Database Design & MySQL

What is a data warehouse?

⇒ A company has 3 departments → Marketing, Sales, Finance

Each department maintains a seperate database this could lead to a situation wherein each department has its own version of the facts, for a ques as "What is the total revenue of the last quarter?" Every department might have a diff ans.

∴ each department draws info from a separate database

↳ Here the data warehouse can prove to be usefull.

A data wairehouse would thus be the central repo of the data of the entire enterprise.

Definition : A data warehouse is a centralized repo where large amount of structured data from multiple sources are stored and manage to support data analysis, reporting etc.

A data warehouse is

1. subject - oriented
2. Integrated - repository
3. non volatile
4. Time Variant

Collection of data in support of managements decisions.

So the data warehouse is a collection of Data. It has following properties:

⇒ Subject Oriented: A data warehouse should contain information about a few well-defined subjects rather than enterprise.

⇒ Integrated: A data wareh ose is the integrated repository of the data, it contains information from various systems within organizatin.

⇒ Non-Volatile: the data values in database cannot be changed without a valid reason

⇒ Time-variant: A data warehouse contains historical data for analysis.

Structure of Data Warehouse

One of the primary methods of designing a data warehouse is "dimensional modelling.
the two key elements of dimensional modelling are facts & dimensions.

In short "Dimensional modelling is a popular way to design data warehouses, aiming to make data easy to retrive and analyze."

facts ⇒ Numerical Data
Dimensions ⇒ Metadata (data explaining some other data)

Dimensional Modelling organizes the data into two main parts:

1) Facts : Facts are the numbers or measurements you want to analyze, they usually represent something measurable like sales amount, quantity sold or total revenue.

→ Facts are stored in fact table where each row represents an event or transaction

eg In Sales database, "Total Sales Amount", "Quantity Sold" are facts because they provide measurable data about each sale.

2) Dimensions : Dimensions are the descriptive details about the facts, they provide the context so we can understand the numbers better and answers questions like "who", "what", "where" & "when"

→ Dimensions are stored in Dimensions table, where each row describes an attribute (like product or time period)

eg In the same sales database

→ Product Dimension : Describe details about each product, like prod name, category, brand.

→ Customer Dimension : Describe customers, including customer name, location, age.

Putting it altogether

A fact table links to dimension table so we can analyze facts with context.

for eg you might have a fact table that records each sale with fields like

→ Sales Amount (fact)

→ Quantity Sold (fact)

→ Links to dimensions like ProductID, CustomerID, DateID

Using these links, we could answer questions like → : What was the total sales amount for Product X in 2023.

eg In BANK

Facts ⟸
- → withdrawl amount
- → Account balance after withdrawl
- → Transaction charge amount
- → Customer ID
- → ATM ID
- → Date of withdrawl

(*) A schema diagram in a data warehouse is a visual representation of structure and organization of data within a warehouse, it shows the relationships b/w different tables, columns, keys and other database objects.

There are commonly 3 types of schema models in Data warehousing:

① Star Schema : It is a simple and widely used schema, a central fact table links to multiple dimension tables, resembling a star shape.

Eg E-Commerce

Date of Purchase	Product	customer	Quantity
05-01-2017	P133a	John	2
06-01-2017	Coke	John	5
-	-	-	-
-	-	-	-

↳ Schema Diag

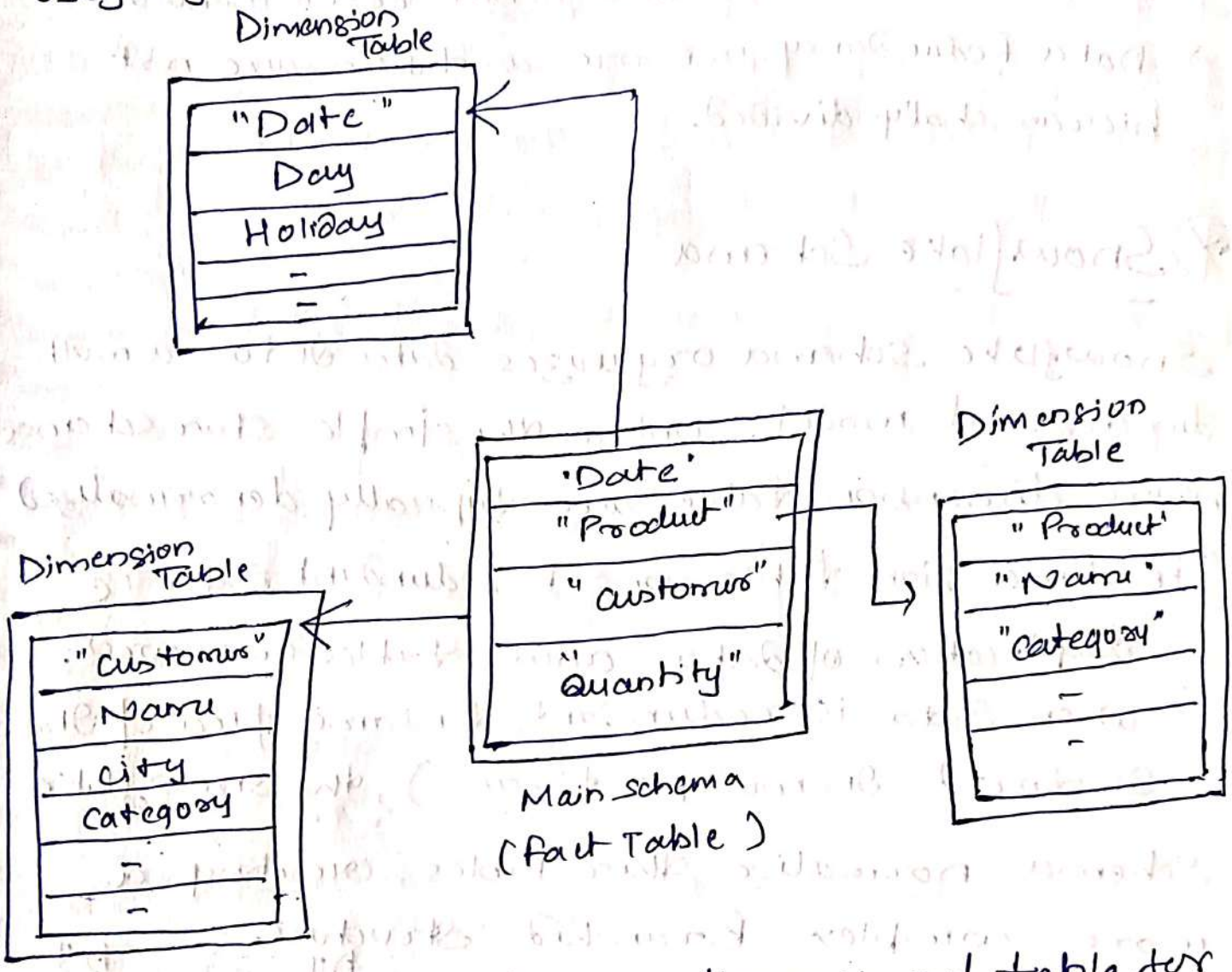
Now suppose we have few Questions?

① What do people from Banglore Purchase in general?

② In general what is the soft drink sales?

These info are not there

So star schema is usefull for augmenting our main data with the additional information usefull for analysis.



Now we can use every dimensional table for analysis.

Benefits of Star Schema

→ Queries use very simple joins while retrieving the data and those by every performance increases.

→ It is simple to retrive data for reporting at any point of time for any period.

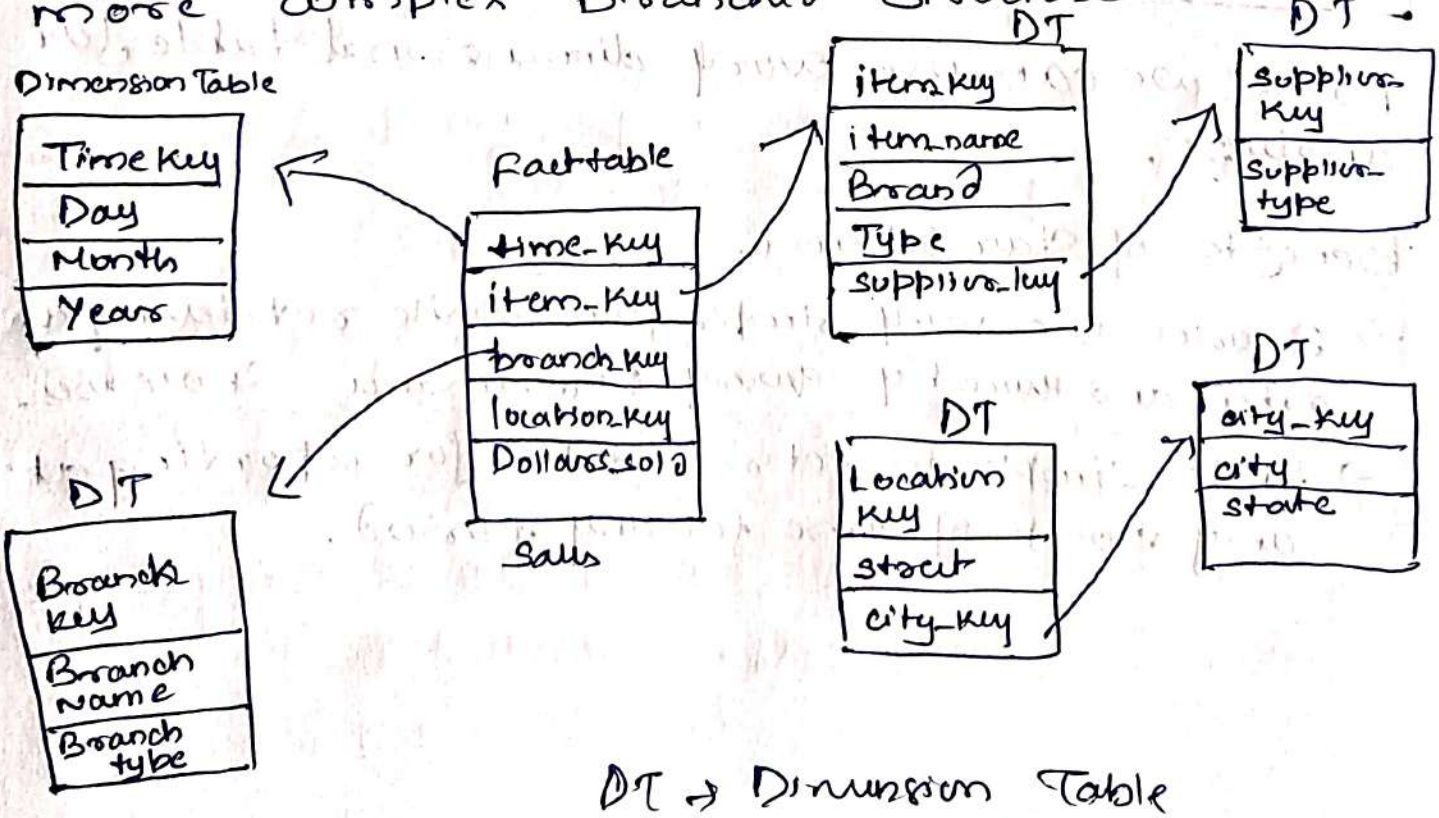
Disadvantages of Star Schema

⇒ If there are many changes in the requirement, the existing star schema is not recommended.

⇒ Data Redundancy is more as tables are not hierarchically divided.

✳ Snowflake Schema

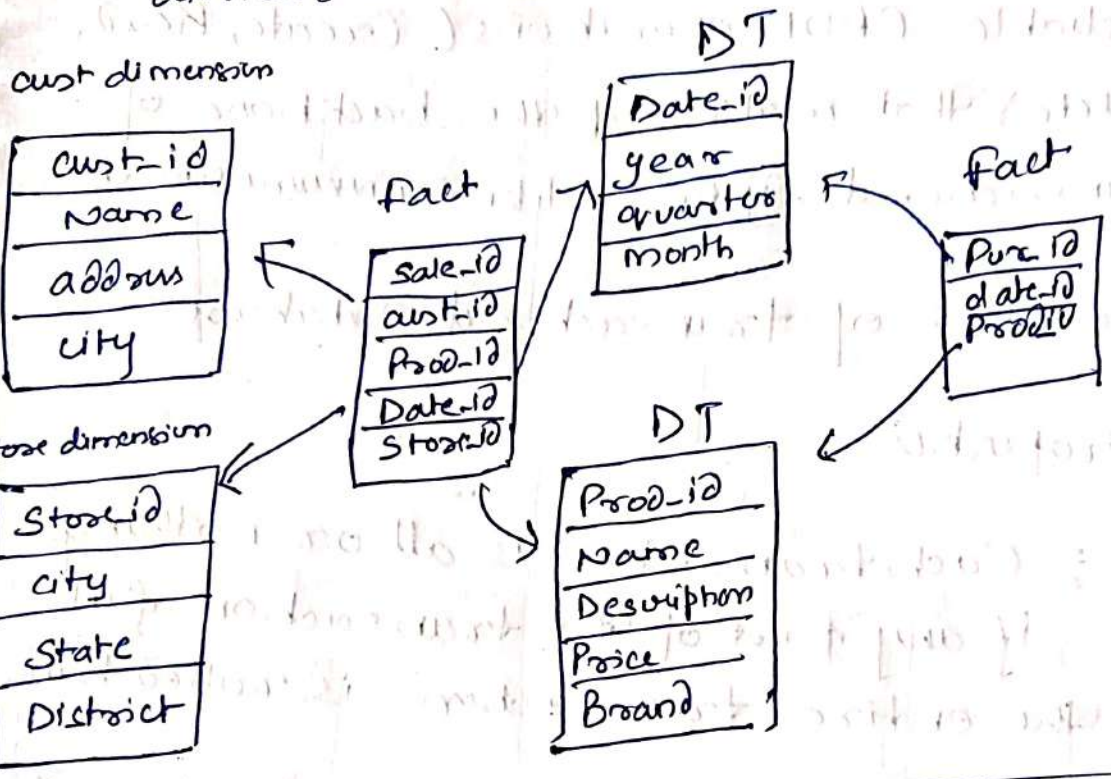
Snowflake schema organizes data into a multi-dimensional model, unlike the simple star schema where dimension table are typically denormalized (ie dimension table consist redundant data ie duplication of data across tables or records. When data is redundant, the same piece of info is stored in multiple places), the snowflake schema normalize these tables, creating a more complex Branched structure



DT → Dimension Table

Galaxy Schema : Also called as fact constellat

It is used when multiple fact tables share dimension tables.



OLAP vs OLTP

Now we will learn what features of Data warehouse differentiate it from a regular database.

The wrong Question Diff b/w Data warehouse and Database

Database is the collection of data, also the data warehouse is a collection of data.

The Right Question Diff b/w DataWarehouse & transaction Database

Transactional Database helps users carry out day to day business transactions

Transactional Database

It is a type of database designed to handle real time transactions efficiently. It's optimized for performing fast and reliable CRUD operations (Create, Read, Update, Delete) that make's up the backbone of online transactional systems like ecommerce sites etc

Key characteristics of transactional database

① ACID Properties

Atomicity : Each transaction is all or nothing if any part of the transaction fails the entire transaction is rolled back.

Consistency : transactions move the database from one valid state to other maintaining predefined rules such as constraints

Isolation : transactions are isolated from one other preventing concurrent transactions from interfering and leading to incorrect data.

Durability : once a transaction is committed, the data is permanently saved, even in the event of system crash.

Egs of transactional databases

→ Relational Database : MySQL, Postgres, Oracle

→ NO SQL Database : MongoDB

Diff b/w Transactional Database & DataWarehouse

Level	transactional DB	Data Warehouse
Purpose	1. to ~~Transform~~ perform business transactions.	1. to conduct analysis and decision making.
Users	2. End users (like us) for business transactions	2. Used by decision make like Managers, CEOs etc
Technology	3. NO ETL	3. ETL process, Extract transform and load process. Data Warehouse ↓ Integrated ↙ Multiple sources ↘ Multiple Database
Front end Tech	1) Webservers 2) Browsers for interaction 3) Mobile Phones 4) Interface with Payment Gateway 5) NO BI	1. BI tools Business Intelligence
Database Design	1) Based on Integrity issues and normalisation 2) No star schema 3) Nature of Access CRUD Creates, retrive, update, delete	1) Based on dimensional Modelling 2) Creation of star schema 3) Nature of Access : Retrive
Also known as	OLTP : Online transaction Processing System	OLAP : Online analytical processing System.

SETL Process Data Warehouse

↳ used to ingest data into schema and perform ops.

Select: Identify the data relevant for analysis. It involves determining which datasets, tables or fields are necessary to achieve specific goals of the analysis.

Extract: Connecting to the particular data source and pulling out the data.

Transform: Modifying the extracted data to standardise it.

Load: Pushing the data into Data Warehouse

Entity Constraints

Constraints are the rules that are used in MySQL to restrict the values that can be stored in the columns of a database.

This ensures data integrity, which is nothing but the accuracy and consistency of the data stored in the database.

(1) Not Null constraint

→ ensures that the column cannot contain a null value.

→ used when a value is required for every row in a column eg (Primary key fields, crucial fields like email in users table)

eg
```
Create table students (
    student_id int NOT NULL,
    name varchar(50) NOT NULL
);
```

2. UNIQUE constraint

→ ensures all values in a column are unique across the table.

→ commonly used for columns like email or username where duplicate values aren't allowed

eg
```
Create table employees (
    emp_id int unique,
    email varchar(100) unique
);
```

Note: the UNIQUE constraint does allow null values, it won't violate the uniqueness rule because null values are treated differently from regular data.

3. Primary Key Constraint

⇒ Combination of not null and unique, it uniquely identifies each row in a table.

⇒ A table can have only one primary key which can be a single column or a combination of columns (composite primary key)

eg
```
Create table orders (
    order_id int Primary key,
    order_date DATE NOT NULL
);
```

4. foreign key constraint

→ enforces a link b/w two tables by referencing a primary key in another table.

→ Maintains referential integrity by ensuring that values in a column or set of cols match values in the referenced table.

eg
```
Create table orders (
    order_id int Primary key,
    customer_id int,
    foreign key (customer_id) References Customers
                                (customer_id)
);
```

5. CHECK Constraint

→ ensures that all the values in col meet a specified condition.

→ useful for setting range limits or enforcing rules on data formats

eg
```
Create table product (
    Product_id int primary key,
    Quantity int check (Quantity >= 0)
);
```

6) Default Constraint

→ Sets a default value for a col, if no value is provided

Eg Create table users (
user_id int Primary key,
registeration_date DATE DEFAULT current_date
);

Referential Constraints

entity type constraints pertain to the values in a single table.

the second type of constraint is called referential constraint, they are used to restrict the values that are taken by a column in one table based on the values that exist in another table.

We have 2 tables

customers

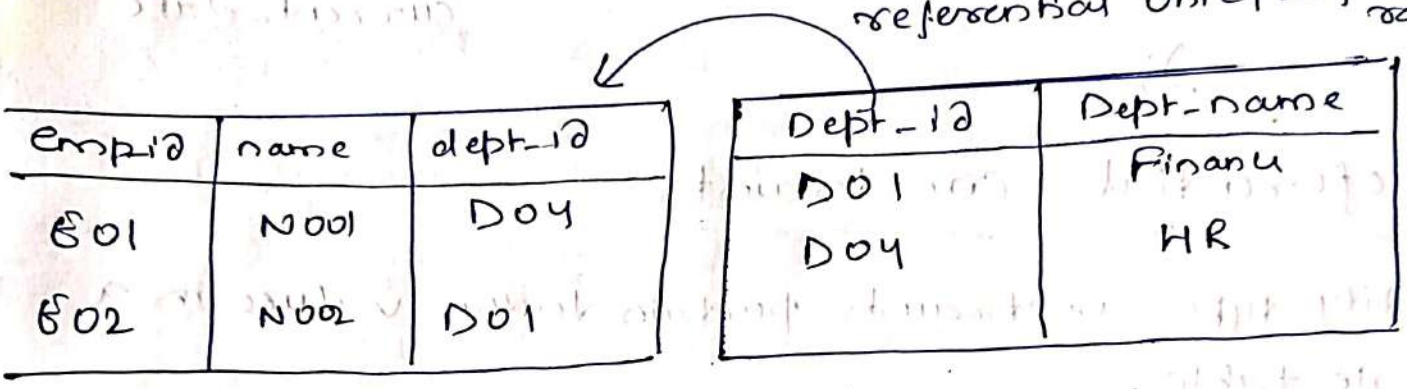
cust_id	firstname	Lastname	Age
1	John	Wick	52
2	Sheldon	cooper	41
3	Charlie	Harper	45

orders

order_id	order_num	cust_id
1	1234	2
2	3142	2
3	1241	3
4		

Cust_id is a foreign key in the "orders" table, because it is used to reference the customers table.

We can easily determine which customer placed that particular order and find out all the details of that customer by looking up the corresponding cust_id in customers table.



Empid	name	dept_id
E01	N001	D04
E02	N002	D01

Dept_id	Dept_name
D01	Finance
D04	HR

According to the referential constraint b/w two tables the value that appears as a foreign key in a table is valid only if it also appears as a primary key in the table to which it appears.

Note: foreign key need not be unique, ∴ many employees can belong to the same department

A table can have any number of foreign key.

Semantic Constraints

Adds rules based on the specific meaning or business rules of the data, for eg

a constraint ensuring an "age" col has values greater than zero, or salary field is within a range.

In Summary

① Entity Constraint

→ Primary key constraint : Ensures each row in a table is unique and not null, making it a unique identifier for rows.

→ Unique Constraint : Guarantees that all values in a column are unique.

→ Not NULL Constraint : Prevents null values in a specific col, ensuring that a value is always provided.

② Referential Constraint

→ foreign key constraint : Enforces a link b/w tables by requiring that the values in a foreign key col correspond to the values in the primary key col of another table.

→ On Delete Cascade : If the referenced row is deleted any rows referencing it are automatically deleted, maintaining referential integrity.

eg 2 tables : Department & Employee

Department table
- Dept_ID (primary key)
- Dept_name

Employee table
- emp_ID (Primary key)
- emp_name
- Dept_ID (Foreign key)

Now if we set On Delete cascade on Employee.Department ID it Means when a row in Dept table is deleted, all rows in employee table

that reference Dept-id will also be deleted.

Eg Create table order-details (
order_id int primary key,
customer_id int References customer_details
(cust-id) on DELETE cascade,
Order_ date Date
);

→ On update cascade

⇒ If primary key is updated, all related foreign keys are automatically updated to match the value.

3. Semantic Constraints

⇒ Check constraint : Enforces a specific condition on the data within a col

Eg age > zero

Eg Create table employee (
eid int(11) Not NULL,
name varchar(40) Default Null
Date_of_joining DATE NOT NULL CHECK
(Date_of_joining > '2019-02-01')
Primary key (eid)
);

ERDs

An entity - relationship diagram or ERD can be thought of as a map of the database schema.

We can visualise the structure of the entire schema and answer the following questions just by looking at it.

→ What are the tables that it contains?
→ What are the columns that each table contains?
→ what is/are the datatypes and constraints for each column?
→ What are the relationships b/w various tables?

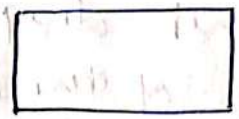
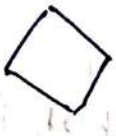
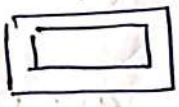
Entity: Small piece of information that we want to track in a database.

Entity type: Collective name given to the bunch of entities

Why use ER Diagrams in DBMS

→ ER Diag represents the ER model in the database making them easy to convert into relations (tables)
→ ER diag requires no technical knowledge and no hardware support
→ It gives standard way of visualizing the data logically

Symbols used In ER models

Rectangle		Entities In ER Model
Ellipse		Attributes In ER model
Diamond		Relationship among entities
Line		Attributes to entities and entity sets with other relation type
Double eclipse		Multivalued Attributes
Double Rectangle		Weak entity

Components of ER Diagram

ER model consist of Entities, Attributes, Relationships among enteties In a Database system.

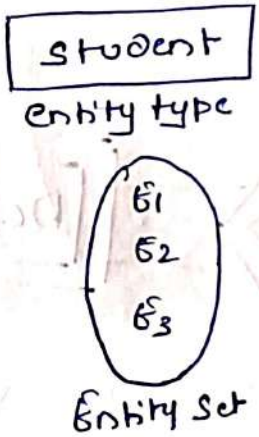
- ER Model
  - Entity
    - strong entity
    - weak entity
  - Attribute
    - key attribute
    - composite attribute
    - Multivalued attribute
    - Derived attribute
  - Relationship
    - one to one
    - one to many
    - many to one
    - Many to many

What is Entity?

An entity may be an object with a physical existence - a particular person, car, house or employee or it may be an object with a conceptual existence a company, a job or a university course.

What is entity set?

An entity is an object of entity type and the set of all the entities is called as entity set.
In ER diagram, entity type is represented as

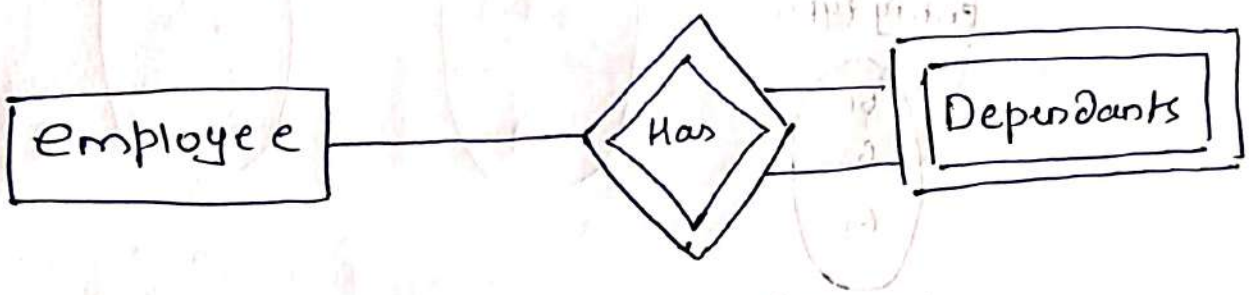


We have 2 types of entities.

① A strong entity is a type of entity that has a key attribute. Strong Entity does not depend on other entity in the schema. It has a primary key, that helps in identifying it uniquely, and it is represented by a rectangle, they are called Strong Entity types.

A weak entity type is an entity that has a key attribute that uniquely identifies each key in the entity set, but some entity type exists for which key attributes can be defined.

eg
A company stores the information of dependents (Parent, children, spouse) of an employee, but the dependents can't exist without an employee so the dependent will be a weak entity type and Employee will be identifying entity type for Dependent which means employee is a strong entity.



Attributes

these are the properties that defines the entity type
eg Roll_no, name, DOB are the attribute that define entity student

Represented as: (Attribute)

Types of Attributes

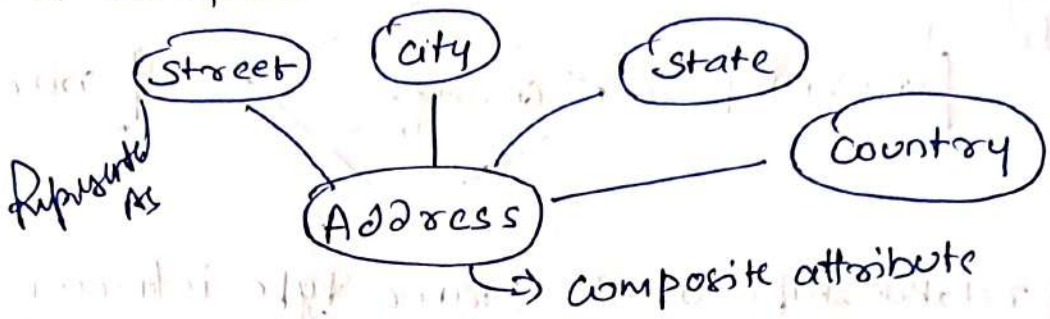
1) Key attribute : the attribute which uniquely identifies each entity in the entity set is called key attribute. for eg roll_no will be unique for each student

Represented as 

2. Composite attribute.

An attribute composed of many other attributes is called a composite attribute.

eg

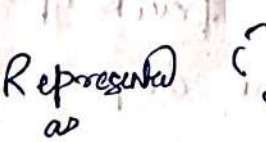


3. Multivalued Attribute

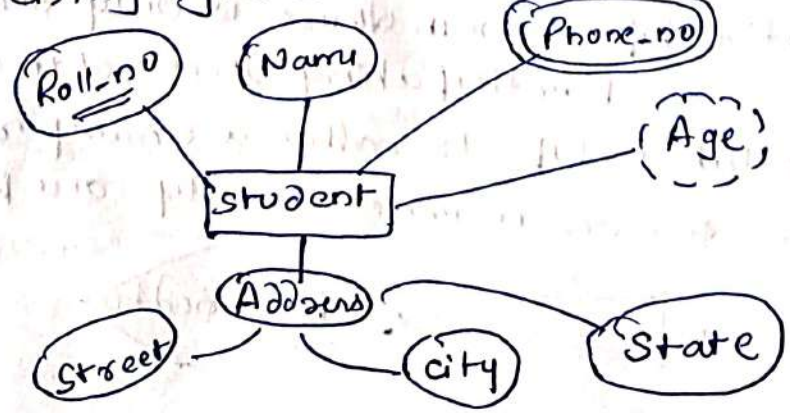
An attribute consisting of more than one value for an entity. eg Phone_no (people contains morethan1)

4. Derived Attribute

An attribute that can be derived from other attributes of the entity type is known as derived attribute



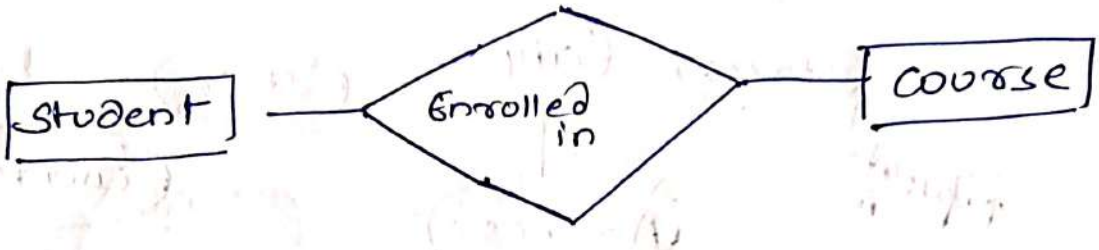
Complete entity type student and its attributes



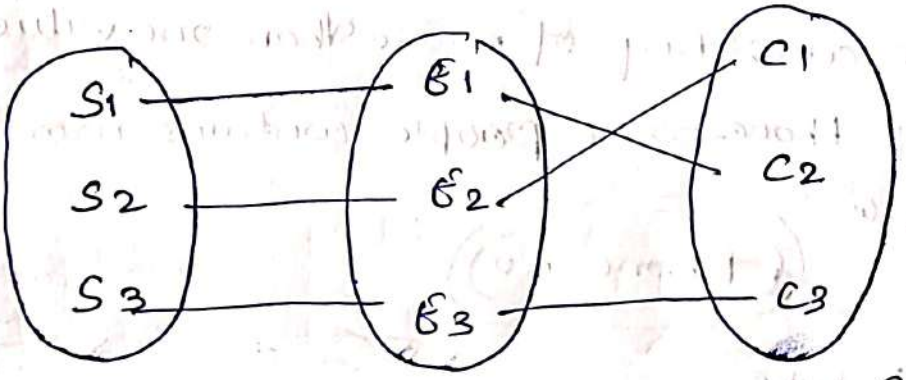
Relationship Type & Relationship Set

A relationship type represents the association b/w entity types. for eg "enrolled in" is a relationship type that exist b/w entity type student and course.

Represented As



A set of relationships of the same type is known as a relationship set.



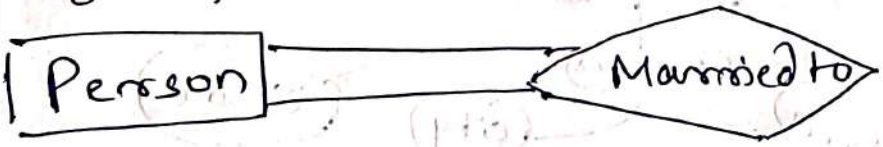
S1 enrolled in C2, S2 enrolled in C1, S3 enrolled in C3

Degree of a relationship set

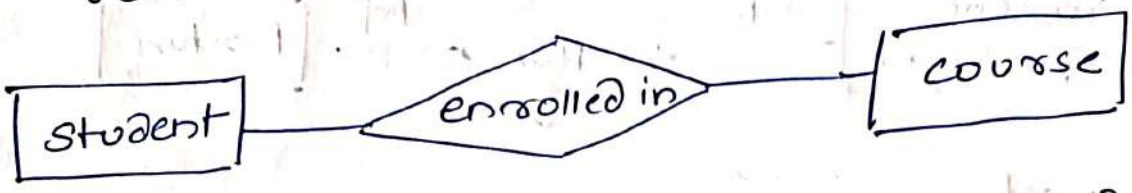
↳ The nos of different entity sets participating in a relationship set is called the degree of a relationship

① Unary Relationship : When there is only one entity set participating in a relation, the relationship is called a unary relationship.

eg One person married to only one person



2. Binary Relationship : When there are two entity sets participating in a relationship, the relationship is called as Binary Relationship.



3. Ternary Relationship : When there are 3 entity sets participating in a relationship.

4. N-ary relationship : When there are n entity set Participating in a relationship.

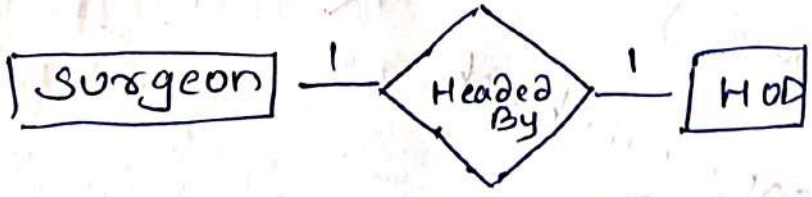
What is Cardinality

The number of times an entity of an entity set participate in a relationship is known as cardinality.

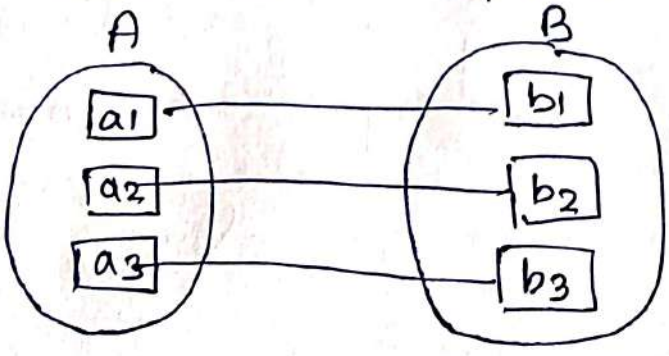
Different types of cardinality are

① One-to-One : When each entity in each entity set can take part only once in the relationship (one to one)

eg Male can Marry one female and vice versa

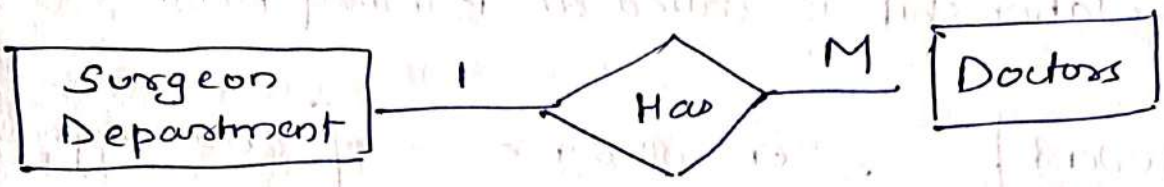


Using sets can be represented as

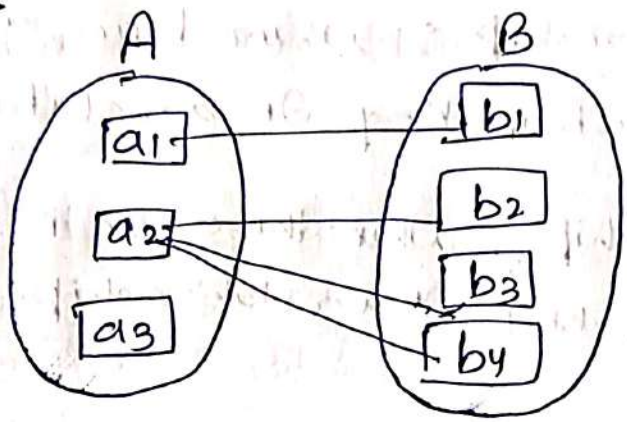


2. One to Many : each entity can be related to more than one entity.

eg

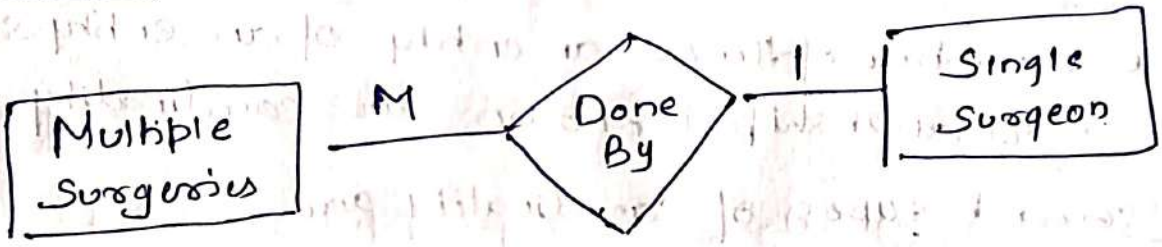


Using sets

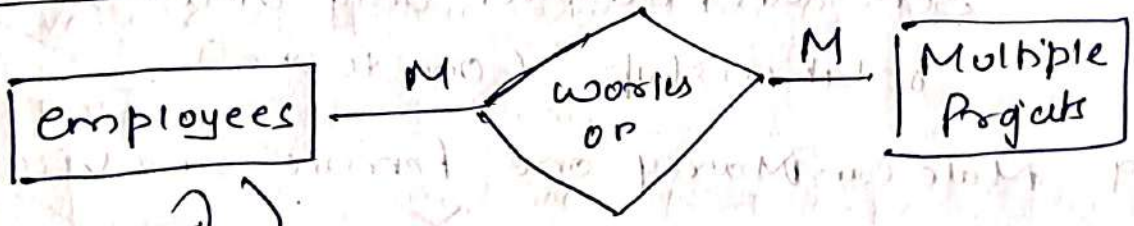


3. Many to one :

eg



4. Many to Many :

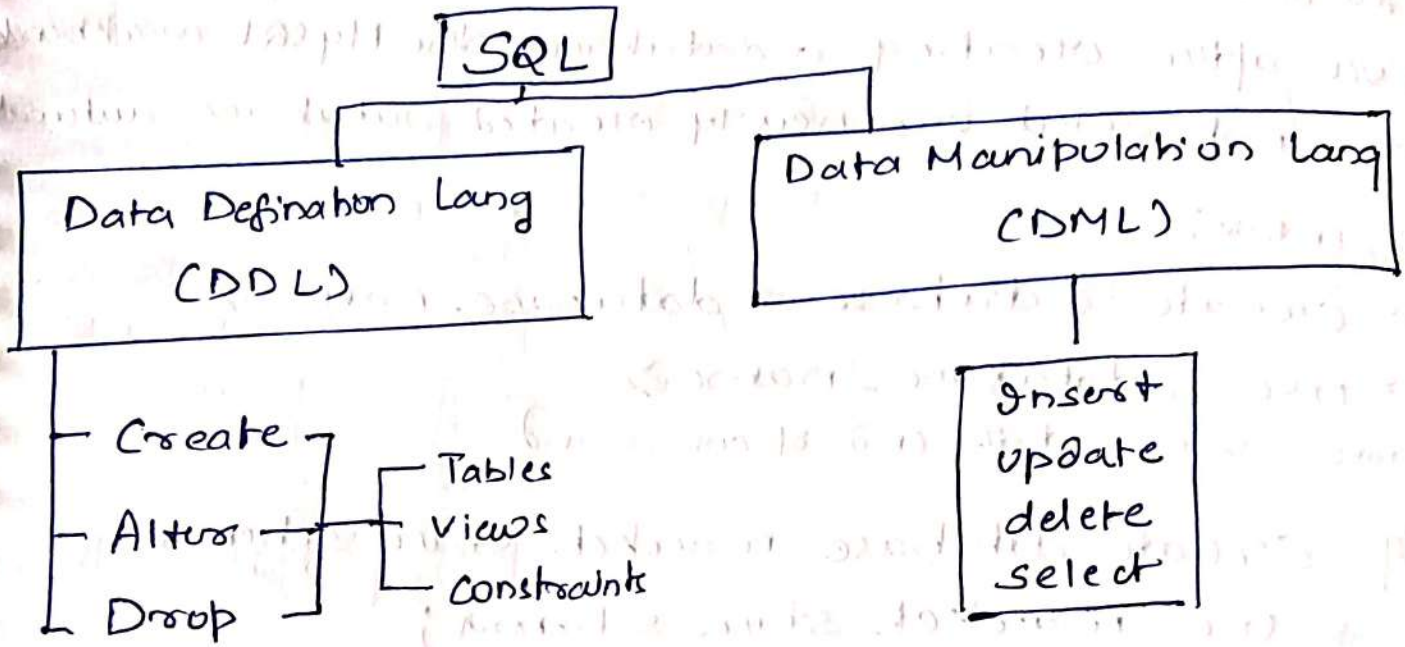


(the root Password)

↳ MySQL Workbench

Introduction to DDL and DML Statements

SQL ⇒ Structured Query Language



So the commands available in SQL can be broadly categorised as follows:

⇒ Data Definaton Language (DDL)

⇒ Data Manipulation Language (DML)

DDL as the name suggests is used to create a new schema as well as modifying an existing schema. The typical commands available in DDL include Create, Alter, Drop

As a data scientist mostly we will be working with DML commands to generate insights

DDL Statements : Demonstration

Now we will learn how to create and use a database (also called schema), we need to specify the database that we need to use, this is because even after creating a database, the MySQL workbench does not select our newly created database automatically.

Syntax:

→ create database <database_name>

→ use <database_name>

Note: use ; at the end of command

Eg ⇒ Create database market_star_schema;

⇒ use market_star_schema;

Creating tables and developing star schema

```
Create table shipping_mode_dimen (
        Ship_mode varchar(25),
        Vehicle_company varchar(25),
        TollRequired boolean
);
```

Varchar : Datatype in MySQL which corresponds to String attribute.

Variable character length → Here 25 refers to the maximum length it can take.

If we want to store fixed length ⇒ char(9)

↳ Max 9 characters

Alter is a command useful for modifying any schema object.

We will add a primary key constraint to an existing table.

> alter table shipping-mode-dimen add constraint Primary Key (ship-mode);

eg create table india (
matches_played int,
matches_won int,
matches_lost int,
net_run_rate Decimal (4,3)
);

Note : In DECIMAL (4,3) ⇒ 4 is the total nos of digits and 3 is the total nos of digits after decimal point

DML Statements : Demonstration

Insert Command

insert into shipping-mode-dimen
values
('Delivery Truck', 'Ashok leyland', false),
('Regular Air', 'Air India', false);

↳ Here we are not specifying the order of the attribute value.

Other format to Insert the value, better way would be

```
Insert into shipping_mode_dimen (Ship_mode,
        Vehicle_company, Toll_reqvired)
Values
    ('Delivery Truck', 'Ashok leyland', false),
    ('Regular Air', 'Air India', false);
```

↳ Here we can mention the order of the attributes

Now we will learn about "update-set-where" & "delete-from-where" commands

update command is used to modify data in the database.

the above inserts we set toll_required as false for Delivery Truck we need to update it as true

```
update shipping_mode_dimen set Toll_required = true
    where Ship_mode = 'Delivery Truck';
```

Delete Air India entry

```
Delete from shipping_mode_dimen where
vehicle_company = 'Air India';
```

↳ delete from table name where condition;
↳ update tablename set col = val where condition;

(syntax)

Note: A value that is a primary key in a table cannot be deleted, if another table has the same value as a foreign key.
So it's necessary to sequence the delete commands.

Modifying Columns

#Alter command to add a new column

alter table shipping_mode_dimen add Vehicle_num Varchar(20);

ee alter table <table_name> and <col_name> <datatype> ''

Now this new col will have null values for all the existing rows.

> update shipping_mode_dimen set vehicle_numbe = "MH-05-R1234";

Dropping a column

Alter table <tablename> drop column <col_name>

Alter table shipping_mode_dimen drop column Vehicle_Number;

↳ # this col is gone forever.

Change the col name "Toll_Required" to "Toll_Amount" also change datatype to int

> alter table shipping_mode_dimen
change Toll_required Toll_Amount int;

↳ keyword

for dropping the table

" drop table <table name> "

> drop table shipping_mode_dimen;

Note : ① Create, alter, drop, rename, truncate are DDL commands

② Select, Insert, update, delete are DML.

Note ! Drop removes the table completely from the database, we can't retrive it back, all the integrity constraints are removed.

Truncate only drops the table rows, it doesn't delete the schema from the database

Querying in MySQL

→ In Database Market_star_schema

SQL statements and operators

to print the entire data of all the customers

select * from cust_dimen;

↳ table name

list names of all the customers

> Select customer_name from cust_dimen;

Print name of all customers along with their city and state

> Select customer_name, city, state from cust_dimen;

#Note : In general the data is shown in the order in which it was inserted into the table.

We have a set of functions that we can use in SQL like for eg Aggregate functions, these functions cumilate the data in multiple ways.

eg # Print the total nos of customers

> Select count(*) as total_customers from cust_dimen;

↳ total_customers
1833

↳ Note : Here we used "count(*) as total_customers"
this is called as Aliasing

eg for Aliasing

Select customer_name AS "Customer Full Name", city as "customer city" from cust_dimen;

↳ Customer Full Name | Customer City

Note : the alias is limited to that specific query only.

we can also apply filter conditions

Eg # Show details of the customers belonging to West Bengal.

> Select * from cust_dimen where State = "West Bengal";

Similarly we can write multiple conditions

> Select * from cust_dimen where State = "West Bengal" and city = "kolkata";

Note: filtering is always evaluated first.

Eg 2 How many customers are from west bengal

> Select count(*) as Bengal_customers from cust_dimen where state = "West Bengal";

Eg 3: Print the names of all customers who are either corporate or belong to Mumbai

> Select Customer_name, city, Customer_segment from cust_dimen where city = 'Mumbai' or Customer_segment = 'Corporate';

List the details of all the customers from southern India, namely TamilNadu, Karnataka, Telangana, Kerala

Select * from cust_dimen where state in ('Tamil Nadu', 'Karnataka', 'Telangana', 'Kerala');

Print the details of all non-small business customers

```
Select * from cust_dimen where customer_segment
        != "Small Business";
```

Negation operator (Not equal to)

Dealing with numeric operators

eg# list the order ids of all those orders which caused losses.

```
Select Ord_id, Profit from market_fact_full where
  Profit < 0;
```

eg# List the orders with '_5' in their order ids and shipping costs BETWEEN 10 and 15. (LIKE)

```
Select ord_id, shipping_cost from market_fact_full
where ord_id like '%\_5%' and shipping_cost
between 10 and 15;
```

eg# Write a query to display cities in the cust_dimen table that begins with letter K

```
Select City from cust_dimen where City like 'K%';
```

Note K% ⇒ cities starts with K

%K ⇒ returns cities ending with K

Between operator eg

```
Select * from market_fact_full where shipping_cost
                    between 10 and 15;
```

The following wildcards are supported in SQL

→ Both % and _ (underscore)

% is a multi character wildcard, whereas
_ is a single character wildcard.

eg "_r%" ⇒ returns words with second character as r followed by zero or more characters at the end.

Note: In Between both are inclusive

Aggregate functions

SQL Provides several aggregate func, each designed for specific data analysis tasks, Here are the most commonly used aggregate func in SQL.

1. SUM() ⇒ calculates the total sum of a numeric column.
2. Avg() ⇒ calculates the avg (mean) of a numeric col.
3. Count() ⇒ count the nos of rows or non null values in a column
4. Min() ⇒ find the minimum value in a column
5. Max() ⇒ find the maximum value in a column
6. Group_concat() or string_agg() depending on SQL version → Concatenates values in a col into a single string.

Example of Aggregate func with Group By

eg table : employees with following columns

- → employee_id
- → department_id
- → salary
- → Job_title

1. SUM() - Calculate total

to find the total salary by department

```
> select department_id, SUM(salary) as total_salary
  from employees group by department_id;
```

2. AVG() - Calculate Average

to find Avg salary by Job_title

```
> select job_title, AVG(salary) as average_salary
  from employees group by job_title;
```

3. Count() → Count Rows or Values

eg count the total nos of employees in each department

```
> select department_id, count(*) as employee_
  count from employees group by
  department_id
```

Eg 2 of count()

to count the non-null values in a specific col

```
Select department_id, count(salary) as non_null_
    _salaries from employees group by
    department_id;
```

this counts only the rows where salary is not null for each department.

4. MIN() → to find minimum value

to find the lowest salary in each dept

```
Select department_id, Min(salary) as minimum_sal
from employees group by department_id;
```

5) MAX() → to find the maximum value

to find the max salary in each department

```
Select department_id, max(salary) as max_sal
    from employees group by department_id;
```

✳ Using GROUP BY with Multiple Columns

Eg. to calculate the avg salary by both department and job title

```
Select department_id, job_title, Avg(salary) as
    avg_salary From employees group by
        department_id, job_title;
```

Group_Concat()

Suppose we have a table employees with 2 cols department-id, employee-name
we want to list all the employee names in each department separated by commas

1) Group-Concat()

the group-concat() in MySQL are used to combine values from multiple rows within each group into a single concatenated string

```
> Select department_id, Group_concat (employee_name
    as employee_names from employees
  Group by department-id
```

op	department_id	employee-names
	1	Alice, Bob, Charlie
	2	Dave, emma, frank

GroupBy and Having

the groupby and Having clauses are commonly used together for grouping data and filtering grouped results.

① Group By : This clause groups rows that have the same values in a specified col, Aggregate func (like SUM, AVG, count etc)

② Having : this clause is used to filter groups based on aggregate values.

While the "where" clause filter rows before grouping, "Having" filters after the grouping and aggregations are completed.

Eg table : Sales

Assume we have a table Sales with following cols

→ Prod_id , → category , → sales_amt, → sales_date

Suppose you want to find the total sales of each Product

↳ we will use Groupby with sum func

> Select product_id , SUM(sales_amount) as total_sales from sales Groupby prod_id

If you only want to see products with total sales above 10000

Select product_id , SUM(sales_amount) As total_sales from sales group by Product_id Having SUM(sales_amount) > 10000 ;

Ordering

In SQL the order by clause is used to sort the resultset based on one or more columns, we can sort the data in ascending / descending order.

Syntax:

```
Select col1, col2, ....
    from table_name
    order by col1 [ASC|DESC], col2 [ASC|DESC],..;
```

By default order by sorts data in ascending order

eg table ⇒ employees with cols as
- → employee_id
- → name
- → salary
- → department

eg # list of employees sorted by salary in descending

```
> Select name, salary from employees order by
  salary Desc;
```

for ascending

```
> Select name, salary from employees order by
  salary;
```

Order by Multiple Columns

We can sort by multiple columns, for eg we want to sort by department in ascending order and then by salary is descending order within each department

> Select name, department, salary from employees order by department Asc, salary desc;

Orderby Aggregate functions

When using aggregate func and groupby we can use order by to sort the result based on the aggregated values

> Select department, SUM(salary) as total_salary from employees group by department order by total_salary Desc;

OrderBy with Aliases

if we use an alias for a column, we can use the alias in the orderby clause

> Select name, salary AS emp_sal from employee order by emp_sal desc;

Ordering Null values

SQL standards vary by database on how Null values are ordered.

→ Some databases put Null values at the beginning of ascending order and the end of descending order.

→ Others allow explicit control over the placement of Null values eg NULLS first or NULLS last

eg In PostGres

```
Select name, salary from employees order by
    salary desc NULLS Last;
```

---

eg we have sales table

→ sale_id → Product_id → amount → sale_date

We want to retrieve the total sales amount for each product that has a total sales amt > than $10000, sort the result by total_sales in descending order and limit the O/P to top 5 prod

```
> Select Product_id, sum(amount) as total_sales
  from sales group by Product_id
    having sum(amount) > 10000 order by
    total_sales desc limit 5;
```

String and Date - Time functions

We have many built in string functions that help with manipulating text data

1. Concat ( ) ⇒ concatenate strings
   ⇒ Combines two or more strings into one.
   ⇒ Syntax: CONCAT (string1, string2, ...)
   eg Select CONCAT (firstname, ' ', last_name) as full_name from employees;

2. Upper ( ) ⇒ Convert to Uppercase
   ⇒ Converts a string to uppercase letters
   ⇒ Syntax : UPPER (String)
   eg Select Upper (name) from products;

3. LOWER ( ) ⇒ Convert to Lowercase
   ⇒ Convert a string to lowercase letters
   ⇒ Syntax: LOWER (string)
   eg Select LOWER (name) from products;

4. SUBSTRING ( ) ⇒ Extract Part of a string
   → extract a position of a string from a specified position for a specified length.
   → Syntax : Substring (String, start, length)
   eg Select Substring (name, 1, 3) From employees;
   ↳ returns 1st 3 letters of the name

5. Length ( ) ⇒ returns the length of a string
→ Returns the number of characters in a string
Syntax : LENGTH (string)
eg Select Length (name) from employees;

6. TRIM ( ) → Removes leading and trailing spaces
→ Removes whitespace from the beginning and end of the string.
→ Syntax: TRIM (string)
eg Select TRIM (name) from employees;

7. Replace ( ) ⇒ Replace the Part of String
→ Replace the occurance of a substring within a string with another substring.
→ Syntax: Replace (string, old_substring, new_substr
eg Select REPLACE (name, ' ', '_') From products;
(replaces space with hyphens)

8. POSITION ( ) → find position of substring
→ finds the position of the first occurance of a substring within a string
Syntax: Position (substring IN string)
eg Select Position ('app' IN name) from products

9. REVERSE ( ) → Reverse the string (REVERSE (string))
eg Select REVERSE (name) from products;

Date Time func in SQL

1. CURRENT_DATE : Returns the current Date
1.2. CURRENT_TIME : Returns the current time
1.3. CURRENT_TIMESTAMP : Returns the current date and time (Date + time).

eg

```
Select current_date;
Select CURRENT_TIME;
Select CURRENT_TIMESTAMP;
```

2. GETDATE()

Returns the current date and time (similar to current_timestamp)

→ Select GETDATE();

3. DATEADD (datepart, number, date)
   → Adds a specific time interval to date
   → datepart → Part of date to increment
   eg year, month, day

eg Select DATEADD (day, 5, '2023-01-01');
Add 5 days to 2023-01-01

4. DATEDIFF (datepart, startdate, enddate)
   ↳ Returns the diff b/w two dates

> Select DATEDIFF(day, '2023-01-01', '2023-01-10');

↳ Returns 9 days

5. YEAR(), MONTH(), DAY()

↳ Returns specific parts of a date

eg Select YEAR('2023-01-01'); → 2023

SELECT MONTH('2023-01-01'); → Returns 1

SELECT DAY('2023-01-01'); → Returns 1

6. SYSDATETIME()

↳ Returns date and time including fractional seconds

→ SELECT SYSDATETIME()

Q Employees table

↳ first_name, last_name, Job_title, NO

write a query to retrieve full names of all the employees along with employee no

eg

employeeNumber	fullName
1002	Diane Murphy

⇒ Select employeeNumber, CONCAT(firstname, ' ', lastname) from employees;

REGEX → Regular expression

Used for pattern matching or string matching

[abc] → a, b or c

[^abc] → any character except a, b, c

[a-z] → a to z

[A-Z] → A to Z

[a-zA-Z] → a to z, A to Z

[0-9] → 0 to 9

→ Quantifiers in Regex

Using quantifiers we can tell the computer about the repetations.

⇒ [ ]? if ? is there it means
↳ whatever the rule is present b/w brackets it can occur 0 or 1 times

⇒ [ ]+ If + sign is there it means
↳ whatever rule is present in b/w brackets it can occur 1 or more times.
(ie atleast 1 time it has to occur)

→ [ ]* → occurs 0 or more nos of times

→ [ ]{n} → occurs n times

[ ] {n,} occurs n or more times

[ ] {y, z} occurs atleast y times but less than z times

Regex Metacharacters

Metacharacters are basically shortforms

Instead of writing [0-9] we can write as \d

for [^0-9] → \D

for [a-z A-Z _ 0-9] → \w

and \W is for [^\w]

Note: \ ⇒ backwood slash tells computer to treat following characters as search characters, because + sign means different

_ ⇒ this means diff, so in order to search

\+ also known as escape character

Eg: mobile number → starts with 8 or 9 and total digits=10

[8 9][0-9]{9}

this has to repeat 9 times

1st digit can be 8 or 9

Eg 2 first char should be uppercase, it should contain lower case alphabets, only one digit is allowed in b/w.

[A-Z][a-z]*[0-9][a-z]*

[a-z]* : occurs 0 or more nos of times

[0-9] : 1 digit

[a-z]* : 0 or more no of times

Eg 3 Email ID

Eg sadya123@gmail.com

1st Part | 2nd | 3rd

1st Part: alphabets +, -, _ } these will come

[a-zA-Z0-9_\-\.]+[@][a-z]+[\.][a-z]{2,3}

for allowing hyphen we need to use escape character eg \-

Small a to z, A to Z, 0 to 9, _, -, . atleast one then [@] for @ then [a-z]+ for gmail then [\.] for . then [a-z]{2,3} for com or in

In SQL we use as

select * from table_name where column_name REGEXP 'Pattern';

Nested Queries

While generating insights from the data we may need to refer to multiple tables in a query, there are two ways to deal with such type of queries:

1. Joins
2. Nested Queries / subqueries

eg Print the order number of the most valuable order (eg Maximum sale) by sales

> Select ord_id, sales from market_fact_full where sales = (select max(sales) from market_fact_full);

Nested Query

Nested Queries are better :: we don't need to hardcode anything here creating a generic form of query.

Note :: When in a query we are comparing it with NULL we are not supposed to use = instead we should use 'is null' or 'is not null'.

for eg:

Select Prod_id from market_fact_full where Prod_base_margin = null ; ✗

Correct way: Select Prod_id from market_fact_full where Prod_base_margin is null; ✓

Eg Select * from prod_dimen where prod_id in (select prod_id from market_fact_full where Prod_base_margin is null);

Note: the subquery is executed first than the main query.

So, in brief Nested Queries also known as subqueries or inner queries are sql queries that are embedded within another SQL query, they are often used to perform operations where the result of one query depends on the outcome of the another query.

Types of Nested Queries

① Single row subquery : these query return only 1 row or 1 col as output

Select name from employees where salary = (select MAX(salary) from employees);

2. Multi row subquery : these return multiple rows and are typically used with operators like in, any or all

eg Select name from employees where dept_id in (select dept_id from departments where loc = "New york");

3) Multi-Column Subqueries : these subqueries return more than 1 column.

eg Select name, department from employees where (department_id, job_id) in (select department_id, Job_id from Job-assignments where active = 1);

(*) COMMON Table Expressions

these are temporary result sets that you can reference within a select, insert, update or delete statement, they help to breakdown the complex queries into more readable parts.

Syntax of CTE (CTE is defined using "with" clause

```
With cte-name AS (
        select col1, col2
            from table-name where condition )
    Select * from cte-name;
```

eg
```
with AvgSalary AS (
    Select dept_id, Avg(salary) as avg_sal
    from employees GROUP BY department_id
)
Select e.name, e.salary from employees e
Join AvgSalary a ON e.department_id =
a.department_id where e.salary > a.avg_salary
```

(*) Derived table

Derived table in SQL are subqueries that are used in the from clause of main query they act as temporary tables that exist only during the execution of the query.

Syntax :

```
Select dt.col1, dt.col2
    From (
        Select col1, col2
        from table_name
        where condition
    ) As dt where dt.col1 > 100;
```

Views ⇒ these are the Virtual tables which can be used

Defination : Can be thought as a saved query that can be used as a table, however it doesn't hold any physical data

Create view view_name as Select col1, col2 from table where condition;

Joins & Set Operations

Set theory

A set is a collection of distinct values, It is denoted by comma-separated values enclosed by curly braces.

eg set A → even nos , set B → Prime nos

$A = \{2, 4, 6, 8\}$, $B = \{2, 3, 5, 7\}$

the union of A and B ($A \cup B$) will contain all the values that are present in either A or B.

$\therefore A \cup B = \{2, 3, 4, 5, 6, 7, 8\}$

the intersection of A and B ($A \cap B$) will contain all the values present in both A and B

$\therefore A \cap B = \{2\}$

the set difference of A and B ($A - B$) will contain the values that are present in A but not in B

$A - B = \{4, 6, 8\}$

Q Total students in class = 50

E + H = 10

E = 32

need to find only Hindi

$A \cup B = A + B - A \cap B$

$50 = A + 32 - 10$

$50 - 22 = A = 28$

28 ⇒ total students enrolled in Hindi

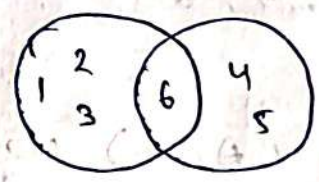
$28 - 10 = 18$ ($\because H - (E+H)$)

Q $A \cap (B \cup A) = ?$

$(A \cap B) \cup (A \cap A) = (A \cap B) \cup A$

$= (A \cup A) \cap (B \cup A)$

$A \cap (B \cup A)$



$A = \{1, 2, 3, 6\}$, $B = \{4, 5, 6\}$

$B \cup A = \{1, 2, 3, 4, 5, 6\}$

$\therefore A \cap (B \cup A) = A$

Types of Joins

In SQL joins are used to combine rows from two or more tables based on a related column b/w them. Joins allow us to query data across multiple tables in a relational database by creating

relationships b/w tables.

Types of Joins in SQL:

① inner join:
- Returns records that have matching values on both tables
- used when we only need rows where there is a match in both tables.

eg Select * from table1 Inner join table2 ON table1.id = table2.id ;

② Left Join or (left_outer join)
- Returns all the records from the left table and the matched records from the right table, if no match is found nulls are returned for columns from the right table.
- used when we want all data from primary (left) table even if there are no matches found, Nulls are returned for cols from right table (if no match)

eg Select * from table1 left join table2 on table1.id = table2_id

③ Right Join:
- Returns all the records from the right table and matched records from left table

eg Select * from table1 right join table2 on table1.id = table2.id ;

4. Full Join

↳ Returns all the records when there is a match on either left or right table if there is no match nulls are returned for non matching rows from either table.

↳ Usefull for combining data when you need to see all possible records regardless of Matches

eg Select * from table1 full join table2 on table1.id = table2.id;

5. Cross join :

↳ Produces the cartesian product of two tables, meaning it joins every row of the first table with every row of the second table.

→ Usefull when you need every combination of rows b/w two tables

eg Select * from table1 coross join table2;

6) Self join :

↳ A join where a table is joined with itself.

↳ Usefull when comparing rows within same tables.

eg Select a.column_name, b.column_name
Form table1 a, table1 b where
a.id = b.Parent_id;

Multijoin

A multijoin refers to a SQL operation where more than two tables are joined together within a single query. This type of join is useful when we need to combine data from multirelated tables.

eg scenario: Imagine a database for an ecommerce platform with the following tables

→ Customers → contains customers details
→ orders → Contains order info
→ Products → Contains Product details
→ OrderDetails → contains order line items linking orders and products

eg
```
Select
    customers.CustomerName,
    Orders.OrderDate,
    Products.ProductName,
    OrderDetails.Quantity
From
    customers
Join
    orders on customers.cust_id = Orders.cust_id
Join
    OrderDetails on Orders.orderid = OrderDetails.ord_id
Join
    Products ON OrderDetails.prod_id = Products.prod_id;
```

ON vs USING

The diff b/w ON and Using is SQL pertains to how conditions are specified on Join clauses, both are used to define how tables are matched when performing joins, but there are subtle distinctions b/w them.

ON ⇒ used to specify join conditions b/w cols of the tables being joined.
It's more flexible than Using because it allows conditions that are not simple col comparisons, you can use any boolean expression

Syntax: Select * from tableA Join tableB ON tableA.col1 = tableB.col2;

USING : It is used when both tables have a col with the same name and you want to join on that column.

Syntax: Select * from TableA Join TableB Using (column_name);

(*) Views : Views in SQL are virtual tables created by a query which allows you to save complex queries for easier access and use later.

A view doesn't store data ... it retrieves data from the underlying tables every time it is queried.

Creating view with Joins

eg Employees table

emp_id	name	dept_id
1	Jhon	101
2	Jane	102
3	Alia	101

Department table

dept_id	department_name
101	Sales
102	Marketing
103	it

Now we want a view that shows each employees name and department name

```
> Create View EmployeeDepartmentView AS
  Select emp.emp_id, emp.name, dept.dept_name
  from employees emp innerjoin department dept
  on emp.dept_id = dept.dept_id;
```

to see the view

```
Select * from EmployeeDepartmentView;
```

to Drop a view => Drop view EmployeeDepartmentView;

✱ SET Operations with SQL

Set operations in SQL allow you to combine the results of two or more select queries.

① UNION

↳ Combines the result of two or more select queries and returns only distinct values (removes duplicates).

Rules:
- → the number and order of columns must be the same in all queries
- → the datatypes of columns must be compatible

eg

TableA

ID	NAME
1	Alice
2	Bob

TableB

ID	Name
2	BOB
3	Charlie

```
Select ID, Name from TableA
UNION
Select ID, Name from TableB;
```

ID	NAME
1	Alice
2	Bob
3	Charlie

② Union All

↳ combines the result of two or more select queries and return all rows, including duplicates

```
Select id, name from table A
Union all
Select id, name from table B
```

ID	Name
1	Alice
2	Bob
2	Bob
3	Charli

Joins Adv eg (Advanced Joins)

* **Natural Join:** Joins tables based on columns with the same name and datatype, it automatically matches columns without specifying condition.

eg Select columns from table1 Natural Join table2;

* **Using Clause:** Specifies the column to join on when columns have the same name in both tables.

eg Select cols from table1 Inner join table2 using (col_name);

* **Join with subqueries:** Allows joining with the result of a subquery.

eg Select main_table.column, subquery_table.column from main_table
Join (Select column from table where condition) subquery_table on main_table.column = subquery_table.column;

Multiple Joins : Joining more than 2 tables,

Eg

```
Select columns from table1
Inner join table2 on table1.col_name = table2.colnam
Innerjoin table3 on table 2.col_name = table3.col-nam
```

OFFSET In SQL

Offset in SQL is used to skip a specific nos of rows in the resultset before starting to return rows, It's often used with limit or fetch to paginate results.

Syntax ⇒ Select columns from table
order by column OFFSET num-of-rows-to-skip ROWS;

Eg Select employee_id, employee_name from employees order by employee_name offset 10 Rows;

↳ Query will skip first 10 rows and return the rows left after the skipped ones.

Eg with limit:

Select * from employees orderby empname offset 2 limit 1;

↳ 3rd record

Rank functions - 1

When we used the GroupBy clause It reduced the no of rows, It also leads to the loss of individual properties of various rows.

to address the two points ie reduction of rows and loss of individual properties of rows, SQL provides a special clause known as "OVER" clause which displays group characteristics in the form of a new column.

"

for eg, If we want to find the top 3 records based on somebody's salary

↳ So we sort it on salary in Descending order then apply limit 3. "

↳ Problems with this approach!

→ It is not a standard way of doing

→ It is very rigid ie. we can only do top3, top5 kind of a query.

↳ So if we convert it to rank type query we will be able to answer multiple things.

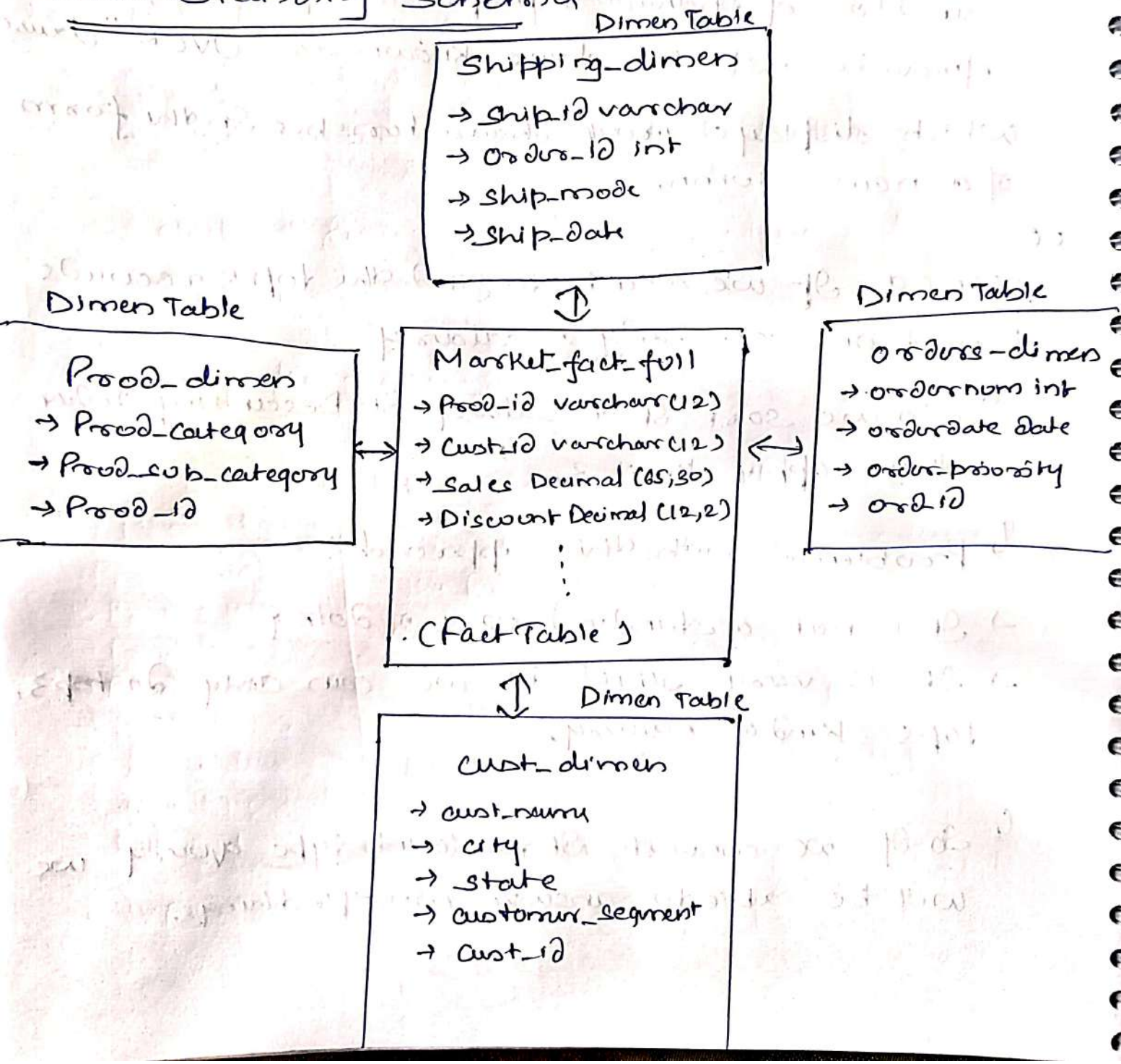
RANK Syntax

RANK ( ) OVER (

PARTITION BY <expression> [{, <expression> ...}]

ORDER BY <expression> [ASC | DESC], [{, <expression> ... }]

Understanding schema



> USE market_star_schema;

we will try to rank all the sales based on order id in descending order

> Select Customer_name, ord_id,
ROUND (sales) as rounded_sales,
RANK () over (ORDER BY SALES DESC)
As sales_rank
From market_fact_full as m
INNER JOIN
cust_dimen as C
ON m.cust_id = C.cust_id
where customer_name = 'RICK WILSON';

---

	o/p	customer_name	ord_id	rounded_sales	sales_rank
		Rick wilson	5136	26095	1
		Rick wilson	3841	8476	2
		Rick wilson	1207	8407	3

Now if we want to filter top 10 records
for this we will use common table expression → for this we use "With" keyword.

```
With rank_info as
(
  Select customer_name,
         ord_id, Round(sales) As rounded_sales,
         RANK() over (order by sales Desc) as
         sales_rank
  From market_fact_full as m
  Inner join
  Cust_dimen as c
  On m.cust_id = c.cust_id
  Where customer_name = 'Rick Wilson'
)
  Select * from rank_info where sales_rank
    <= 10;
```

↳ top 10 records fetched.

Eg 2 Given a table name Products with following cols

Products
- → Prod_code
- → Prod_name
- → Prod_line
- → Prod_scale
- → Prod_vendor
- → Prod_desc
- → quantityInStock
- → buyPrice

→ Write a query to retrive the ranks of the Products in decreasing order of quantityInStock.

```
Select quantityInStock,
RANK() over (order by
quantityInStock Desc) as
quantityRank from products;
```

↳ O/P

quantityInStock	quantityRank
9995	1
6534	2
5839	3

(#) We have dense_rank and percent_rank functions.

let's say we have 5 students in a class

↳ 3 students scored 98/100
&
4th, 5th student scored 95, 90

Now if we apply rank the 1st 3 students in the rank func will get 1, 1, 1 then 2 and 3 is skipped because we already covered 3 students 4th and 5th student will get 4, 5 as ranks.

Instead of this approach, if we want to give the rank to 4th and 5th student as 2nd, 3rd given that they got 2nd heighest, 3rd Heighest marks → for this we use dense_rank

What Does percent rank do?

↳ Converts the entire ranking into % age, whole class will get nos from 0 to 1, similar to percentile

Types of Rank:

⇒ RANK(): Rank of current row within its partition, with gaps.

⇒ Dense_rank(): Rank of current row within its partition without gaps.

⇒ Percent_rank(): Percentage Rank Value

DENSE_RANK and PERCENT_RANK syntax

```
Dense_rank ( ) over (
    Partition by <expression> [{, <expression>...}]
    Order by <expression> [ASC | DESC],
                [{, <expression>...}]

Percent_rank ( ) over (
    Partition by <expression> [{, <expression>...}].
    Order by <expression> [ASC | DESC],
                [{, <expression>...}]
```

Now let's understand how to use different rank functions & queries to solve various Problems.

Eg for dense_rank

```
>Select ord_id,
        discount,
        customer_name,
        RANK ( ) over ( order by discount Desc ) as
        disc_rank,
        DENSE_RANK ( ) over ( order by discount Desc)
         As disc_dense_rank
From    market_fact_full as m Innerjoin cust_dimen as
C on    m.cust_id = c.cust_id where customer_name = "Rick Wilson";
```

Ord_id	discount	cust_name	disc_rank	disc_dense_ra
Ord_3855	0.10	Rick wilson	1	1
Ord_5186	0.09	Rick wilson	2	2
Ord_1209	0.09	Rick wilson	2	2
Ord_3841	0.08	Rick wilson	4	3

RANK() DENSE_RAN

We learnt about the 'RANK', 'DENSE_RANK', 'PERCENT_RANK' functions, in this segment we will learn about fourth type

→ Using RANK() and Dense_RANK we can get the rows having exact same rank, so to avoid this we have a concept of Row Number, It gives a unique value for each and every row

ROW_NUMBER syntax

```
Row_NUMBER() over (
        Partition By <expression> [ ,<expression>..]
        order By <expression> [ASC | DESC], [ ,
                                <expression>... ]
    )
```

We can use "Row_NUMBER()" for the following type use cases.

→ to determine the top 10 selling products out of a very large variety of Products

→ to determine the top 3 winners in a car race.

Eg We want to rank various customers based on the nos of orders placed.

```
> Select customer_name,
  Count (Distinct ord_id),
  RANK () over (order by count (Distinct ord_id)
                DESC),
  DENSE_RANK () over (order by count (
                Distinct ord_id) DESC),
  ROW_NUMBER () over (order by count (
                Distinct ord_id) Desc)
From market_fact_full as m
Inner Join
cust_dimen AS C
 ON m.cust_id = c.cust_id
Group By customer_name;
```

(*) Partitioning

We know that the 'rank()' and 'dense()' func are used with Over clause, However this may not be enough if we want to rank groups of rows based on certain criteria,

for eg we can rank the top 10 Batsmen in the world using rank() function, what if we want to findout top 10 batsmen from each team

↳ for this use case we use "partition" and "over" clauses together.

-- Partitioning example --

first creating data

```
Select ship_mode, Month(ship_date) As shipping_
    month, count(*) as shipments
From
Shipping_dimen
Group By ship_mode,
        Month(ship_date);
```

>

ship_mode	shipping_month	shipments
Regular Air	10	467
Regular Air	11	467
Delivery Truck	6	91
Regular Air	5	564
Delivery Truck	9	86
		488

On top of this we have to rank

```
With shipping_summary As
(
  Select ship_mode, MONTH (ship_date) As
        shipping_month,
    COUNT(*) as shipments
  From
    shipping_dimen
  Group by ship_mode,
        Month (ship_date)
)
Select *, RANK() over (Partition BY
  Ship_mode ORDER BY shipments DESC) As
  shipping_rank
From shipping_summary;
```

Ship_mode	shipping_month	shipments	shipping_rank
Delivery Truck	12	107	1
Delivery Truck	4	105	2
Delivery Truck	3	105	2
Express Air	12	96	1
Express Air	7	91	2
Express Air	4	86	3

Rank got reset here

Over function :

the over func in SQL is used to perform with the window func to perform calculations across a set of rows related to the current row, without requiring grouping into a single result row.

this is particularly useful in analytical queries where you want to get results on a row by row basis with some aggregation or calculation applied.

Common window function with Over

① ROW_NUMBER : Assigns a unique sequential integer to rows within a partition.

eg
```
Select employee_id, dept_id, salary,
ROW_NUMBER() over ( Partition By
dept_id ORDER BY Salary DESC )
As row_num from employees;
```

Here, ROW_NUMBER() will restart the numbering for each department, ordered by salary in descending order

2. RANK : Similar to ROW_NUMBER, but if two rows have the same value, they get the same rank, and the next rank is skipped.

```
> Select employee_id, dept_id, salary,
    RANK () over ( Partition By dept_id order by
    salary DESC ) As rank from employees;
```

---

3. DENSE_RANK : Similar to RANK, but without skipping ranks if there are ties.

```
> select employee_id, dept_id, salary,
  DENSE_RANK ( ) OVER ( PARTITION BY dept_id
      ORDER BY salary DESC ) As dense_rank
  from employees;
```

---

4. SUM, AVG, MIN, MAX : Aggregate func that provide a running total, average, minimum, or maximum for a specific window

```
> Select employee_id, salary,
    SUM (salary) OVER ( order by employee_id )
  As running_total from employees;
```

Components of the Over Clause

→ Partition By : Divides the result set into Partitions, similar to groups in aggregate

→ ORDER BY : Defines the order of rows within each partition.

→ Rows : Specifies the range of rows for each calculation.

Eg = Table named → orderdetails with following cols

OrderDetails
→ OrderNumber
→ ProductCode
→ quantityOrdered
→ PriceEach
→ OrderLineNumber

Q-there are some orders that have multiple order amounts corresponding to same order, write a query to retrive the individual and total order amounts for each order along with order numbers

expected o/p

OrderNumber	individualOrderAmount	totalOrderAmount
10154	2332.13	4465.85
10154	2133.72	4465.85

```
> Select
    OrderNumber,
    (quantityOrdered * priceEach) As individualOrderAmt,
    SUM (quantityOrder * priceEach) Over (PartitionBy
      OrderNumber) As totalOrderAmount
  From OrderDetails
OrderBy OrderNumber ASC, individualOrderAmount DESC;
```

Window In SQL

In SQL a window function is a feature that allows you to perform calculations across a set of table rows related to the current row, without needing to use GroupBy.

The concept is called window because the calculations are done over a subset or "window" of rows within a resultset

Key Concepts:

1. Window func: functions that perform calculations across the rows in a specified window or Partition.

Common window funcs are:

→ ROW_NUMBER() : Assigns a unique row number within the partition.

→ RANK(), DENSE_RANK() : Assigns a rank to each row within a partition

→ SUM(), AVG(), MIN(), MAX() : Performs aggregate calculations within a ~~rank~~ partition.

2. OVER() : Specifies the window (or range of rows) to use with the window function

→ Partitioning (PARTITION BY) : Divides the data into partitions or subsets for each calculation eg Partition by department, country etc.

→ Ordering (Order By) : Determines the order in which rows within partition are arranged.

eg Let's say we have "Sales" table with cols sales_person_id, sales_amount, sales_date, we want to calculate running total of sales_amount for each salesperson

```
> Select
    sales_person_id,
    sales_date,
    SUM( sales_amount) OVER( Partition By
        sales_person_id order by sales_date Desc)
        As running_total
From
Sales;
```

Named Window

A Named window function in SQL is a feature that allows you to define a reusable window specification with a name, which can then be referenced in multiple window functions within a query, particularly useful for simplifying complex queries.

Syntax:

We use "window" clause followed by window_name and the window_specification, you then refer to the named window in the over clause

ee

```
WINDOW window_name AS (window_spec)
```

Here window-spec let's us define details like Partition By, Order By etc

Eg

We have a table "Sales" with columns salesperson_id, sale_amount, sale_date

We want to calculate running total and Moving avg for each salesperson

eg Select
Salesperson_ID,
Sale_date,
Sale_amount,
Sum (sale_amount) over w as total
Avg (sale_amount) over w as average
From
sales
WINDOW w As (Partition By salesperson
id Order by sales_d
;

FRAMES

frames in SQL is used with window func in SQL. Whenever we use a window function, it creates a "window" or "partition" depending upon the column mentioned after the "partition by" clause in the "over" clause and then it applies the window func to each of those partitions and inside those partitions, we can create a subset of records using the frame clause.

∴ A frame clause specifies a subset

Frames are useful when you want to calculate values that depend on the nearby rows.

Using frames with window functions

frames work with window func in SQL
A window func like SUM() or AVG() let's you perform calculation over a window of rows around each row. When we define a frame we tell SQL which rows around the current row should be included in the calculation.

Eg
```
Select employee_id,
       salary,
       SUM(salary) over (ORDER BY employee_id)
                 AS running_total
  from employees;
```

Here SUM() calculates the running total of salaries and the order by employee_id sorts the rows by employee_id.

What does a frame DO?

A frame lets you control the specific range of rows in each window

→ which rows to start and end with (e.g. start from beginning look at only the current row or look at the last two rows).

→ How far forward or backward to look (eg include two rows before and one row after the current row).

Defining frames with ROWS and RANGE

- ROWS : Looks at the fixed number of rows relative to the current row.
- Range : Looks at rows with specific value in the ordered column, allowing you to include row with the same value.

frame options

eg : Running total using a frame

to get a running total of salaries for each employee, you can include all rows from start up to 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;
```

→ UNBOUNDED PRECEDING : Include all the rows from start

→ Current row ⇒ Includes the current row

So for each employee, this frame adds up the salaries of all employees up to that point.

Eg 2 : Moving Avg of salaries Over a few rows to calculate a moving avg of salaries for each employee based on two rows before and the current row.

```
? Select employee_id, salary, AVG(salary) OVER
    (ORDER BY employee_id ROWS BETWEEN 2
     PRECEDING AND CURRENT ROW) As Moving-
                                    avg
  from employes
```

Eg

Sales_id	employee_id	Sale_amount	Sale_date
1	101	100	2024-01-01
2	102	150	2024-01-02
3	101	200	2024-01-03
4	103	300	2024-01-04
5	101	250	2024-01-05
6	102	400	2024-01-06

to calculate the running total of sales upto each sale date

```
Select sales_id, sale_date, sale_amount,
    SUM(sale_amount) OVER ( order by sale_date
        ROWS BETWEEN UNBOUNDED
        PRECEDING AND CURRENT ROW) AS
    Running_total
    From sales ;
```

O/P

sales_id	sale_date	sale_amount	running_total
1	2024-01-01	100	100
2	2024-01-02	150	250
3	2024-01-03	200	450
4	2024-01-04	300	750
5	2024-01-05	250	1000
6	2024-01-06	400	1400

Eg2 # calculating the 3 day moving avg of sales

```
select sales_id, sale_date, sale_amount, AVG(sale_amount
OVER (ORDER BY sale_date ROWS BETWEEN 2 PRECEDING
AND current row) as moving_avg from sales;
```

O/P

sales_id	sale_date	sale_amount	moving_avg
1	2024-01-01	100	100
2	2024-01-02	150	125
3	2024-01-03	200	150
4	2024-01-04	300	216.67
5	2024-01-05	250	250
6	2024-01-06	400	316

SQL eg

```
Select ship_date , SUM ( shipping_cost) AS
daily_cost_total
From
market_fact_full as m Inner join shipping_
dimen As s on m.ship_id =S.ship_id
Group By ship_date;
```

(

Ship_date	daily_total
2010-10-20	93.92
2010-11-24	40.57
-	-

-- Frames examples --

```
With daily_shipping_summary as
(  Select ship_date ,
        SUM (shipping_cost) As daily_total
   From
   market_fact_full as m
   Inner Join
   shipping_dimen as s
    on m.ship_id = s.ship_id
   Group By ship_date
)  Select * ,
   SUM ( daily_total ) OVER W1 as running_total,
```

```
AVG (daily_total) Over W2 as moving_avg
from daily_shipping_summary
window W1 as ( Order by daily_total Rows
        UNBOUNDED PRECEDING ),
W2 AS (ORDER BY daily_total Rows 6 Preceding)
;
```

RANGE Clause

Range clause within a window function defines a specific range of values relative to the current row for calculations like averages, sums, and other aggregations. Unlike the rows clause which counts a fixed number of rows, the range clause looks at values in a specified column.

⇒ the RANGE clause is used in the OVER() part of window func

Syntax

```
< window_func > Over (
        order by <column>
        Range between <n> Preceding AND <M>
                                Following )
```

Eg Kohli on ODIs

table given below contains the number of runs scored by Virat Kohli over a time period

Kohli_Batting

year	runs
2008	159
2009	328
2010	995
2011	1382
2012	1028
.	.

to calculate
4 Moving Avg

years	Runs	Moving Avg
2015	623	1145.4

```
Select
    year, runs,
    AVG(RUNS) OVER ( ORDER BY year
    RANGE BETWEEN 4 Preceding AND
    current row ) AS Moving_average
From
Kohli_batting
Order By
    Years;
```

So some necessary keyword in frames are Unbounded preceding, following, between etc

(Eg)

Row1
Row2
Row3 ← ← current cursor
Row4
Row5
Row6
Row7

↳ order by x Rows 2 following

↳ Row 4 and Row 5
current Row ke Neeche ke 2 ro

(Eg)

Row 1
Row 2
:
Row 9
Row 10

→ current cursor → Row 6

↳ order by x Rows Between 2 Preceding and 3 following

↳ Row 4 to Row 9

(Eg)

Row 1
Row 2
Row 3
Row 4
:
Row 10

→ current cursor → Row 6

↳ order by x Desc Rows Between unbounded Preceding And 2 following

↳ Row 4 to Row 11

∴ the Desc keyword has been used the order of the rows on which the frame will be created will be reversed.

Lead and Lag functions

the Lead and Lag functions are usefull in SQL and data analysis for accessing data from a row before or after the current the row without needing a self join or complex subquery they are often used in window functions, where data is organized based on specific partition or orderings, making it easier to analyze sequential data patterns.

LEAD Function

the LEAD func allows you to access data from a subsequent row in your dataset, based on the specific offset, for eg you might want to see the next row's value in each current row, useful for comparisons with future datapoints.

Syntax

```
LEAD (col_name, offset, default_value) over
    ( Partition by Partition_col order by
        order_columns)
```

→ Col_name : the column to retrieve the data from

→ offset : How many rows ahead to look (default is 1)

→ default_value : Value to return if the requested row doesn't exist (default is NULL)

eg = We have table Sales with columns "day" & "revenue"

day	revenue
Mon	100
Tue	150
Wed	200
Thurs	250
Fri	300

we want to see each day revenue along with revenue of the next day

> Select day, revenue, LEAD (revenue, 1, 0) over (order By day) as next_day_revenue from sales;

↳ O/P

day	revenue	next_day_revenue
Mon	100	150
Tue	150	200
Wed	200	250
Thurs	250	300
Fri	300	0

Note : for friday, next_day_revenue is 0 because there is no following day's revenue.

LAG func

The LAG func works similarly but access data from a preceding row, this is useful when we want to compare each row with previous value

Syntax :

LAG (col_name, offset, default_value) over (Partition By partition_col Order By order_column)

Eg # We have same Sales table, we want to compare each day's revenue with Prev day's revenue

```
Select day, revenue, LAG(revenue, 1, 0)
OVER (Order by day) as prev_day_revenue
```

day	revenue	Prev_Day_revenue
Mon	100	0
Tue	150	100
Wed	200	150
Thur	250	200
Fri	300	250

Applications

→ LEAD and LAG are excellent for tracking trends over time, such as changes in sales, stock prices etc.

(*) CASE Statements

We encountered if else and case statements in python, java, similarly we have it in SQL too.

CASE statement Syntax

```
CASE
    when condition1 then result1
    when condition 2 then result2
    :
    :
    when condition N then result N
    ELSE result
END as col_name
```

Eg Demo # In the market fact full table we will be fetching market_fact_id, profit columns

Select market_fact_id, profit from market_fact_full;

↳

Market_fact_id	Profit
1	-3051
2	4.56
3	1148.56
4	729.0

Now we want to categorise the profit into 4 buckets (Huge loss, bearable loss, decent profit, great profit)

-- Case when example --

/* Profit < -500 → Huge loss
Profit -500 to 0 → Bearable loss
Profit 0 to 500 → decent profit
Profit > 500 → great profit */

> Select market_fact_id,
Profit,
CASE
when profit < -500 Then 'Huge Loss'
when profit Between -500 AND 0
Then 'Bearable loss'
when profit Between 0 and 500
Then 'Decent Profit'
ELSE 'Great Profit'
END AS Profit_type
From
market_fact_full;

eg Below is the table for income tax slabs

Tax slabs

Taxable Income Slabs	Tax Slabs
Upto Rs 2.5 lakh.	A
Rs 250001 to Rs 500000	B
Rs 500001 to Rs 1000000	C
Rs 1000000 and above	D

Salaries

Name	Salary (in lpa)
Sundar Pichai	6.8
Jeff Bezos	8.7
Bill Gates	9
Ambani	14

Add another col to output the tax slabs along with given data.

```
Select Name, Salary,
CASE
  When Salary <= 2.5 then "A",
  When Salary Between 2.5 and 5
       Then 'B'
  When Salary Between 5.1 And 10 Then 'C'
End As Tax_slab
from Salaries;
```

Common table expression CTEs

CTE is a temporary named result set in SQL often used to simplify complex queries and improve readibility

→ CTEs are created using "with" clause and can be referenced within the main query or other CTEs.

CTEs are usefull when you need to use the same set of data multiple times within a query.

Syntax

```
With cte_name as (
    Select col1, col2
        From table_name
        Where condition
)
Select * from cte_name;
```

Defining the query for CTE

Eg

```
with ProductRevenue AS (
    Select product_id, SUM (quantity x price) As
        total_revenue from sales
        Group by product_id
) Select product_id, total_revenue from
    ProductRevenue
    where total_revenue > 5000;
```

Eg Tax Slabs 2

Tax Rates

Taxable Income Slabs	Income tax rates
upto rs 2.5 lakh	Nil
rs 250001 to 500000	5% of (Total Income - 250000)
rs 500001 to 1000000	12500 + 20% (total Inc - 500000)
rs 1000001 and above	112500 + 30% (total Inc - 1000000)

```
Select name, salary,
  CASE
    when salary <= 2.5 then 0
    when salary between 2.5 and 5 then
                ROUND(0.05 * (salary - 2.5) * 100000
    when salary between 5 and 10 then
                Round(12500 + 0.2 * (salary - 5) * 1000
    Else Round(112500 + 0.3 * (salary - 10) * 100000)
  End as Tax_amount
From salaries;
```

UDFs (User Defined functions)

You already know how to deploy various in built MySQL functions, such as sum(), avg() and concat().

Now it's possible to find the sum of two numbers in MySQL without using sum() func, Now we can use arithmetic + operator.

However there are operations that you may want to repeat multiple times in a piece of code because they do not have inbuilt func → for this we use user-defined functions (UDFs) or stored func

Syntax

* Create func is a DDL statement

Syntax:

```
Create function function_name ( func_parameter1,
                                func_parameter2, ... )
    RETURN datatype [characteristics]
/* func_body */
Begin
    <MySQL statements>
    Return expression;
END
```

Note: the func body must always contain one return statement.

-- example --

```
DELIMITER $$
CREATE FUNCTION profitType (Profit int)
RETURNS VARCHAR (30) DETERMINISTIC
BEGIN
DECLARE message VARCHAR(30);
IF Profit < -500 THEN
        SET message = 'Huge Loss'
ELSE IF Profit BETWEEN -500 and 0 THEN
        SET message = 'Bearable Loss'
ELSE If Profit BETWEEN 0 and 500 then
        Set message = 'Decent Profit'
```

```
ELSE
    SET message = 'Great Profit'
END IF;
RETURN message;
END;
$$
DELIMITER ;
```

Calling func

```
Select ProfitType(10);
Select ProfitType(-20);
```

Note: A stored func returns one and only one value.
Deterministic keyword is used to ensure that the output is the same for the same input values

Stored Procedures

We want to write certain snippets which keeps on getting used again and again.

Syntax

```
DELIMITER $$
CREATE PROCEDURE PROCEDURE_name (<Parameter list>)
BEGIN
    SQL statement of Stored Procedure ;   # Can be N nos of SQL Statement
END $$
DELIMITER ;

# Call Statement
CALL Procedure_name;
```

Eg

```
DELIMITER $$
Create procedure get_sales_customers (sales_input
                                      int)
BEGIN
    Select DISTINCT cust_id, ROUND(sales) AS
           sales_amount
    From
        market_fact_full
    where ROUND(sales) > sales_input
    order by sales;
END $$
DELIMITER ;

CALL get_sales_customers (300)
```

Note : # For Dropping the Procedure

```
DROP PROCEDURE <Procedure_name>
```

Difference b/w UDF and Stored Procedure

UDF	Stored Procedure
→ It supports only one input	→ It supports input/output and input-output Params
→ It can be called using select statement	→ It can be called using CALL func
→ It must return a value	→ It need not return a value
→ only select operation is allowed	→ All database operations allowed

Best Practices

① Write appropriate comments

In MySQL for single line comments

↳ -- single line comment

For Multiline Comments

```
/* comment section
with multiple lines
*/
```

② Always use table aliases when SQL statement involves more than one source

↳ In Joins we use (AS)

③ Assign simple descriptive names for columns

④ Write all keywords in CAPS and variable names in small case

⑤ To use proper indentation

⑥ use new line for different section of the query

⑦ Use SQL formatter (Many available online) for beautifying the query

Indexing

One of the ways in which you can greatly reduce the runtime of your queries is by using Indexing.

Indexing is the process of referring to only the required values directly, instead of going through entire table. This prevents the query engine from looking up the values one by one, instead it returns the exact value that you want right away.

Indexing is necessary for querying extremely large datasets, a primary key is an index because it helps in identifying each record in a table uniquely.

Note: Although you can not view Indexes whenever you execute queries, they occupy some storage space, Indexes are internal to database engine.

When we are writing our select Query

```
Select <columns>
from ____
where Salary < 10000
or DeptId = 5;
```

the most inefficient way to fetch the record line by line and making decision on a row by row basis.

But Notice the where clause here

```
Select <columns>
from employee
where salary < 10000
or DEPT_ID = 5;
```

([ ) Attributes where clause में use hote हैं उनही पर Indexing करते हैं।

MYSQL has a specific DDL command called as "CREATE INDEX" → used for specifying the attribute on which you want to create an Index.

INDEX SYNTAX

→ Index is a DDL statement

Creating an Index

```
CREATE INDEX index_name ON table_name
                (column1, column2, ... );
```

we can also use Alter command

```
Alter table table_name ADD INDEX index_name
                (column1, column2, ... );
```

Dropping an Index:

Alter table table_name DROP INDEX index_name;

-- Index Demo --

first creating a temporary table from an existing table.

> Create table market_fact_temp AS
Select * from market_fact_full;

Create index (using combination of cust_id, ship_id, Prod_id

> Create index filter_index on market_fact_temp
(cust_id, ship_id, prod_id);

for dropping the index

> Alter table market_fact_temp Drop index
filter_index;

the "where" command in SQL works efficiently when we are using indices because the index will enable us to get the exact rows that you need without processing entire table row by row.

CLUSTERED VS NON CLUSTERED INDEXING

Clustered Index	Non-clustered Index
→ this is mostly the primary key of the table	→ It is a combination of one or more columns of the table.
→ It is present within the table	→ the unique list of keys will be outside the table
→ there is no requirement for a separate mapping as this index is part of the table.	→ the external table will point to different sections of the main table
→ relatively faster	→ relatively slower

Order of Execution of the Query

the order in which the various SQL statements appear in the query is as follows

1) Select
2) From
3) [Join]
4) Where
5) Group By
6) HAVING
7) Window
8) OrderBy

} the order in which various statements are executed by the database engine is not same.

However the order in which the various statements are executed by the database engine is not the same

From (including joins)
↳ Where
↳ GroupBy
↳ Having
↳ Window Func
↳ Select
↳ Distinct
↳ Order By
↳ Limit & Offset

Some of the imp points that you should keep in mind while writing a query

→ Use inner joins wherever possible to avoid having unnecessary rows in resultant table.

→ Avoid using Distinct while using group by clause as it slows down query processing

Joins Vs Nested Queries

EMP

Eid	Name	DID

DEPT

DID	DNAME
1	HR
2	IT

Now we want to get [illegible] of the emp whose department name is HR

One way using subquery

```
Select name from Emp
where DID in (
        select DID
        from Dept
        where DName='HR');
```

using Joins

```
select Name from
Emp innerjoin
Dept on EMP.DID
  = Dept.DID
where DName =
      'HR';
```

⊛ Profitability Analysis

Problem Statement : Growth team wants to understand sustainable (profitable) product categories.

→ Sustainability can be achieved when we make better profits or at least positive profits

→ We can look at the profits per product category.

→ We can look at profits per product subcategory.

→ We can check Avg profit per order

→ Also consider Avg profit % per order.

Tables that we will use :

⇛ market_fact_full

⇛ Prod_dimen

⇛ Orders_dimen

Schema that we will use :

- Shipping_dimen
  - → Ship_id VARCHAR (12)
  - → Order_id int (11)
  - → Ship_mode varchar (25)
  - → Ship_date DATE

- market_fact_full
  - → Prod_id Varchar (12)
  - → cust_id Varchar (12)
  - → Sales Decimal
  - → Discount Decimal
  - → Order Quantity
  - → Profit
  - → Shipping_cost
  - → Product_Base_Margin
  - → Market_fact_id
  - → Ord_id
  - → Ship_id

- Prod_dimen
  - → Product_category
  - → Product_Subcategory
  - → Prod_id

- Order_dimen
  - → Order_number
  - → Order_date
  - → Order_priority
  - → Ord_id

- Cust_dimen
  - → Customer_name
  - → City
  - → State
  - → Customer_segment
  - → cust_id

Problem statement : Growth years ...
Sustainable (profitable) product categories

→ Sustainability can be achieved when we make better Profits or atleast positive profits.

↳ we can look at the profits per category
↳ we can look at the profits per product subcategory
↳ we can check avg profit per order
↳ we can check Profit % per order

first get the data first

```
Select P. Product_category,
       P. Product_Sub_Category,
       SUM(m. Profit) As Profits
FROM
    market_fact_full as m
INNER JOIN
       Prod_dimen as P
ON  m.prod_id = P.prod_id
GROUP BY
       P. Product_category
       P. Product_Sub_category
Order By SUM(m. Profit) ;
```

→ O/P

Prod_cat	subcateg.	Prof
—	—	—
—	—	—
—	—	—

We are getting Profits made at a Product_category and Product_subcategory level but we aren't able to get any insights out of this.

↳ -ve Profits are at top.

(#) for Profits Per category

```
select P. product_category, SUM(m. Profit) As Profits
From market_fact_full as m inner join Prod_dimen
as P on m.prod_id = P.prod_id Group by P. Product_
Category Order By SUM(m. profit);
```

O/P

Product_category	Profit
Furniture	117433.03
Office Supplies	518021.43
Technology	886313.52

All product categories report good profits, Hence they do not contribute to the solution, but we can not identify the reasons for any loss the company is facing

∴ Profit Per product category is still an important ~~subcategory~~ metric that can be used for creating business reports, Profit per product subcategory is a useful metric as it shows subcategory that report heavy loss.

Profit Per Product Subcategory

```
Select P. Product_Category,
       P. Product_Sub_category,
       SUM( m. profit ) As Profit
From
    Market_fact_full as M
Inner Join
    Prod_dimen as P
ON  m. prod_id = P. Prod_id
Group By  P. product_category
          P. product_subcategory
Order by  SUM ( m. profit )
```

Exploring ord_id, order_number from order table

```
> select ord_id, order_number
  from
      orders_dimen
  group by
          ord_id,
          order_number;
```

from this we are not able to understand wether order id to order no is a unique mapping.

let's go with another approach

```
Select
    Count(*) as rec_count,
    Count ( Distinct ord_id ) as ord_id_count,
    Count ( Distinct ord_number ) as ord_num_cou
from
    orders_dimen
```

O/P

rec_count	ord_id_count	ord_num_count
5506	5506	5496

→ Multiple order ids can have same order num