Week - 7 Basics of SQL - II

Module 1 : Querying in MYSQL

Introduction:-

In the previous sessions, you learnt what databases are, what they are composed of, and how to create a database along with its tables, constraints and values. In this session, we will dive into the core of SQL and you will learn all about the statements that are used for querying data. You will also develop the thought process that is required to extract the data needed for analysis and to drive business decisions.

In this session

You will learn how to write queries using the **select**, **from**, **where**, **group by**, **having**, **order by** and **limit** statements. A query may generally contain all or some of these keywords. You will also get an understanding of the need for the various types of operators and functions, which make it easy to address seemingly complex data requirements.

Eventually, Professor Ramanathan will talk about subqueries, CTEs (Common Table Expressions) and views, which help with solving even more complex problem statements.

SQL Statements and Operators:-

Consider a data set of the Indian census containing at least a billion rows, with data on every individual's height, weight, monthly income, and hundreds of more such attributes. How do you even start with the analysis of such an extensive data set to capture any trends in the data? The SQL clauses that you will learn in this segment and throughout this session will help you with this.

So, in this segment, you will learn about the basic constructs of the SELECT query. You will specifically learn about the following:

- 'Select' Clause
- 'From' Clause
- 'Where' Clause
- Operators: Arithmetic, Comparison and Logical

Now, it is important that you go through the next videos carefully, as they will form the foundation for the entire SQL course.

Before proceeding to the operations, download the following SQL script and set up the schema on which all the operations are performed.

Also, please download the attached script file, which contains the questions

that are solved in the upcoming videos.

you learnt about a few basic queries using the **select** and **from** statements. After watching the next couple of videos, you will be able to filter data based on certain given criteria and group it into categories, if required. You will also learn how to use operators in queries. In the next video, you will have hands-on using the basic SQL commands.

Operators can be used for various functions: pattern matching, returning a subset of the data based on its properties or comparing different values to get a sense of the distribution of the data. Pattern matching is an important concept in text-based processing. As you learnt, in SQL, certain characters are reserved as wildcards, which can match any number of preceding or trailing characters.

Additional Resources:

1. SQL Operators is an excellent resource where you can have a quick glance at the various operators that are supported in SQL.

In the next segment, you will learn about the aggragate functions.

Aggregate Functions:-

As you go about examining and analysing data of varying magnitudes, you will quickly realise the need for grouping similar types of values together and looking at them as one group. For example, consider a table that has data, which consists of the marks scored by students in their 12th board exams. While you would want to know how the students performed in all the subjects put together, it is equally important to see how they performed in each subject. You can derive even further insights if you group these students by state. Hence, it is imperative that you learn the usage of aggregate functions in your queries. So, let's go ahead and learn about it.

Points to ponder:—>**(difference between *count(*) and count(Name_of_specific_column))***
count(*) and count(Name_of_specific_column)- Remember count(*) will count the no of rows including NULL values while count(Name_of_specific_column) will only count the number of rows in that particular column while excluding the NULL values in that column, which is mentioned as parameter of count() function.

So, in this segment, you learnt how to use the **group by** clause. In brief, we use 'group by' when we need to find the aggregate values of a column C1 'grouped by' a certain column C2.

count() is just one of many aggregate functions that are available in the MySQL

Workbench. Feel free to explore other functions, such as **min()**, **max()** and **avg()**. Try using them in queries and examine the results that you obtain. You will realise that aggregate functions play a crucial role in determining outliers in the data and analysing any existing trends in it.

please keep this in mind whenever you write any query because not all versions of the MySQL Workbench in all operating systems allow non-aggregate columns in the 'select' clause whenever you use a 'group by' statement.

Explanation of above line in details using query as an example:-

select Customer_Segment,count(Customer_Name) as customer_from_Bihar from cust_dimen where State = 'Bihar' group by Customer_Segment;

From the above query it is evident that group by is done on column name "Customer_Segment", so your select query should not have any other column name except that which is mentioned as in group by clause and except aggregate function. Finally, any other column name which is non-aggregate columns in the 'select' clause whenever you use a 'group by' statement may cause issue in query execution.

you have learnt about the various aggregate functions in this segment. In the next segment, you will learn about another operation i.e. Ordering.

Ordering:-

Quite often, you would want to display the retrieved records in a particular order, for example, in increasing order of income, joining date or alphabetical order. This is commonly useful when you are making a report or presenting the data to someone else. Let's try to understand this.

The Having Clause:-

You have already learnt how to filter individual values based on a given condition. But how do you do this for grouped values? Suppose your manager asks you to count all the employees whose salaries are more than the average salary in their respective departments. Now, intuitively, you know that two aggregate functions would be used here: **count()** and **avg().** You decide to apply the 'where' condition on the average salary of the department, but to your surprise, the query fails. This is exactly what the '**having**' clause is for. So, in the upcoming video, you will learn how the 'having' clause is used in a query.

To summarise, you are now aware of all the SQL clauses: 'select', 'from', 'where', 'group by', 'order by' and 'having'.

The 'having' clause is typically used when you have to apply a filter condition on an 'aggregated value'. This is because the 'where' clause is applied before aggregation takes place and, thus, it is not useful when you want to apply a filter on an aggregated value.

In other words, the 'having' clause is equivalent to a 'where' clause after the 'group by' clause has been executed but before the 'select' clause is executed. You can read this StackOverflow answer to understand the 'having' clause better.

It is important to not get confused between the 'having' and 'where' clauses. For example, if you want to display the list of all the employees whose salary >= 30,000, then you can use the 'where' clause, since there is no aggregation taking place in this query. But if you want to display the list of all the employees whose salary <= the average salary, where avg() is the aggregation function, then you will have to use the 'having' clause.

String and Date–Time Functions:-

you will learn all about string functions. They are used for manipulating string data and make it more understandable for analysis. For example, given two strings 'robertdowneyjr' and 'Robert Downey Jr.', which one is more readable? Of course, the latter one.

Similar to string functions, SQL also has functions to manipulate the date and time values in a database. These are quite useful for analysing time-series data. For example, consider a database pertaining to the students of the university from which you graduated. Now, suppose you have data on the total number of students who used the library every day. You can use date functions to determine whether the library was busier on any particular day of the week as compared with the other days, or maybe a particular month (for example, exam time).

As you must have realised, functions are extremely important in SQL. They help us extract appropriate information from a single value or a group of values, and even present the information in a better format. Data in a more readable format can make the work of an analyst much easier in the long run.

Regular Expressions:-

You have already learnt how to perform pattern matching using the **like** operator along with **wildcards**. Now, the 'like' operator and the wildcards may fall short for some advanced use cases. One such example that you can consider is email validation. Given a string, how can you determine whether an email is valid or not? This may not even be possible to achieve using 'like'. In fact, this is a slightly complicated requirement to meet even for regular expressions. It definitely makes the work of an analyst much easier,

though.

Nested Queries:-

By now, you know that a database is a collection of multiple related tables. Now, while generating insights from data, you may need to refer to these multiple tables in a query. There are two ways to deal with such types of queries:

1. **Joins**
2. **Nested queries/Subqueries**

you will learn about nested queries/subqueries.Subqueries are a generic form of writing queries wherein you do not need to specify some value explicitly (either minimum or maximum, or some other value). If any update is made to a table, then the subquery would still return the required output as it is independent of any particular value in the table. Now, let's move on to the next and learn more about nested queries.

To summarise, in this segment, you learnt about nested queries, which are typically used when you have to select columns from one table based on the filter conditions from another table. In such cases, you write a subquery inside the 'where' clause, instead of a specific value.

CTEs:-

Imagine you are working as an analyst at a large bank (if you aren't already, that is). Let's say you are required to find out your top 10 clients out of thousands. Of these clients, your company wants to give some cash handouts to the one that has the least turnover in the current year. You can definitely use a simple query to achieve this [by using the min() function and the 'where' clause], but what if you want to derive multiple insights from the data of your top 10 clients? In this case, it is better to use a **Common Table Expression (CTE)**, so that you can get the data of only those clients and perform all of your analysis on a subset of the entire data.

A CTE is used to create a temporary table, which is smaller than the existing table. This smaller table cannot be individually queried, that returns an error. The CTE has to be used as part of the main query.

Views:-

In the previous segment, you learnt that CTEs are temporary tables that you can use in order to query data from a part of a huge data set. Now, what if you want to use the same subset of data for multiple queries? In such a case, instead of writing a CTE over and over again, you can store the required data on which you want to perform your analysis. This is what views are used for. You

will understand the importance of using views.

Note that it is not necessary that views would always be preferred to CTEs. When you know for sure that you need to subset data from a table only once, you should use a CTE to avoid extra memory and space usage.

Summary:-

In this session, you understood the crux of SQL queries. These are the basic concepts of SQL, which will be tested rigorously in your interviews. Hence, we urge you to practise all you can and develop a deep understanding of the various SQL statements.

Now, you learnt that a complete query (in a sense) has the following basic outline:

```
select (attributes)
from (table)
where (filter_condition)
group by (attributes_to_be_grouped_upon)
having (filter_condition_on_grouped_values)
order by (values)
limit (no_of_values_to_display);
```

You also learnt about the following important concepts related to basic SQL querying:

- Relational, arithmetic and logical operators
- Aggregate functions
- Regular expressions
- Nested queries
- Common Table Expressions
- Views
- Advantages of views over CTEs

In the next session, you will finally learn about joins and set operations.

Module 2 : Joins and Set Operations

Introduction:-

Previously, you learnt about nested queries, which are used to retrieve data from multiple tables. However, as you must have noticed, a nested query refers to only one table at a time. What if you want to refer to multiple tables in a single query? In such a case, you can use joins. Joins are a handy tool for outputting data from multiple tables in a single table.

You will also understand the set theory and some types of set operations. After this session, you will be able to perform set operations on your data using some SQL keywords. For example, you will be able to get the output of two separate queries in a single query using the 'union' operator.

In this session

You will learn the basics of set theory. The basic set operations are as follows:

- **Union**
- **Intersection**
- **Difference**

You will also learn about various types of **joins**, such as follows:

- **Inner join**
- **Left join**
- **Right join**

Set Theory:-

There are many set operations in practice, but in this segment, you will learn about **union**, **intersection** and **difference**. A set is a collection of distinct values. It is denoted by comma-separated values enclosed by curly braces.

Consider two sets A and B containing even numbers and prime numbers, respectively. For ease of understanding, let's consider only values less than 10. Therefore, the sets will be A = { 2, 4, 6, 8} and B = { 2, 3, 5, 7 }.

The union of A and B (denoted by **A** ∪ **B**) will contain all the values that are present in either A or B. Therefore, A ∪ B = { 2, 3, 4, 5, 6, 7, 8 }.

The intersection of A and B (also denoted by **A** ∩ **B**) will contain all the values that are present in both A and B. Therefore, A ∩ B = { 2 }.

The set difference of A and B (also denoted by **A - B**) will contain all the values that are present in A but not in B. Therefore, A - B = { 4, 6, 8 }.

Types of Joins:-

Consider a database that contains data about a music festival that you are organising. Now, while designing the database, you may have tables pertaining to the following:

- Data about the artists who would be performing at the festival
- Details of the stages on which the artists would perform
- Timings for each performance on each stage
- Details of organisers and other details of security guards, volunteers, photographers, etc.

- Details of attendees, including name, age, address, ticket category, etc.

As you can see, the design of such a database can become quite complicated. How would you analyse the data of the artists and attendees together? This information may be required to calculate the feasibility of the event in terms of cost. This is where joins can be used to combine multiple tables and make the analysis of such data easier for you.

there are mainly two types of joins: inner joins and outer joins. While an inner join searches tables for matching or overlapping data, an outer join also returns rows for which there is no match between the tables.

Types of Joins: A Demonstration:-

Now that you have a theoretical understanding of joins in MySQL, next, Professor Ramanathan will walk you through various examples where you would need to use inner joins. The use case may be as simple as just returning some data from two tables. It may even be as advanced as analysing multiple tables together in a single query and identifying any trends that emerge.

The general syntax of a query using an inner join statement is as follows:

select <column_1>, <column_2>
from table_1 a
inner join table_2 b
on a.<common_column> = b.<common_column>;

Earlier in this session, you learnt about inner joins and writing join queries. You must have noticed that only two tables were referred to in the join. But what if you have a complicated problem statement that needs referencing three, four, five or more tables? In such situations, you can write join queries using **multi-joins**.

In brief, you can use multi-joins to join multiple tables using common attributes between pairs of tables. This is possible because the result of a join is also a table, which you can join further to another table (with a common attribute).

ERDs can be useful for understanding the links between tables, which, in turn, can be quite helpful in writing multi-way join queries.

Outer Joins: A Demonstration:-

Now that you know when to use an outer join in a query, Professor Ramanathan will take you through some example queries using left and right joins. Using a left join will return all values from the left table and matching values from the right table. If there is no match for any value, the right table will return null for

that value.

To summarise, an outer join is used when you want to display the rows in one table even if they do not have a corresponding entry in the other table. Also, an outer join is of two types: left outer join and right outer join. It does not really matter which table you treat as left or right, i.e., you can choose the table that you are more comfortable with.

Views with Joins:-

Imagine you have two tables Menu and Ingredients. The Menu table contains data about different food items and their prices, while the Ingredients table contains data about the ingredients as well as their quantities used in each food item. It turns out that you need to analyse food products that cost above ? 200 repeatedly to understand if there is any way to drive down their costs. Is there a way to avoid writing join statements every single time for each query? Well, it turns out there is. Watch the video given below and find out how you can do this.

Views with joins are especially useful if you need to store data from multiple columns in a single place for ready reference. You can even use subqueries and CTEs in views. Feel free to explore such use cases and try writing queries to get the exact results for further analysis.

By now, you have mastered the most important aspect of writing SQL queries, joins. Do keep practising different queries involving multiple joins, as joins form a majority of the questions that are generally asked in interviews.

Set Operations with SQL:-

Can you recall the types of set operations that you came across at the beginning of this session? They were union, intersection and difference. In MySQL, these operations are achieved using the keywords 'union' and 'union all'. Unfortunately, MySQL Workbench does not support 'intersect' and 'minus' keywords; these operations are performed with a combination of other SQL clauses that you are familiar with by now. You may find the two links provided below helpful for reading more about these two operations.

From the next video, you will get an understanding of the difference between using the 'union' and 'union all' operators and learn how they are implemented in queries.

Always make sure that the tables or the results of your queries are union-compatible before you perform a union operation on them. Two tables are union-compatible if:

1. They have the same number of attributes, and
2. The attribute types are compatible, i.e., the corresponding attributes have the same data type.

Additional resources

1. MySQL INTERSECT is a useful link to help you understand how you can emulate intersect operations in MySQL.
2. Similarly, you can implement minus operations in MySQL. You can go to this MySQL MINUS to know more about it.

Summary:-

You learnt the following topics in this session:

- Some basic set operations such as union, intersection and difference
- Types of joins: inner joins, left joins and right joins
- Writing simple and complex queries involving multiple joins and views
- Using 'union' and 'union all' in queries

Armed with all this knowledge and experience of writing numerous queries, you are now primed to dive into some of the more advanced features in MySQL. In the next module, you will learn about many of them in great detail.