

Database Management System (DBMS)

A Database Management System (DBMS) is basically a software where you can store, retrieve, define, and manage your data in a database.

Most Used DBMS:

- MySQL
- Oracle
- Microsoft SQL
- PostgreSQL
- Mongo DB

Advantages of DBMS

1. DBMS has lots of techniques to store, manipulate and retrieve data.
2. DBMS considered as a most efficient handler to balance the data.
3. A DBMS uses lots of powerful functions to store, manipulate and retrieve data efficiently.
4. Data Integrity and security is one of the strongest parts of DBMS.
5. The DBMS use data integrity to protect data and maintains the privacy.
6. Helps to reduced Application Development Time.

RDBMS - A Relational Database Management System is used for the database management system (DBMS). The concept is based on the relational model as introduced by E.F. Codd.

- 1) The data in RDBMS is stored in database object called tables. A table is a collection of related data entries, and it consists of columns and rows.
- 2) A record, also called a row is each individual entry that exists in a table.
- 3) A column is a vertical entity in a table that contains all information associated with a specific field in a table.

What is SQL?

- 1) SQL stands for Structured Query Language.
- 2) SQL lets you access and manipulate databases.
- 3) SQL became a standard of the American National Standards Institute (ANSI) in 1986, and of the International Organization for Standardization (ISO) in 1987.

What Can SQL do?

- SQL can execute queries against a database.
- SQL can retrieve data from a database.
- SQL can insert records in a database.
- SQL can update records in a database.
- SQL can delete records from a database.
- SQL can create new databases.
- SQL can create new tables in a database.
- SQL can create stored procedures in a database.
- SQL can create views in a database.
- SQL can set permissions on tables, procedures, and views.

MySQL Data Types:

The data type of a column defines what value the column can hold: integer, character, money, date and time, binary and so on.

In MySQL there are three main data types: String, Numeric, and Date and Time.

- String: char, varchar etc.
- Numeric: int, float, bool etc.
- Date and time: date, datetime etc.

STRINGS

CHAR (10) – Fixed Length

VARCHAR (100) – Variable length, max: 65535

MEDIUMTEXT –Max 16 mb

LONGTEXT – Max 4 mb

TINYTEXT – Max 255 bytes

TEXT – Max 64 KB

NUMERIC

TINYINT: 1 bytes

SMALLINT: 2 bytes

MEDIUMINT: 3 bytes

INT: 4b – whole numbers

BIGINT: 8 bytes

DECIMAL (precision, scale): DECIMAL(3,2) => 1.56

DATE AND TIME

DATE: 2022-01-01

TIME: 22:52:11

DATETIME: 2022-01-01 22:52:11

TIMESTAMP: 2010-01-01 22:52:11 (limited from 1970-01-01 to 2038-01-01)

YEAR: 2022

BOOLEAN

BOOL: 0/1 – True/False

Commonly used data types in SQL:

- int: used for the integer value
- float: used to specify a decimal point number
- bool: used to specify Boolean values true and false.
- char: fixed length string that can contain numbers, letters, and special characters.
- varchar: variable length string that can contain numbers, letters, and special characters.
- date: date format YYYY-MM-DD
- datetime: date and time combination, format is YYYY-MM-DD HH:MM:SS

Primary Key:

- A primary key is a unique column we set in a table to easily identify and locate data in queries.
- A table can have only one primary key, which should be unique and NOT NULL.

Foreign Key:

- A Foreign Key is a column used to link two or more tables together.
- A table can have any number of foreign keys, can contain duplicate and NULL values.

Constraints:

- Constraints are used to specify rules for data in a table.

- This ensures the accuracy and reliability of the data in the table.
- Constraints can be specified when the table is created with the create table statement, or after the table is create with the ALTER TABLE statement.

SYNTAX:

```
Create TABLE table_name (
    column1 datatype constraint,
    column2 datatype constraint,
    column3 datatype constraint,
    ....
);
```

Query:

```
CREATE TABLE customer (
    "ID" int8 PRIMARY KEY,
    "NAME" varchar (50) NOT NULL,
    "AGE" int NOT NULL,
    "City" char (50),
    "Salary" numeric
);
```

Insert Values in Table

The Insert into statement is used to insert new records in a table.

Syntax:

```
Insert into table_name
(column1, column2, column3,...column
values
value1, value2, value3,...valueN
);
```

```
INSERT INTO public.customer("ID", "NAME", "AGE", "City", "Salary")
VALUES
(1, 'Sam', 26, 'Delhi', 9000),
(2, 'Ram', 19, 'Bangalore', 11000),
(3, 'Pam', 31, 'Mumbai', 6000),
(4, 'Jam', 42, 'Pune', 10000);
```

Update values in Table:

Update command is used to update existing rows in a table.

Update table name

```
set "column_name1" = 'value1', "column_name2" = 'value2'
```

```
where "ID" = 'value'
```

```
UPDATE customer
```

```
SET custname = 'Xam', age = 32
```

```
WHERE custid = 4;
```

Delete Values in Table:

The Delete statement is used to delete existing records in a table.

Syntax: Delete from table_name where condition;

Delete from customer where custid = 3;

ALTER Table:

The Alter Table statement is used to add, delete or modify columns in an existing table.

1) Alter Table - Add Column Syntax

Alter Table table_name Add Column column_name;

2) Alter Table - Drop Column Syntax

Alter Table table_name Drop column column_name;

3) Alter Table - Alter/Modify Column Syntax

Alter Table table_name

Alter Column column_name datatype;

Alter Table customer Add Column pin_code int;

Alter Table customer drop column pin_code;

Drop and Truncate Table

The Drop table command deletes a table in the database.

Drop Table table_name.

The Truncate Table command delete the data inside a table, but not the table itself.

Truncate Table table_name;

Commonly used constraints in SQL:

1. NOT NULL - Ensures that a column cannot have a NULL value.
2. UNIQUE - Ensures that all values in a column are different.
3. PRIMARY KEY - A combination of a NOT NULL and UNIQUE.
4. FOREIGN KEY - Prevents actions that would destroy links between table (used to link multiple tables together).
5. CHECK - Ensures that the values in a column satisfies a specific condition.
6. DEFAULT - Sets a default value for a column if no value specified.
7. CREATE INDEX - Used to create and retrieve data from the database very quickly.

Structure of a SQL Query

- 1) Select – <select list> - defines which columns to return
- 2) From - <table source> - defines tables to query
- 3) Where - <search condition> - filter rows using a predicate
- 4) Group by - <group by list> - arranges rows by groups
- 5) Having - <search condition> - filters groups using a predicate
- 6) Order by - <order by list> - sorts the output
- 7) Limit - <limit the display result > - display limit

Syntax Order vs Execution Order

What the Query looks like	How it's executed	Why it works this way
Select	From	SQL starts with which table your query is taking data from
From	Where	This is how SQL filters on rows
Where	Group BY	This is where your SQL query checks if you have an aggregation
Group BY	Having	Having requires a group by statement
Having	Select	Only after all these calculations have been made will SQL "Select" which columns you want to see returned
Order BY	Order BY	This sorts the data returned
Limit	Limit	Lastly, you can limit the number of rows returned

Select Statement:

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

Select column_name from table_name;

Select * from table_name;

To select distinct/unique fields available in the table

Select distinct column_name from table_name;

Where Clause:

The WHERE clause is used to filter records.

It is used to extract only those records that fulfil a specified condition.

Syntax:

Select name from classroom where grade = 'A';

Operators in SQL:

The SQL reserved words and characters are called operators, which are used with WHERE clause in a SQL query.

Most used operators:

1. Arithmetic Operators: Addition (+), Subtraction (-), Multiplication (*), Division (/), Modulus (%).
2. Comparison Operators: Equal (=), Not Equal (!=), Greater Than (>), Greater than Equals (>=).
3. Logical Operators: Perform the Boolean operations, example: ALL, IN, BETWEEN, LIKE, AND, OR, NOT, ANY
4. Bitwise Operators: Bitwise AND (&), Bitwise OR (|).

Limit Clause:

The LIMIT clause is used to set an upper limit on the number of tuples returned by SQL.

Ex: select column_name from table_name limit 5; - the code will return 5 rows of data.

Order BY Clause:

The Order BY is used to sort the result set in ascending (ASC) or descending (DESC).

EX: select column_name from table_name order by column_name e ASC; the code will sort the output data by column name in ascending order.

Functions in SQL:

Functions in SQL are the database objects that contains a set of SQL statements to perform a specific task.

A function accepts input parameters, perform actions, and then return the result.

Types of Function:

1. System Defined Function: these are built in functions, ex: rand(), round(), upper(), lower(), count(), sum(), max(), min(), avg() etc.
2. User Defined Function: Once you define a function, you can call it in the same way as the built in functions.

Most used String Functions:

String functions are used to perform an operation on input string and return an output string

- UPPER() - converts the value of a field to uppercase
- LOWER() - converts the value of a field to lowercase
- LENGTH() - returns the length of the value in a text field
- SUBSTRING() - extracts a substring from a string
- NOW() - returns the current system date and time
- FORMAT() - used to set the format of a field
- CONCAT() - adds two or more strings together
- REPLACE() - Replaces all occurrences of a substring within a string, with a new substring
- TRIM() - removes leading and trailing spaces (or other specified characters) from a string

Most Used Aggregate Functions:

Aggregate functions perform a calculation on multiple values and returns a single value.

And Aggregate functions are often used with GROUP BY & SELECT statement.

- COUNT() - Returns number of values.
- SUM() - Returns sum of all values.
- AVG() - Returns Average value.
- MAX() - Returns maximum value.
- MIN() - Returns minimum value.
- ROUND() - Rounds a number to a specified number of decimal places.

Group By Statement:

The Group BY statement group rows that have the same values into summary rows.

It is often used with aggregate functions (COUNT(), MAX(), MIN(), SUM(), AVG()) to group the result set by one or more columns.

Syntax:

```
select mode, sum(amount) as Total
from payment
group by mode
```

Having Clause:

The HAVING clause is used to apply a filter on the result of GROUP BY based on the specified condition.

The WHERE clause places conditions on the selected columns, whereas the HAVING clause places conditions on groups created by the GROUP BY clause.

Syntax:

```
SELECT mode, COUNT(amount) as total
from payment
group by mode
having count(amount) >= 3
order by total desc
```

SQL Order of Execution

- FROM
- WHERE
- GROUP BY
- HAVING
- SELECT
- ORDER BY
- LIMIT

TIMESTAMP:

The Timestamp is used for values that contain both date and time parts.

- Time contains only time, format HH:MM:SS
- Date contains on date, format YYYY-MM-DD
- Year contains on year, format YYYY or YY
- Timestamp contains date and time, format YYYY-MM-DD HH:MM:SS
- Timestamptz contains date, time and time zone

TIMESTAMP functions/operators:

Below are the TIMESTAMP functions and operators in SQL.

- SHOW TIMEZONE
- SELECT NOW()
- SELECT TIMEOFDAY()
- SELECT CURRENT_TIME
- SELECT CURRENT_DATE

EXTRACT FUNCTION:

The EXTRACT() function extracts apart from a given date value.

Syntax: SELECT EXTRACT(MONTH FROM date field) FROM Table

- YEAR
- QUARTER
- MONTH
- WEEK
- DAY
- HOUR
- MINUTE
- DOW – day of week
- DOY – day of year

SQL Join:

- Join means to combine something.

Subhadip Chatterjee

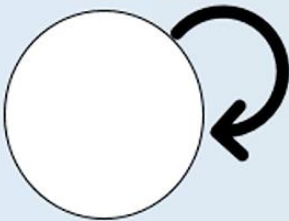
- A join clause is used to combine data from two or more tables, based on a related column between them.

Types of JOINS:

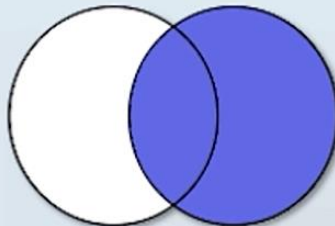
- Inner Join
- Left Join
- Right Join
- Full Join

Joins in MySQL

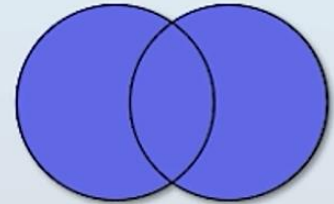
Self Join



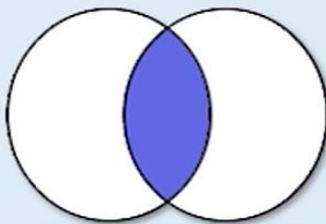
Right Join



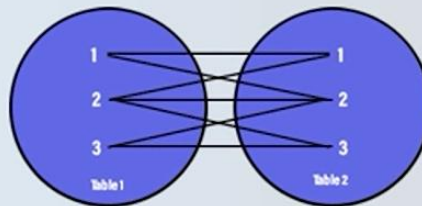
Full Join



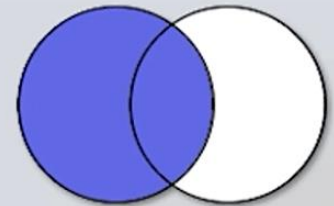
Inner Join



Cross Join



Left Join



1. Inner Join: Returns records that have matching values in both tables.

Syntax:

```
select column_name(s)
from table a
inner join table b
on tablea.col_name = tableb.col_name
```

```
select c.first_name, p.amount, p.mode
from customer c
inner join payment p
on c.customer_id = p.customer_id
where c.first_name = 'Madan';
```

2. Left Join: Returns all records from the left table and the matched records from the right table.

Syntax:

```
select column_name(s)
from table a
left join table b
on tablea.col_name = tableb.col_name
```



```
select * from customer c
left join payment p
on c.customer_id = p.customer_id;
```

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

Syntax: select column_name(s)
from table b
right join table a
on tableb.col_name = tablea.col_name

```
select * from customer c
right join payment p
on c.customer_id = p.customer_id;
```

4. **Full Join/Full Outer Join:** Returns all records when there is a match in either left or right table.

```
select * from customer c
full outer join payment p
on c.customer_id = p.customer_id;
```

Self Join:

- A self join is a regular join in which a table is joined to itself.
- Self Joins are powerful for comparing values in a column of rows with the same table.

Syntax:

```
select column_name(s)
from table AS t1
Join table AS t2
on T1.col_name = T2.col_name
```

```
select * from emp as e1
join emp as e2
on e2.empid=e1.manager_id;
```

```
select e1.empname as employee_name,
e2.empname as manager_name
from emp as e1
join emp as e2
on e2.empid=e1.manager_id;
```

Union: The SQL Union clause/operator is used to combine/concatenate the results of two or more select statements without returning any duplicate rows and keeps unique records.

To use this UNION clause each statement must have

- The same number of columns selected and expressions
- The same data type and
- Have them in the same order.

Syntax:

```
select column_name(s) from table a
UNION
select column_name(s) from table b
```

Union ALL:

In Union ALL everything is same as UNION, it combines/concatenate two or more table but keeps all records, including duplicates.

Syntax:

```
select column_name(s) from table a
UNION ALL
select column_name(s) from table b;
```

Sub Query:

- A Subquery or Inner Query or a Nested Query allows us to create complex query on the output of another query.
- Subquery syntax involved two SELECT statements.

Syntax:

```
Select column_name(s)
From table name
Where column_name operator
```

Exercise of Subquery:

Find the details of customers, whose payment amount is more than the average of total amount paid by all customers.

Let's divide this query into 2 parts.

1. Find the average amount.
2. Filter the customers whose amount > average amount



```
select * from payment
where
amount > (select round(avg(amount)) as avg_amount from payment);
```

customer_id [PK] bigint	amount bigint	mode character varying (50)	payment_date date
1	60	Cash	2020-09-24
3	90	Credit Card	2020-07-07
9	80	Cash	2021-08-25
10	50	Mobile Payment	2021-11-03
11	70	Cash	2022-11-01
12	60	Netbanking	2022-09-11
14	50	Credit Card	2022-05-14
4	50	Debit Card	2020-02-12

Window Function:

- Window Function applies aggregate, ranking and analytic functions over a particular window (set of rows).

- And OVER clause is used with window functions to define that window.

Syntax:

```
Select column_name(s),  
       fun() over ([<Partition by clause>  
                  [<Order by clause>  
                  [<Row or Range clause>]  
From table_name;
```

Select a Function:

- Aggregate Functions
- Ranking Functions
- Analytic Functions

Define a Window:

- Partition BY
- Order BY
- Rows

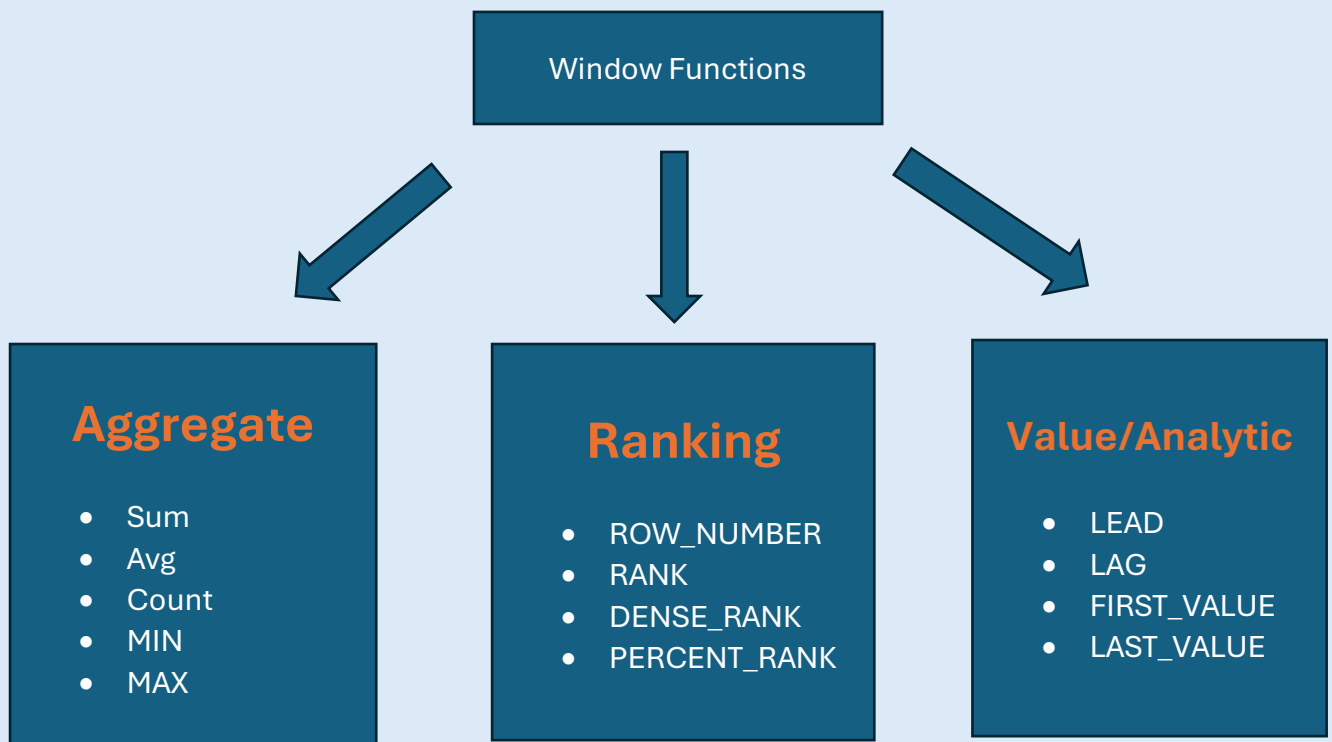
Window Function Terms:

Let's look at some definitions.

- **Window Function:** Applies aggregate, ranking and analytic functions over a particular window, for example: sum, avg, or row_number.
- **Expression:** is the name of the column that we want the window function operated on. This may and may not be necessary depending on what window function is used.
- **OVER:** is just to signify that this is a window function.
- **PARTITION BY:** Divides the rows into partitions so we can specify which rows to use to compute the window function.
- **ORDER BY:** is used so that we can order the rows within each partition. This is optional and does not have to be specified.
- **ROWS:** can be used if we want to further limit the rows within our partition. This is optional and usually not used.

Window Function Types:

There are no official division of the SQL Window functions into categories but high level we can divide into three types.



Window Function Query:

```
select new_id, new_cat,  
sum(new_id) over(order by new_id rows between unbounded preceding and unbounded following) as  
"Total",  
avg(new_id) over(order by new_id rows between unbounded preceding and unbounded following) as  
"Average",  
count(new_id) over(order by new_id rows between unbounded preceding and unbounded following)  
as "Count",  
min(new_id) over(order by new_id rows between unbounded preceding and unbounded following) as  
"Min",  
max(new_id) over(order by new_id rows between unbounded preceding and unbounded following) as  
"Max"  
from test_data;
```

Note: Above we have used “rows between unbounded preceding and unbounded following” which will give a single output based in all input values / partition (if used).

new_id bigint	new_cat character varying (50)	Total numeric	Average numeric	Count bigint	Min bigint	Max bigint
100	Agni	4800	342.8571428571428571	14	100	700
100	Agni	4800	342.8571428571428571	14	100	700
200	Agni	4800	342.8571428571428571	14	100	700
200	Agni	4800	342.8571428571428571	14	100	700
200	Vayu	4800	342.8571428571428571	14	100	700
200	Vayu	4800	342.8571428571428571	14	100	700
300	Vayu	4800	342.8571428571428571	14	100	700
300	Vayu	4800	342.8571428571428571	14	100	700
500	Dharti	4800	342.8571428571428571	14	100	700
500	Dharti	4800	342.8571428571428571	14	100	700
500	Vayu	4800	342.8571428571428571	14	100	700
500	Dharti	4800	342.8571428571428571	14	100	700
500	Vayu	4800	342.8571428571428571	14	100	700
700	Dharti	4800	342.8571428571428571	14	100	700

Rank, Dense Rank, Row_number, Percent Rank query

```
select new_id,
row_number() over(order by new_id) as "Row_Number",
rank() over(order by new_id) as "Rank",
dense_rank() over(order by new_id) as "Dense_Rank",
percent_rank() over(order by new_id) as "Percent_Rank"
from test_data;
```

new_id bigint	Row_Number bigint	Rank bigint	Dense_Rank bigint	Percent_Rank double precision
100	1	1	1	0
100	2	1	1	0
200	3	3	2	0.15384615384615385
200	4	3	2	0.15384615384615385
200	5	3	2	0.15384615384615385
200	6	3	2	0.15384615384615385
300	7	7	3	0.46153846153846156
300	8	7	3	0.46153846153846156
500	9	9	4	0.6153846153846154
500	10	9	4	0.6153846153846154
500	11	9	4	0.6153846153846154
500	12	9	4	0.6153846153846154
500	13	9	4	0.6153846153846154
700	14	14	5	1

Analytic Functions: first_value, last_value, lead, lag

```
select new_id,  
first_value(new_id) over(order by new_id) as "First_Value",  
last_value(new_id) over(order by new_id) as "Last_Value",  
lead(new_id) over(order by new_id) as "Lead",  
lag(new_id) over(order by new_id) as "Lag"  
from test_data;
```

Note: If you just want the single last value from the whole column use “rows between unbounded preceding and unbounded following”.

new_id bigint	First_Value bigint	Last_Value bigint	Lead bigint	Lag bigint
100	100	100	100	[null]
100	100	100	200	100
200	100	200	200	100
200	100	200	200	200
200	100	200	200	200
200	100	200	300	200
300	100	300	300	200
300	100	300	500	300
500	100	500	500	300
500	100	500	500	500
500	100	500	500	500
500	100	500	500	500
500	100	500	700	500
700	100	700	[null]	500

Note: The Lag function is used to get a value from the row that comes before the current row. The Lead function is exactly opposite, it’s used to get the value from the row that comes after the current row.

Offset the lead and lag values by 2 in the output columns.

```
select new_id,  
lead(new_id,2) over(order by new_id) as "Lead_by2",  
lag(new_id,2) over(order by new_id) as "Lag_by2"  
from test_data;
```

new_id bigint	Lead_by2 bigint	Lag_by2 bigint
100	200	[null]
100	200	[null]
200	200	100
200	200	100
200	300	200
200	300	200
300	500	200
300	500	200
500	500	300
500	500	300
500	500	500
500	700	500
500	[null]	500
700	[null]	500

CASE Expression:

- The CASE expression goes through conditions and returns a value when the first condition is met (like if-then-else statement). If no conditions are true, it returns the value in the ELSE clause.
- If there is no ELSE part and no conditions are true, it returns NULL.

General CASE Syntax:

```
CASE  
  WHEN condition1 THEN result1  
  WHEN condition2 THEN result2  
  WHEN conditionN THEN result  
  ELSE other_result  
END;
```

Query:

```
select customer_id, amount,  
case  
    when amount > 50 then 'Expensive Product'  
    when amount = 50 then 'Moderate Product'  
    else 'Inexpensive Product'  
END as Product_Status  
from payment;
```

customer_id [PK] bigint	amount bigint	product_status text
1	60	Expensive Product
2	30	Inexpensive Product
3	90	Expensive Product
5	40	Inexpensive Product
7	10	Inexpensive Product
8	30	Inexpensive Product
9	80	Expensive Product
10	50	Moderate Product
11	70	Expensive Product
12	60	Expensive Product
13	30	Inexpensive Product
14	50	Moderate Product
15	30	Inexpensive Product
4	50	Moderate Product
6	40	Inexpensive Product

CASE Expression Syntax:

```
CASE Expression  
    WHEN value1 THEN result1  
    WHEN value2 THEN result2  
    WHEN valueN THEN resultN  
    Else other_result  
END;
```

Query:

```
select customer_id, amount,  
case amount  
    when 70 then 'Prime Customer'  
    when 50 then 'Plus Customer'  
    else 'Regular Customer'  
end as customer_status  
from payment
```

Subhadip Chatterjee

Here amount is passing in as Expression after case

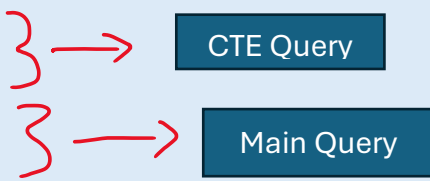
customer_id [PK] bigint	amount bigint	customer_status text
1	60	Regular Customer
2	30	Regular Customer
3	90	Regular Customer
5	40	Regular Customer
7	10	Regular Customer
8	30	Regular Customer
9	80	Regular Customer
10	50	Plus Customer
11	70	Prime Customer
12	60	Regular Customer
13	30	Regular Customer
14	50	Plus Customer
15	30	Regular Customer
4	50	Plus Customer
6	40	Regular Customer

Common Table Expression (CTE):

- A common table expression, or CTE is a temporary named result set created from a simple SELECT statement that can be used in a subsequent SELECT statement.
- We can define CTEs by adding a WITH clause directly before SELECT, INSERT, UPDATE, DELETE, or MERGE statement.
- The with clause can include one or more CTEs separated by commas.

Syntax:

```
WITH my_cte as (  
    SELECT a,b,c  
    From table1)  
Select a,c  
From my_cte
```



- The name of the CTE is my_cte, and the CTE query is SELECT a,b,c from table1.
- The CTE starts with the WITH keyword, after which you specify the name of your CTE, then the content of the query in parentheses.
- The main query comes after the closing parentheses and refers to the CTE.
- Here, the main query (also known as the outer query) is SELECT a,c from my_cte.

Query:

```
with my_cte as (  
    SELECT *,  
    AVG(p.amount) OVER (ORDER BY p.customer_id) AS "Average Price",  
    COUNT(c.address_id) OVER (ORDER BY c.customer_id) AS "Count"  
FROM  
    payment AS p  
INNER JOIN  
    customer AS c  
ON  
    p.customer_id = c.customer_id  
)  
select first_name, last_name from my_cte
```

first_name character varying (50) 🔒	last_name character varying (50) 🔒
Mary	Smith
Patricia	Johnson
Madan	Mohan
Elizabeth	Brown
Maria	Miller
Susan	Wilson
R	Madhav
Dorothy	Taylor
Lisa	Anderson
Nancy	Thomas
Karen	Jackson
Betty	White
Helen	Harris
Barbara	Jones
Jennifer	Davis

Multiple CTEs Query:

```
WITH my_cp AS (  
    SELECT p.*,  
    AVG(p.amount) OVER (ORDER BY p.customer_id) AS "Average Price",  
    COUNT(c.address_id) OVER (ORDER BY c.customer_id) AS "Count"  
FROM  
    payment AS p  
INNER JOIN  
    customer AS c  
ON  
    p.customer_id = c.customer_id  
)
```

Subhadip Chatterjee

```
my_ca AS (  
  SELECT c.*, a.city, cc.country  
  FROM  
    customer AS c  
  INNER JOIN  
    address AS a  
  ON  
    a.address_id = c.address_id  
  INNER JOIN  
    country AS cc  
  ON  
    cc.country_id = a.country_id  
)  
SELECT cp.first_name, cp.last_name, ca.city, ca.country, cp.amount  
FROM  
  my_ca AS ca  
INNER JOIN  
  my_cp AS cp  
ON  
  ca.customer_id = cp.customer_id;
```

```
2) WITH my_cte AS (  
  SELECT mode, MAX(amount) AS highest_price, SUM(amount) AS total_price  
  FROM payment  
  GROUP BY mode  
)  
SELECT payment.*, my.highest_price, my.total_price  
FROM payment  
JOIN my_cte my  
ON payment.mode = my.mode  
ORDER BY payment.mode;
```

customer_id [PK] bigint	amount bigint	mode character varying (50)	payment_date date	highest_price bigint	total_price numeric
11	70	Cash	2022-11-01	80	220
1	60	Cash	2020-09-24	80	220
7	10	Cash	2021-08-25	80	220
9	80	Cash	2021-08-25	80	220
14	50	Credit Card	2022-05-14	90	200
2	30	Credit Card	2020-04-27	90	200
3	90	Credit Card	2020-07-07	90	200
15	30	Credit Card	2022-09-25	90	200
6	40	Debit Card	2021-06-28	50	90
4	50	Debit Card	2020-02-12	50	90
10	50	Mobile Payment	2021-11-03	50	120
8	30	Mobile Payment	2021-06-17	50	120
5	40	Mobile Payment	2020-11-20	50	120
13	30	Netbanking	2022-12-10	60	90
12	60	Netbanking	2022-09-11	60	90

Recursive CTE:

A recursive CTE references itself. It returns the result subset, then it repeatedly (recursively) references itself, and stops when it returns all the results.

A recursive CTE has three elements:

- **Non Recursive Term:** It's a CTE query definition that forms the base result set of the CTE structure.
- **Recursive Term:** One or more CTE query definitions joined with non-recursive term using UNION or UNION ALL operator.
- **Termination Check:** The recursion stops when no rows are returned from the previous iteration.

Syntax:

```
WITH RECURSIVE cte_name AS (
    CTE_query_definition          -- non recursive term -- base query/anchor member
    UNION ALL
    Recursive_query_definition    -- recursive term -- recursive query/recursive member
)
SELECT * FROM cte_name
```

Example:

```
WITH RECURSIVE my_cte AS (
    SELECT 1 AS n          --- base query

    UNION ALL

    SELECT (n+1) FROM my_cte --- recursive query
    WHERE n < 3)          --- condition check
SELECT * FROM my_cte;
```

	n integer	🔒
1		1
2		2
3		3

Recursive CTE Use Case:

- Count up until three.
- Finding Bosses and Hierarchical Level for All Employees.
- Finding Routes Between Cities.
- Finding Ancestors.

Query:

```
with recursive emp_cte as (
  -- Anchor Query
  select e.emp_id, e.emp_name, e.manager_id
  from employees as e
  where emp_id = 7

  union all

  -- Recursive Query
  select e.emp_id, e.emp_name, e.manager_id
  from employees as e
  join emp_cte
  on e.emp_id = emp_cte.manager_id
)
select * from emp_cte;
```

emp_id integer	emp_name character varying	manager_id integer
7	Damodar	6
6	Keshav	5
5	Shiva	4
4	Arjun	3
3	Tom	2
2	Same	1
1	Madhav	[null]

Temp Tables

1. Temporary tables are special types of tables that let you store a temporary result set in memory.
2. You can reuse this temp table multiple times in a single session.
3. Great for storing complex queries that you want to reuse.
4. Temp tables are great for reducing the complexity of queries, storing intermediate result sets, and improving performance.

Creating a new temp table

```
26 • CREATE TEMPORARY TABLE temp_table
27   (first_name varchar(50),
28    last_name varchar(50),
29    favorite_movie varchar(100)
30   );
31
32 • INSERT INTO temp_table
33   VALUES ('Alex','Freberg','Lord of the Rings: The Twin Towers');
```

Creating a new temp table from an existing table.

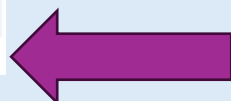
```
37 • CREATE TEMPORARY TABLE temp_table_2
38   SELECT *
39   FROM employees
40   WHERE salary > 50000;
```

Roll Up: Roll Up when added to the group by statement allows you to get a sum of actual output of your aggregations.

Quick way to adding up all the aggregated column all together.

```
select customer_id, sum(tip) as total_tips
from customer_orders
group by customer_id with rollup;
```

customer_id	total_tips
100101	12
100102	11
100103	6
100104	3
100105	1
100106	3
100108	1
100110	1
NULL	38



Total sum of all the total tips column, this can be added by the roll up which is added in the group by

Regular Expressions

- Regular Expressions also called Regex, is a sequence of characters, used to search and locate specific sequences of characters that match a provided pattern.
- Similar to the LIKE statement, but more specific and complex.

Subhadip Chatterjee

```
Select phone
From z_regular_expression
Where phone regexp '[0-9]{3}-[0-9]{3}-[0-9]{4}';
```

```
Select * from customers
Where first_name regexp 'k.{3}n';
```

Regular Expression Methods in Query, read to understand.

```
select *
from customers
where first_name like '%k%';
```

```
select *
from customers
where first_name regexp 'k';
```

```
select first_name,
regexp_replace(first_name,'a','b')
from customers;
```

```
select first_name,
regexp_like(first_name,'a')
from customers;
```

```
select first_name,
regexp_instr(first_name,'a')
from customers;
```

```
select first_name,
regexp_substr(first_name,'char')
from customers;
```

Regular Expression Metacharacters

[-. ^ \$ * + ?]

```
select *
from customers
where first_name regexp '[a-c]';
---- searching for name contains a,b,c letters
```

Query to change a column name of a table

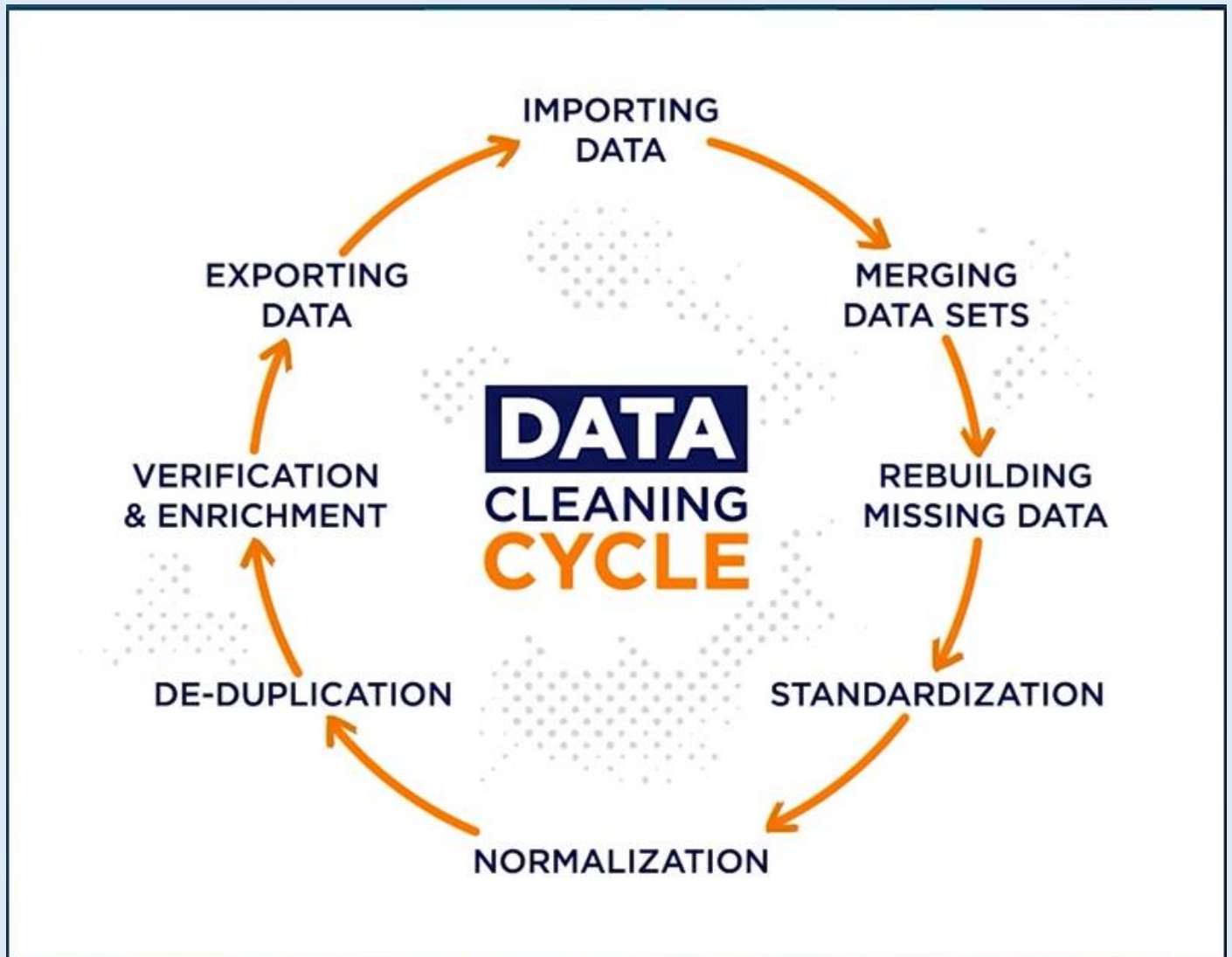
```
alter table product_suggestions rename column `ï»¿product_suggestion_id` to
`product_suggestion_id`;
```

Data Cleaning:

Subhadip Chatterjee

Data cleaning is the process of identifying and correcting or removing invalid, incorrect, or incomplete data from a dataset.

There are several techniques including removing duplicated data, normalization, standardization, populating values, and more.



Delete duplicates value from the database using SubQuery

```
delete from customer_sweepstakes_staging
where sweepstake_id in (
select sweepstake_id
from (select sweepstake_id,
row_number() over(partition by customer_id order by customer_id) as row_num
from bakery.customer_sweepstakes_staging) as table_row
where row_num > 1);
```

Query to clean the phone numbers, for a small base

```
select phone, regexp_replace(phone, '([()-/+]', '')
from customer_sweepstakes_staging;
```

Subhadip Chatterjee


```
update customer_sweepstakes_staging  
set phone = regexp_replace(phone, '[-/+]', '');
```

Stored Procedure

1. Stored procedures are a way to save SQL code that can be reused over and over again.
2. You use a stored procedure by “Calling” it which executes the code saved within the stored procedure.
3. Great for storing complex SQL code, simplifying repetitive code, and enhanced performance.

```
52      DELIMITER $$  
53 •    CREATE PROCEDURE large_order_totals2()  
54      BEGIN  
55          SELECT *  
56          FROM bakery.customer_orders  
57          WHERE order_total > 10;  
58          SELECT *  
59          FROM bakery.customer_orders  
60          WHERE order_total > 5;  
61      END $$  
62  
63      DELIMITER ;  
64  
65 •    CALL large_order_totals2();
```

Parameters in Stored Procedures

1. Parameters are variables that are passed as an input or output to a stored procedure.
2. They allow the stored procedure to accept input values from the calling program, to perform certain actions based on those input values.
3. IN parameters are used to pass input values to the stored procedure.
4. OUT parameters are used to return output values from the stored procedure.

```

71      DELIMITER $$
72 •    CREATE PROCEDURE orders_by_product(p_product_id int)
73      BEGIN
74          SELECT *
75          FROM bakery.customer_orders
76          -- and we change it down here too.
77          WHERE product_id = p_product_id;
78      END $$
79
80 •    CALL orders_by_product(1001);

```

Triggers

1. Triggers are a named database object that is associated with a table and automatically executes in response to certain events on that table.
2. Trigger can be insert, update or delete operation.
3. Triggers can be used for automation, auditing, replication, data integrity and more.

```

87      DELIMITER $$
88 •    CREATE TRIGGER update_invoice_with_payments
89          AFTER INSERT ON bakery2.client_payments
90          FOR EACH ROW
91      BEGIN
92          UPDATE client_invoices
93          SET total_paid = total_paid + NEW.amount_paid
94          WHERE invoice_id = NEW.invoice_id;
95      END $$

```

Events

1. An event is a scheduled task that runs at a specific time or interval.
2. Events can be used to perform a variety of tasks, such as generating reports, updating data, daily maintenance, or sending notifications.

```
102     DELIMITER $$
103 •   CREATE EVENT delete_old_customer_orders
104     ON SCHEDULE EVERY 30 SECOND
105     DO BEGIN
106         DELETE FROM bakery.customer_orders
107         WHERE order_date < NOW() - INTERVAL 5 YEAR;
108     END $$
109
110     DELIMITER ;
```

Indexing

1. Indexes in MySQL are data structures that are used to improve the speed of the queries on database tables.
2. MySQL can use the index to quickly find the rows that match the query criteria, rather than scanning the entire table.
3. Indexes can significantly improve the performance of queries on large tables.

```
115
116 •   CREATE INDEX idx_state_ALand
117     ON bakery4.ushouseholdincome (State_Name(10), ALand);
118
```

Best Practices of Indexes

1. Composite Indexes are good, in general composite indexes are perform much better especially if you are filtering in more than one column
2. Order Matters
3. Avoiding using select *
4. Looking at your where clause
5. Full table scans vs indexes