What is SQL?
SQL stands for Structured Query Language
SQL lets you access and manipulate databases

SQL Aggregate Functions

An aggregate function is a function that performs a calculation on a set of values, and returns a single value.

Aggregate functions are often used with the GROUP BY clause of the SELECT statement. The GROUP BY clause splits the result-set into groups of values and the aggregate function can be used to return a single value for each group.

The most commonly used SQL aggregate functions are:

- **MIN() -** returns the smallest value within the selected column
- **MAX() -** returns the largest value within the selected column
- **COUNT() -** returns the number of rows in a set
- **SUM() -** returns the total sum of a numerical column
- **AVG() -** returns the average value of a numerical column

The SQL GROUP BY Statement

The GROUP BY statement groups rows that have the same values into summary rows, like "find the number of customers in each country".

The **GROUP BY statement is often used with aggregate functions (COUNT(), MAX(), MIN(), SUM(), AVG())** to group the result-set by one or more columns.

The SQL HAVING Clause

The HAVING clause was added to SQL because the WHERE keyword cannot be used with aggregate functions.
Note :-- We can use **HAVING Clause** with **GROUP BY Statement**

The SQL Indexes

SQL Indexes are used to s**peed up the process of data retrieval.** They hold pointers that refer to the data stored in a database, which makes it easier to locate the required data records in a database table.
While an index speeds up the performance of data retrieval queries (SELECT statement), it slows down the performance of data input queries **(UPDATE and INSERT statements).**

When should indexes be avoided?

Indexes should not be **used on small tables.**

They should not be used on tables that have frequent, **large batch updates or insert operations.**

Indexes should not be used on columns that contain a high number of NULL values.
Columns that are frequently manipulated should not be indexed.

What is the difference between the WHERE and HAVING clauses?

When GROUP BY is not used, the WHERE and HAVING clauses are essentially equivalent. However, when GROUP BY is used: The WHERE clause is used to filter records from a result. The filtering occurs before any groupings are made.

The HAVING clause is used to filter values from a group (i.e., to check conditions after aggregation into groups has been performed).

What does UNION do? What is the difference between UNION and UNION ALL

UNION will omit /skip duplicate records whereas **UNION ALL will include duplicate records.** It is important to note that the **performance of UNION ALL will typically be better than UNION**

Explain each of the ACID properties that collectively guarantee that database transactions are processed reliably.

ACID (Atomicity, Consistency, Isolation, Durability) is a set of properties that guarantee that database transactions are processed reliably. They are defined as follows:

Atomicity :-- Atomicity requires that each transaction be **"all or nothing"**: if **one part of the transaction fails, the entire transaction fails, and the database state is left unchanged.** An atomic system must guarantee atomicity in each and every situation, including power failures, errors, and crashes.

Consistency :-- The consistency property ensures that any transaction will bring the database from one valid state to another. **Any data written to the database must be valid according to all defined rules, including constraints, cascades, triggers, and any combination thereof.**

Isolation :-- The isolation property ensures that the concurrent execution of transactions results in a system state that would be obtained if transactions were executed serially, i.e., one after the other. **Providing isolation is the main goal of concurrency control. Depending on concurrency control method** (i.e. if it uses strict - as opposed to relaxed - serializability), **the effects of an incomplete transaction might not even be visible to another transaction.**

Durability :-- Durability means that once a transaction has been committed, it will remain so, even in the event of power loss, crashes, or errors. In a relational database, for instance, once a group of SQL statements execute, the results need to be stored permanently (even if the database crashes immediately thereafter). To defend against power loss, transactions (or their effects) must be recorded in a non-volatile memory.

What is INDEX :--> SQL Indexes are **used in Relational databases to quickly reterieve data.**

They are two types of Indexes **(Cluster Index & Non-cluster Index)**

1. Cluster Index :-- B-Tree Clustered index arrange the rows physically in the memory in sorterd order and **when Primary key is created automatically the Clustered index is created. An advantage of clustered index is that searching for a range of values will be fast.** A clustered index is internally maintained using B-Tree data structure leaf node of the b tree of clustered indexes that will contain the table data, **you can create only one clustered index for a table.**

How to create Cluster Index :--
Answer :-- Create CLUSTERED index my_cluster_index on Employee(emp_id);

2. Non-cluster Index :-- you can create multiple non-cluster index for table.

Whena a Unique key is created automatically, a non-clustered index is created.

A Non-clustered index will not arrange the rows physically in the memory in sorterd order.

An advantage of Non-clustered index is searching for the values that are in a range.

you can create maximum 999 non-clustered index on a table.

A Non-clustered index is also maintained by B-Tree data structure.but leaf nodes of a B-Tree or non-clustered index contain the pointers to the pages that contain the table data and not the table data directly.

Non-cluster Index is slower than cluster index.

How to create Non-Cluster Index :--
Answer :-- Create NONCLUSTERED index my_NONcluster_index on Employee(emp_id);

Difference between Clustered Index & Non Clustered Index

Clustered Index :--
1. This will arrange the rows physically in the memory in sorted order
2. This will fast in searching for the range of values
3. one cluster Index for table
4. Leaf node of the 3 tier of Clustered index contains table data

Non Clustered Index :--
1. This will not arrange the rows physically in the memory in sorted order.
2. This will be fast in searching for the values that are not in the range.
3. you can create 999 non-clustered index for tables.
4. Leaf nodes of b-tree of non-clustered index contain pointers to get the

contained pointers that contain two table data, and not the table data directly.

Composite Index :-- Composite Index is also called multi column index and it is **mostly used when you want to search data base on multile condition.**

Example :-- SELECT ID, Name, District, Population FROM World.city WHERE CountryCode='DEU' AND Population > 1000000;

Explanation :-- If you observe there are two condition **(CountryCode and Population).**
so if you create the one single index for this multi columns **(CountryCode and Population)** , that is called Composite Index and gives good performance for searching data.

How to add Composite Index :--
ALTER TABLE World.city ADD INDEX (CountryCode and Population);

How to add Single Index :--
ALTER TABLE World.city ADD INDEX (CountryCode);

Redundant Index :-- Redundant Index is duplicate key with another column for best performance.

For example :--

```
ALTER TABLE EMPLOYEE ADD INDEX (empId );
ALTER TABLE EMPLOYEE ADD INDEX (empId, dob);
ALTER TABLE EMPLOYEE ADD INDEX (empId, dob, mobile);
```

What is partitioning :-- The process of dividing data in a database into logical parts is called partitioning.

partitioning is a crucial element to keep the performance for databases with very large tables. When table size increase , then performance get slow and we can't increase table size more than 32GB. so partitioning is helpful to divide a large table into small small tables and it helps query processor to scan much smaller tables and indexes to find the needed data. When you get longer response time then we can use partitioning. **we can do partitioning based on those table columns which mostly used in conditionals clause.**

suppose we want to find records based on employee joining date, and if we are executing below query.
for example :-- Select * from employee where emp_join_date=**'01-01-2024'**

so we can do partitioning based on (emp_join_date) column.

There are two types of partitioning
1. Declarative partitioning :-- This is the latest and easiest way of partitioning.

2. Inheratince partitioning :-- This is the older and very hard way of partitioning.

Note : we normally use Declarative partitioning

there are four methods to partition the tables according to different use cases **(1. Light, 2. Range, 3. Hash, 4. Composite)** partition.

Range partitioning :- it is most common partitioning used for better performance. In RANGE partitioning you can partition values within a given range and Ranges should be contiguous but not overlapping.

How to create PARTITION BY RANGE

Create Database :-- create database test

Create Table with partition :--

```
create table customers (id integer , name text, dob date) PARTITION BY RANGE( YEAR(joined) ) (
    PARTITION emp_partition_p0 VALUES LESS THAN (1960),
    PARTITION emp_partition_p1 VALUES LESS THAN (1970),
    PARTITION emp_partition_p2 VALUES LESS THAN (1980),
    PARTITION emp_partition_p3 VALUES LESS THAN (1990),
    PARTITION emp_partition_p4 VALUES LESS THAN MAXVALUE
);
```

or second way

Create Table with partition :--

```
create table customers (id integer , name text, dob date) PARTITION BY RANGE( YEAR(joined) );
Create Actual Partition :-- create table emp_partition_p0 of customers for values
 from (MINVALUE) to (1960);
Create Actual Partition :-- create table emp_partition_p1 of customers for values from (1960)
 to (1970);
Create Actual Partition :-- create table emp_partition_p2 of customers for values from (1970)
 to (1980);
Create Actual Partition :-- create table emp_partition_p3 of customers for values from (1980)
 to (1990);
Create Actual Partition :-- create table emp_partition_p3 of customers for values from (1990)
 to (MAXVALUE);
```

How to see partition details of tables :-- execute below query to see partition details of tables

select tableoid::regclass, * from customers;

LIST COLUMNS partitioning :-- it also good partitioning and commonly used for better performance.

Example , suppose you have district and city wise large data and you perform some quereies base on **district and cities columns** then **you can create below table and partition**

```
CREATE TABLE customers (
    fname VARCHAR(25),
    district varchar(15)
    city VARCHAR(15)
)
PARTITION BY LIST COLUMNS(city) (
    PARTITION pDistrict_city_1 VALUES IN('chakan', 'daund', 'saswad'),
    PARTITION pDistrict_city_2 VALUES IN('thane', 'kalyan', 'dadar'),
    PARTITION pDistrict_city_3 VALUES IN('parola', 'prakasha', 'chopala','desaipura')
);

insert into customers (fname,district,city)
 ('mayuresh', 'nandurbar','desaipura')
,('jayesh', 'nandurbar','parola')
,('ashu', 'pune','chakan')
,('yogesh', 'pune','saswad')
,('sameer', 'mumbai','kalyan')
,('yogesh', 'mumbai','thane');

Select * from customers where district='pune' and city='saswad';
```

composite partitioning :- it is most common partitioning used for better performance. **Subpartitioning—also known as composite partitioning.**
You can partition table combining *RANGE* and *HASH* for better results

Important points of composite partitioning are

- Each partition must have the same number of subpartitions.
- Each SUBPARTITION clause must include (at a minimum) a name for the subpartition. Otherwise, you may set any desired option for the subpartition or allow it to assume its default setting for that option.
- Subpartition names must be unique across the entire table.

```
CREATE TABLE purchase (id INT, item VARCHAR(30), purchase_date DATE)
    PARTITION BY RANGE( YEAR(purchase_date) )
    SUBPARTITION BY HASH( TO_DAYS(purchase_date) )
    SUBPARTITIONS 2 (
        PARTITION p0 VALUES LESS THAN (2000),
        PARTITION p1 VALUES LESS THAN (2010),
        PARTITION p2 VALUES LESS THAN MAXVALUE
    );
```

Sharding :-- Sharding is a database server partitioning technique that can be used to **distribute data across different servers in order to improve performance and scalability. Sharding is an alternative approach for scaling databases, which divides the database into smaller pieces called shards.** Each shards can then be hosted on a separate server, which helps distribute the load among them.

Database sharding is the process of storing a large database across multiple machines. A single machine, or database server, can store and process only a limited amount of data. Database sharding overcomes this limitation by splitting data into smaller chunks, called shards, and storing them across several database servers. All database servers usually have the same underlying technologies, and they work together to store and process large volumes of data.

SQL JOINS :--

Sql have **(full join, left join, right join, inner join , cross join)** types.

Full Join :-- Full join return rows when **there are matching rows in any one of the tables.** This means it returns all the rows from the left-hand side table and all the rows from the right-hand side table.

Returns all rows for which there is a match in EITHER of the tables. Conceptually, a FULL JOIN combines the effect of applying both a LEFT JOIN and a

RIGHT JOIN. **its result set is equivalent to performing a UNION of the results of left and right outer queries.**

Left Join :-- Left join returns all rows from the Left table and those which are shared between the tables. If there are no matching rows in the right table, it will still return all the rows from the left table. **Returns all rows from the left table, and the matched rows from the right table.**

the results will contain all records from the left table, even if the JOIN condition doesn’t find any matching records in the right table. This means that if the ON clause doesn’t match any records in the right table, the JOIN will still return a row in the result for that record in the left table, but with NULL in each column from the right table.

Right Join :-- Right join returns all rows from the right table and those which are shared between the tables. If there are no matching rows in the left table, it will still return all the rows from the right table. **Returns all rows from the right table, and the matched rows from the left table.**

the results will contain all records from the right table, even if the JOIN condition doesn’t find any matching records in the left table. This means that if the ON clause doesn’t match any records in the left table, the JOIN will still return a row in the result for that record in the right table, but with NULL in each column from the left table.

Inner Join :-- Rows are returned when **there is at least one match of rows between the tables.** Returns all rows for which there is at least one match in BOTH tables. This is the default type of join if no specific JOIN type is specified.

CROSS JOIN :-- An SQL CROSS JOIN is used when you need to find out all the possibilities of combining two tables, where the result set includes every row from each contributing table. The CROSS JOIN clause returns the Cartesian product of rows from the joined tables. Returns all records where each row from the first table is combined with each row from the second table.

The CROSS JOIN query in SQL is used to generate all combinations of records in two tables. For example, you have two columns: size and color, and you need a result set to display all the possible paired combinations of those—that's where the CROSS JOIN will come in handy.

What is Hibernate N+1 Problem

Hibernate N+1 problem occurs when you use FetchType.LAZY for your entity associations. If you perform a query to select n-entities and if you try to call any access method of your entity's lazy association, Hibernate will perform n-additional queries to load lazily fetched objects.

“N+1 problem” refers to a performance issue that can occur when retrieving entities and their associated relationships from a database.

In one-to-many relationship , we faces the **“N+1 problem” .**the N+1 problem can happen if the first query populates the primary object and the second query populates all the child objects for each of the unique primary objects returned.

The cause of the N+1 problem is the eager loading nature of Hibernate. Hibernate will always load the **@ManyToOne children by default (FetchType.EAGER).** So it will load each post object if it already hasn’t loaded it from the database.

NOTE :-- using EntityGraph or using select query with fetch join, we can solve **“N+1 problem”.**

Practical

```
create database practice;

use practice;

create table emp(eid int(11) NOT NULL AUTO_INCREMENT, nme varchar(15), salary int, PRIMARY KEY (`eid`));

insert into emp (nme,salary) values('abc',1000),('abc',2000),('abc',13000),('abc',100),('abc',5000);
```

Question :-- Find 2 minimum and 2 maximum SQL

```
 Select e.* from (
 (SELECT e2.salary  FROM emp e2 order by salary asc limit 2 ) union
  (SELECT e2.salary  FROM emp e2 order by salary desc limit 2)) e ;
```

Question :-- Find 1 minimum and 1 maximum SQL

```
SELECT e.nme, e.salary
FROM emp e
WHERE e.salary = (SELECT MAX(e2.salary) FROM emp e2) OR
      e.salary = (SELECT MIN(e2.salary) FROM emp e2);
```

Question :-- How do you get the Nth-highest salary from the Employee table without a subquery or CTE?

```
SELECT salary from emp order by salary DESC LIMIT 2,1
```

Question :-- Find Maximum and Minimum salary by Department in SQL Server

```
CREATE TABLE table1(
eid int(11) NOT NULL AUTO_INCREMENT,
nme varchar(15), salary int, depId int, PRIMARY KEY (`eid`)) ;

INSERT table1 VALUES (1, 'test1', 1200, 1), (2, 'test2', 1500, 1),(3, 'test3', 1300, 2),
(4, 'test4', 2000, 3),(5, 'test5', 1000, 2),(6, 'test6', 1300, 2);

select depid, max(salary) as MaxSalary, min(Salary) as MinSalary from table1 group by
DepId;
```

Question :-- You have two tables, “employees” and “departments,” and you need to retrieve the names of employees along with their department names.

```
SELECT e.first_name, e.last_name, d.department_name FROM employees e INNER JOIN
departments d ON e.department_id = d.department_id;
```