Definitions

- **Data**
  - A collection of information (Raw facts and figures). E.g. Numbers, test, images, etc.
- **SQL**
  - Structured query language (SQL) is a programming language for storing and processing information in a relational database.
  - Supports CSV, Excel, Commands.
- **Database**
  - A databse is an organised collection of structured information or data, typically stored electronically in a computer system.
  - A database is usually controlled by a database management system (DBMS).
- **DBMS**
  - A Database Management System (DBMS) is a software system that allows users to create, define, manipulate and manage databases. It provides a way for organizations to store, organize and retrieve large amounts of data quickly and efficiently in an organized manner.
  - E.g. MySQL, PostgreSQL, Oracle, MS SQL Server
- **RDBMS**
  - Relational Database Management System, is a type of database management system (DBMS) that organizes data into tables with rows and columns, where relationships between tables are defined using common attributes.
  - It uses the relational model of data, which structures data into tables that can be linked together, making it efficient for storing and retrieving related information.
  - RDBMS uses SQL (Structured Query Language) to manage and query the data.
  - In a database different tables should not have the same name.
  - Columns are fields (y-axis).
  - Rows are records (x-axis).
- **Primary Key**
  - Not Null + Unique value
- **Foreign Key**
  - Not Null + Unique value related to primary key in another table
- **Candidate Key**
  - Have all the pre-requists of primary key, but we don't consider it as primary key

---

Types of Data

- **1. Structured Data** : Data that is formated properly and can be easily interpreted by anyone. It is supported by DBMS/RDBMS.
- **2. Semi-Structured Data** : Partially structured data. E.g. emails, json files, xml/html files, csv files.
- **3. Unstructured Data** : Unformated Data. E.g. audio, video, images, etc.

Types of Databases

- **Tables**

- **RDBMS**
  - Any RDBMS must support the following relations
    - One to One i.e. 1 : 1
    - One to Many i.e. 1 : Many
    - Many to Many i.e. Many : Many
- **No SQL Database**
  - Stores data that can't be stored in tables. E.g. images, emails, social media posts, steaming, chat messages, charts, etc.
  - E.g. MongoDB, CouchDB, GraphDB, ApacheDB

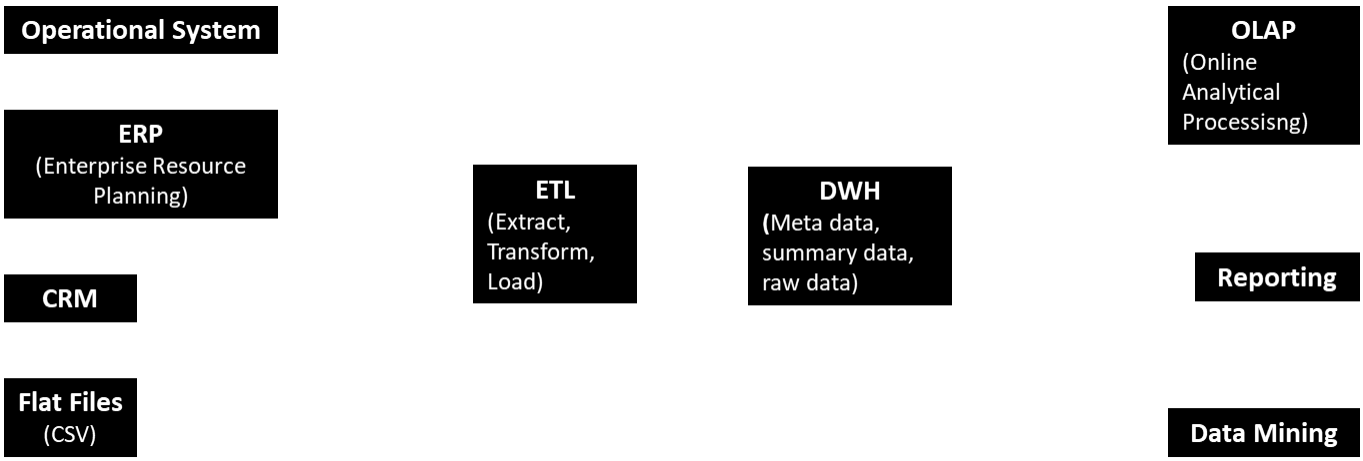
Data Warehouse (DWH)

- A data warehouse is a centralized repository designed for storing large amounts of data from various sources, enabling organizations to perform in-depth analysis and generate reports for better decision-making
- Only the data which is structured/formated is stored in Data warehouse.
- Data Warehouse is an appliance for storing and analyzing data, and reporting.
- It is designed for query and analysis.
- It usually contains historical data derived from transaction data or other sources.

Real - Time Data Warehouse

- Server rooms
- DWH is like an on-premise server, its safer than working with thirdparty servers.

Data Warehouse Architecture



Data Warhousing Process

1. **Data Cleaning:** It includes filling in missing values, smoothing noisy data removing outliers and resolving incosistencies.
2. **Data infegeration:** It includes infegeration of multiple database, data cubes and files.
3. **Data transformation:** Converts data fron host format to warehouse format
4. **Data loading:** Sort, summarize, consolidate, compute, views, check integrity

5. **Data refreshing:** Propagates the updates from data sources to the warehouse.

Characteristics Of Datawarehouse

1. **Subject ariented:** It can be used to analyze a particular subject area
2. **Integerated:** It integerates data from. multiple data sources.
3. **Time variant:** Historical data is kept in a data warehouse.
4. **Non volatile:** Once data is in the warehouse, it will not changes

Components of data warehouse:

- **Central Database:** A database serves as the foundation of your data warehouse
- **Data Integeration:** Data is pulled from many Source system and modified.
- **Meta Data:** It specifies the source, usage, values and other features of the data
- **Access Tools:** It allows users to interact with data in your data warehouse. E.g. OLAP tools.

Applications of Data Warehouse

- Services, Banking, INsurance, Finance, Retailing, Education, Manufacturing, Healthcare

---

Data Mart

- Subsets of DWH, they are comprised completely of only specific departments/category/business units.
- It exists independently without being integerated into a data warehouse.
- they provide quick and easy access to data for individual category, eliminating the need to search through the entire data warehouse.

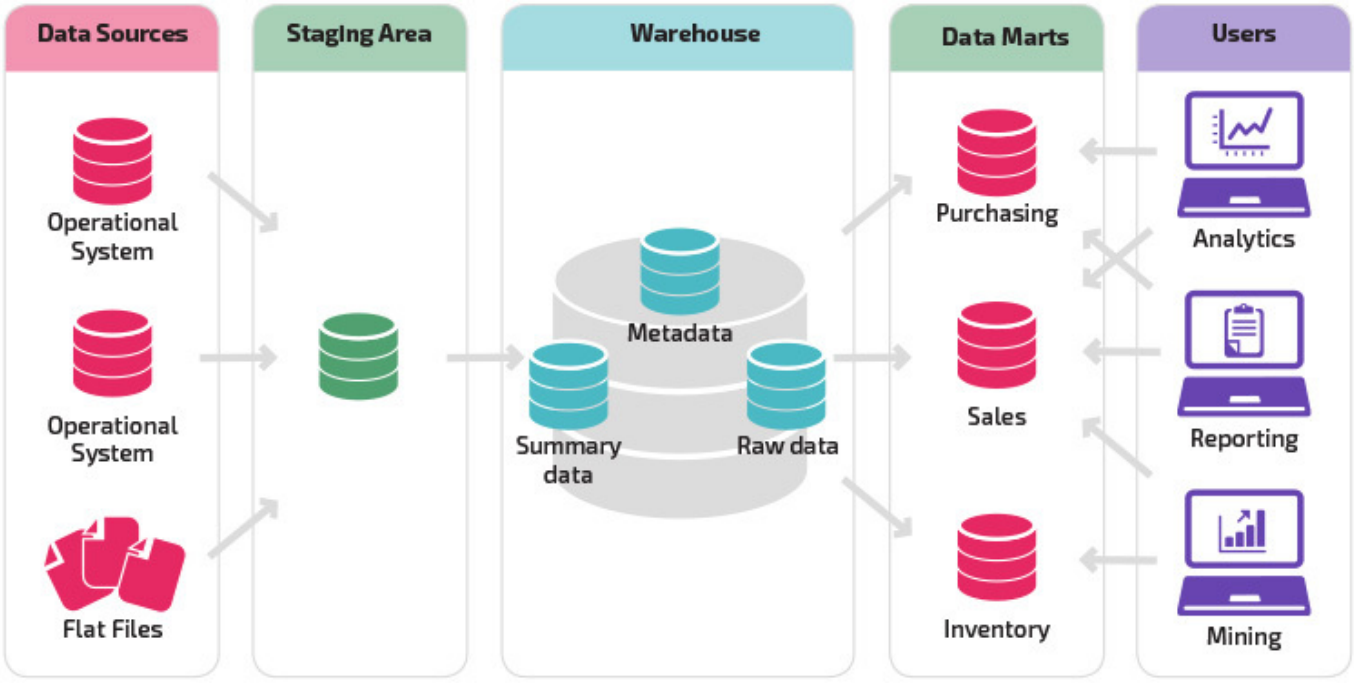
Uses of Data Mart

- Indentifying trends and patterns
- Making informed decisions
- improving the operational efficiency

Benefits of Data Mart

- Improved access of data
- Faster analysis
- Reduced Costs

Data Mart Architecture



Data Mart vs Data Warehouse

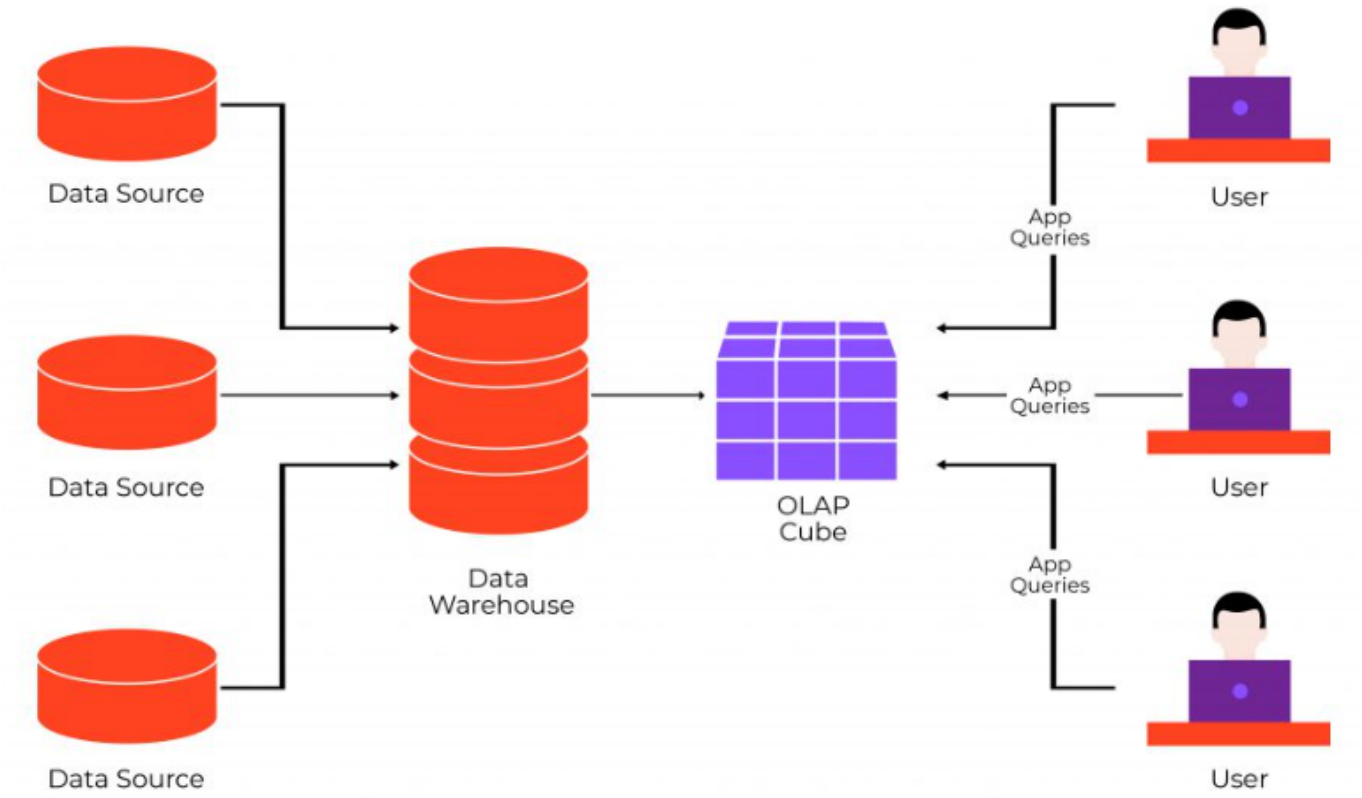
Feature	Data Mart	Data Warehouse
Main Definition	A **subset** of a Data Warehouse. Focused on a specific business line or department.	A **central repository** of data collected from various departments across an organization.
Size	Typically **less than 100 GB**	Usually **more than 100 GB**
Scope	Covers a **single department or business unit**	Covers the **entire organization**
Time to Build	Takes **a few weeks to a few months**	Takes **many months or even years**
Organization	Managed by **individual departments**	Managed by **central IT or data team**

DBMS vs Data Warehouse

DBMS	Data Warehouse
It is Transaction oriented	It is Subject oriented
Contains detailed data	Contains historic data
It captures data	It analyzes data
Application Oriented	Subject Oriented

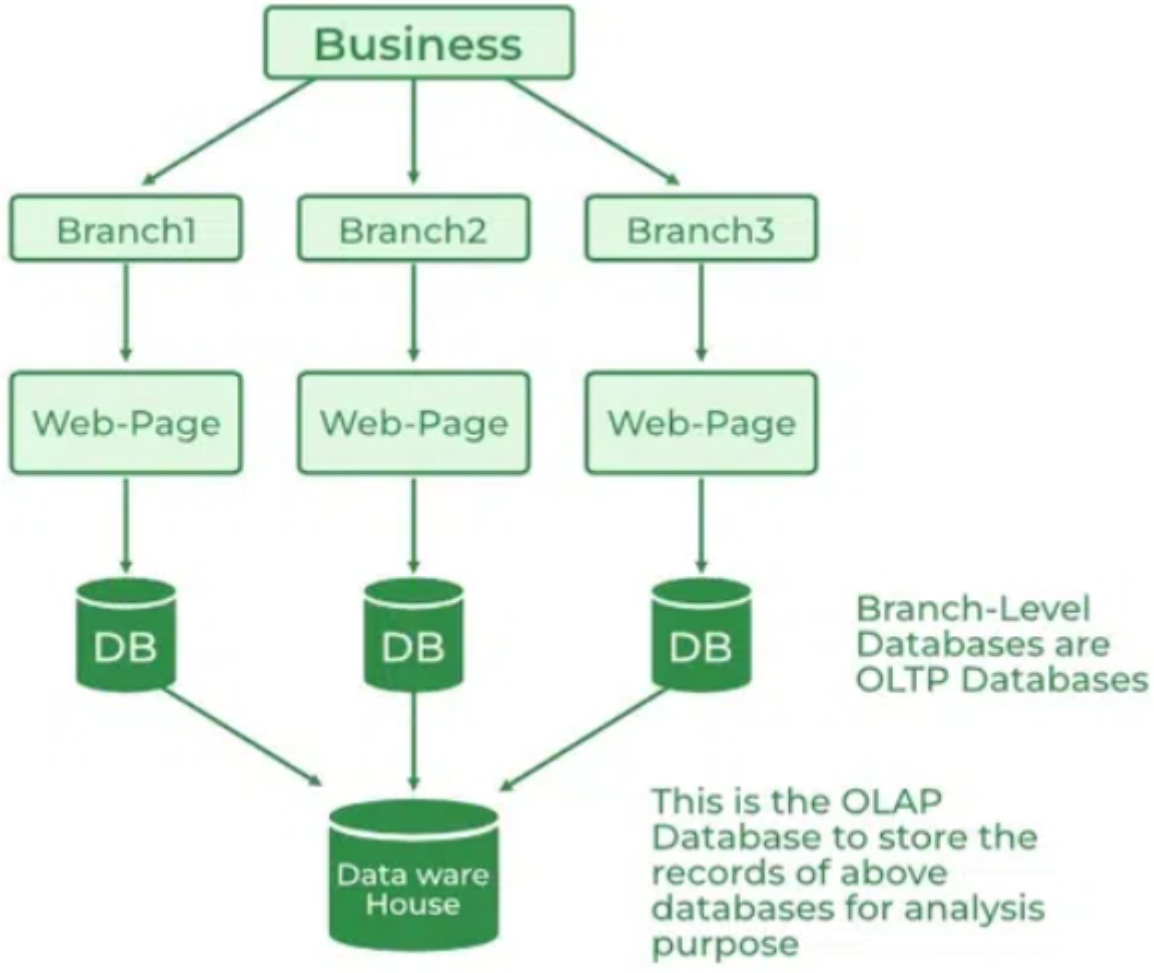
OLAP (Online Analytical Processing)

- OLAP is used for data preparation in data data warehouses.
- It is a computing method that allows users to easily and selectively extract data.
- It is used to query data in order to analyze it from different point of views.
- OLAP business intelligence often helps in trend analysis, financial reporting, sales forecasting, budgeting and other planning processes.
- OLAP helps in making data mart.
- OLAP maintains data in a cubical structure.
- Examples of OLAP Tools: IBM Cognos, Oracle OLAP, Tableau, Domo, Sisense, and Reveal.



OLTP (Online Transactional Processing)

- It facilitates and manageges transaction oriented applications (typically involving data entry and retrieval).
- OLTP enabels large number od database transactions(insertion, deletion or query of data in a DB) made by large number of people over the internet.
- OLTP system drives many of the financial transactions made everyday including online transaction, ATM transaction, E-commerce and in-store purchase.
- Examples include:
    - Database Management Systems (DBMS): PostgreSQL, MySQL, Oracle Database, Microsoft SQL Server, CockroachDB, IBM Db2, MongoDB.
    - Applications: ATMs, online banking, e-commerce, ticketing systems, and recordkeeping systems like health records and inventory control.



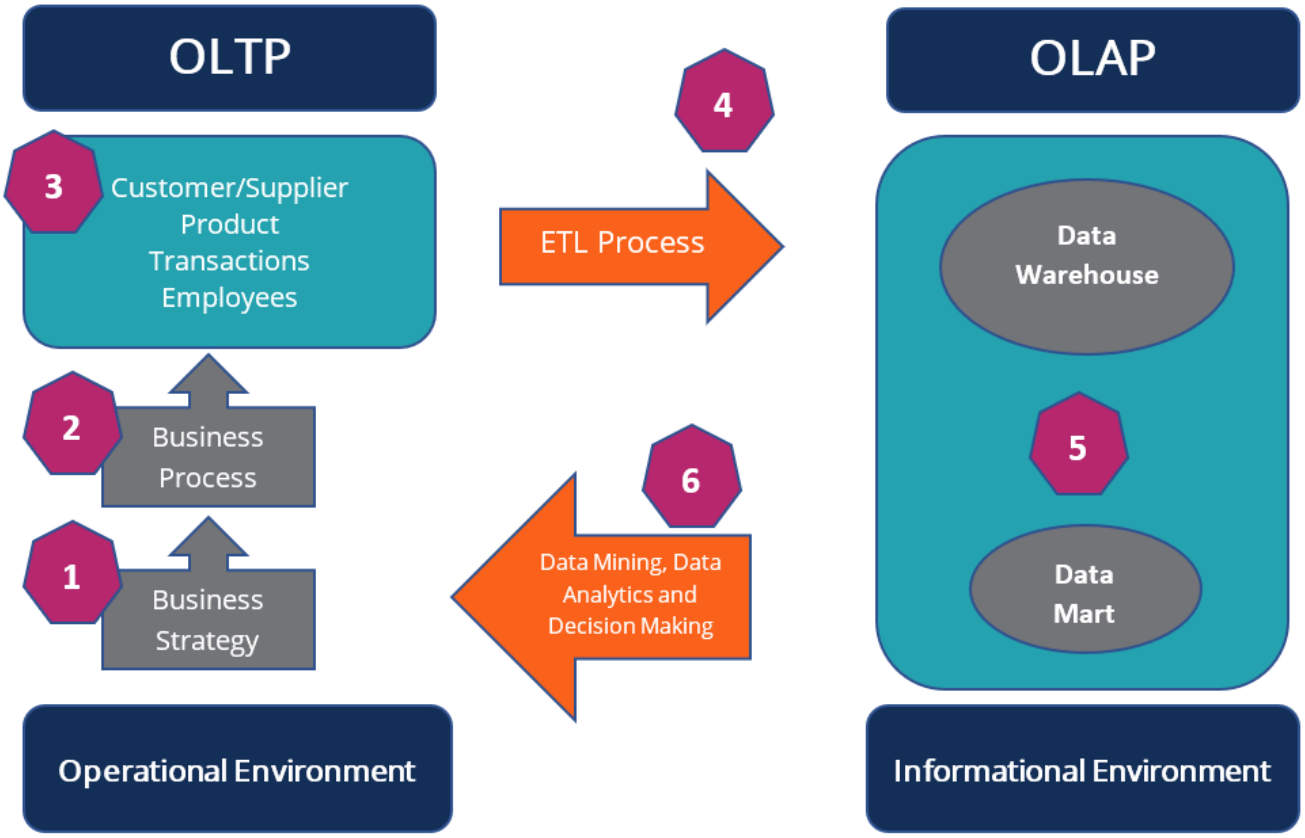
OLTP vs OLAP

Feature	OLTP	OLAP
Full Form	Online Transaction Processing	Online Analytical Processing
Purpose	Handles **day-to-day transactions**	Used for **data analysis and decision making**
Users	**Clerks, cashiers, DBAs**	**Analysts, managers, executives**
Operations	**Insert, Update, Delete** (short, atomic transactions)	**Read-heavy operations** like complex queries and analysis
Data Volume	Processes **large numbers of small transactions**	Works with **fewer but complex and large queries**
Response Time	**Milliseconds to seconds**	**Seconds to minutes (can be longer)**
Database Design	**Highly normalized** (to reduce redundancy)	**De-normalized** (for faster read access and joins)
Data Source	Real-time **current data**	Historical **aggregated data**

Feature	OLTP	OLAP
Examples	Banking systems, e-commerce, retail POS systems	Data warehouses, business reporting systems

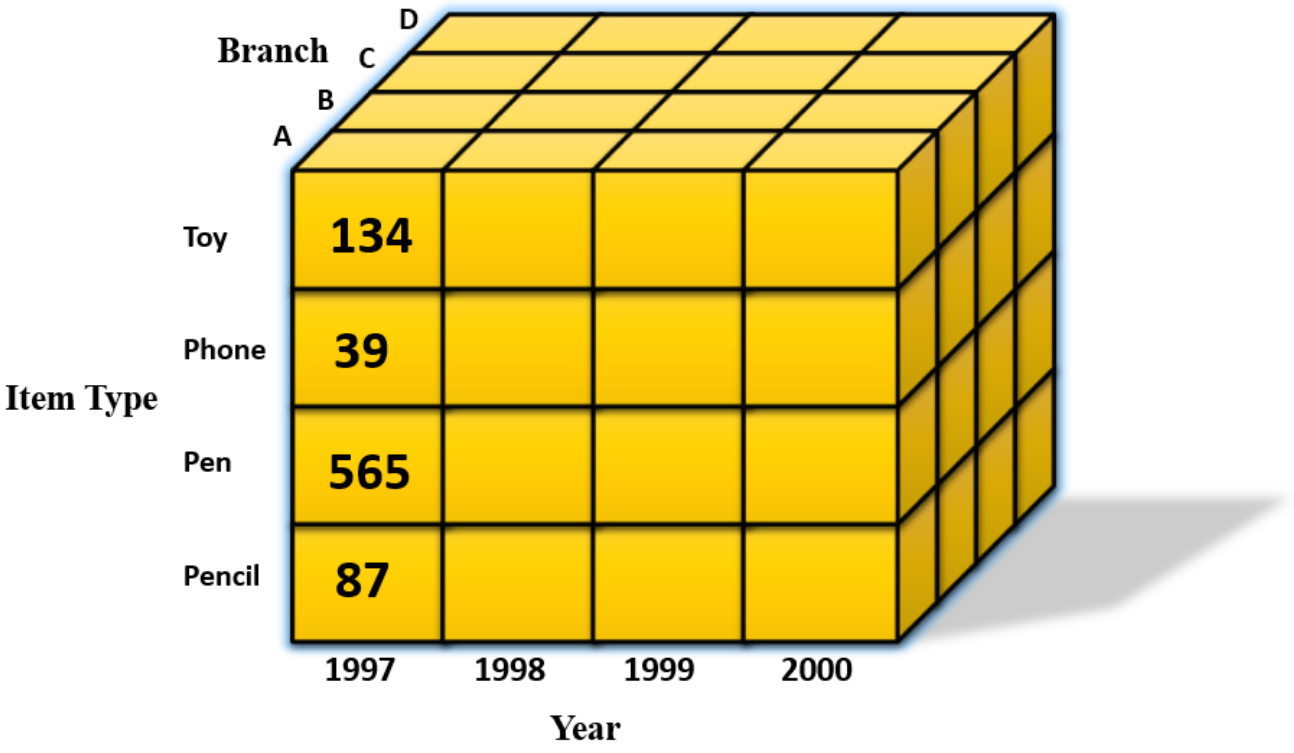
OLTP and OLAP working together

- user will give queries to the data warehouse/data mart through OLTP Database for analytical pr=urpose and the query will go to OLAP cube to acces data.



Multidimensioanl Data Model

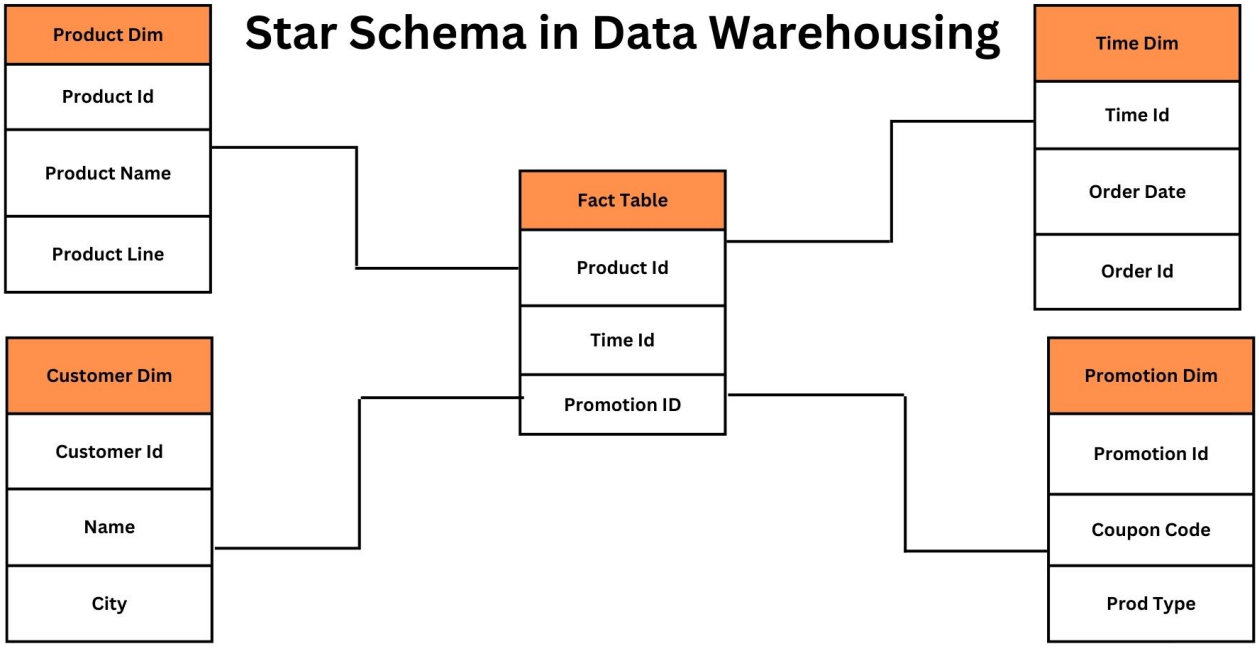
- The multidimensional datamodel is used to store the data in the form of **Data cubes**.
- A data cube allows data to be viewed in multiple dimensions.
- Here the dimensions are the entities with respect to which an organisation keeps the records.
- It provides machanism to store data and a way for business analysis.
- **Dimensions** and **Facts** are two components of multidimensional data model.
- **Dimensions** : are the text attributes to analyze data
- **Facts** : are the numeric volume to analyze business.
- It helps to provide fast and accurate data-related answers to complex business queries.



Schema for Multidimensional Model

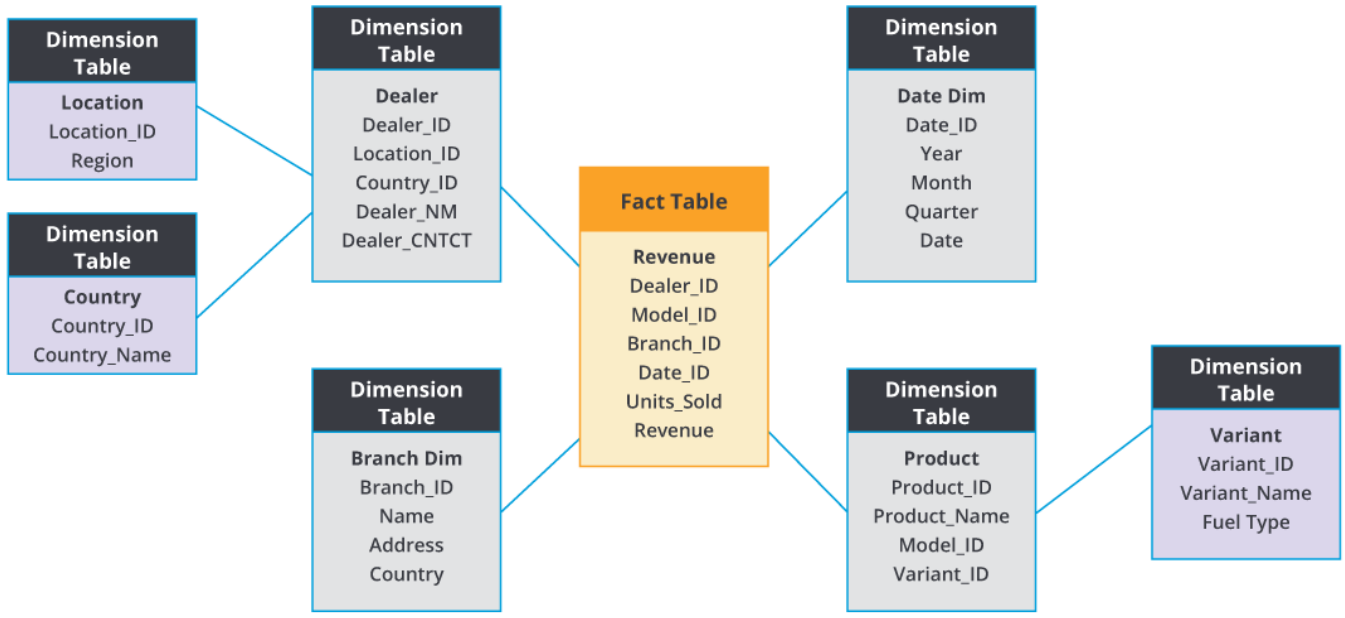
1. Star Schema

- It is the simplest data warhouse schema because it resembles a star.
- In star every Dimensional table is connected to the Fact table using primary key(dimensional table) to foreign key(Fact table) relationship.



2. Snowflake Schema

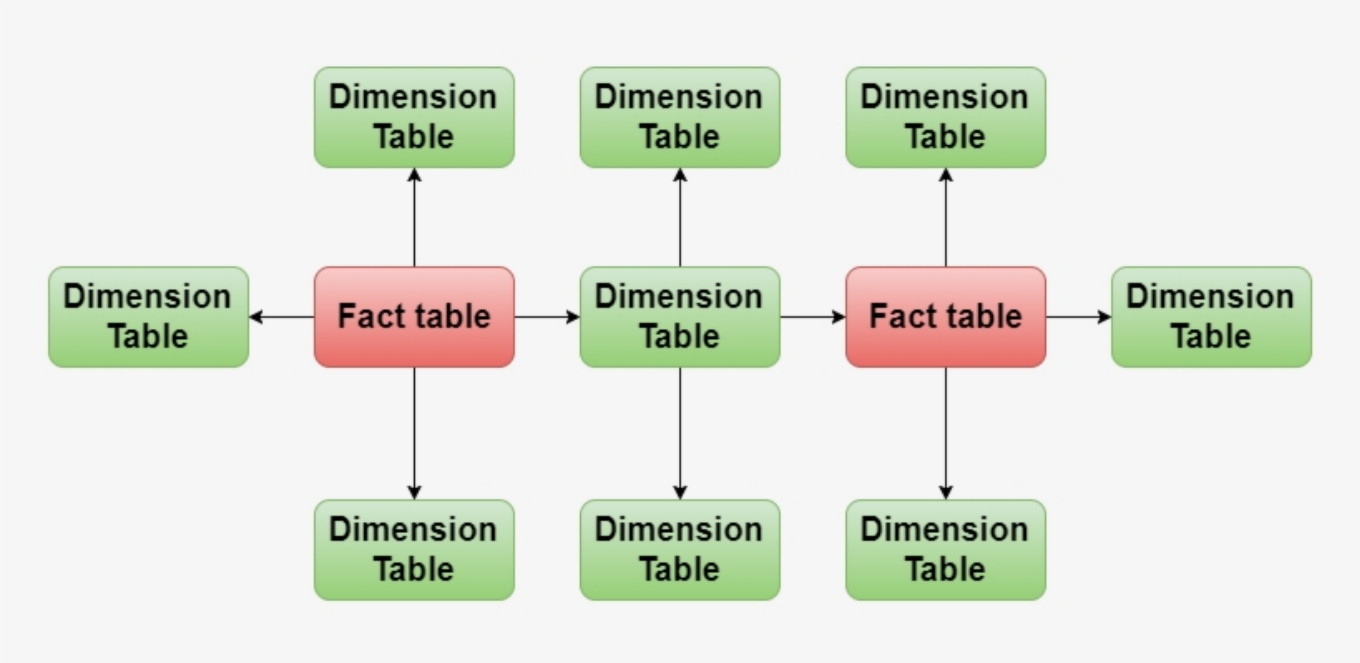
- In Snowflake schema dimensional tables are connected to other dimensional tables.
- It is more complex than Star Schema.



Example of Snowflake Schema

3. Fact Constellation Schema

- This schema has multiple fact tables, which dhare many fimensional tables.
- this type of schema can be viewed as a star and snowflake schema, hence it is also called **Galaxy Schema** or Fact Constellation Schema.
- This main disadvantage of this shema is that it has more complex design.



SQL Query Execution Order

Step	Clause	Description
1	`FROM`	Identifies source tables and joins them if needed
2	`WHERE`	Filters rows before grouping
3	`GROUP BY`	Groups rows based on specified columns
4	`HAVING`	Filters groups (after aggregation)
5	`SELECT`	Selects columns or expressions to return
6	`DISTINCT`	Removes duplicate rows (if specified)
7	`ORDER BY`	Sorts the result set
8	`LIMIT / OFFSET`	Limits the number of rows returned (if used, e.g., in MySQL/PostgreSQL)

SQL Command Types

1. DDL (Data Definition Language)

- Used to define and manage database structure (tables, schemas, etc.)
- **CREATE** : used to create new tables/databases/views/stored procedures/functions/triggers/indexes
- **ALTER** : used to modify design of a table (ADD/REMOVE/MODIFY)
- **DROP** : Remove Table/databases/views/stored procedures/functions/triggers/indexes
- **TRUNCATE** : Remove all rows from a table without logging individual row deletions. Faster than DELETE.

2. DML (Data Manipulation Language)

- Used to manipulate data in tables.
- **INSERT** : Inserting new records
- **UPDATE** : Modifying existing records
- **DELETE** : Delete specific rows from a table (with WHERE clause).

3. TCL (Transaction Control Language)

- Used to manage transactions (group of SQL operations).
- **BEGIN TRANSACTION** : Starts a new transaction block.
- **COMMIT** : Permanently saves all changes made during the transaction.
- **ROLLBACK** : Undoes changes since the last COMMIT.
- **SAVE** : Sets a point in a transaction to which you can roll back to later.

4. DCL (Data Control Language)

- Used to control access and permissions to data in the database.

- **GRANT** : Give privileges/access to users/roles (e.g., SELECT, INSERT, EXECUTE).
- **REVOKE** : Remove access/privileges granted to users/roles.

```
-- Create a new user (assumes login already exists)
CREATE USER John FOR LOGIN JohnLogin;

-- Grant SELECT permission on the Employees table to John
GRANT SELECT ON Employees TO John;

-- Revoke SELECT permission from John
REVOKE SELECT ON Employees FROM John;
```

5. DQL (Data Query Language)

- Used to query and fetch data from the database.
- **SELECT** : Retrieves data from one or more tables/views. It's the only DQL command.

Every table or any object created in DB will have schema called as **DBO.<OBJECT_Name>**

- example of objects are : Database object, tables, views,index,stored procedures, functions, etc.

SELECT COMMAND

- Used to get data from table.
- Syntax :

```
SELECT <column names with commas> from <Table name>
```

- Example :

```
-- " * " is used to fetch all records
SELECT *
FROM Employee_Table

Select Emp_name, Emp_ID
from Employee_Table
```

SQL Clauses

Clause	Purpose

Clause	Purpose
`FROM`	Specifies the **table** to retrieve data from.
`WHERE`	Filters rows **before** grouping or selection.
`GROUP BY`	Groups rows that have the same values in specified columns.
`HAVING`	Filters **groups** after `GROUP BY` (like `WHERE` but for aggregates).
`ORDER BY`	Sorts the result set by one or more columns.

FROM Clause

- The `FROM` clause specifies the **table(s)** from which to retrieve data.

☑ Key Points:

- Always follows the `SELECT` statement.
- Can include **single or multiple tables** (joins).
- Acts as the **starting point** for data retrieval.

📋 Syntax:

```
SELECT <column_name>
FROM <table_name>;
```

📌 Examples:

```
-- Select all columns from Employee_Table
SELECT *
FROM Employee_Table;

-- Select specific columns from a table
SELECT EmpName, Salary
FROM Employee_Table;
```

WHERE Clause

- The `WHERE` clause is used to **filter rows** based on a condition **before** any grouping or aggregation happens.

☑ Key Points:

- Always written **after the `FROM` clause** and **before** `GROUP BY`, `HAVING`, or `ORDER BY`.
- Used to filter individual rows before aggregation.
- Multiple conditions can be combined using logical operators like `AND`, `OR`, and `NOT`.
- For string comparisons, values must be enclosed in single quotes `' '`.

📝 Syntax:

```
SELECT <column_name>
FROM <table_name>
WHERE <condition>;
```

📌 Examples:

```
-- Select all rows where department number is 10
SELECT *
FROM Employee_Table
WHERE DeptNo = 10;

-- Select all employees whose job is 'Clerk'
SELECT *
FROM Employee_Table
WHERE Job = 'Clerk';
```

GROUP BY Clause

- The `GROUP BY` clause is used to **group rows** that have the same values in specified columns, typically used with **aggregate functions**.
- used to prepare summary report in dashboards/analysis.

☑ Key Points:

- Comes **after** `WHERE` and **before** `HAVING`.
- Must include all non-aggregated columns in the `SELECT` list.
- Enables use of functions like `COUNT()`, `SUM()`, `AVG()`, `MAX()`, `MIN()`.

📝 Syntax:

```
SELECT <column>, AGG_FUNCTION(<column>)
FROM <table>
WHERE <condition>
GROUP BY <column>;
```

📌 Examples:

```
-- Get total salary by department
SELECT DeptNo, SUM(Salary)
FROM Employee_Table
GROUP BY DeptNo;
```

```
-- Count employees in each job role
SELECT Job, COUNT(*)
FROM Employee_Table
GROUP BY Job;
```

HAVING Clause

- The `HAVING` clause is used to **filter groups** after `GROUP BY` has been applied. It works like `WHERE`, but **for aggregated data**.

☑ Key Points:

- Always used **after** `GROUP BY`.
- Cannot be used without `GROUP BY` when filtering aggregates.
- Supports aggregate functions.

📋 Syntax:

```
SELECT <column>, AGG_FUNCTION(<column>)
FROM <table>
GROUP BY <column>
HAVING <aggregate_condition>;
```

📌 Examples:

```
-- Show departments with total salary greater than 50000
SELECT DeptNo, SUM(Salary)
FROM Employee_Table
GROUP BY DeptNo
HAVING SUM(Salary) > 50000;

-- Show jobs with more than 3 employees
SELECT Job, COUNT(*)
FROM Employee_Table
GROUP BY Job
HAVING COUNT(*) > 3;
```

ORDER BY Clause

- The `ORDER BY` clause is used to **sort** the result set by one or more columns.

☑ Key Points:

- Always appears at the **end** of the query.
- Default sorting is **ascending (`ASC`)**, descending order is specified using `DESC`.
- Can sort by column name or **column position** (index in the SELECT list).

📝 Syntax:

```
SELECT <columns>
FROM <table>
ORDER BY <column> [ASC|DESC];
```

📌 Examples:

```
-- Sort employees by salary in ascending order
SELECT EmpName, Salary
FROM Employee_Table
ORDER BY Salary;

-- Sort employees by salary in descending order
SELECT EmpName, Salary
FROM Employee_Table
ORDER BY Salary DESC;

-- Sort by multiple columns
SELECT EmpName, DeptNo, Salary
FROM Employee_Table
ORDER BY DeptNo ASC, Salary DESC;
```

SQL Operators

- SQL operators are used in `WHERE`, `HAVING`, and other clauses to filter or manipulate data. They are categorized into the following types:

📊 1. Comparison Operators

Operator	Description	Example
`=`	Equal to	`Salary = 50000`
`!=` or `<>`	Not equal to	`DeptNo != 10`
`>`	Greater than	`Salary > 60000`
`<`	Less than	`Salary < 30000`
`>=`	Greater than or equal to	`Experience >= 5`
`<=`	Less than or equal to	`Age <= 40`
`BETWEEN`,`NOT BETWEEN`	Between a range (inclusive)	`Salary BETWEEN 30000 AND 60000`
`LIKE`,`NOT LIKE`	Pattern matching	`Name LIKE 'A%'`

Operator	Description	Example
`IN`,`NOT IN`	When we want to filter `OR` condition or multiple values in 1 column	`DeptNo IN (10, 20, 30)`
`IS NULL`,`IS NOT NULL`	Checks if the value is NULL	`Manager_ID IS NULL`

- 0 is a value, NULL means there is no value
- for matching strings we can also use lowercase
- `%` in string means 'n' characters at that place, e.g. `A%` can be 'Adi', 'Abhinav', etc. and `%A` can be 'Vandana', 'Sharma', 'data',etc.
- `%O%` it contains 'O'
- `S%T` it starts with 's' and ends with 't'
- `_` mean 1 single charachter.
- `_____` (i.e. 4 times _) means 4 characters, e.g. `_O___` means length is 4 and 2nd charchter is 'O'

🧠 2. Logical Operators

Operator	Description	Example
`AND`	True if **both** conditions are true	`DeptNo = 10 AND Job = 'Clerk'`
`OR`	True if **any one** condition is true	`DeptNo = 10 OR DeptNo = 20`
`NOT`	Negates a condition	`NOT (Job = 'Manager')`

🧪 3. Arithmetic Operators

Operator	Description	Example
`+`	Addition	`Salary + Bonus`
`-`	Subtraction	`Salary - Deduction`
`*`	Multiplication	`Salary * 0.1`
`/`	Division	`Salary / 12`
`%`	Modulus (remainder) *(varies)*	`Salary % 1000` *(may not work in all DBs)*

🔁 4. Assignment Operator (Used in `UPDATE`)

Operator	Description	Example
`=`	Assigns a value	`UPDATE Employee SET Salary = 50000`

🔍 5. Pattern Matching Operators (Used with `LIKE`)

Symbol	Description	Example
`%`	Zero or more characters	`Name LIKE 'A%'` (starts with A)

Symbol	Description	Example
_	Exactly one character	`Name LIKE '_a%'`

📝 Example Query Using Multiple Operators:

```
SELECT EmpName, Salary, DeptNo
FROM Employee_Table
WHERE Salary > 30000 AND DeptNo IN (10, 20)
ORDER BY Salary DESC;
```

SQL Aggregation Functions

- Aggregation (or aggregate) functions perform **calculations on multiple rows** and return a **single summary value**. They are commonly used with `GROUP BY` and `HAVING`.
- They can't be used in where clause because where clause works before aggregation takes place.
- `SUM`, `AVG`, `MAX`, `MIN`, `COUNT`

🔢 1. COUNT()

- Returns the **number of rows** that match the condition.

☑ Syntax:

```
SELECT COUNT(*) FROM <table>;
SELECT COUNT(column_name) FROM <table>;
```

📌 Examples:

```
-- Count all rows in the table
SELECT COUNT(*) FROM Employee_Table;

-- Count non-null job titles
SELECT COUNT(Job) FROM Employee_Table;
```

💰 2. SUM()

- Returns the **total sum** of a numeric column.

☑ Syntax:

```
SELECT SUM(column_name) FROM <table>;
```

📌 Example:

```
-- Total salary of all employees
SELECT SUM(Salary) FROM Employee_Table;
```

📊 3. AVG()

- Returns the **average value** of a numeric column.

☑ Syntax:

```
SELECT AVG(column_name) FROM <table>;
```

📌 Example:

```
-- Average salary of employees
SELECT AVG(Salary) FROM Employee_Table;
```

📈 4. MAX()

- Returns the **maximum value** in a column.

☑ Syntax:

```
SELECT MAX(column_name) FROM <table>;
```

📌 Example:

```
-- Highest salary among employees
SELECT MAX(Salary) FROM Employee_Table;
```

📉 5. MIN()

- Returns the **minimum value** in a column.

☑ Syntax:

```
SELECT MIN(column_name) FROM <table>;
```

📌 Example:

```
-- Lowest salary among employees
SELECT MIN(Salary) FROM Employee_Table;
```

🧠 Aggregation with GROUP BY

You can combine these functions with `GROUP BY` to summarize data by categories.

📌 Example:

```
-- Total salary by department
SELECT DeptNo, SUM(Salary)
FROM Employee_Table
GROUP BY DeptNo;

-- Count of employees in each job role
SELECT Job, COUNT(*)
FROM Employee_Table
GROUP BY Job;
```

ROLLUP function

- it extends the functionality of group by.
- adds a row to the group by output.
- when applied on 1 column only adds `grand total` i.e `1 row`
- when applied on more than 1 column adds `grand total` according to first column and `subtotals` according to 2nd column (and so on for further columns)

📝 Syntax:

```
group by Rollup(<col name>) -- must be in paranthesis
```

COALESCE function

- it replaces null value in the specified column with the specified value

📝 Syntax:

```
select COALESCE(<col name>, <value>) as <col name>,
from table
```

📌 Example:

```
-- ADD another row with null as index and null values as data which will be
replaced by 'Grand Total' due to colaesce
SELECT COALESCE(REGION, 'GRAND TOTAL') AS REGION
FROM SUM(SALES) AS 'TOTAL SALES' -- USE ' ' FOR SPACE OR SPECIAL CHARACTERS
FROM SALESORDER
GROUP BY ROLLUP(REGION)
```

ISNULL function

- used to replace null values in a specified column with specified value
- In DB calculation of anyvalue with 'NULL' is equal to 'NULL' only.
- `COALESCE` will do the same but we generally use that on text column, `ISNULL` is generally used on numerical value.

📌 Example:

```
SELECT *, SAL + ISNULL(COMM,0) AS TOTAL_SALES
FROM EMP
```

🕓 DATE AND TIME FUNCTIONS

- Date components : **Day**. **Month**, **Year**, **Quarter**, **Weekday**, **Weeknum**
- Time Components : **Hours**, **Minutes**, **Seconds**, **Milisec**, **Timezone**

🗓️ 1. DATEPART()

- Returns an **integer** representing the specified part of a date (e.g., year, month, day, hour).

☑ Syntax:

```
SELECT DATEPART(part, date_value);
```

📌 Example:

```
SELECT
    DATEPART(YEAR, '2025-06-11 14:35:25.123') AS YearPart,
    DATEPART(QUARTER, '2025-06-11 14:35:25.123') AS QuarterPart,
    DATEPART(MONTH, '2025-06-11 14:35:25.123') AS MonthPart,
    DATEPART(DAY, '2025-06-11 14:35:25.123') AS DayPart,
    DATEPART(WEEK, '2025-06-11 14:35:25.123') AS WeekPart,
    DATEPART(WEEKDAY, '2025-06-11 14:35:25.123') AS WeekdayPart,
    DATEPART(HOUR, '2025-06-11 14:35:25.123') AS HourPart,
    DATEPART(MINUTE, '2025-06-11 14:35:25.123') AS MinutePart,
    DATEPART(SECOND, '2025-06-11 14:35:25.123') AS SecondPart,
    DATEPART(MILLISECOND, '2025-06-11 14:35:25.123') AS MillisecondPart;
```

- Common parts:
  - `YEAR`, `QUARTER`, `MONTH`, `DAY`, `WEEK`, `WEEKDAY`
  - `HOUR`, `MINUTE`, `SECOND`, `MILLISECOND`

🗓️ 2. DATENAME()

- Returns the **string name** of the specified part of a date.

☑ Syntax:

```
SELECT DATENAME(part, date_value);
```

📌 Example:

```
SELECT
    DATENAME(MONTH, '2025-06-11 14:35:25.123') AS MonthName,
    DATENAME(WEEKDAY, '2025-06-11 14:35:25.123') AS WeekdayName,
```

- DATENAME() returns strings, e.g., 'June', 'Wednesday', not numbers.
- only for Month and Weekday if used on other parts then it will work like DATEPART
- Common parts:
  - `MONTH` returns 'June'
  - `WEEKDAY` returns 'Wednesday'

⏳ 3. DATEDIFF()

- Returns the **difference between two dates** in the specified unit.

☑ Syntax:

```
SELECT DATEDIFF(part, start_date, end_date);
```

📌 Example:

```
SELECT
    DATEDIFF(YEAR, '2022-01-01', '2025-06-11') AS YearDiff,
    DATEDIFF(QUARTER, '2022-01-01', '2025-06-11') AS QuarterDiff,
    DATEDIFF(MONTH, '2022-01-01', '2025-06-11') AS MonthDiff,
    DATEDIFF(DAY, '2022-01-01', '2025-06-11') AS DayDiff,
    DATEDIFF(WEEK, '2022-01-01', '2025-06-11') AS WeekDiff,
    DATEDIFF(HOUR, '2022-01-01', '2025-06-11') AS HourDiff,
    DATEDIFF(MINUTE, '2022-01-01', '2025-06-11') AS MinuteDiff,
    DATEDIFF(SECOND, '2022-01-01', '2025-06-11') AS SecondDiff;
```

- `GetDate()` is a predefined function that will give current system date.

➕ 4. DATEADD()

- Adds a specific number of units (e.g., days, months) to a date.

☑ Syntax:

```
SELECT DATEADD(part, number, date_value);
```

📌 Example:

```
SELECT
    DATEADD(YEAR, 2, '2025-06-11') AS AddYear,
    DATEADD(QUARTER, 1, '2025-06-11') AS AddQuarter,
    DATEADD(MONTH, 3, '2025-06-11') AS AddMonth,
    DATEADD(DAY, 10, '2025-06-11') AS AddDay,
    DATEADD(WEEK, 2, '2025-06-11') AS AddWeek,
    DATEADD(HOUR, 5, '2025-06-11 14:00:00') AS AddHour,
    DATEADD(MINUTE, 30, '2025-06-11 14:00:00') AS AddMinute,
    DATEADD(SECOND, 45, '2025-06-11 14:00:00') AS AddSecond;

-- Adding multiples parts like day, month or year in one query
SELECT DATEADD(MONTH, 2, DATEADD(DAY, 10, GETDATE())) AS NewDate
```

📆 5. EOMONTH()

- Add months and Returns the **last day of the month** for a given date.

☑ Syntax:

```
SELECT EOMONTH(date_value, no_of_months_to_add);
```

📌 Example:

```
SELECT EOMONTH('2025-06-11', 1) AS EndOfNextMonth;

-- End of the month for a given date without adding months
SELECT EOMONTH('2025-06-11') AS MonthEnd;

- Multiple Use Cases in One Query
SELECT
    EOMONTH('2025-06-11') AS EndOfCurrentMonth,
    EOMONTH('2025-06-11', 1) AS EndOfNextMonth,
    EOMONTH('2025-06-11', -1) AS EndOfPreviousMonth;
```

🌍 6. SWITCHOFFSET()

- Adjusts a datetimeoffset value to a new time zone **offset**.

☑ Syntax:

```
SELECT SWITCHOFFSET(datetimeoffset_value, timezone_offset);
```

📌 Example:

```
-- Change the timezone offset to +05:30 (IST)
SELECT SWITCHOFFSET(SYSDATETIMEOFFSET(), '+05:30') AS IST_Time;
```

> Useful with `SYSDATETIMEOFFSET()` to work with multiple time zones. `SYSDATETIMEOFFSET()` gives current date and time with timezone

---

🔲 SQL WINDOW FUNCTIONS

- Window functions **perform calculations across a set of table rows** related to the current row, **without collapsing rows** (unlike aggregate functions).

🎯 1. ROW_NUMBER(), RANK(), DENSE_RANK()

- **ROW_NUMBER()** : Assigns a unique **sequential integer** to rows within a partition.
- **RANK()** : Leaves gaps in ranking if there are same values.
- **DENSE_RANK()** : No gaps.

☑ Syntax:

```
SELECT ROW_NUMBER() OVER (PARTITION BY column ORDER BY column) AS RowNum
RANK() OVER (PARTITION BY column ORDER BY column)
DENSE_RANK() OVER (PARTITION BY column ORDER BY column)
FROM <table>;
```

📌 Example:

```
SELECT
    EmployeeID,
    EmployeeName,
    Department,
    Salary,

    ROW_NUMBER() OVER (PARTITION BY Department ORDER BY Salary DESC) AS RowNum,
    RANK() OVER (PARTITION BY Department ORDER BY Salary DESC) AS RankVal,
    DENSE_RANK() OVER (PARTITION BY Department ORDER BY Salary DESC) AS
DenseRankVal

FROM Employee_Table;
```

📌 Example:

```
SELECT
    EmployeeID,
    EmployeeName,
    Department,
    Salary,

    ROW_NUMBER() OVER (PARTITION BY Department ORDER BY Salary DESC) AS RowNum,
    RANK() OVER (PARTITION BY Department ORDER BY Salary DESC) AS RankVal,
    DENSE_RANK() OVER (PARTITION BY Department ORDER BY Salary DESC) AS
DenseRankVal

FROM Employee_Table;
```

- 🧹 Sample Output:

EmployeeID	Department	Salary	RowNum	RankVal	DenseRankVal
101	IT	90000	1	1	1
102	IT	90000	2	1	1
103	IT	85000	3	3	2
104	HR	75000	1	1	1
105	HR	70000	2	2	2

- 🔍 What each function does:

Function	Behavior
`ROW_NUMBER()`	Gives a unique row number regardless of ties.
`RANK()`	Gives same rank for ties, **but skips numbers** (gaps).
`DENSE_RANK()`	Gives same rank for ties, **but does not skip numbers** (no gaps).

🔢 2. NTILE()

- **NTILE(n)**: Divides rows into `n` buckets (quantiles).

☑ Syntax:

```
NTILE(n) OVER (PARTITION BY column ORDER BY column)
```

📌 Example:

```
-- Ranking and bucketing employees by salary in each department
SELECT
    EmployeeID,
    Department,
    Salary,
    RANK() OVER (PARTITION BY Department ORDER BY Salary DESC) AS RankNum,
    DENSE_RANK() OVER (PARTITION BY Department ORDER BY Salary DESC) AS DenseRank,
    NTILE(4) OVER (PARTITION BY Department ORDER BY Salary DESC) AS Quartile
FROM Employee_Table;
```

🔲 3. LAG() & LEAD()

- **LAG()**: Get **previous row's value**
- **LEAD()**: Get **next row's value**

☑ Syntax:

```
LAG(column, offset, default) OVER (ORDER BY column)
LEAD(column, offset, default) OVER (ORDER BY column)
-- default_value: (optional) value to return when the offset goes out of bounds
(default is NULL).
```

📌 Example:

```sql
-- Compare current salary with previous and next
SELECT
    EmployeeID,
    Salary,
    LAG(Salary, 1, 0) OVER (ORDER BY EmployeeID) AS PrevSalary,
    LEAD(Salary, 1, 0) OVER (ORDER BY EmployeeID) AS NextSalary
FROM Employee_Table;
```

🔢 4. FIRST_VALUE() & LAST_VALUE()

- Return the **first or last value** in a window frame.

☑ Syntax:

```sql
FIRST_VALUE(column) OVER (PARTITION BY column ORDER BY column)
LAST_VALUE(column) OVER (PARTITION BY column ORDER BY column ROWS BETWEEN
UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING)
```

📌 Example:

```sql
-- Get first and last salary in each department
SELECT
    EmployeeID,
    Department,
    Salary,
    FIRST_VALUE(Salary) OVER (PARTITION BY Department ORDER BY Salary) AS
FirstSal,
    LAST_VALUE(Salary) OVER (
        PARTITION BY Department
        ORDER BY Salary
        ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING
    ) AS LastSal
FROM Employee_Table;
```

📊 5. Aggregate Functions with OVER()

Use aggregate functions like `SUM()`, `AVG()`, `MIN()`, `MAX()` **without collapsing rows**.

☑ Syntax:

```sql
SELECT SUM(column) OVER (PARTITION BY column) AS Total
```

📌 Example:

```sql
-- Total and average salary per department
SELECT
    EmployeeID,
    Department,
    Salary,
    SUM(Salary) OVER (PARTITION BY Department) AS DeptTotalSalary,
    AVG(Salary) OVER (PARTITION BY Department) AS DeptAvgSalary,
    MIN(Salary) OVER (PARTITION BY Department) AS DeptMinSalary,
    MAX(Salary) OVER (PARTITION BY Department) AS DeptMaxSalary
FROM Employee_Table;
```

📌 All in One Query (Ultimate Window Function Demo)

```sql
SELECT
    EmployeeID,
    Department,
    Salary,

    ROW_NUMBER() OVER (PARTITION BY Department ORDER BY Salary DESC) AS RowNum,
    RANK() OVER (PARTITION BY Department ORDER BY Salary DESC) AS RankNum,
    DENSE_RANK() OVER (PARTITION BY Department ORDER BY Salary DESC) AS DenseRank,
    NTILE(4) OVER (PARTITION BY Department ORDER BY Salary DESC) AS Quartile,

    LAG(Salary, 1, 0) OVER (PARTITION BY Department ORDER BY Salary DESC) AS
PrevSalary,
    LEAD(Salary, 1, 0) OVER (PARTITION BY Department ORDER BY Salary DESC) AS
NextSalary,

    FIRST_VALUE(Salary) OVER (PARTITION BY Department ORDER BY Salary ASC) AS
FirstSal,
    LAST_VALUE(Salary) OVER (
        PARTITION BY Department ORDER BY Salary ASC
        ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING
    ) AS LastSal,

    SUM(Salary) OVER (PARTITION BY Department) AS DeptTotal,
    AVG(Salary) OVER (PARTITION BY Department) AS DeptAvg,
    MIN(Salary) OVER (PARTITION BY Department) AS DeptMin,
    MAX(Salary) OVER (PARTITION BY Department) AS DeptMax

FROM Employee_Table;
```

📝 SQL Server TEXT FUNCTIONS

🎀 1. LEN()

- Returns the **length of a string** (excluding trailing spaces).

☑ Syntax:

```
SELECT LEN(column_name) FROM <table>;
```

📌 Example:

```
SELECT LEN('SQL Server') AS LengthOfString;
-- Output: 10
```

🔠 2. UPPER() and LOWER()

- **UPPER()** converts text to **uppercase**.
- **LOWER()** converts text to **lowercase**.

📌 Example:

```
SELECT
    UPPER('sql server') AS UpperCaseText,
    LOWER('SQL SERVER') AS LowerCaseText;
```

🔍 3. CHARINDEX()

- Returns the **starting position** of a substring inside a string.

☑ Syntax:

```
CHARINDEX('substring', 'main_string')
```

📌 Example:

```
SELECT CHARINDEX('Server', 'SQL Server') AS Position;
-- Output: 5
```

🧬 4. SUBSTRING()

- Extracts a **portion** of a string.

☑ Syntax:

```
SUBSTRING(string, start_position, length)
```

📌 Example:

```
SELECT SUBSTRING('SQL Server 2025', 5, 6) AS SubPart;
-- Output: Server
```

🔗 5. CONCAT()

- Combines multiple strings into one.

📌 Example:

```
SELECT CONCAT('SQL', ' ', 'Server') AS FullText;
-- Output: SQL Server
```

🧹 6. LTRIM() & RTRIM()

- **LTRIM()** removes leading spaces.
- **RTRIM()** removes trailing spaces.

📌 Example:

```
SELECT
    LTRIM('   Hello') AS LeftTrimmed,
    RTRIM('World   ') AS RightTrimmed;
```

🧱 7. REPLACE()

- Replaces all occurrences of a substring.

☑ Syntax:

```
REPLACE('original_string', 'search_string', 'replace_string')
```

📌 Example:

```
SELECT REPLACE('SQL 2019', '2019', '2025') AS NewVersion;
-- Output: SQL 2025
```

🔠 8. LEFT() and RIGHT()

- Extracts a specified number of characters from the **left** or **right**.

📌 Example:

```
SELECT
    LEFT('SQLServer', 3) AS LeftPart,     -- Output: SQL
    RIGHT('SQLServer', 6) AS RightPart;   -- Output: Server
```

🧠 9. ASCII() and CHAR()

- `ASCII()` returns the **ASCII code** of the first character.
- `CHAR()` returns the **character** from an ASCII code.

📌 Example:

```
SELECT
    ASCII('A') AS AsciiValue,    -- Output: 65
    CHAR(66) AS CharFromAscii;   -- Output: B
```

🧾 10. REPLICATE()

- Repeats a string a given number of times.

📌 Example:

```
SELECT REPLICATE('*', 5) AS Stars;
-- Output: *****
```

📌 All Common String Functions Together in One Query

```
SELECT
    ' SQL Server 2025 ' AS OriginalText,

    LEN(' SQL Server 2025 ') AS Length,
    UPPER('sql server') AS UpperText,
    LOWER('SQL SERVER') AS LowerText,
    CHARINDEX('Server', 'SQL Server 2025') AS ServerPosition,
    SUBSTRING('SQL Server 2025', 5, 6) AS ExtractedWord,
    CONCAT('SQL', ' ', 'Server') AS CombinedText,
    LTRIM('   Hello') AS LTrimmed,
    RTRIM('World   ') AS RTrimmed,
    REPLACE('SQL 2019', '2019', '2025') AS ReplacedText,
    LEFT('SQLServer', 3) AS LeftSide,
    RIGHT('SQLServer', 6) AS RightSide,
    ASCII('A') AS AsciiCode,
```

```
    CHAR(66) AS CharFromAscii,
    REPLICATE('*', 5) AS StarsRepeated;
```

🧮 SQL Server MATH FUNCTIONS

➕ 1. ABS()

- Returns the **absolute value**.

☑ Syntax:

```
SELECT ABS(-25) AS AbsoluteValue;
-- Output: 25
```

🔢 2. CEILING() and FLOOR()

- **CEILING()**: Rounds **up** to the nearest integer.
- **FLOOR()**: Rounds **down** to the nearest integer.

📌 Example:

```
SELECT
    CEILING(12.3) AS RoundUp,    -- Output: 13
    FLOOR(12.7) AS RoundDown;    -- Output: 12
```

⏳ 3. ROUND()

- Rounds to the specified number of decimal places.

☑ Syntax:

```
ROUND(number, decimal_places)
```

📌 Example:

```
SELECT ROUND(123.4567, 2) AS RoundedVal;
-- Output: 123.46
```

🧠 4. POWER() and SQRT()

- **POWER(x, y)**: x to the power y.
- **SQRT()**: Square root.

📌 Example:

```
SELECT
    POWER(2, 4) AS PowerValue,      -- Output: 16
    SQRT(49) AS SquareRootValue;    -- Output: 7
```

💯 5. PI(), EXP(), LOG(), LOG10()

- **PI()**: Returns π.
- **EXP(x)**: e to the power of x.
- **LOG(x)**: Natural logarithm.
- **LOG10(x)**: Base-10 logarithm.

📌 Example:

```
SELECT
    PI() AS PiValue,
    EXP(1) AS Exponential,
    LOG(10) AS NaturalLog,
    LOG10(100) AS LogBase10;
```

📌 All Common Math Functions Together in One Query

```
SELECT
    ABS(-10) AS AbsoluteVal,
    CEILING(12.3) AS CeilingVal,
    FLOOR(12.7) AS FloorVal,
    ROUND(123.4567, 2) AS RoundedVal,
    POWER(3, 2) AS PowerVal,
    SQRT(81) AS SqrtVal,
    PI() AS PiValue,
    EXP(1) AS ExpVal,
    LOG(10) AS NaturalLog,
    LOG10(100) AS LogBase10;
```

🗒️ SQL Server CONDITIONAL FUNCTIONS

📍 1. CASE WHEN

- Acts like an **IF-THEN-ELSE** statement.

☑ Syntax:

```
CASE
    WHEN condition1 THEN result1
    WHEN condition2 THEN result2
    ELSE default_result
END
```

📌 Example:

```
SELECT
    Salary,
    CASE
        WHEN Salary >= 80000 THEN 'High'
        WHEN Salary >= 50000 THEN 'Medium'
        ELSE 'Low'
    END AS SalaryLevel
FROM Employee_Table;
```

🧾 2. IIF()

- Shorter version of `CASE`; works like a single-line IF.

☑ Syntax:

```
IIF(condition, true_value, false_value)
```

📌 Example:

```
SELECT
    Salary,
    IIF(Salary > 60000, 'Eligible', 'Not Eligible') AS BonusStatus
FROM Employee_Table;

-- multiple IIFs
SELECT
    Salary,
    IIF(Salary > 60000, 'Grade A', IIF(Salary > 30000, 'GRade B', 'Grade C')) AS
GRADES
FROM Employee_Table;
```

🤯 3. NULLIF()

- Returns **NULL** if two expressions are equal; else returns the first expression.

📌 Example:

```
SELECT NULLIF(100, 100) AS Result;  -- Output: NULL
SELECT NULLIF(100, 200) AS Result;  -- Output: 100
```

📌 All Common Conditional Functions Together in One Query

```
SELECT
    EmployeeID,
    Salary,

    -- CASE for salary levels
    CASE
        WHEN Salary >= 80000 THEN 'High'
        WHEN Salary >= 50000 THEN 'Medium'
        ELSE 'Low'
    END AS SalaryLevel,

    -- IIF for bonus eligibility
    IIF(Salary >= 60000, 'Eligible', 'Not Eligible') AS BonusStatus,

    -- NULLIF example
    NULLIF(Salary, 0) AS NullIfSalaryZero,

FROM Employee_Table;
```

JOINS

- Basically used to get information from multiple tables in 1 select statement.
- The tables must have a common column to join them.
- When we use `Join` then `Inner Join` is applied by default.

Types of joins

Join Type	Returns
`INNER JOIN`	Only matching rows from both tables.
`LEFT JOIN`	All rows from the left table + matching rows from the right (NULL if no match).
`RIGHT JOIN`	All rows from the right table + matching rows from the left (NULL if no match).
`FULL JOIN`	All rows from both tables (NULLs where there's no match).

1. INNER JOIN

- Returns only the **matching rows** from both tables.

📋 Syntax:

```
SELECT a.column1, b.column2
FROM TableA a
INNER JOIN TableB b
ON a.common_column = b.common_column;
```

📌 Example:

```
SELECT E.EmpName, D.DeptName
FROM Employee E
INNER JOIN Department D
ON E.DeptID = D.DeptID;
-- giving alias to tables make it easier to work with them when we have to write
larger queries
-- an alias is limited to one script only
```

☑ Only employees who belong to a valid department will be shown.

2. LEFT JOIN (LEFT OUTER JOIN)

- Returns **all rows from the left table** and **matched rows from the right**. If no match, NULLs are returned for the right table.

📋 Syntax:

```
SELECT a.column1, b.column2
FROM TableA a
LEFT JOIN TableB b
ON a.common_column = b.common_column;
```

📌 Example:

```
SELECT E.EmpName, D.DeptName
FROM Employee E
LEFT JOIN Department D
ON E.DeptID = D.DeptID;
```

☑ Includes employees even if they don't belong to any department.

3. RIGHT JOIN (RIGHT OUTER JOIN)

- Returns **all rows from the right table** and **matched rows from the left**. If no match, NULLs are returned for the left table.

📋 Syntax:

```
SELECT a.column1, b.column2
FROM TableA a
RIGHT JOIN TableB b
ON a.common_column = b.common_column;
```

📌 Example:

```
SELECT E.EmpName, D.DeptName
FROM Employee E
RIGHT JOIN Department D
ON E.DeptID = D.DeptID;
```

> ☑ Includes all departments even if they have no employees.

4. FULL JOIN (FULL OUTER JOIN)

- Returns **all rows from both tables**. If there's no match, NULLs are shown for non-matching sides.

📋 Syntax:

```
SELECT a.column1, b.column2
FROM TableA a
FULL JOIN TableB b
ON a.common_column = b.common_column;
```

📌 Example:

```
SELECT E.EmpName, D.DeptName
FROM Employee E
FULL OUTER JOIN Department D
ON E.DeptID = D.DeptID;
```

> ☑ Includes all employees and all departments — even unmatched ones.

SELF JOIN

- A table that joins to itself.

📋 Syntax:

```
SELECT a.column1, b.column2
FROM TableA a
INNER JOIN TableA b
ON a.common_column = b.common_column;
```

📌 Example:

```
SELECT *
FROM EMP A
INNER JOIN EMP B
ON A.MGR = B.MGR;
```

How to Join Multiple Tables

```
SELECT * FROM EMP A
LEFT JOIN DEPT B
ON A.EMPID = B.EMPID
INNER JOIN PRODUCT P
ON P.DEPTNO = B.DEPTNO
```

UPDATE JOIN

- It will update values from table A to table B, which satisfy join consition
- INNER join only

📌 Example:

```
UPDATE Emp
SET Emp.Salary = Emp_New.Salary,
    Emp.Job = Emp_New.Job
FROM Emp_New
INNER JOIN Emp
ON Emp_New.IDNO = Emp.EmpNo
```

MERGE JOIN

- It will update if data is present in records and add data to records if not present.

📌 Example:

```
MERGE EMP AS TARGET
USING EMP_NEW AS SOURCE
```

```
ON TARGET.EMPNO = SOURCE.IDNO
WHEN MATCHED THEN UPDATE
SET TARGET.SAL = SOURCE.SAL,
    TARGET.JOB = SOURCE.JOB
WHEN NOT MATCHED BY TARGET THEN INSERT (EMPNO, SAL, JOB)
VALUES (SOURCE.IDNO, SOURCE.SAL, SOURCE.JOB)
WHEN NOT MATCHED BY SOURCE THEN DELETE:
```

- Mention the table name in `WHEN NOT MATCHED`.

SQL SET OPERATORS

- Combine results from two or more `SELECT` statements.
- **Types of set operators in SQL:**
  - `UNION` : Appends without repetition (removes duplicates).
  - `UNION ALL` : Appends all rows (includes duplicates).
  - `INTERSECT` : Returns only the **common** records.
  - `EXCEPT` : Returns records from the **first query only**, that are not in the second.

☑ Conditions

- The **number of columns** in both queries must be **equal**.
- The **data types** of corresponding columns must be **compatible**.
- **Column names** in the result set are taken from the **first query**.

◇ UNION

- Removes duplicates between results from both queries.

📌 Example:

```
SELECT * FROM EMP WHERE DEPTNO = 10
UNION
SELECT * FROM EMP WHERE JOB = 'CLERK';
```

◇ UNION ALL

- Includes **all rows**, even duplicates.

📌 Example:

```
SELECT * FROM EMP WHERE DEPTNO = 10
UNION ALL
```

```
SELECT * FROM EMP WHERE JOB = 'CLERK';
```

◇ INTERSECT

- Returns **only the common rows** between the two queries.

📌 Example:

```
SELECT * FROM EMP WHERE DEPTNO = 10
INTERSECT
SELECT * FROM EMP WHERE JOB = 'CLERK';
```

◇ EXCEPT

- Returns rows from the **first query** that are **not present** in the second.

📌 Example:

```
SELECT * FROM EMP WHERE DEPTNO = 10
EXCEPT
SELECT * FROM EMP WHERE JOB = 'CLERK';
```

SQL Data Types

Data Type	Description	Example
`VARCHAR(n)`	Variable-length **text**, up to **n characters**	`EmpName VARCHAR(50)`
`CHAR(n)`	Fixed-length **text**, always stores **n characters**	`Gender CHAR(1)`
`TEXT`	Long variable-length text (non-indexed in many databases)	`Comments TEXT`
`INT` / `INTEGER`	Whole numbers (positive or negative)	`Age INT`
`SMALLINT`	Smaller-range whole numbers	`Rating SMALLINT`
`BIGINT`	Very large whole numbers	`Population BIGINT`
`DECIMAL(p,s)`	Exact numeric with precision `p` and `s` no. of decimals	`Salary DECIMAL(10,2)`
`NUMERIC(p,s)`	Same as `DECIMAL` (standard SQL), if s not mentioned then no decimals	`Price NUMERIC(8,2)`

Data Type	Description	Example
`FLOAT`	Approximate floating-point number	`Temperature FLOAT`
`REAL` / `DOUBLE`	More precision than `FLOAT`	`Longitude DOUBLE`
`BOOLEAN`	Logical TRUE or FALSE	`IsActive BOOLEAN`
`DATE`	Stores a calendar date (YYYY-MM-DD)	`BirthDate DATE`
`TIME`	Stores time (HH:MM:SS)	`LoginTime TIME`
`DATETIME` / `TIMESTAMP`	Stores both date and time	`CreatedAt DATETIME`
`BLOB`	Binary Large Object (used for images, audio, etc.)	`Photo BLOB`
`NVARCHAR(n)`	Variable-length **Unicode** text (for multi-language support)	`CityName NVARCHAR(100)`
`NCHAR(n)`	Fixed-length **Unicode** text	`Language NCHAR(2)`
`ENUM('a','b',...)`	A string object with a predefined set of values (MySQL specific)	`Gender ENUM('Male', 'Female')`
`SET('a','b',...)`	A string object that can store **multiple** predefined values (MySQL specific)	`Hobbies SET('Music','Sports','Books')`
`JSON`	Stores JSON-formatted data (MySQL/PostgreSQL)	`Preferences JSON`
`XML`	Stores XML data (used in SQL Server)	`ConfigData XML`
`YEAR`	Stores a year in 2-digit or 4-digit format (MySQL-specific)	`JoinYear YEAR`

📝 Notes:

- Use `VARCHAR` for **names, emails, addresses** etc.
- Use `INT`, `DECIMAL`, or `FLOAT` for **numeric data**.
- Use `DATE`, `TIME`, or `DATETIME` for **temporal data**.
- Choose data types carefully to **optimize performance and storage**.

DDL Commands (Data Definition Language)

- Used to define and manage database structure (tables, schemas, etc.)
- **CREATE** : used to create new tables/databases/views/stored procedures/functions/triggers/indexes
- **ALTER** : used to modify design of a table (ADD/REMOVE/MODIFY)
- **DROP** : Remove Table/databases/views/stored procedures/functions/triggers/indexes
- **TRUNCATE** : Remove all rows from a table without logging individual row deletions. Faster than DELETE.

CREATE : To Create a Table

Syntax:

```
CREATE TABLE <TABLE NAME> (
                            <COL NAME>  <DATA TYPE>,
                            <COL NAME>  <DATA TYPE>,
                            <COL NAME>  <DATA TYPE>,
                            ...
                          )
```

Example:

```
CREATE TABLE EMP_001
(
EMPID  VARCHAR(50)  NOT NULL, -- NOT NULL IS A CONTRAINT
ENAME  VARCHAR(100),
SAL    NUMERIC(10,2),
DOJ    DATE,
DNAME  VARCHAR(50)
)
```

- To see table description :

```
SP_HELP <TABLE NAME>  -- SP stands for stored procedure
```

- A stored procedure is a group of SQL statements that are stored in a database and can be executed as a single unit

What Can `CREATE` Be Used For in SQL?

Statement	Purpose
`CREATE DATABASE`	Creates a new database
`CREATE TABLE`	Creates a new table with specified columns and data types
`CREATE VIEW`	Creates a virtual table (view) based on a SELECT query
`CREATE INDEX`	Creates an index on one or more columns to improve query performance
`CREATE UNIQUE INDEX`	Creates an index that ensures all values in the column(s) are unique
`CREATE SCHEMA`	Creates a new schema (logical container for database objects)
`CREATE PROCEDURE`	Defines a stored procedure (set of SQL statements that can be reused)
`CREATE FUNCTION`	Defines a user-defined function that returns a value

Statement	Purpose
`CREATE TRIGGER`	Creates a trigger that runs automatically on specified table events
`CREATE ROLE`	Creates a database role for managing permissions
`CREATE USER`	Creates a new database user (depends on DBMS and user permissions)
`CREATE TYPE`	Creates a user-defined data type (used in SQL Server, PostgreSQL, etc.)
`CREATE SEQUENCE`	Creates a sequence object for generating numeric values (like auto-increment)
`CREATE SYNONYM` (SQL Server)	Creates an alias or alternative name for another database object

📌 Examples

1. Create a Database

```
CREATE DATABASE SchoolDB;
```

2. Create a Table

```
CREATE TABLE Students (
  StudentID INT PRIMARY KEY,
  Name VARCHAR(100),
  Age INT,
  EnrolledDate DATE
);
```

3. Create a View

```
CREATE VIEW ActiveStudents AS
SELECT Name, Age FROM Students WHERE EnrolledDate >= '2023-01-01';
```

4. Create an Index

```
CREATE INDEX idx_name ON Students(Name);
```

5. Create a Stored Procedure

```
CREATE PROCEDURE GetStudentByID @ID INT
AS
BEGIN
  SELECT * FROM Students WHERE StudentID = @ID;
END;
```

DROP : To Remove a Table

📋 Syntax:

```
DROP TABLE <TABLE NAME>
```

📌 Example:

```
DROP TABLE EMP_OO1
```

Add New Columns to the Table

📋 Syntax:

```
ALTER TABLE <TABLE NAME> ADD <COL NAME> <DATA TYPE>
```

📌 Example:

```
ALTER TABLE EMP_001 ADD EMAILID VARCHAR(50)
```

Remove Columns

📋 Syntax:

```
ALTER TABLE <TABLE NAME> DROP COLUMN <COL NAME>
```

📌 Example:

```
ALTER TABLE EMP_001 DROP COLUMN CONTACT_01
```

Modify Data Type of Existing Column**

📝 Syntax:

```
ALTER TABLE <TABLE NAME> ALTER COLUMN <COL NAME> <NEW DATA TYPE>
```

📌 Example:

```
ALTER TABLE EMP_001 ALTER COLUMN EMAILID VARCHAR(50)
```

Rename Column Name

📝 Syntax:

```
SP_RENMAE '<TABLE NAME>.<OLD COL NAME>', '<NEW COL NAME>'
```

📌 Example:

```
SP_RENAME 'EMP_001.EMAILID', 'EMAIL'
```

Here's a detailed explanation of the `TRUNCATE` statement in SQL:

TRUNCATE

- The `TRUNCATE` statement is used to **quickly delete all rows** from a table **without logging each row deletion**.
- It is faster and uses fewer system and transaction log resources than `DELETE`.
-

📝 Syntax:

```
TRUNCATE TABLE table_name;
```

🆚 `TRUNCATE` vs `DELETE` vs `DROP`

Feature	`TRUNCATE`	`DELETE`	`DROP`
Action	Removes **all rows**	Removes **selected rows** (or all if no WHERE)	Deletes **entire table structure**
`WHERE` clause	❌ Not allowed	☑ Allowed	❌ Not applicable

Feature	TRUNCATE	DELETE	DROP
Speed	☑ Very fast	❌ Slower (row-by-row logging)	☑ Very fast
Rollback	❌ Usually **cannot be rolled back** (unless inside a transaction in some DBMSs)	☑ Can be rolled back	❌ Cannot be rolled back
Resets identity?	☑ Yes (in most DBMSs like SQL Server)	❌ No	❌ N/A
Affects schema?	❌ No	❌ No	☑ Yes (removes the table)
Triggers fired?	❌ No	☑ Yes	❌ N/A

📌 Example

```
-- Before
SELECT COUNT(*) FROM Employees;  -- Output: 1000

-- Truncate the table
TRUNCATE TABLE Employees;

-- After
SELECT COUNT(*) FROM Employees;  -- Output: 0
```

- You **cannot use** `TRUNCATE` on a table that is **referenced by a foreign key constraint**.
- You **cannot truncate** a **specific set of rows** — it's all or nothing.
- In SQL Server: `TRUNCATE` resets `IDENTITY` columns to the seed value.

DML Commands (Data Manipulation language)

- Used to manipulate data in tables.
- **INSERT** : Inserting new records
- **UPDATE** : Modifying existing records
- **DELETE** : Delete specific rows from a table (with WHERE clause)

INSERT

- Used to insert new rows in a table
- The order of inserting data must be the same as order of the columns present in a table

Method 1 : Supply all the values

📝 **Syntax:**

```
INSERT INTO <TABLE>
VALUES ( VALUE1, VALUE2,...)
-- In case I only have a few values then I will mention other values as 'NUll'
```

📌 **Example:**

```
INSERT INTO EMP_001
VALUES ('A1101','ABCD',1000,'01-JAN-2024','HR','A@GMAIL.COM',98121*****)

INSERT INTO EMP_001
VALUES ('A1101','ABCD',1000,'NUll','NUll','NUll','NUll')
```

Method 2 : Supply columns for which we are inserting the values

- Rest of the values will be 'NULL' by default

📝 **Syntax:**

```
INSERT INTO <TABLE>(<COL 1>, <COL 2>, ... ,<COL n>)
VALUES ( VALUE1, VALUE2,...,VALUEN),
```

📌 **Example:**

```
INSERT INTO EMP_001(EMPID, ENAME)
VALUES ('A1104','PQRS')
```

Method 3 : Enter Multiple Records

📝 **Syntax:**

```
INSERT INTO <TABLE>(<COL 1>, <COL 2>, ... ,<COL n>)
VALUES ( VALUE1, VALUE2,...,VALUEN),
VALUES ( VALUE1, VALUE2,...,VALUEN),
...
VALUES ( VALUE1, VALUE2,...,VALUEN)
```

📌 **Example:**

```
INSERT INTO EMP_001(EMPID, ENAME)
VALUES ('A1104','PQRS')
VALUES ('A1105','QWER')
VALUES ('A1106','A')
```

DELETE

- Delete all the records, or specified records

📋 Syntax:

```
-- all records
DELETE FROM <TABLE>

-- delete specified records
DELETE FROM <TABLE> WHERE <CONDITION>
```

📌 Example:

```
-- all records
DELETE FROM EMP_001

-- delete specified records
DELETE FROM EMP_001 WHERE EMPID = 'A1106'
```

UPDATE

- Update existing records

📋 Syntax:

```
-- all records
UPDATE <TABLE> SET <COL> = <VALUE>

-- UPDATE specified records
UPDATE <TABLE> SET <COL> = <VALUE>
WHERE <CONDITION>
```

📌 Example:

```
-- all records
UPDATE EMP_001 SET SAL = 25000
```

```
-- UPDATE specified records
UPDATE EMP_001 SET SAL = 25000
WHERE EMPID = 'B1101'
```

CONSTRAINTS

- Rules on columns in a table.
- Examples : no duplicates, salary>=25000, date between 1st Jan 2025 to 31st Dec 2025, Ename should not appect null values.
- For Every rulw there is a key word that we need to apply.
- list of constraints:
    - **NOT NULL**
    - **UNIQUE**
    - **PRIMARY KEY**
    - **FOREIGN KEY**
    - **DEFAULT**
    - **CHECK**

UNIQUE

- No duplicates allowed in a column.

📝 Syntax:

```
CREATE TABLE Users (
                    Email VARCHAR(255) UNIQUE
                   );
```

NOT NULL

- Null Values are not allowed in a column

📝 Syntax:

```
CREATE TABLE Student (
                      StudentID  INT           NOT NULL,
                      Name       VARCHAR(100)  NOT NULL
                     );
```

NOT NULL + UNIQUE

📝 Syntax:

```
CREATE TABLE Student (
                      ID          INT            NOT NULL  UNIQUE,
                      Name        VARCHAR(100)   NOT NULL
                     );
```

CHECK

- Condition based on validation.

📝 Syntax:

```
CREATE TABLE Accounts (
                       AccID      INT               PRIMARY KEY,
                       Balance    DECIMAL(10, 2)    CHECK (Balance >= 0)
                      );
```

PRIMARY KEY

- When applied on column, column should not accept null and duplicate values i.e. NOT NULL + UNIQUE
- Only 1 Primary key per table.

📝 Syntax:

```
CREATE TABLE Employees (
                        EmpID    INT            PRIMARY KEY,
                        Name     VARCHAR(100)
                       );
```

FOREIGN KEY

- It is the refrence to a column in another table (Primary key)
- While adding values in foreign key column make sure that the same value exists in primary key column.
- It accepts a duplicate value

📝 Syntax:

```
CREATE TABLE Departments (
                          DeptID      INT            PRIMARY KEY,
                          DeptName    VARCHAR(100)
                         );

CREATE TABLE Employees (
                        EmpID       INT             PRIMARY KEY,
                        Name        VARCHAR(100),
                        DeptID      INT,
```

```
                        FOREIGN KEY (DeptID) REFERENCES Departments(DeptID)
                       );
-- OR
CREATE TABLE Employees (
                        EmpID      INT             PRIMARY KEY,
                        Name       VARCHAR(100),
                        DeptID     INT             FOREIGN KEY REFERENCES
Departments(DeptID)
                       );
```

DEFAULT

- When mention what default value needs to be taken instead of null values.

📝 Syntax:

```
CREATE TABLE Orders (
                     OrderID   INT           PRIMARY KEY,
                     Status    VARCHAR(20)   DEFAULT 'Pending'
                    );
```

🧱 Add constraints to an existing table:

📝 Syntax:

```
-- Add a UNIQUE constraint
ALTER TABLE Users
ADD CONSTRAINT uc_email UNIQUE (Email);

-- Add a FOREIGN KEY
ALTER TABLE Employees
ADD CONSTRAINT fk_dept FOREIGN KEY (DeptID) REFERENCES Departments(DeptID);
```

View Constraints available on a Table

📝 Syntax:

```
SP_HELP <Table Name>
-- In this see at bottom
-- Here we can see the name of the constraint
```

Drop Constraints

📝 Syntax:

```
ALTER TABLE <TABLE NAME> DROP CONSTRAINT <CONSTRAINT NAME>

-- EXAMPLE :
ALTER TABLE CS_002 DROP CONSTRAINT UQ_CS_002_B87DC94ADIF4B2E9
```

☑ Applying Constraints on a Group of Columns

1. Composite `PRIMARY KEY`

- Used when a combination of two or more columns **uniquely identifies each row**.

📝 Syntax:

```
CREATE TABLE Enrollment (
  StudentID INT,
  CourseID INT,
  EnrollmentDate DATE,
  PRIMARY KEY (StudentID, CourseID)
);
```

☑ This ensures:

- Each `StudentID` can enroll in multiple courses.
- Each `CourseID` can be taken by multiple students.
- But a student **cannot enroll in the same course twice**.

2. Composite `UNIQUE` Constraint

- Used when a **combination of columns** must be **unique**, but not necessarily the primary key.

📝 Syntax:

```
CREATE TABLE EmployeeContacts (
  EmpID INT,
  ContactType VARCHAR(50),
  ContactValue VARCHAR(100),
  CONSTRAINT uc_emp_contact UNIQUE (EmpID, ContactType)
);
```

☑ This ensures:

- An employee (`EmpID`) can have multiple contacts (email, phone, etc.).
- But **each contact type is unique per employee**.

🧱 Adding Multi-Column Constraint to an Existing Table

Composite UNIQUE (example):

```
ALTER TABLE EmployeeContacts
ADD CONSTRAINT uc_emp_contact UNIQUE (EmpID, ContactType);
```

Composite PRIMARY KEY (example):

```
ALTER TABLE Enrollment
ADD CONSTRAINT pk_enrollment PRIMARY KEY (StudentID, CourseID);
```

⚠ Notes:

- Composite `PRIMARY KEY` **automatically sets** `NOT NULL` on the columns involved.
- Composite `UNIQUE` **does not set** `NOT NULL`, so you can have one `NULL` per column combination unless explicitly disallowed.
- You **cannot use the same column in multiple** `PRIMARY KEY` **constraints**.

TCL (TRANSACTION CONTROL LANGUAGE)

- Used to manage transactions (group of SQL operations).
- **BEGIN TRANSACTION** : Starts a new transaction block.
- **COMMIT** : Permanently saves all changes made during the transaction, send to server/finalize
- **ROLLBACK** : Undoes changes since the last COMMIT.
- **SAVE** : Sets a point in a transaction to which you can roll back to later.

📌 EXAMPLE 1

```
BEGIN TRANSACTION -- Select and execute only one time
DELETE FROM Emp
SELECT * FROM Emp
ROLLBACK TRANSACTION -- If there are no check-point then it will revert the
changes and END Transaction
```

📌 EXAMPLE 2

```
BEGIN TRANSACTION
DELETE FROM Emp WHERE EmpNo = 8888
SAVE TRANSACTION A1 -- Creates a checkpoint
UPDATE Emp SET Salary = 20000
SAVE TRANSACTION A2
ROLLBACK TRANSACTION A1 -- Removes changes after A1
```

```
COMMIT TRANSACTION -- After commit we can't rollback
ROLLBACK TRANSACTION -- This rollback request won't work as transaction is already
commited, so there are no open transactions
```

CTE (COMMON TABLE EXPRESSION)

- We can use one select statement in other select statement normally.
- So we will store the result of the query in a 'with' statement.
- a `WITH` statement is used to store the result of a query and work on columns created in it (in runtime).
- we have to select the query from `with` to `input` to `output` all at one, we can't execute it individually.
- It creates a temporary table -> gives output -> table is destroyed

📌 Example:

```
WITH AB -- ANY TEMP NAME [TEMP RECORD SET]
AS
(SELECT *, SAL*12 AS ANNUAL_SAL FROM EMP) -- INPUT
SELECT * FROM AB WHERE ANNUAL_SAL > 30000 -- OUTPUT
```

TEMPORARY TABLES

- Stores data temporarily.
- Once we close the session, the temporary tables are destroyed.
- Temp table must have # before their names.
- All temp tables created are accessible only by the same user.

Local Temp Tables

- Any temp table created with single # before table name are limited to that session only.

📌 Example:

```
SELECT * INTO #AB1 -- SYNTAX
FROM ( SELECT A.Ename, B.Dname
       FROM Emp A
       LEFT JOIN Dept B
       ON A.Deptno = B.Deptno
     ) ABCD -- ALIAS NAME MUST BE GIVEN

SELECT A.Ename, B.Dname FROM #AB1
```

Global Temp Tables

- Any temp table created with double # (i.e. ##) before table name are accessible across all sessions till the session is active.

📌 Example:

```
SELECT * INTO ##AB1 -- SYNTAX
FROM ( SELECT A.Ename, B.Dname
       FROM Emp A
       LEFT JOIN Dept B
       ON A.Deptno = B.Deptno
     ) ABCD -- ALIAS NAME MUST BE GIVEN

SELECT A.Ename, B.Dname FROM #AB1
```

VIEWS

- An alias name for the query
- View is not a table
- The query will be stored in it.
- Views don'r store the result of the query.
- Views are parmanent like tables.

📌 Example:

```
CREATE VIEW VW_001 as -- VM_001 is view name
SELECT * FROM Emp

SELECT * FROM VM_001

DROP VIEW VM_001  -- TO DROP VIEW

SP_HELPTEXT VM_001  -- SHOWS THE QUERY STORED IN IT

-- We will give view name to the user,
-- and user can write query using it
-- and the query stored in views will be executed, we do this security reasons
```

STORED PROCEDURES (SP)

- Used for creating a batch of SQL statements.
- SPs are used as backend of web/desktop applications.
- SP can be reused for different inputs.

Syntax:

```
CREATE PROCEDURE <PROCEDURE ANME>
AS
BEGIN
...
END

-- FOR EXECUTING SP
EXEC <SP NAME>

-- TO DROP PROCEDURE
DROP PROCEDURE <SP NAME>

-- TO SEE CODE
SP_HELPTEXT <SP NAME>
```

Example:

```
-- WRITE DOWN ADDITION/SUBTRACTION PROGRAM
CREATE PROCEDURE ADDITION
AS
BEGIN
  DECLARE @X INT = 100 -- `DECLARE` : TO DECLARE VARIABLES
  DECLARE @Y INT = 300
  PRINT @X + @Y        -- `PRINT` : TO PRINT
  PRINT @X - @Y
END

EXEC ADDITION
DROP PROCEDURE ADDITION

-- CHECK FOR EMPNO. IF RECORD EXITS THEN DELETE ELSE PRINT 'NO RECORDS'
CREATE PROCEDURE DELETE_RECORDS (@EMPNO INT) -- USER INPUT
AS
BEGIN
  IF EXITS ( SELECT * FROM EMP
             WHERE EMPNO = @EMPNO )
    BEGIN                              -- IF MORE THAN 1 LINE PRESENT
      DELETE FROM EMP WHERE WHERE EMPNO = @EMPNO
    END
  ELSE
    PRINT 'NO RECORDS'
END

EXEC DELETE_RECORDS 1000 -- USER INPUT
```

ERROR HANDLING

```
CREATE PROCEDURE PR_01 (@X INT, @Y VARCHAR(20), @Z INT)
AS
BEGIN
  BEGIN TRY                                 -- TRY START
    INSERT INTO EMP(EMPNO, ENAME, DEPTNO)
    VALUES (@X, @Y, @Z)
  END TRY                                   -- TRY END
  BEGIN CATCH                               -- CATCH START
    PRINT ERROR_MESSAGE()  -- DISPLAY ERROR MESSAGE
    PRINT ERROR_LINE()     -- DESPLAY ERROR LINE NO.
  END CATCH                                 -- CATCH END
END

EXEC PR_01 1111,'ABCD', 50
```

TRIGGERS

- These are special sps executed automatically
- It executes when user executes insert/ update/ delete statement
- Triggers are used to monitor the activity in the log (update/ insert/ delete, etc.)
- Special SPs are used to trigger email/messages if any changes happened to our DB table.
- Triggers are permanent until deleted.
- They are used for DML commands.

📝 Syntax:

```
CREATE TRIGGER <TRIGGER NAME> ON <TABLE NAME>
FOR <DML COMMAND>
AS
BEGIN
...
END

-- TO DROP TRIGGER
DROP TRIGGER <TRIGGER ANME>

-- TO SEE INFO ABOUT TRIGGER
SELECT * FROM SYS.TRIGGERS

-- TO SEE CODE
SP_HELPTEXT <TRIGGER NAME>
```

📌 Example:

```
-- CREATE A TRIGGER TO GIVE INSERTED RECORD IN A TABLE
CREATE TRIGGER INSERT_LOG ON EMP
FOR INSERT
AS
BEGIN
  SELECT * FROM INSERTED -- IT IS A PREDEFINED SYSTEM TABLE
END                      -- WHICH WILL GIVE US THE RECORD THAT
                         -- IS INSERTED IN THE TABLE
```

```
-- CREATE A TRIGGER TO GIVE DELETED RECORD IN A TABLE
CREATE TRIGGER TRIG_02 ON EMP
FOR DELETE
AS
BEGIN
  SELECT * FROM DELETED -- PREDEFINED TABLE FOR DELETE
END
```

- There are no predefined table for update.
- We can't add triggers in transactions, TCL will only support dml commands.

```
-- CREATE A LOG TABLE ENTRY
-- CREATE A TABLE EMP_LOG FIRST
CREATE TABLE EMP_LOG (
  EMPNO INT,
  ENAME VARCHAR(100),
  SAL DECIMAL(10,2),
  ACTION_TIME DATETIME,
  ACTION_TYPE VARCHAR(10)
)

-- Create a trigger for all DML operations
CREATE TRIGGER SHOW_LOG
ON EMP
FOR INSERT, UPDATE, DELETE
AS
BEGIN
  -- Log INSERTS
  INSERT INTO EMP_LOG (EMPNO, ENAME, SAL, ACTION_TIME, ACTION_TYPE)
  SELECT EMPNO, ENAME, SAL, GETDATE(), 'INSERT'
  FROM INSERTED

  -- Log UPDATES
  INSERT INTO EMP_LOG (EMPNO, ENAME, SAL, ACTION_TIME, ACTION_TYPE)
  SELECT EMPNO, ENAME, SAL, GETDATE(), 'UPDATE'
  FROM INSERTED
  WHERE EXISTS (
```

```
    SELECT 1 FROM DELETED d WHERE d.EMPNO = INSERTED.EMPNO
  )

  -- Log DELETES
  INSERT INTO EMP_LOG (EMPNO, ENAME, SAL, ACTION_TIME, ACTION_TYPE)
  SELECT EMPNO, ENAME, SAL, GETDATE(), 'DELETE'
  FROM DELETED
END
```

🔁 LOOPS in SQL

- Loops are used when you want to **repeat a block of SQL statements** until a **certain condition** is met.

🧠 Where are Loops Used?

- Loops are typically used in:
  - **Stored procedures**
  - **Functions**
  - **Triggers**
  - **Control-flow logic (T-SQL, PL/SQL, etc.)**

☑️ Types of Loops (T-SQL / SQL Server)

Loop Type	Purpose
`WHILE` loop	Repeats as long as a condition is true
`GOTO`	Jumps to a labeled section of code (less preferred)
`CURSOR` loop	Loops through rows in a result set

🧪 Example 1: **WHILE Loop (SQL Server / T-SQL)**

```
DECLARE @Counter INT = 1;

WHILE @Counter <= 5
BEGIN -- MANDATORY TO SET SCOPE OF LOOP
    PRINT 'Loop iteration: ' + CAST(@Counter AS VARCHAR);
    SET @Counter = @Counter + 1;
END;
```

🟢 Output:

```
Loop iteration: 1
Loop iteration: 2
Loop iteration: 3
```

```
Loop iteration: 4
Loop iteration: 5
```

Example 2: **Using CURSOR (Row-by-row Loop)**

```
-- print odd rows from sales order table in 1 to 500 records
-- and how much time it is taking
DECLARE @Name VARCHAR(100);

DECLARE name_cursor CURSOR FOR
SELECT Name FROM Employees;

OPEN name_cursor;
FETCH NEXT FROM name_cursor INTO @Name;

WHILE @@FETCH_STATUS = 0
BEGIN
    PRINT 'Employee Name: ' + @Name;
    FETCH NEXT FROM name_cursor INTO @Name;
END;

CLOSE name_cursor;
DEALLOCATE name_cursor;
```

```
-- Drop the procedure if it already exists
IF OBJECT_ID('dbo.PrintOddSalesOrders', 'P') IS NOT NULL
    DROP PROCEDURE dbo.PrintOddSalesOrders;
GO

-- Create the stored procedure
CREATE PROCEDURE dbo.PrintOddSalesOrders
AS
BEGIN
    -- Declare variables for OrderID and row tracking
    DECLARE @OrderID INT;
    DECLARE @RowNum INT = 1;

    -- Declare variables to track execution time
    DECLARE @StartTime DATETIME = GETDATE();
    DECLARE @EndTime DATETIME;

    -- Declare the cursor to fetch first 500 records from SalesOrder table
    DECLARE SalesCursor CURSOR FOR
    SELECT TOP 500 OrderID              -- Select first 500 OrderIDs
    FROM SalesOrder
    ORDER BY OrderID;                   -- Ensure consistent row order

    -- Open the cursor to begin processing
    OPEN SalesCursor;
```

```
    -- Fetch the first row into @OrderID
    FETCH NEXT FROM SalesCursor INTO @OrderID;

    -- Loop while there are still rows to process
    WHILE @@FETCH_STATUS = 0
    BEGIN
        -- Check if the current row number is odd
        IF @RowNum % 2 = 1
        BEGIN
            -- Print the row number and OrderID
            PRINT 'Odd Row ' + CAST(@RowNum AS VARCHAR) + ': OrderID = ' +
CAST(@OrderID AS VARCHAR);
        END

        -- Increment the row number counter
        SET @RowNum = @RowNum + 1;

        -- Fetch the next row into @OrderID
        FETCH NEXT FROM SalesCursor INTO @OrderID;
    END

    -- Close the cursor after use
    CLOSE SalesCursor;

    -- Deallocate the cursor to free resources
    DEALLOCATE SalesCursor;

    -- Capture the end time
    SET @EndTime = GETDATE();

    -- Print the start and end time
    PRINT 'Start Time: ' + CAST(@StartTime AS VARCHAR);
    PRINT 'End Time: ' + CAST(@EndTime AS VARCHAR);

    -- Print the duration in seconds
    PRINT 'Total Duration (seconds): ' + CAST(DATEDIFF(SECOND, @StartTime,
@EndTime) AS VARCHAR);
END;
GO
```

⚠ Note:

- **Avoid loops** if you can write your logic using **set-based queries** (`SELECT`, `UPDATE`, `JOIN`, etc.). Loops are slower.
- Use **cursors** only when row-wise processing is **absolutely necessary**.

OUTPUT PARAMETERS

- Any parameter fixed with output keyword.

- It sends the result back.

📌 Example:

```
-- Procedure to add two numbers and return the result in an OUTPUT parameter
CREATE PROCEDURE ADDITION (@X INT, @Y INT, @Z INT OUTPUT)
AS
BEGIN
    SET @Z = @X + @Y
END
GO

-- Procedure to call ADDITION and print the result
CREATE PROCEDURE RESULT (@A INT, @B INT)
AS
BEGIN
    DECLARE @C INT  -- Fixed typo: was 'DECALRE'
    EXEC ADDITION @A, @B, @C OUTPUT
    PRINT @C
END
GO

-- Execute the RESULT procedure with inputs 3 and 4
EXEC RESULT 3, 4
```

FUNCTIONS

- To develop our own functions (user defined functions)
- **Scalar Value Functions**
  - Functions that return 1 value are scalar value functions.
- **Table Value Functions**
  - Functions that return a table are Table value functions.
- **System Function example**
  - `SELECT SUM(SAL) FROM EMP` : `SUM` is function and `SAL` is input.

☑ SCALAR VALUE FUNCTION

- A **Scalar Value Function** is a **user-defined function (UDF)** that returns a **single value** (like `INT`, `VARCHAR`, `DATE`, etc.).

📋 Syntax:

```
CREATE FUNCTION FunctionName (@parameter DataType, ...)
RETURNS ReturnDataType
AS
BEGIN
```

```
    -- logic here
    RETURN <single value>
END;

-- SYNTAX FOR CALLING A FUNCTION
SELECT DBO.FunctionName

-- DROP FUNCTION
DROP FUNCTION FunctionName
```

- To alter a function use `ALTER FUNCTION`

📌 Example:

```
-- Get Bonus for Employee Based on Salary
-- Accepts salary as input AND returns a bonus as 10% of the salary.

-- Create scalar value function
CREATE FUNCTION dbo.GetBonus (@Salary DECIMAL(10, 2))
RETURNS DECIMAL(10, 2)
AS
BEGIN
    DECLARE @Bonus DECIMAL(10, 2);

    -- Bonus is 10% of the salary
    SET @Bonus = @Salary * 0.10;

    RETURN @Bonus;
END;
```

- ▶️ **Usage Example:**

```
-- Using the scalar function in a SELECT query
SELECT EmpName, Salary, dbo.GetBonus(Salary) AS Bonus
FROM Employees;
```

- ☑️ Benefits:

- Code reuse
- Clean and modular logic
- Can be used anywhere an expression is valid

Great! Let's now cover **Table-Valued Functions (TVFs)** — another important type of **User-Defined Function (UDF)** in SQL Server.

☑️ TABLE VALUE FUNCTION (TVF)

- A **Table-Valued Function** returns a **table** instead of a single scalar value. It can be:

- **Inline** (single `SELECT` statement — more efficient)
- **Multi-statement** (can include multiple logic blocks — more flexible)

☑ 1. INLINE TABLE-VALUED FUNCTION

📋 Syntax:

```
CREATE FUNCTION FunctionName (@param DataType, ...)
RETURNS TABLE
AS
RETURN (
    SELECT ... FROM ... WHERE ...
);
```

📌 Example:

```
-- Get Employees from a Specific Department

-- Create Inline Table-Valued Function
CREATE FUNCTION dbo.GetEmployeesByDept (@DeptNo INT)
RETURNS TABLE
AS
RETURN
(
    SELECT EmpID, EmpName, DeptNo, Salary
    FROM Employees
    WHERE DeptNo = @DeptNo
);
```

- ▶ **Usage:**

```
SELECT * FROM dbo.GetEmployeesByDept(10);
```

☑ 2. MULTI-STATEMENT TABLE-VALUED FUNCTION

📋 Syntax:

```
CREATE FUNCTION FunctionName (@param DataType, ...)
RETURNS @ResultTable TABLE (ColumnDef1, ColumnDef2, ...)
AS
BEGIN
    -- Insert logic into the return table
    INSERT INTO @ResultTable
    SELECT ... FROM ...
```

```
    RETURN;
END;
```

📌 **Example:**

```
-- Return Employees with Bonus Calculated

-- Create Multi-Statement TVF
CREATE FUNCTION dbo.GetEmployeesWithBonus ()
RETURNS @Result TABLE (
    EmpID INT,
    EmpName VARCHAR(100),
    Salary DECIMAL(10,2),
    Bonus DECIMAL(10,2)
)
AS
BEGIN
    INSERT INTO @Result
    SELECT EmpID, EmpName, Salary, Salary * 0.1 AS Bonus
    FROM Employees;

    RETURN;
END;
```

- ▶️ **Usage:**

```
SELECT * FROM dbo.GetEmployeesWithBonus();
```

- 📌 Summary Table

Function Type	Returns	Use Case
Scalar Value Function	Single value	Compute and return one result
Inline Table-Valued	Table	Simple `SELECT`-based table logic
Multi-Statement TVF	Table	Complex logic returning a table

🗂️ INDEXES in SQL

- An **index** is a **performance optimization** feature in SQL. It creates a **lookup structure** (like an index in a book) to allow **faster searching, filtering, and sorting** of data in a table.

☑️ Why Use Indexes?

Without Index	With Index
Table scan — checks every row	Uses B-tree/B+-tree to find rows fast
Slower on large tables	Much faster for `WHERE`, `JOIN`, `ORDER BY`

🔍 Types of Indexes

Type	Description
Clustered Index	Sorts and stores the actual data rows in table based on the indexed column
Non-Clustered Index	Creates a separate structure with a pointer to the actual data rows
Unique Index	Prevents duplicate values in the indexed column(s)
Composite Index	An index on multiple columns (e.g. `Index(col1, col2)`)
Full-text Index	Special index for text searching
Filtered Index	Applies index only to rows that meet a specific `WHERE` condition

📓 Syntax & Examples

☑ 1. Create Clustered Index

- (Only one allowed per table)

```
CREATE CLUSTERED INDEX idx_emp_id
ON Employees (EmpID);
```

📌 This sorts the **entire table** by `EmpID`.

☑ 2. Create Non-Clustered Index

```
CREATE NONCLUSTERED INDEX idx_emp_name
ON Employees (EmpName);
```

📌 This creates a **separate structure** to speed up searches by `EmpName`.

☑ 3. Create Composite Index

```
CREATE NONCLUSTERED INDEX idx_dept_sal
ON Employees (DeptNo, Salary);
```

📌 Helps with queries filtering or sorting on both `DeptNo` and `Salary`.

☑ 4. **Create Unique Index**

```
CREATE UNIQUE INDEX idx_unique_email
ON Employees (Email);
```

📌 Ensures no two employees can have the same `Email`.

☑ 5. **Drop Index**

```
DROP INDEX idx_emp_name ON Employees;
```

⚡ **Performance Tip**

- Always create indexes on:

- Columns in `WHERE`, `JOIN`, `ORDER BY`, `GROUP BY`
- Foreign keys
- High-read, low-write scenarios

⚠ **Drawbacks of Indexes**

Drawback	**Explanation**
Slower `INSERT/UPDATE/DELETE`	Because indexes need to be updated
Uses extra disk space	For each index structure
Too many indexes = slower	SQL optimizer may get confused

- In SQL **you don't need to manually "call" or "invoke" an index** — SQL Server (or any RDBMS) automatically decides **whether and how** to use available indexes using the **query optimizer**.

☑ How Indexes Are Used

- Automatically When you write a query like:

```
SELECT * FROM Employees WHERE EmpName = 'John';
```

➡ If you have an index on `EmpName`, **SQL Server automatically checks** whether that index improves performance and will use it **internally** to speed up the search.

❓ How to Know if Index Is Being Used?

- You can use **query execution plans** to **see** whether the index was used:

🔍 In SQL Server Management Studio (SSMS):

1. Write your query.
2. Click on **"Include Actual Execution Plan"** (`Ctrl + M`).
3. Execute the query.
4. Look for operators like:
    - **Index Seek** ☑ → Best (uses index efficiently)
    - **Index Scan** ⚠ → Index exists but scanning rows
    - **Table Scan** ❌ → No index used (full table search)

📌 Example:

```
-- Let's say we have this index:
CREATE NONCLUSTERED INDEX idx_empname ON Employees (EmpName);

-- Now this query will automatically use it:
SELECT * FROM Employees WHERE EmpName = 'Alice';
```

- You don't need to do anything extra — **the optimizer picks it**.

🔧 Forcing SQL to Use or Ignore Index (Advanced Use)

☑ Force an Index (not recommended unless you're sure):

```
SELECT * FROM Employees WITH (INDEX(idx_empname))
WHERE EmpName = 'Alice';
```

❌ Ignore Index:

```
SELECT * FROM Employees WITH (INDEX(0))
WHERE EmpName = 'Alice';
```

> ⚠ Use index hints **only when necessary**, because they override the optimizer and can hurt performance in other cases.

☑ Summary

Task	What You Do
Use Index	Write your query as usual
Check if used	Use Execution Plan in SSMS
Force index (if needed)	Use `WITH (INDEX(index_name))`

🔁 Conversion Functions in SQL Server

- Conversion functions are used to **change data from one type to another**, like from `VARCHAR` to `INT`, `DATETIME` to `VARCHAR`, etc.

🧩 Common Conversion Functions:

Function	Purpose	Supports Formatting?	Notes
`CAST()`	ANSI-standard type conversion	❌ No	Preferred for portability
`CONVERT()`	SQL Server-specific type conversion	☑ Yes (style codes)	Useful for formatting date/time
`PARSE()`	Converts string to date/time or number	☑ Yes (uses culture)	Slower, used when parsing user input
`TRY_CAST()` / `TRY_CONVERT()` / `TRY_PARSE()`	Like above but returns `NULL` on failure	☑ Yes	Safer for unpredictable input

☑ 1. `CAST()`

📋 Syntax:

```
CAST(expression AS target_data_type)
```

📌 Example:

```
SELECT CAST('123' AS INT) AS ConvertedInt;            -- returns 123
SELECT CAST(123.45 AS VARCHAR(10)) AS TextVal;        -- returns '123.45'
```

☑ 2. `CONVERT()`

📋 Syntax:

```
CONVERT(target_data_type, expression [, style_code])
```

- `[, style_code]` is optional

📌 Example:

```
-- Convert float to int
SELECT CONVERT(INT, 123.89) AS IntegerPart;           -- returns 124 (rounded)
```

```
-- Convert date to string
SELECT CONVERT(VARCHAR(20), GETDATE(), 103) AS UKDate;  -- returns '09/06/2025'
```

📝 Common `style_code` for date formatting:

Style Code	Format Type	Example Output
101	USA `mm/dd/yyyy`	`06/09/2025`
103	UK `dd/mm/yyyy`	`09/06/2025`
120	ISO `yyyy-mm-dd`	`2025-06-09 13:45:00`

☑ 3. `PARSE()`

> `PARSE()` is useful when converting **locale-aware strings** to numbers or dates.

- Sometimes cast and convert can't recognize the formate then use parse.
- But parse uses a lot of memory so don't use it unless cast/convert fails.

📝 Syntax:

```
PARSE(expression AS data_type [USING culture])
```

📌 Example:

```
SELECT PARSE('June 9, 2025' AS DATETIME) AS ParsedDate;

SELECT PARSE('1,234.56' AS FLOAT USING 'en-US') AS ParsedUS;
SELECT PARSE('1.234,56' AS FLOAT USING 'de-DE') AS ParsedDE;
```

☑ Bonus: TRY Versions

- These are **safe conversions** — instead of failing with an error, they return `NULL` if conversion fails.

📝 Examples:

```
SELECT TRY_CAST('abc' AS INT) AS SafeCast;      -- returns NULL
SELECT TRY_CONVERT(DATE, '2025-06-09') AS SafeDate;
SELECT TRY_PARSE('09/06/2025' AS DATE USING 'en-GB') AS UKParsed;
```

☑ Summary Table

Function	Use For	Style Formatting	Fails on Error?

Function	Use For	Style Formatting	Fails on Error?
`CAST()`	Simple, portable conversions	❌ No	☑ Yes
`CONVERT()`	Date/Time formatting	☑ Yes	☑ Yes
`PARSE()`	Locale-based string parsing	☑ Yes (culture)	☑ Yes
`TRY_CAST()`	Safe numeric/date conversions	❌ No	❌ No (returns NULL)
`TRY_CONVERT()`	Safe version of `CONVERT()`	☑ Yes	❌ No
`TRY_PARSE()`	Safe version of `PARSE()`	☑ Yes	❌ No

🔁 Recursive CTE (Common Table Expression)

- A **recursive CTE** is used to perform **hierarchical or recursive operations**, such as **traversing parent-child relationships**.

📋 Syntax:

```
WITH CTE_Name (columns...) AS (
    -- Anchor member (base query)
    SELECT ...
    FROM ...
    WHERE ...

    UNION ALL

    -- Recursive member (refers back to CTE)
    SELECT ...
    FROM CTE_Name
    JOIN ... ON ...
)
SELECT * FROM CTE_Name;
```

📌 Example:

```
-- Get hierarchy from Employee table

-- Table: Employees(EmpID, EmpName, ManagerID)

WITH EmpHierarchy AS (
    -- Base level: top managers (no manager)
    SELECT EmpID, EmpName, ManagerID, 1 AS Level
    FROM Employees
    WHERE ManagerID IS NULL

    UNION ALL
```

```
    -- Recursively add subordinates
    SELECT e.EmpID, e.EmpName, e.ManagerID, eh.Level + 1
    FROM Employees e
    INNER JOIN EmpHierarchy eh ON e.ManagerID = eh.EmpID
)
SELECT * FROM EmpHierarchy;
```

🔁 Eliminating Duplicates

☑ Using `DISTINCT`

```
SELECT DISTINCT column1, column2 FROM table;
```

☑ Using `ROW_NUMBER()` and CTE to delete duplicates:

```
WITH CTE AS (
    SELECT *,
           ROW_NUMBER() OVER (PARTITION BY column1, column2 ORDER BY ID) AS rn
    FROM table
)
DELETE FROM CTE WHERE rn > 1;
```

📐 Normalization in SQL

- Normalization is the process of **organizing data** in a database to:
- Eliminate redundancy
- Ensure data integrity

☑ 1NF (First Normal Form)

- Each cell must hold a **single atomic value**.
- No repeating groups or arrays.

❌ Bad:

EmpID	EmpName	Skills
1	Alice	Java, Python

☑ Good:

EmpID	EmpName	Skill
1	Alice	Java
1	Alice	Python

☑ 2NF (Second Normal Form)

- Must satisfy 1NF.
- **No partial dependency** (i.e., non-key attribute depends only on part of composite key).

❌ Bad:

- In a table with a composite key `(StudentID, CourseID)`:

StudentID	CourseID	StudentName

- Here, `StudentName` depends only on `StudentID` ⇒ **partial dependency**

☑ Good:

- Split into two tables:

- `Students(StudentID, StudentName)`
- `Enrollments(StudentID, CourseID)`

☑ 3NF (Third Normal Form)

☑ Rules:

- Must be in **2NF**
- **No transitive dependency**: A non-key column should not depend on another non-key column.

🔍 Example (violating 3NF)

StudentID	StudentName	DepartmentID	DepartmentName
101	Alice	D01	Computer Sci
102	Bob	D02	Electronics

Here:

- `DepartmentName` depends on `DepartmentID` (which is not a primary key of the table) ⇒ **transitive dependency**

☑ Fix (Apply 3NF)

Table 1: `Students(StudentID, StudentName, DepartmentID)` **Table 2:** `Departments(DepartmentID, DepartmentName)`

☑ BCNF (Boyce-Codd Normal Form)

> BCNF is a **stronger version** of 3NF.

☑ Rules:

- Must be in **3NF**
- For every functional dependency **A** → **B**, `A` must be a **super key**

🔍 Example (violating BCNF)

Course	Instructor	Room
DBMS	John	101
DBMS	John	102

- Assume:

- `Course → Instructor`
- But `(Course, Room)` is the primary key ⇒ Instructor is functionally dependent on **non-super key** ⇒ violates **BCNF**

☑ Fix:

Table 1: `CourseInstructor(Course, Instructor)` **Table 2:** `CourseRoom(Course, Room)`

☑ 4NF (Fourth Normal Form)

> 4NF deals with **multi-valued dependencies** — when one column depends on the key but is **independent of other columns**.

☑ Rules:

- Must be in **BCNF**
- No **multi-valued dependencies** in a single table

🔍 Example (violating 4NF)

StudentID	Hobby	Language
101	Painting	English
101	Music	English
101	Painting	French

Here:

- Hobbies and Languages are **independent multi-valued attributes** ⇒ violates 4NF

☑ Fix:

Table 1: `StudentHobbies(StudentID, Hobby)` **Table 2:** `StudentLanguages(StudentID, Language)`

📊 Summary Comparison Table

Normal Form	Rule	Eliminates
1NF	Atomic values (no arrays/multiple values per column)	Repeating groups
2NF	No partial dependency (non-key attribute depends on whole key)	Partial dependencies
3NF	No transitive dependency	Transitive dependencies
BCNF	For every A → B, A is a super key	All anomalies from 3NF not covered
4NF	No multi-valued dependency	Multivalued dependency anomalies

🔄 Denormalization

- The **reverse of normalization** — combining tables to:

- Improve **read/query performance**
- Avoid **complex joins**
- But can introduce **redundancy**

☑ Denormalized Example:

Instead of storing department name in a separate table:

```
Employees(EmpID, EmpName, DeptID, DeptName)  -- Redundant DeptName
```

Denormalization is common in **reporting, data warehousing**, or **read-heavy systems**.

🔁 Advantages of Normalisation

Advantage	Explanation
Eliminates Redundancy	No duplicate data in multiple places
Ensures Data Consistency	Same information appears the same throughout the DB
Easier Maintenance	One place to update; fewer chances for mistakes
Prevents Anomalies	Insert, update, delete anomalies are avoided
Improves Performance (Write)	Smaller tables = faster updates/inserts

Advantage	Explanation
Better Organization	Easier to read, maintain, and scale
Enforces Integrity with FK/PK	Business rules and relationships are properly maintained

---

🧪 ACID Properties in DBMS

> ACID stands for **Atomicity, Consistency, Isolation, and Durability** — the four key properties of a **database transaction**.

1. ⚛️ Atomicity

- A **transaction is all-or-nothing**.
- Either **all operations succeed**, or **none are applied**.

🔄 Example:

```
BEGIN TRANSACTION
  UPDATE Accounts SET Balance = Balance - 500 WHERE AccountID = 101;
  UPDATE Accounts SET Balance = Balance + 500 WHERE AccountID = 102;
COMMIT;
```

If any step fails (e.g., due to insufficient funds), the whole transaction **rolls back**.

---

2. 🎯 Consistency

- The database must go **from one valid state to another**.
- **All rules, constraints, and relationships** are preserved.

🌓 Example:

- A `CHECK constraint` ensures salary cannot be negative.
- If a transaction violates this, it fails and the DB remains consistent.

---

3. 🔒 Isolation

- Concurrent transactions should **not interfere** with each other.
- Each transaction should **appear to run independently**.

🔄 Example:

Two users transferring money at the same time won't see each other's intermediate states.

SQL provides **isolation levels** like:

- `READ UNCOMMITTED`
- `READ COMMITTED`
- `REPEATABLE READ`
- `SERIALIZABLE`

Higher isolation = fewer concurrency issues but may affect performance.

---

4. 💾 Durability

- Once a transaction is **committed**, it is **permanently saved**.
- Even if there is a **power failure** or **crash**, the data will not be lost.

🔐 Example:

After committing a transfer of ₹500, the change remains safe on disk even if the system crashes immediately.

📊 Summary Table

Property	Explanation	Real-Life Analogy
Atomicity	All operations in a transaction happen or none at all	Booking a movie ticket: seat is confirmed or not
Consistency	Database remains valid after any transaction	No double seat booking or negative balance allowed
Isolation	Transactions don't affect each other	Multiple people booking tickets simultaneously
Durability	Committed changes persist even after a crash or restart	Once booked, your ticket is stored securely

---