableA A
FULL OUTER JOIN TableB B
ON A.Key = B.Key
```

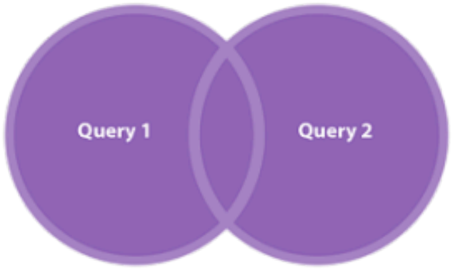



```
SELECT <select_list>
FROM TableA A
FULL OUTER JOIN TableB B
ON A.Key = B.Key
WHERE A.Key IS NULL
OR B.Key IS NULL
```

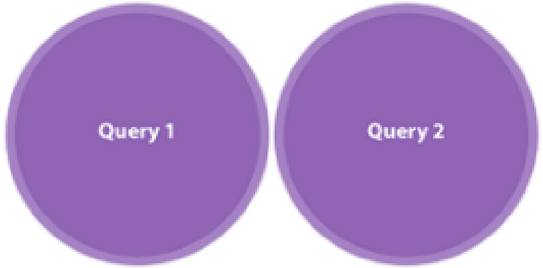
# SET Operations

are used to combine or manipulate the result sets of multiple SELECT queries.

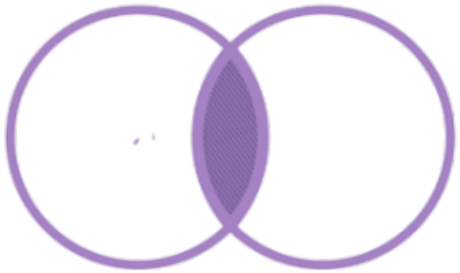
## Types



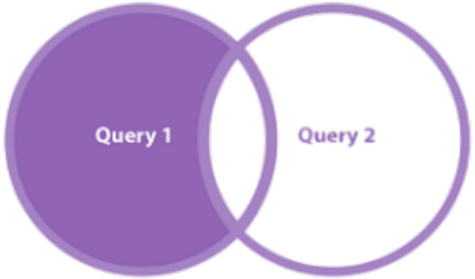

**Union All**




**Union**



**Intersect**



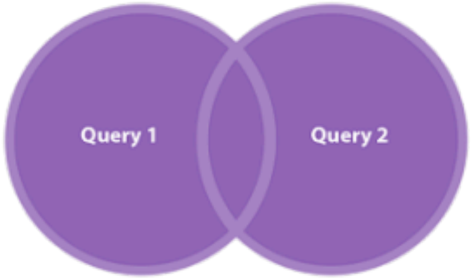

**Except (Minus)**

Set operations compare entire result sets, while joins compare specific columns based on relationships between tables.

## Union All

combines all the rows including duplicates from result sets of two or more SELECT queries.

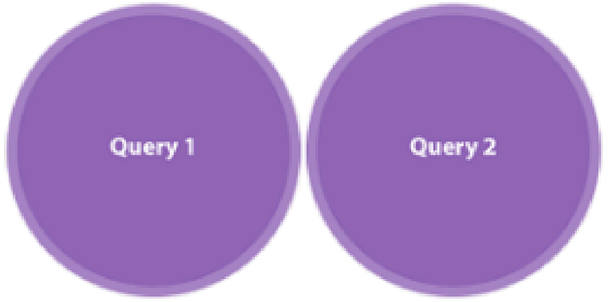
```
SELECT CustomerName FROM
Customers UNION ALL
SELECT SupplierName FROM
Suppliers;
```




## Union

combines all the rows except duplicates from result sets of two or more SELECT queries.

```
SELECT CustomerName FROM
Customers UNION
SELECT SupplierName FROM
Suppliers;
```

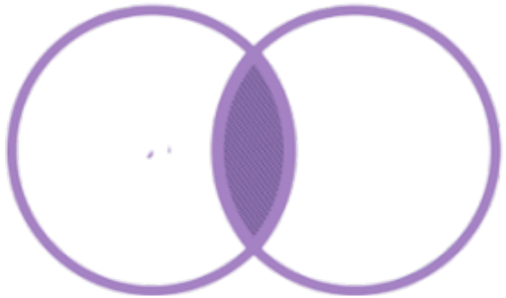


## Intersect

It returns the common rows that exist in the result sets of two or more SELECT queries.

- It only returns distinct that appear in all result sets.

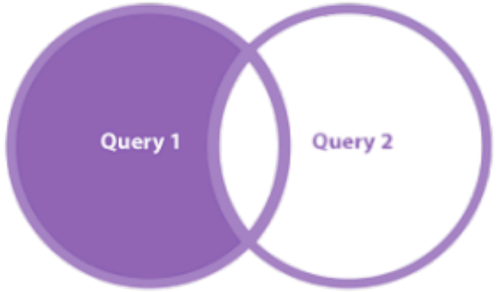
```
SELECT CustomerName FROM
Customers INTERSECT
SELECT SupplierName FROM
Suppliers;
```



## Except or Minus

It returns the distinct rows that are present in the result set of the first SELECT query but not in the result set of the second SELECT query.

```
SELECT CustomerName FROM
Customers EXCEPT
SELECT SupplierName FROM
Suppliers;
```



## Group by

It is used to group rows from one or more columns of a table

- Mostly used with **aggregate** functions

```
SELECT customer_id, SUM(quantity) AS total_quantity
FROM orders
GROUP BY customer_id;
```

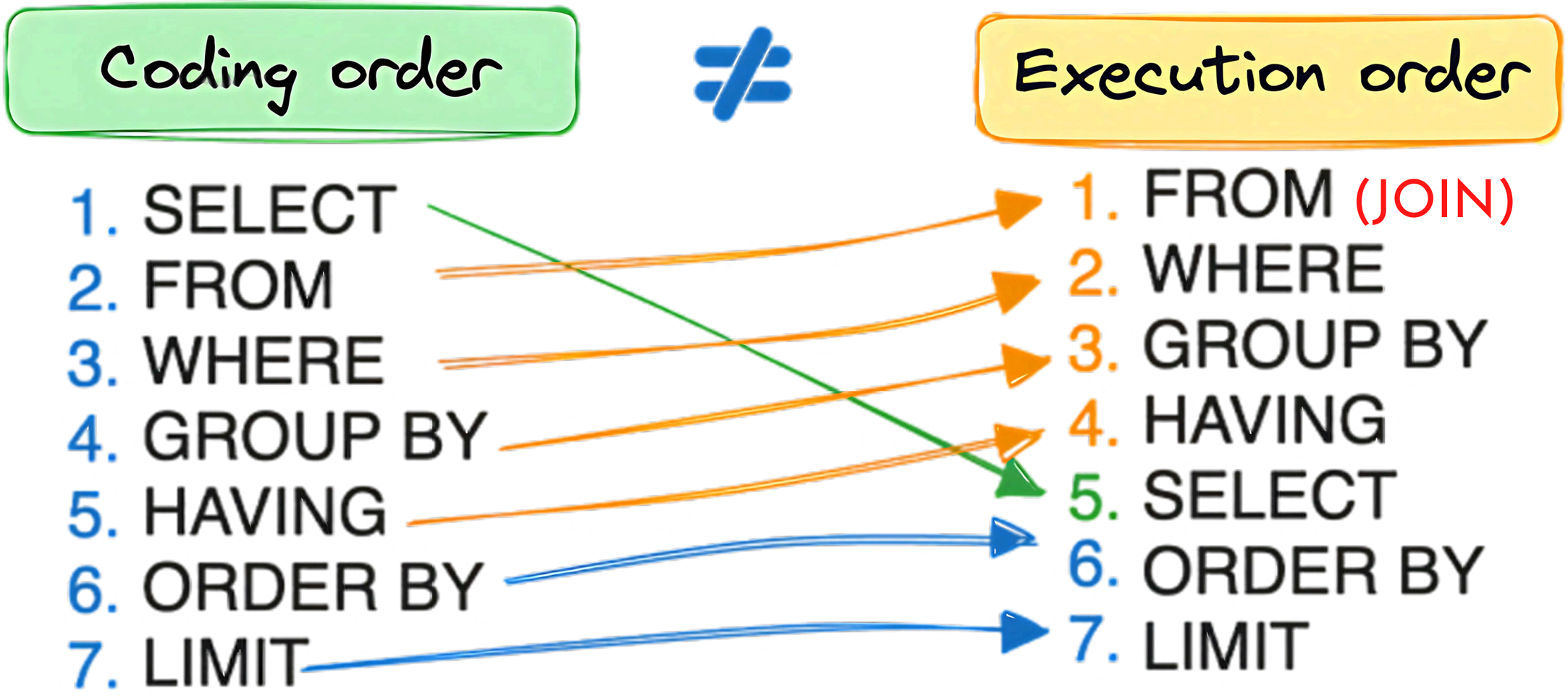
## Having

The HAVING clause is used to apply a **filter** on the results of GROUP BY

- Where Clause specifically for group by

```
SELECT customer_id, SUM(quantity)
FROM orders
GROUP BY customer_id
HAVING SUM(quantity) > 10;
```

# Order of Execution



# Aggregate Functions

performs a calculation on multiple values and returns a single value.

- These are often used with GROUP BY & SELECT statement

| Aggregate Function | Description | Usage in Query |
|---|---|---|
| COUNT | Returns the number of rows in a group. | SELECT COUNT(column_name) AS count_result FROM table_name; |
| SUM | Calculates the sum of values in a group. | SELECT SUM(column_name) AS sum_result FROM table_name; |
| AVG | Calculates the average of values in a group. | SELECT AVG(column_name) AS average_result FROM table_name; |
| MIN | Returns the minimum value in a group. | SELECT MIN(column_name) AS min_result FROM table_name; |
| MAX | Returns the maximum value in a group. | SELECT MAX(column_name) AS max_result FROM table_name; |
| STDEV | Calculates the sample standard deviation of values. | SELECT STDEV(column_name) AS stdev_result FROM table_name; |
| VAR | Calculates the sample variance of values. | SELECT VAR(column_name) AS var_result FROM table_name; |

# Date Time Functions

| Function | Description | Example Usage |
|---|---|---|
| GETDATE() | Returns the current date and time. | SELECT GETDATE(); |
| NOW() | Returns the current date and time. | SELECT NOW(); |
| DATE_ADD() | Adds a specified interval to a date or datetime value. | SELECT DATE_ADD('2024-03-31', INTERVAL 1 DAY); |
| DATEDIFF() | Calculates the difference between two dates. | SELECT DATEDIFF('2024-03-31', '2024-03-01'); |
| CURRENT_DATE() | Returns the current date. | SELECT CURRENT_DATE(); |
| CURRENT_TIME() | Returns the current time. | SELECT CURRENT_TIME(); |
| TO_DATE() | Converts a string into a date value according to a specified format. | SELECT TO_DATE('2024-06-29', 'YYYY-MM-DD'); |
| ISDATE() | Checks if a value is a valid date. | SELECT ISDATE(value); |

# Extract Datetime Function

The EXTRACT() function extracts a part from a given date value

| Function | Description | Example Usage |
|---|---|---|
| EXTRACT(YEAR FROM date_expr) | Extracts the year component from a date or datetime expression. | 'SELECT EXTRACT(YEAR FROM '2024-03-31'); |
| EXTRACT(MONTH FROM date_expr) | Extracts the month component from a date or datetime expression. | 'SELECT EXTRACT(MONTH FROM '2024-03-31'); |
| EXTRACT(DAY FROM date_expr) | Extracts the day component from a date or datetime expression. | 'SELECT EXTRACT(DAY FROM '2024-03-31'); |
| EXTRACT(HOUR FROM date_expr) | Extracts the hour component from a datetime expression. | 'SELECT EXTRACT(HOUR FROM '2024-03-31 12:30:45'); |
| EXTRACT(MINUTE FROM date_expr) | Extracts the minute component from a datetime expression. | 'SELECT EXTRACT(MINUTE FROM '2024-03-31 12:30:45'); |
| EXTRACT(DOW FROM date_expr) | Extracts the day of the week (0 = Sunday, 1 = Monday, ..., 6 = Saturday) from a date expression. | 'SELECT EXTRACT(DOW FROM '2024-03-31'); |

# String Functions

| Function | Description | Example Usage |
|---|---|---|
| CONCAT() | Concatenates two or more strings. | 'SELECT CONCAT( 'hello', ' world'); |
| SUBSTRING() | Extracts a substring from a string. | 'SELECT SUBSTRING( 'hello world', 1, 5); |
| UPPER() | Converts a string to uppercase. | 'SELECT UPPER( 'hello'); |
| LOWER() | Converts a string to lowercase. | 'SELECT LOWER( 'HELLO' ); |
| TRIM() | Removes leading and trailing spaces from a string. | 'SELECT TRIM(' hello '); |
| REPLACE() | Replaces occurrences of a substring in a string. | 'SELECT REPLACE( 'hello world', 'world', 'everyone'); |
| LENGTH() | Returns the length of a string. | 'SELECT LENGTH( 'hello'); |
| LEFT() | Returns the leftmost characters from a string. | 'SELECT LEFT( 'hello world', 5); |
| RIGHT() | Returns the rightmost characters from a string. | 'SELECT RIGHT ('hello world', 5); |
| COALESCE() | Is useful for replacing null values with a default value. | 'SELECT COALESCE(NULL, 'default value'); |
| POSITION() | Returns the position of a substring in a string. | 'SELECT POSITION( 'world' IN 'hello'); |

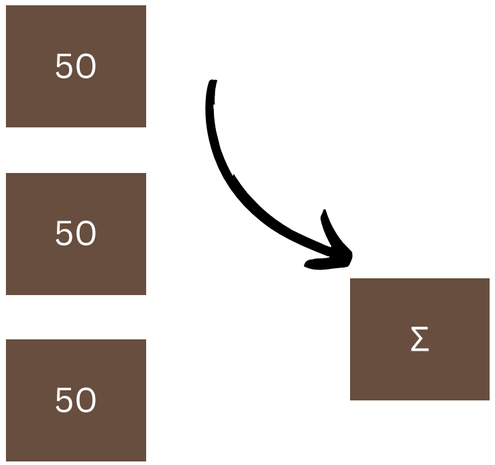
# Window Functions

Function applied over window frame

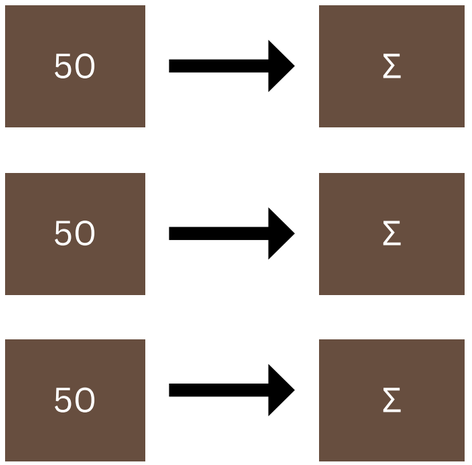
applies aggregation, ranking, analytic, distribution functions over a window frame

- **Window Frame** = Set of Rows or Subset of Partition
- **Over** = defines a window frame by **partition**, **order** and framing of **rows**

## Aggregate




## Window




Give output one row per aggregation

rows maintain their separate identities

Note: In window function over clause is mandatory to use whereas partition by & order by is optional

# Window Function Syntax

```
SELECT column_name(s),
       fun( ) OVER (  [ <PARTITION BY Clause> ]
                      [ <ORDER BY Clause> ]
                      [ <ROW or RANGE Clause> ]  )
FROM table_name
```

## Functions

- Aggregate functions
- Ranking functions
- Value functions
- Statistic functions

## Define a Window

- Partition by
- Order by
- Rows or Range

# Working of Windows Function

- **Partitioning**: The first step in using a window function is to partition the data into groups based on certain criteria. This partitioning is done using the PARTITION BY (optional) clause. Rows within each partition will be treated as a separate group for the window function.
- **Ordering**: Once the data is partitioned, it's often helpful to order the rows within each partition to define the window frame. Ordering is done using the ORDER BY (mandatory) clause. This determines the order in which the window function will process the rows within each partition.
- **Window Frame**: The window frame defines the subset of rows within each partition that the window function will operate on. It's defined by the combination of the PARTITION BY and ORDER BY clauses. You can specify whether the window frame includes all rows in the partition, a fixed number of rows preceding or following the current row, or rows between a specified range.
- **Applying the Function**: Once the window frame is defined, the window function is applied to the rows within the frame. The function calculates a result for each row based on the values within its window frame.
- **Result**: Finally, the result of the window function is returned for each row in the query result set. The result is typically displayed as an additional column alongside the original data.

# Window Function Types

- **Aggregate**
  - Min, Max, Sum, Avg, Count
- **Value**

| Function | Definition | Example Query |
|---|---|---|
| LEAD | Accesses data from the next row within the same result set. | `` `SELECT product_id, sales_date, sales_amount, LEAD(sales_amount, 1) OVER (ORDER BY sales_date) AS next_sales FROM sales_table;` `` |
| LAG | Accesses data from the previous row within the same result set. | `` `SELECT product_id, sales_date, sales_amount, LAG(sales_amount, 1) OVER (ORDER BY sales_date) AS prev_sales FROM sales_table;` `` |
| FIRST_VALUE | Returns the first value in an ordered set of values. | `` `SELECT product_id, sales_date, sales_amount, FIRST_VALUE(sales_amount) OVER (ORDER BY sales_date) AS first_sales FROM sales_table;` `` |
| LAST_VALUE | Returns the last value in an ordered set of values. | `` `SELECT product_id, sales_date, sales_amount, LAST_VALUE(sales_amount) OVER (ORDER BY sales_date) AS last_sales FROM sales_table;` `` |
| NTILE | Divides the result set into a specified number of approximately equal groups or "tiles". | `` `SELECT product_id, sales_date, sales_amount, NTILE(4) OVER (ORDER BY sales_amount) AS quartile FROM sales_table;` `` |
| NTH_VALUE | Returns the value of the expression from the nth row of the window frame. | `` `SELECT product_id, sales_date, sales_amount, NTH_VALUE(sales_amount, 2) OVER (ORDER BY sales_date) AS second_sales FROM sales_table;` `` |

- **Ranking**

| Function | Definition | Example Query |
|---|---|---|
| ROW_NUMBER() | Assigns a unique sequential integer to each row in the result set. | `` `SELECT ROW_NUMBER() OVER (ORDER BY column_name) AS row_num FROM table_name;` `` |
| RANK() | Assigns a rank value to each distinct row within a partition of a window frame, with no gaps in rank values. | `` `SELECT RANK() OVER (ORDER BY column_name) AS rank_num FROM table_name;` `` |
| DENSE_RANK() | Assigns a rank value to each row within a partition of a window frame, with no gaps in rank values. Rows with equal values receive the same rank. | `` `SELECT DENSE_RANK() OVER (ORDER BY column_name) AS dense_rank_num FROM table_name;` `` |

- **Statistic**

| Function | Definition | Example Query |
|---|---|---|
| CUME_DIST() | Returns the cumulative distribution of a value within a partition, indicating the proportion of rows with values less than or equal to the current row's value. | `` `SELECT CUME_DIST(column_name) OVER (ORDER BY column_name) AS cume_dist_value FROM table_name;` `` |
| PERCENT_RANK() | Returns the relative rank of a value within a partition, indicating the percentage of values less than the current row's value. | `` `SELECT PERCENT_RANK() OVER (ORDER BY column_name) AS percent_rank_value FROM table_name;` `` |

# Rank vs Dense Rank

## Rank

Skip

| name | age | rank_age |
|---|---|---|
| string | 15 | 1 |
| string | 23 | 2 |
| string | 23 | 2 |
| string | 34 | 4 |
| string | 41 | 5 |

VS

## Dense Rank

| name | age | dense_rank_age |
|---|---|---|
| string | 15 | 1 |
| string | 23 | 2 |
| string | 23 | 2 |
| string | 34 | 3 |
| string | 41 | 4 |

# Unbounded in Windows Partition

## Unbounded preceding

This refers to all rows from the beginning of the partition up to and including the current row.

## Unbounded following

This refers to all rows from the current row up to the end of the partition.

Used with Range or Rows

all rows from the beginning of the partition up to and including the current row

```
SELECT
 employee_id,
 salary,
 SUM(salary) OVER (ORDER BY employee_id ROWS BETWEEN UNBOUNDED PRECEDING
AND CURRENT ROW) AS running_total
FROM
 employees;
```

# Row vs Range in Window Function

ROWS BETWEEN 1 PRECEDING
AND 1 FOLLOWING

| city | sold | month |
|---|---|---|
| Paris | 300 | 1 |
| Rome | 200 | 1 |
| Paris | 500 | 2 |
| Rome | 100 | 4 |
| Paris | 200 | 4 |
| Paris | 300 | 5 |
| Rome | 200 | 5 |
| London | 200 | 5 |
| London | 100 | 6 |
| Rome | 300 | 6 |

current row →

1 row before the current row and
1 row after the current row

RANGE BETWEEN 1 PRECEDING
AND 1 FOLLOWING

| city | sold | month |
|---|---|---|
| Paris | 300 | 1 |
| Rome | 200 | 1 |
| Paris | 500 | 2 |
| Rome | 100 | 4 |
| Paris | 200 | 4 |
| Paris | 300 | 5 |
| Rome | 200 | 5 |
| London | 200 | 5 |
| London | 100 | 6 |
| Rome | 300 | 6 |

current row →

values in the range between 3 and 5
ORDER BY must contain a single expression

# Aggregate Function Example

```
SELECT new_id, new_cat,
SUM(new_id) OVER( PARTITION BY new_cat ORDER BY new_id ) AS "Total",
AVG(new_id) OVER( PARTITION BY new_cat ORDER BY new_id ) AS "Average",
COUNT(new_id) OVER( PARTITION BY new_cat ORDER BY new_id ) AS "Count",
MIN(new_id) OVER( PARTITION BY new_cat ORDER BY new_id ) AS "Min",
MAX(new_id) OVER( PARTITION BY new_cat ORDER BY new_id ) AS "Max"
FROM test_data
```

| new_id | new_cat | Total | Average | Count | Min | Max |
|---|---|---|---|---|---|---|
| 100 | Agni | 300 | 150 | 2 | 100 | 200 |
| 200 | Agni | 300 | 150 | 2 | 100 | 200 |
| 500 | Dharti | 1200 | 600 | 2 | 500 | 700 |
| 700 | Dharti | 1200 | 600 | 2 | 500 | 700 |
| 200 | Vayu | 1000 | 333.33333 | 3 | 200 | 500 |
| 300 | Vayu | 1000 | 333.33333 | 3 | 200 | 500 |
| 500 | Vayu | 1000 | 333.33333 | 3 | 200 | 500 |

# Ranking Function Example

```
SELECT new_id,
ROW_NUMBER() OVER(ORDER BY new_id) AS "ROW_NUMBER",
RANK() OVER(ORDER BY new_id) AS "RANK",
DENSE_RANK() OVER(ORDER BY new_id) AS "DENSE_RANK",
PERCENT_RANK() OVER(ORDER BY new_id) AS "PERCENT_RANK"
FROM test_data
```

| new_id | ROW_NUMBER | RANK | DENSE_RANK | PERCENT_RANK |
|---|---|---|---|---|
| 100 | 1 | 1 | 1 | 0 |
| 200 | 2 | 2 | 2 | 0.166 |
| 200 | 3 | 2 | 2 | 0.166 |
| 300 | 4 | 4 | 3 | 0.5 |
| 500 | 5 | 5 | 4 | 0.666 |
| 500 | 6 | 5 | 4 | 0.666 |
| 700 | 7 | 7 | 5 | 1 |

# Value Function Example

```
SELECT new_id,
FIRST_VALUE(new_id) OVER( ORDER BY new_id) AS "FIRST_VALUE",
LAST_VALUE(new_id) OVER( ORDER BY new_id) AS "LAST_VALUE",
LEAD(new_id) OVER( ORDER BY new_id) AS "LEAD",
LAG(new_id) OVER( ORDER BY new_id) AS "LAG"
FROM test_data
```

| new_id | FIRST_VALUE | LAST_VALUE | LEAD | LAG |
|---|---|---|---|---|
| 100 | 100 | 100 | 200 | null |
| 200 | 100 | 200 | 200 | 100 |
| 200 | 100 | 200 | 300 | 200 |
| 300 | 100 | 300 | 500 | 200 |
| 500 | 100 | 500 | 500 | 300 |
| 500 | 100 | 500 | 700 | 500 |
| 700 | 100 | 700 | null | 500 |

# System Functions

| Function | Definition | Query Example |
|---|---|---|
| CAST | Converts an expression from one data type to another using a specific type conversion. | `` `SELECT CAST('2022-01-01' AS DATE)` `` - Converts the string '2022-01-01' to a DATE data type. |
| CONVERT<br>Format | Converts an expression from one data type to another, with the option to specify a style for the output. | `` `SELECT CONVERT(VARCHAR, GETDATE(), 101)` `` - Converts the current date and time into a VARCHAR string with the format 'mm/dd/yyyy'. |
| CHOOSE | Returns the item at the specified index from a list of values. | `` `SELECT CHOOSE(3, 'First', 'Second', 'Third', 'Fourth')` `` - Returns 'Third' as it corresponds to the index 3 in the list of provided values. |
| ISNULL | Replaces NULL with the specified replacement value. | `` `SELECT ISNULL(NULL, 'Replacement Value')` `` - Returns 'Replacement Value' as NULL is replaced with the specified value. |
| ISNUMERIC | Determines whether an expression is a valid numeric type. | `` `SELECT ISNUMERIC('123')` `` - Returns 1 if '123' can be converted to a numeric data type; otherwise, returns 0. |
| IIF | Returns one of two values, depending on whether the Boolean expression evaluates to true or false. | `` `SELECT IIF(1 > 0, 'True', 'False')` `` - Returns 'True' as the condition '1 > 0' evaluates to true. |

## Cast vs Convert

CAST and CONVERT are both the functions that can be used to change the data type of a value in SQL Server. CAST is usually preferable since it is more concise. However, there are some situations where CONVERT may be necessary, such as when you need to use the style argument.

Convert Formats

```
8   -- hh:mm:ss
101 -- mm/dd/yyyy
102 -- yyyy.mm.dd
103 -- dd/mm/yyyy
104 -- dd.mm.yyyy
```

# Declare Variable

Variables in SQL are defined by using DECLARE statement

```
Declare @TJ varchar(50)
set @TJ = 'My Name is Tajamul'

Declare @Number bigint
set @Number = 700654

select @TJ
select @Number
```

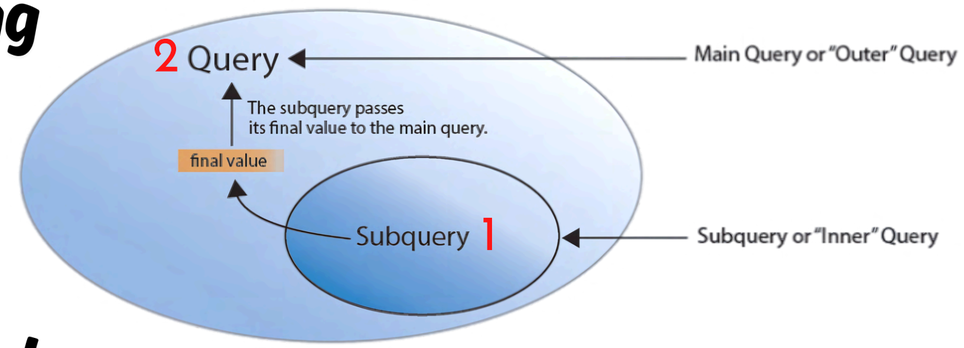
```
%
Results  Messages

(No column name)
My Name is Tajamul

(No column name)
7006542939
```

# Sub Query Nested

also known as a **nested query**, is a query nested within another query. It allows us to retrieve data based on the results of another query.

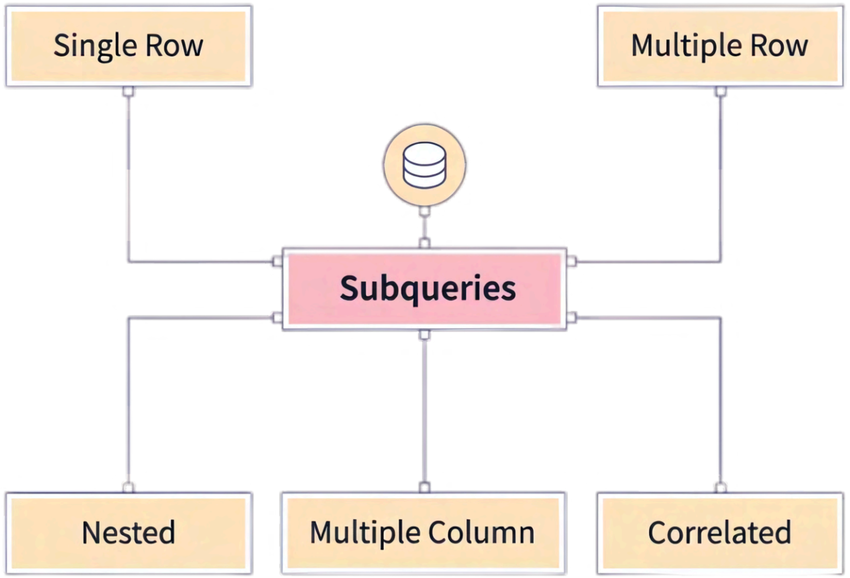
- Sub query syntax involves two SELECT statements
- It can be added after keywords like WHERE or ON, with comparison operators (>, <, =).

## Working




## Example

```
SELECT *
FROM table_name
WHERE <=
(SELECT column_name FROM table_name WHERE ...);
```



# CTE WITH

also known as Common Table Expressions, is a named temporary table or named result set that can be used multiple times within a single query.

- It used tempdb but is efficient than temporary table
- It allows us to break down complex queries into smaller, more manageable parts.

## Example

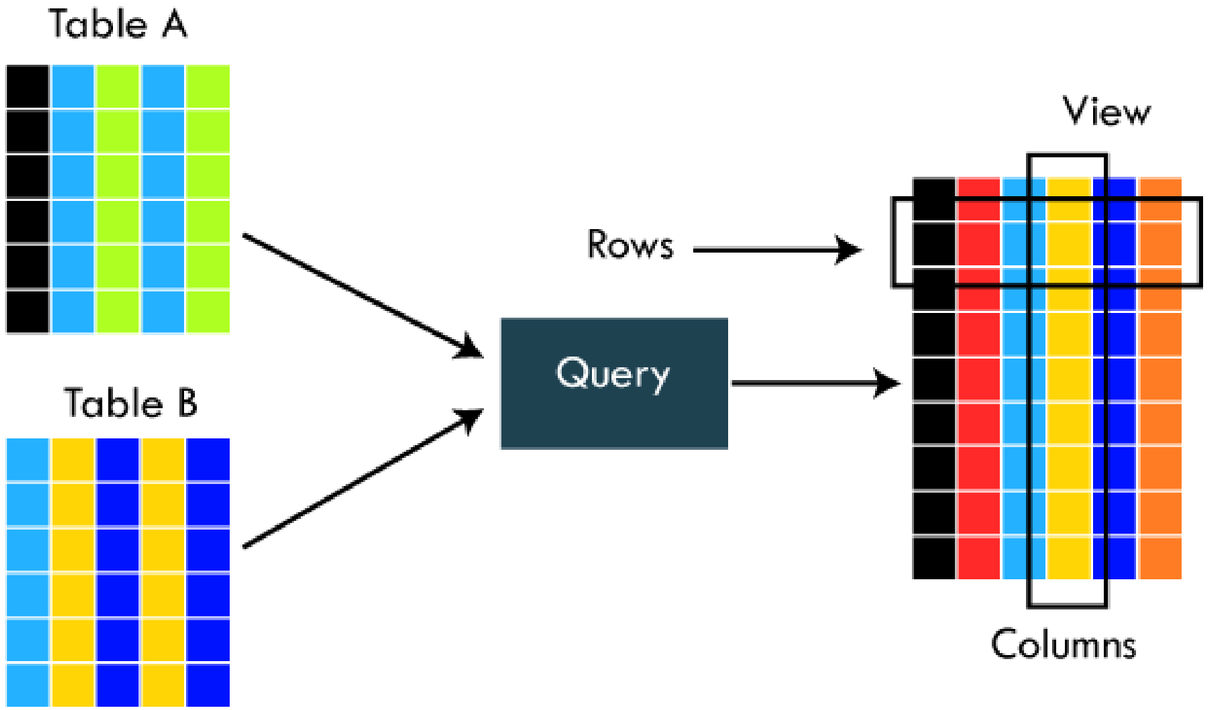
```
WITH Sales_CTE AS (
    SELECT
        customer_id,
        SUM(amount) AS total_sales
    FROM
        sales
    GROUP BY
        customer_id
)
SELECT customer_id, total_sales FROM Sales_CTE WHERE total_sales > 1000;
```

# Sub Query vs CTE

| Feature | Subquery | CTE |
|---|---|---|
| Definition | A nested query that is executed within the context of another query. | A named temporary table that can be used within a SQL query. |
| Location | Can be used in any part of a SQL query. | Defined before the main query. |
| Reusability | Can only be used once in a query. | Can be referenced multiple times in a query. |
| Readability | Can be difficult to read and write, especially for complex queries. | More readable and easier to maintain, especially for complex queries. |
| Performance | Can be less efficient, as they may be executed multiple times, depending on how they are used. | More efficient, as they are only executed once, even if they are referenced multiple times in the query. |

# Views

- View is like a virtual table that is based on the result set of a SELECT query.
- View does not store any data, difference between table & view is that table can store data but view can never stores data
- Changes will be done in view table not in main table




## Syntax

```
CREATE VIEW my_view AS
SELECT column1, column2
FROM table
WHERE condition;

Select * FROM my_view
```

## Rules for View

- Cannot change column name.
- Cannot change column data type
- Cannot change order of columns
- But we can add new column at the end

## Use

- Data Integrity and Security
- To simplify SQL complex queries

If main table is changed we need to refresh view to see changes

## Example

```
create view order_summary
as
select o.ord_id, o.date, p.prod_name, c.cust_name
, (p.price * o.quantity) - ((p.price * o.quantity) * disc_percent::float/100) as cost
from tb_customer_data c
join tb_order_details o on o.cust_id = c.cust_id
join tb_product_info p on p.prod_id = o.prod_id;

alter view order_summary rename column date to order_date;
alter view order_summary rename to order_summary_2;


drop view order_summary_2;
```

Changes are done in View only

# View Problem

## Create View

Once you executed if you try to re-execute it will show error

```
create view order_summary
as
select o.ord_id, o.date, p.prod_name, c.cust_name
, (p.price * o.quantity) - ((p.price * o.quantity) * disc_percent::float/100) as cost
from tb_customer_data c
join tb_order_details o on o.cust_id = c.cust_id
join tb_product_info p on p.prod_id = o.prod_id;
```

## Create or Replaced View

Once you executed if you try to re-execute multiple times it will run without any error

```
create or replace view order_summary
as
select o.ord_id, o.date, p.prod_name, c.cust_name
, (p.price * o.quantity) - ((p.price * o.quantity) * disc_percent::float/100) as cost
from tb_customer_data c
join tb_order_details o on o.cust_id = c.cust_id
join tb_product_info p on p.prod_id = o.prod_id;
```

# Indexes

Lookup

Indexing in SQL organizes data using a B-tree data structure to swiftly locate information in tables, enhancing performance for read-intensive operations.

| Index Type | Definition | Example |
| --- | --- | --- |
| Primary Index | Index automatically created when a primary key is defined on a table. | `` `CREATE TABLE Employees (EmployeeID INT PRIMARY KEY);` `` |
| Unique Index | Ensures values in the indexed column(s) are unique. | `` `CREATE UNIQUE INDEX idx_unique_email ON Employees (Email);` `` |
| Clustered Index | Sorts and stores data rows in the table based on the indexed column(s). | `` `CREATE CLUSTERED INDEX idx_clustered_employeeid ON Employees (EmployeeID);` `` |
| Non-Clustered Index | Creates a separate object storing indexed column values and row pointers. | `` `CREATE NONCLUSTERED INDEX idx_lastname ON Employees (LastName);` `` |
| Composite Index | Index on multiple columns. | `` `CREATE INDEX idx_composite_name ON Employees (FirstName, LastName);` `` |

# Stored Procedures

A precompiled collection of SQL statements stored in the database and executed as a single unit.

- **Improved Performance**: Precompiled and stored on the server, they reduce parsing time and enhance execution speed.
- **Enhanced Security**: Users execute procedures without direct table access, reducing SQL injection risks.
- **Code Reusability**: Encapsulate business logic for reuse across applications.
- **Reduced Network Traffic**: Execute with minimal data transfer, sending only procedure names and parameters.

## Syntax

```
CREATE PROCEDURE sp_GetEmployeeByID @EmployeeID INT
AS
BEGIN
-- SQL statements inside the stored procedure
SELECT * FROM Employees
WHERE EmployeeID = @EmployeeID;
-- Execute the stored procedure
EXEC sp_GetEmployeeByID @EmployeeID = 1;
```

# Triggers

A database object that automatically executes in response to specified events (INSERT, UPDATE, DELETE) on a table.

Purpose: automate actions based on data changes, ensuring consistency, integrity, and tracking of operations within the database.

## Syntax

| Type | Definition | Example Query |
|---|---|---|
| Insert Trigger | Fires automatically after an INSERT operation on a table. | `` `sql CREATE TRIGGER trg_after_insert_employee ON Employees AFTER INSERT AS BEGIN INSERT INTO AuditTrail (EmployeeID, Action, ActionDate) VALUES ((SELECT EmployeeID FROM inserted), 'Insert', GETDATE()); END;` `` |
| Update Trigger | Fires automatically after an UPDATE operation on a table. | `` `sql CREATE TRIGGER trg_after_update_employee ON Employees AFTER UPDATE AS BEGIN INSERT INTO AuditTrail (EmployeeID, Action, ActionDate) VALUES ((SELECT EmployeeID FROM inserted), 'Update', GETDATE()); END;` `` |
| Delete Trigger | Fires automatically before a DELETE operation on a table. | `` `sql CREATE TRIGGER trg_before_delete_employee ON Employees BEFORE DELETE AS BEGIN DELETE FROM AuditTrail WHERE EmployeeID IN (SELECT EmployeeID FROM deleted); END;` `` |

# Temporary Tables

also known as temp tables are on the fly tables used to do complex calculations without storing data in database. They are stored in **tempdb**

| TEMP TABLE CHARACTERISTIC | LOCAL TEMPORARY TABLE | GLOBAL TEMPORARY TABLE |
|---|---|---|
| Scope | Automatically deleted at the end of the session | Automatically deleted at the end of the session that created it and all other sessions that are referencing it |
| Visibility | Visible in the session where they are created | Visible in all sessions |
| Use Cases | Tasks specific to a single-user session | For collaborative tasks when multiple sessions share the same temporary data |
| Security | More secure due to isolation level | Less secure as there's no isolation from other users |
| Performance | No contention and concurrency problems | Possible contention and concurrency problems |

## Local

```
Select * into #localtb
Delete from #localtb
WHERE ID = 1
Drop table #localtb
```

## Global

```
Create Table ##Globaltb
(
Roll No INT,
City Varchar(20)
)
```

Data Blogs

Follow Tajamul Khan

# SQL Cheat Sheet

## Data Definition

**CREATE TABLE**: Creates a new table.
**CREATE TABLE table_name (id INT PRIMARY KEY, name VARCHAR(50));**

**ALTER TABLE**: Modifies an existing table.
**ALTER TABLE table_name ADD column2 INT;**

**DROP TABLE**: Deletes a table.
**DROP TABLE table_name;**

**CREATE INDEX**: Creates an index on a table.
**CREATE INDEX idx_name ON table_name (column1);**

**DROP INDEX**: Removes an index.
**DROP INDEX idx_name ON table_name;**

**CREATE VIEW**: Creates virtual table based on query.
**CREATE VIEW view_name AS SELECT column1, column2 FROM table_name;**

**DROP VIEW**: Deletes a view.
**DROP VIEW view_name;**

**RENAME TABLE**: Renames an existing table.
**RENAME TABLE old_table_nm TO new_table_name;**

## Select Data

**SELECT**: Retrieves specific columns from a table.
**SELECT column1, column2 FROM table_name;**

**DISTINCT**: Removes duplicate rows from the result.
**SELECT DISTINCT column1 FROM table_name;**

**WHERE**: Filters rows based on a condition.
**SELECT * FROM table_name WHERE column1 = 'v1';**

**ORDER BY**: Sorts result set by one or more columns.
**SELECT * FROM table_nm ORDER BY column1 ASC;**

**LIMIT / FETCH**: Limits the number of rows returned.
**SELECT * FROM table_name LIMIT 10;**

**LIKE**: Searches for patterns in text columns.
**SELECT * FROM table_name WHERE col1 LIKE 'A%';**

**IN**: Filters rows with specific values.
**SELECT * FROM table_nm WHERE col1 IN ('v1', 'v2');**

**BETWEEN**: Filters rows within a range of values.
**SELECT * FROM table WHERE c1 BETWEEN 1 AND 20;**

# Aggregate Data

**COUNT()**: Returns the number of rows.
**SELECT COUNT(*) FROM table_name;**

**SUM()**: Calculates the sum of a numeric column.
**SELECT SUM(column1) FROM table_name;**

**AVG()**: Calculates the average of a numeric column.
**SELECT AVG(column1) FROM table_name;**

**MIN()**: Returns the smallest value in a column.
**SELECT MIN(column1) FROM table_name;**

**MAX()**: Returns the largest value in a column.
**SELECT MAX(column1) FROM table_name;**

**GROUP BY**: Groups rows for aggregation.
**SELECT col1, COUNT(*) FROM t1 GROUP BY col1;**

**HAVING**: Filters grouped rows based on a condition.
**SELECT column1, COUNT(*) FROM t1 GROUP BY column1 HAVING COUNT(*) > 5;**

**DISTINCT COUNT()**: Counts unique values in column.
**SELECT COUNT(DISTINCT col1) FROM table_name;**

# Data Manipulation

**INSERT INTO**: Adds new rows to a table.
**INSERT INTO table_name (column1, column2) VALUES ('value1', 'value2');**

**UPDATE**: Updates existing rows in a table.
**UPDATE table_name SET col1 = 'value' WHERE id = 1;**

**DELETE**: Removes rows from a table.
**DELETE FROM table_name WHERE column1 = 'value';**

**MERGE**: Combines INSERT, UPDATE, and DELETE based on a condition.
**MERGE INTO table_name USING source_table ON condition WHEN MATCHED THEN UPDATE SET column1 = value WHEN NOT MATCHED THEN INSERT (columns) VALUES (values);**

**TRUNCATE**: Removes all rows from a table without logging.
**TRUNCATE TABLE table_name;**

**REPLACE**: Deletes existing rows and inserts new rows (MySQL-specific).
**REPLACE INTO table_name VALUES (value1, value2);**

# Transactions

**Commit Transaction:** Finalizes changes when all operations succeed.

```
START TRANSACTION;
UPDATE accounts SET balance = 1000 WHERE id = 1;
WHERE id = 2; COMMIT;
```

**Execute a Stored Procedure:** Undoes changes if an error occurs or the transaction is not committed.

```
START TRANSACTION;
UPDATE accounts SET balance = 1000 WHERE id = 1;
ROLLBACK;
```

**Using Savepoints:** Set a rollback point within a transaction, allowing partial rollback without affecting the whole transaction.

```
START TRANSACTION;
UPDATE accounts SET balance = 1000 WHERE id = 1;
SAVEPOINT sp1;
UPDATE accounts SET balance = 2000 WHERE id = 3;
-- Simulate failure
ROLLBACK TO SAVEPOINT sp1;
UPDATE accounts SET balance = 1000 WHERE id = 2;
COMMIT;
```



# Set Operations

**UNION:** Combines results from two queries, removing duplicates.

```
SELECT column1 FROM table1 UNION SELECT
column1 FROM table2;
```

**UNION ALL:** Combines results from two queries, including duplicates.

```
SELECT column1 FROM table1 UNION ALLSELECT
column1 FROM table2;
```

**INTERSECT:** Returns common rows from both queries.

```
SELECT column1 FROM table1 INTERSECT SELECT
column1 FROM table2;
```

**EXCEPT (or MINUS):** Returns rows from the first query that are not in the second query.

```
SELECT column1 FROM table1 EXCEPTSELECT
column1 FROM table2;
```

# Table Joins

**INNER JOIN:** matching values in both tables.
**SELECT * FROM table1 INNER JOIN table2 ON table1.id = table2.id;**

**LEFT JOIN**: Returns all rows from the left table and matching rows from the right table.
**SELECT * FROM table1 LEFT JOIN table2 ON table1.id = table2.id;**

**RIGHT JOIN**: Returns all rows from the right table and matching rows from the left table.
**SELECT * FROM table1 RIGHT JOIN table2 ON table1.id = table2.id;**

**FULL OUTER JOIN**: Returns rows when there is a match in either table.
**SELECT * FROM table1 FULL OUTER JOIN table2 ON table1.id = table2.id;**

**CROSS JOIN**: Cartesian product of both tables.
**SELECT * FROM table1 CROSS JOIN table2;**

**SELF JOIN**: Joins a table with itself.
**SELECT a.column1, b.column1 FROM table_name a, table_name b WHERE a.id = b.parent_id;**

# Other Functions

**CONCAT()**: Concatenates strings.
**SELECT CONCAT(first_name, ' ', last_name) FROM table_name;**

**SUBSTRING()**: Extracts a substring from a string.
**SELECT SUBSTRING(column1, 1, 5) FROM table_nm;**

**LENGTH()**: Returns the length of a string.
**SELECT LENGTH(column1) FROM table_name;**

**ROUND()**: Rounds a number to a specified number of decimal places.
**SELECT ROUND(column1, 2) FROM table_name;**

**NOW()**: Returns the current timestamp.
**SELECT NOW();**

**DATE_ADD()**: Adds a time interval to a date.
**SELECT DATE_ADD(NOW(), INTERVAL 7 DAY);**

**COALESCE()**: Returns the first non-null value.
**SELECT COALESCE(column1, column2) FROM table_name;**

**IFNULL()**: Replaces NULL values with desired value.
**SELECT IFNULL(col1, 'default') FROM table_name;**

# Window Functions

**ROW_NUMBER**: Assigns a unique number to each row in a result set.
**SELECT ROW_NUMBER() OVER (PARTITION BY department ORDER BY salary DESC) AS row_num FROM employees;**

**RANK:** Assigns a rank to each row, with gaps for ties.
**SELECT RANK() OVER (PARTITION BY department ORDER BY salary DESC) AS rank FROM employees;**

**DENSE_RANK:** Assigns a rank to each row without gaps for ties.
**SELECT DENSE_RANK() OVER (PARTITION BY department ORDER BY salary DESC) AS dense_rank FROM employees;**

**NTILE:** Divides rows into equal parts.
**SELECT NTILE(4) OVER (ORDER BY salary) AS quartile FROM employees;**

**LEAD():** Accesses subsequent rows' data.
**SELECT name, salary, LEAD(salary) OVER (ORDER BY salary) AS next_salary FROM employees;**

**LAG():** Accesses subsequent rows' data.
**SELECT name, salary, LAG(salary) OVER (ORDER BY salary) AS previous_salary FROM employees;**

# Stored Procedures

**Create a Stored Procedure:**

```
CREATE PROCEDURE sp_GetEmployeeByID
@EmployeeID INT
AS
BEGIN
-- SQL statements inside the stored procedure
SELECT * FROM Employees
WHERE EmployeeID = @EmployeeID;
```

**Execute a Stored Procedure:**

```
EXEC sp_GetEmployeeByID @EmployeeID = 1;
```

**Stored Procedure with OUT Parameter:**

```
CREATE PROCEDURE GetEmployeeCount (OUT
emp_count INT) BEGINSELECT COUNT(*) INTO
emp_count FROM employees; END;
```

**Drop a Stored Procedure:**

```
DROP PROCEDURE GetEmployeeDetails;
```

# Triggers

**Create a Trigger (Before Insert):**

```
CREATE TRIGGER set_created_at
BEFORE INSERT ON employees
FOR EACH ROW
SET NEW.created_at = NOW();
```

**After Update Trigger:**

```
CREATE TRIGGER log_updates
AFTER UPDATE ON employees
FOR EACH ROW
INSERT INTO audit_log(emp_id, old_salary,
new_salary, updated_at)
VALUES (OLD.id, OLD.salary, NEW.salary, NOW());
```

**After Delete Trigger:**

```
CREATE TRIGGER log_deletes
AFTER DELETE ON employees
FOR EACH ROW
INSERT INTO audit_log(emp_id, old_salary,
new_salary, deleted_at)
VALUES (OLD.id, OLD.salary, NULL, NOW());
```

# Subquery

**Scalar Subquery:** Returns a single value.

```
SELECT name, salary
FROM employees
WHERE salary > (SELECT AVG(salary) FROM
employees);
```

**Correlated Subquery:**

```
SELECT e1.name, e1.salary
FROM employees e1
WHERE e1.salary > (SELECT AVG(e2.salary) FROM
employees e2 WHERE e1.department =
e2.department);
```

# CTE

**With a Single CTE:**

```
WITH DepartmentSalary AS (
   SELECT department, AVG(salary) AS avg_salary
   FROM employees
   GROUP BY department
)
SELECT *
FROM DepartmentSalary
WHERE avg_salary > 50000;
```

**Recursive CTE:**

```
WITH RECURSIVE Numbers AS (
   SELECT 1 AS num
   UNION ALL
   SELECT num + 1
   FROM Numbers
   WHERE num < 10
)
SELECT * FROM Numbers;
```

# Indexes

**Create an Index:**

```
CREATE INDEX idx_department ON
employees(department);
```

**Unique Index:** `CREATE UNIQUE INDEX idx_unique_email ON employees(email);`

**Drop an Index:**

```
DROP INDEX idx_department;
```

**Clustered Index (SQL Server):**

```
CREATE CLUSTERED INDEX idx_salary ON
employees(salary);
```

**Using EXPLAIN to Optimize:**

```
EXPLAIN SELECT * FROM employees WHERE salary >
50000;
```

Data Blogs

Follow Tajamul Khan 

# FREE DATA RESOURCES

# Free Projects

**FREE MACHINE LEARNING PROJECTS**

**FREE EDA PROJECTS**

**FREE STATISTICS PROJECTS**

## Download projects for your portfolio!

Data Blogs

Follow Tajamul Khan

# Free Ebooks

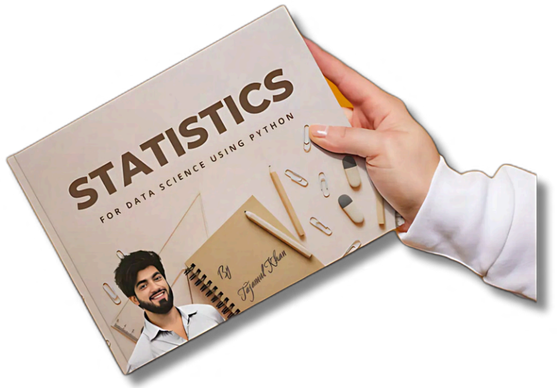
## Power BI


Download

## EDA


Download

## Statistics




Download

## Excel


Download

# Download your copy now!

# Free Resources

**Notes & Tips**

**Free Blogs**

**Free Projects**

## Follow to stay updated!

Drop your Review!

# Tajamul Khan