Advanced SQL and Best Practices

Module 1 : Window Function

Introduction:-

In the previous module, you learnt about databases and their various constituents. You also understood the steps involved in designing a database, which are as follows:

- Ideating and designing an ERD
- Adding a database schema, its various tables, the values in each table and the constraints
- Querying a database using several SQL statements to extract valuable insights from the data

So far, you have covered all the basic concepts of SQL. However, you should also be aware of some advanced concepts of SQL such as window functions, query optimisation, case statements, stored functions and cursors and the best practices for writing queries. These are useful tools for handling several use cases and will help you solve complex questions easily.

In the previous module, you learnt how to order employees by their salary in an example data set. One method to do this would be to categorise them by their values. For example, you can divide them into the following categories:

- Employees earning less than ₹2,50,000 per year
- Employees earning greater than or equal to ₹2,50,000 per year

The first category of employees would be exempted from paying any tax and need not go through some of the additional checks that the second category of employees would be required to go through. You can solve this problem statement using a filter condition in a 'where' clause.

Now, imagine you are working in a bank and need to identify and classify your customers on the following criteria:

- Top 10% of customers: Platinum
- Next 10% of customers: Gold
- Next 20% of customer: Silver
- Rest of the customers: Regular

Your bank intends to roll out different schemes for these classes of customers in exchange for their loyalty. But how do you solve such a seemingly complex problem statement? You will find the answer to this question in the upcoming sessions.

In this session

You will be introduced to the concept of windowing functions. You will learn about the 'over' and 'partition' clauses used to implement windowing. After going through this session, you should be able to use window functions such as rank(), dense_rank() and percent_rank() in your queries.

You will also be introduced to the concept of named windows. You will learn about frames and how they move within a window. Next, you will learn about the various applications of windowing, including one to calculate an element known as a moving average. Finally, you will learn about the 'lead' and 'lag' functions that are used to fetch data from succeeding and preceding rows, respectively.

In the upcoming video, you will get a brief glance at the topics that will be covered in this module.

Rank Functions - I:-

In the previous module, you learnt about and executed the 'group by' clause for collecting the facts about certain categories. Now, you may have noticed that using the 'group by' clause reduces the number of rows. It also leads to a loss of individual properties of various rows.

To address the two points mentioned above, i.e., reduction of rows and loss of individual properties of rows, SQL provides a special clause known as the 'over' clause, which displays group characteristics in the form of a new column. This leads to the preservation of the individual characteristics of different rows while displaying the group characteristics simultaneously. Let's watch the upcoming video to understand the need for rank functions in SQL.

In this video, you understood why rank functions are required in SQL. In the next video, Shreyas will walk you through the syntax for writing queries with rank functions.

So, as you learnt in this video, the syntax used for writing a rank function is as follows:

- RANK() OVER (
- PARTITION **BY** <expression>[{,<expression>...}]
- **ORDER BY** <expression> [**ASC|DESC**], [{,<expression>...}])

Note: You will learn more about the 'partition by' clause later in this session. In the upcoming video, you will get a demonstration of window functions (also known as analytic functions). You will also understand the importance of ranking values based on the required criteria.

In this video, you learnt how to use rank functions with the help of some examples.

So, how well have you understood the use of rank functions? Test yourself by writing the query for the question given below.

There are some more rank functions which help in ranking values according to the given criteria. You will learn about them in the next segment.

Rank Functions - II:-

In the previous segment, you learnt how rank functions are implemented to rank values based on certain criteria. In the upcoming videos, Shreyas will demonstrate the difference between the 'rank' and 'dense rank' functions. You will also learn about another type of rank function, which is known as the 'per cent rank' function.

In this video, you learnt that there are different types of rank functions, which are as follows:

- **RANK():** Rank of the current row within its partition, with gaps
- **DENSE_RANK():** Rank of the current row within its partition, without gaps
- **PERCENT_RANK():** Percentage rank value, which always lies between 0 and 1

The syntax for writing the 'dense rank' and 'per cent rank' functions are as follows:

- 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>...}]
-)
-

In the upcoming video, you will understand how to use different rank functions in queries to solve various problems.

So, as you saw in the video, the 'rank' function need not have consecutive values, but the 'dense rank' function must have these values. For example, consider the table given below. It contains the marks obtained by the top three students in the 12th board exams in India conducted by the CBSE board.

CBSE Marks (12th)

Name	Marks (out of 500)	Rank	Dense Rank
Shubham Agarwal	495	1	1
Paritosh Sinha	495	1	1
Dilip Kumar	492	3	2

Notice how Dilip's rank is 3 but his dense rank is 2. This is because the rank increases whenever the previous entries have similar values. If 10 students, instead of two, had scored 495 marks, then his rank would have become 11, but his dense rank would have remained 2.

There is a fourth type of rank function: the 'row number' function. You will learn more about this in the next segment.

Rank Functions - III:-

In the previous segment, you learnt about the 'rank', 'dense rank' and 'per cent rank' functions. In this segment, you will learn about the fourth type of rank function: the 'row number' function. In the upcoming video, you will learn more about this function and its uses.

So, as Shreyas mentioned, you can use the 'row number' function for the following use cases:

- To determine the top 10 selling products out of a large variety of products
- To determine the top three winners in a car race
- To find the top five areas in different cities in terms of GDP growth

The main advantage of the 'row number' function over all the other types of rank functions is that it returns unique values. The syntax for writing the 'row number' function is as follows:

- ROW_NUMBER() OVER (
- PARTITION **BY** <expression>[{,<expression>...}]
- **ORDER BY** <expression> [**ASC|DESC**], [{,<expression>...}]
-)

Let's take a look at the demonstration of all the rank functions in the following video.

Now that you know how to use different rank functions in your query, it's time to understand what the 'partition by' clause does in the multiple queries that you wrote. You will learn more about partitions in the next segment.

To learn more about the 'row number' function, you can refer to the link provided below.

Partitioning:-

You must have noticed that there was a 'partition by' clause in the syntax for the rank functions. But what exactly is this clause used for? You will learn more about this in the upcoming video.

As you learnt in this video, the 'rank' and 'dense rank' functions are used with the 'over' clause. However, this may not be enough if you want to rank groups of rows based on certain criteria. For example, you can rank the top 10 batsmen in the world using the 'rank' function. But what if you want to find out the top three batsmen from each team? This is where you would want to use the 'partition' and 'over' clauses together. So, let's watch the next video and learn more about these clauses and their uses.

So, now that you have learnt about window functions and partitioning, try to attempt the following question.

Try to answer the question given below to get more comfortable with the syntax. You may find it a bit difficult initially, but partitions and windows are a useful tool to solve questions that require you to rank values, especially when you need to rank values separately based on certain criteria (for example, state or gender).

In the next segment, you will learn about named windows and how they shorten long queries, making them more readable.

Named Windows:-

In the previous segment, you learnt that we can use multiple window functions in the same query. Now, you may have noticed that in one of the queries, we used the same window in both the window functions. This makes the code heavier and difficult to tweak as and when the need arises. So, in this segment, you will learn how to create, store and use named windows.

Let's watch the upcoming video as Shreyas explains this in detail.

So, as you learnt in this video, the same window can be used to define multiple 'over' clauses. You can define the window once, give it a name and then refer to the name in the 'over' clauses. A named window makes it easier to experiment with multiple window definitions and observe their effects on the query results. You only need to modify the window definition in the 'window' clause, rather than using multiple 'over' clause definitions.

The syntax for writing a named window is as follows:

- WINDOW window_name **AS** (window_spec)
- [, window_name **AS** (window_spec)] ...

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

1. SELECT
2. FROM
3. JOIN
4. WHERE
5. GROUP BY
6. HAVING
7. WINDOW
8. ORDER BY

In the next segment, you will learn another construct in SQL known as a frame. It is used to group data from multiple consecutive rows together and perform operations on them.

The MySQL documentation is a good resource in case you need more clarity on any concept(s). You can refer to the link provided below to learn more about named windows.

Eg. of window :

```
SELECT *,
RANK() OVER w 'Rank',
DENSE_RANK() OVER w 'Dense Rank',
PERCENT_RANK() OVER w 'Percent Rank'
FROM shipping_dimen
WINDOW w AS (
  PARTITION BY ship_mode
  ORDER BY COUNT(*)
);
```

Frames - I:-

In the previous segment, you were introduced to the concept of windowing functions and windows. You also learnt how to use basic aggregate functions, including the 'count()' function, in windowing.

In this segment, you will learn about the concept of frames and understand how frames move while a query is being executed. You will also learn how to implement moving averages in SQL. You can get a rough idea of the importance of calculating moving averages, especially in the stock market sector, by clicking on the link provided below and quickly going through the article on Investopedia.

- Moving Averages in the Stock Market

Note that you will only be calculating simple moving averages in this course. In the upcoming video, Shreyas will explain the concept of frames with the help of an example.

In this video, you learnt how frames are useful in SQL and when they can be applied to answer relevant questions asked by stakeholders. In the next video, you will see a demonstration of how to use frames in SQL.

So, in this video, you learnt how to use frames in SQL. Now, attempt the following question to test your understanding of moving averages.

Try to solve the question given below; it will help you analyse Kohli's batting in ODIs over a period of five years. You will be required to display 3 columns: '**Year**', '**Runs**' and '**5 Year Moving Average**' for the given data.

[Note: Please use the column names in lower case format as mentioned in the code stub]

If you look at the output obtained after writing the correct query, you will notice that Kohli's five-year average has been about 1,000 since 2013. This indicates that he has been a consistent player. Similarly, you can write queries to obtain one-year moving averages to extract and understand any trend in the data in the question given above.

In the next segment, Shreyas will break down the frame clause and analyse its components. This will help you to understand frames better and use the appropriate keywords and solve problems.

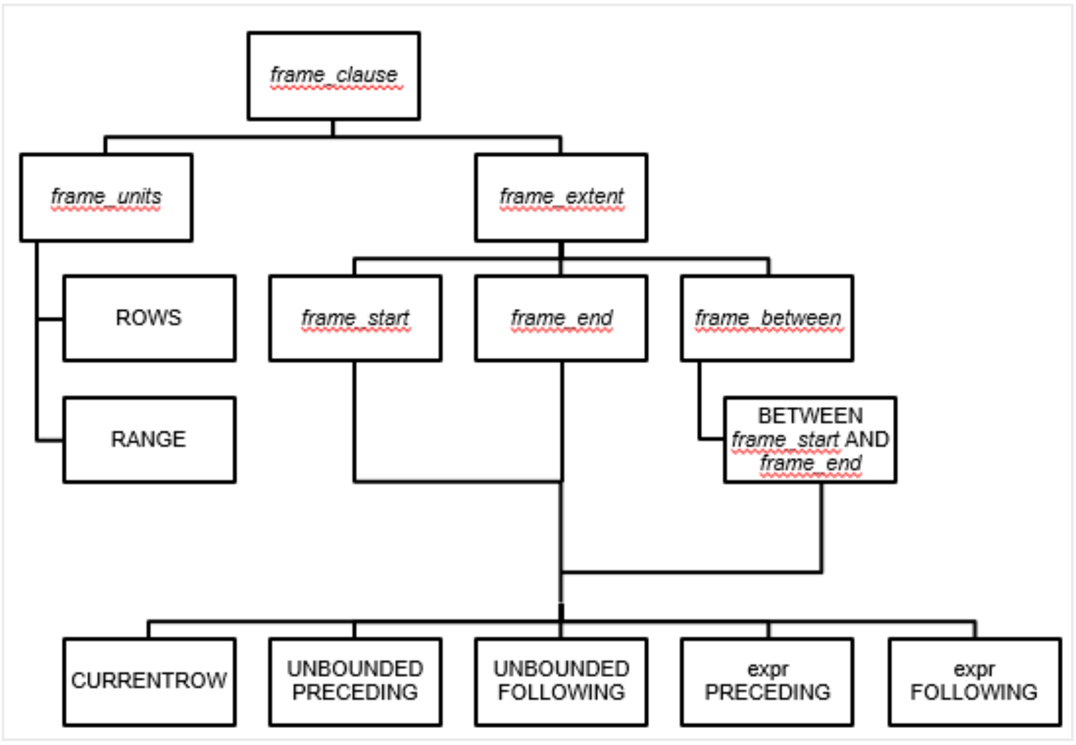
Eg. of frames

```
select *, avg(runs) over (order by year rows 4 preceding) '5 Year Moving
Average'
from kohli_batting;
```

Frames - II:-

In the previous segment, you learnt how to write a simple query using a frame. Now, let's watch the upcoming video to understand the structure of a frame in depth and learn about its possible constituents.

As you learnt in this video, a query that uses a frame has multiple components. These components are shown in the diagram given below.



You also learnt about the usage of several keywords in the 'frame' clause such as UNBOUNDED, PRECEDING, FOLLOWING, BETWEEN and so on. Note that the order of the rows gets reversed when you use the DESC keyword in the query. Now, before moving on to the next segment, try to solve the questions given below to test your knowledge of frames.

In the next segment, you will learn about yet another set of window functions. These are the lead and lag functions.

To learn more about the different parts of a frame, you can refer to the link provided below.

- Window Frame Clause

Lead and Lag Functions:-

In the earlier segments, you learnt how to calculate moving averages in SQL using frames. Identifying frequent customers is a commonly observed business requirement for retail chains. The 'lead' and 'lag' functions can be quite helpful in this case. You can compare the date on which a particular customer purchased an item from a store with the next date on which they ordered an item again. In the upcoming video, you will learn more about the 'lead' and 'lag' functions as well as their use cases.

So, as explained in this video, another use case of the 'lead' and 'lag' functions is to determine whether consecutive orders were shipped using the same shipping mode. This can be extremely helpful in optimising the shipping and delivery of products.

The syntax for using the 'lead' and 'lag' functions are as follows:

- LEAD(expr[, **offset**[, **default**]])
- OVER (Window_specification | Window_name)

- LAG(expr[, **offset**[, **default**]])
- OVER (Window_specification | Window_name)

In the next video, you will see a practical example of determining the frequency at which a customer named Rick Wilson orders products, using the 'lead' and 'lag' functions on the 'market star schema'.

With this, you have come to the end of this session. You learnt about various concepts in this session, so let's summarise the topics and the syntax for some of the advanced SQL concepts in the final segment.

Summary:-

In this session, you learnt about some of the advanced concepts in SQL. Let’s watch the upcoming video to revisit the topics that were covered in this session one by one.

Rank functions: The different types of rank functions are as follows:

- **RANK():** Rank of the current row within its partition, with gaps
- **DENSE_RANK():** Rank of the current row within its partition, without gaps
- **PERCENT_RANK():** Percentage rank value; it will always lie between 0 and 1
- **ROW_NUMBER():** Assigns unique numeric values to each row, starting from 1

Rank function syntax: The syntax for the 'rank' function is as follows:

- RANK() OVER (
- PARTITION **BY** <expression>[{,<expression>...}]
- **ORDER BY** <expression> [**ASC|DESC**], [{,<expression>...}]
-)

Named windows: A named window makes it easier to define and reuse multiple window functions. The syntax for a named window

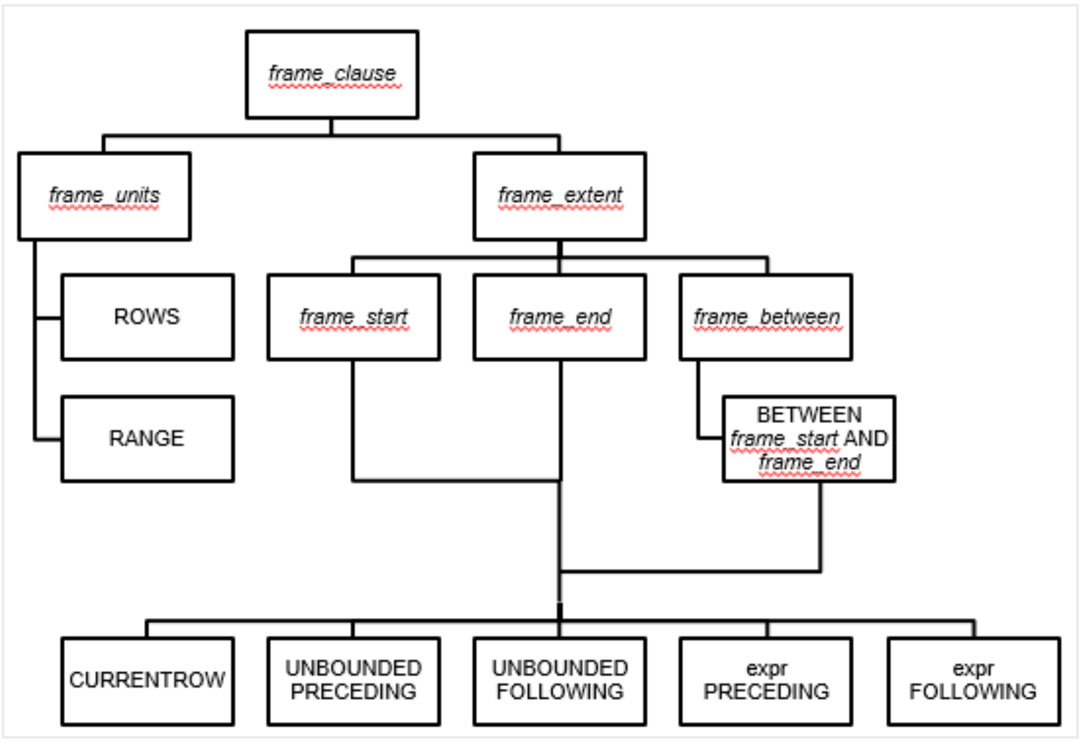
is as follows:

- WINDOW window_name **AS** (window_spec)
- [, window_name **AS** (window_spec)] ...

Order of SQL statements: The order in which the various SQL statements appear in a query is as follows:

1. SELECT
2. FROM
3. JOIN
4. WHERE
5. GROUP BY
6. HAVING
7. WINDOW
8. ORDER BY

Frames: Frames are used to subset a set of consecutive rows and calculate moving averages. A query using a frame has multiple components as shown in the diagram given below.



Lead and lag functions: These functions are used to compare a row value with the next or the previous row value. The syntax for the 'lead' function is as follows:

- LEAD(expr[, **offset**[, **default**]])
- OVER (Window_specification | Window_name)

Module 2 : Case Statements, Stored Routines and Cursors

Introduction:-

So far, you have learnt about some of the basic and advanced SQL concepts. So, let's continue with this module and learn more about case statements and stored routines. You will also understand the importance of using these and other SQL features in this session.

In this session

You will learn about case statements, which are used for classifying your results based on certain conditions. You will also learn about stored routines such as user-defined functions and stored procedures, which are used for storing and reusing large queries to solve complex problem statements efficiently. Finally, you will learn about cursors and their uses.

Case Statements - I:-

You may have encountered if-else and case statements in some programming languages. Now, suppose you are in school and have to follow a set time-table, according to which you need to bring the textbooks of certain subjects on certain days to class. Consider the following time-table:

- Monday: English and Hindi
- Tuesday: English and Maths
- Wednesday: Science and Social Science
- Thursday: Maths and Social Science
- Friday: Hindi and Science

This is a basic example where you can determine the subjects that you would be studying in class on certain days by just looking at the list. However, the data around us is enormous and complex. There may be several conditions that you may need to set and many actions to be performed according to each of these conditions.

In the upcoming video, you will learn about case statements in detail.

As you learnt in the video, the syntax for writing a case statement is as follows:

- **CASE**
- **WHEN** condition1 **THEN** result1
- **WHEN** condition2 **THEN** result2
- .
- .
- **WHEN** conditionN **THEN** resultN
- **ELSE result**

- **END AS column_name**;

So, now that you know the syntax for writing a case statement, in the next video, Shreyas will walk you through a practical example where you will classify orders based on the amount of profit or loss incurred.

Now, solve the coding question given below to understand the importance of case statements in solving actual business problems. You will realise how easy it is to classify groups of values based on some given conditions. Such classifications help companies take important business decisions.

In the next segment, you will learn how case statements can be applied with other SQL concepts to solve complex questions.

Eg. of case in sql:-

```
use upgrad;

# Write your code below
Select Name,
       salary as 'Salary(in LPA)',
       case
          when salary <= 2.5 then 'A'
          when salary between 2.5 and 5 then 'B'
          when salary between 5 and 10 then 'C'
          else 'D'
        end as 'Tax Slab'
From
Salaries;
```

Case Statements - II:-

In the previous segment, you saw a relatively simple example of a case statement. In the video, you will learn how a complex problem can be solved using case statements, CTEs and joins in conjunction. While solving this problem, you will realise how important it is to understand basic SQL concepts and when and where to apply them.

[Note: The right subheading in the video given above should be 'Case Statements'.]

In the previous session, you classified some employees into various tax slabs according to their salaries. Now, try to solve the following question, which is slightly more difficult and requires you to use case statements.

In the next segment, you will learn about another construct in SQL called a stored routine. You will learn why stored routines are important and also implement them in queries.

Eg. of case in sql:-

```
use upgrad;

# Write your code below
select *,
round(case
when salary <= 2.5
then 0
when salary > 2.5 and salary <= 5
then 0.05 * (salary - 2.5) * pow(10,5)
when salary > 5 and salary <= 10
then (0.125 + 0.2 * (salary - 5)) * pow(10,5)
when salary > 10
then (1.125 + 0.3 * (salary - 10)) * pow(10,5) end) as 'Tax Amounts'
from salaries;
```

UDFs:-

You already know how to deploy various in-built MySQL functions, such as sum(), avg() and concat(), to make querying easier. Now, it is possible to find the sum of two numbers in MySQL without using the sum() function. You can use the arithmetic addition operator (+) for this purpose. Similarly, you can use the arithmetic addition and division operators to find the average of two numbers.

The sum() and avg() functions are preferred because they are easier to use and also increase the readability of the code. Another important factor is reusability. You do not need to see the entire definition behind the sum() function every time you use it; all you need is the name sum() to invoke the function whenever you want to determine the sum of two numbers.

However, there are operations that you may want to repeat multiple times in a piece of code because they do not have an in-built function. This is where you must use user-defined functions (UDFs) or stored functions. In the upcoming video, Shreyas will explain the syntax of UDFs.

As you learnt in this video, the syntax for writing a UDF is as follows:

- **DELIMITER** $$
-
- **CREATE FUNCTION** function_name(func_parameter1, func_parameter2, ...)
-   **RETURN** datatype [**characteristics**]
- /*   func_body   */

- **BEGIN**
- <**SQL** Statements>
- **RETURN** expression;
- **END** ; $$
-
- **DELIMITER** ;

- **CALL** function_name;

Remember the following points:

- The CREATE FUNCTION is also a DDL statement.
- The function body must contain one RETURN statement.

So, now that you have understood the importance of UDFs and are familiar with the syntax, in the upcoming video, you will see a practical demonstration of the same. You will solve the same problem statement that was discussed in the segments on case Statements.

So, as you learnt in this video, a delimiter is a marker for the end of each command that you send to the MySQL command-line client. Whenever you are inside a UDF, you need to define another delimiter and reset it to the default '**;**' (semicolon) after the function ends.

There is another stored routine in SQL - a stored procedure. How does it differ from a UDF? You will find out in the next segment.

Stored Procedures:-

In the previous segment, you were introduced to a UDF, which is a type of stored routine. There is another type of stored routine in SQL: a stored procedure. This is similar to a UDF but differs in certain ways. Watch the next video to learn more about stored procedures.

As you learnt in this video, the syntax for writing a stored procedure is as follows:

- **DELIMITER** $$
-
- **CREATE PROCEDURE** Procedure_name (<Paramter List>)
- **BEGIN**
- <**SQL** Statements>
- **END** $$

-
- **DELIMITER** ;

- **CALL** Procedure_name;

In the next video, you will see a demonstration of how stored procedures are used in SQL. Note that this is a simple example, but stored procedures can become quite complex according to the business use case.

In this video, you learnt how to use stored procedures in SQL. Although the syntax of a stored procedure is similar to that of UDFs, there are many differences between the two. In the next video, you will learn about these differences in detail.

The differences between UDFs and stored procedures are summarised in the table given below.

UDF	Stored Procedure
1. It supports only the input parameter, not the output.	1. It supports input, output and input-output parameters.
2. It cannot call a stored procedure.	2. It can call a UDF.
3. It can be called using any SELECT statement.	3. It can be called using only a CALL statement.
4. It must return a value.	4. It need not return a value.
5. Only the 'select' operation is allowed.	5. All database operations are allowed.

Keep in mind the above differences whenever you need to determine the right kind of stored routine to use. Now that you have learnt all about stored routines, you will learn about another concept which is used in advanced applications of SQL, which is the cursor.

Cursors:-

While stored routines are a great way to store and reuse logic in code as required, they do have some disadvantages. You will learn what these are. You will also be introduced to the concept of cursors. Cursors are used to individually process each row that is returned in a query. In the upcoming video, Professor Ramanathan will take you through an example and explain how cursors are used in it.

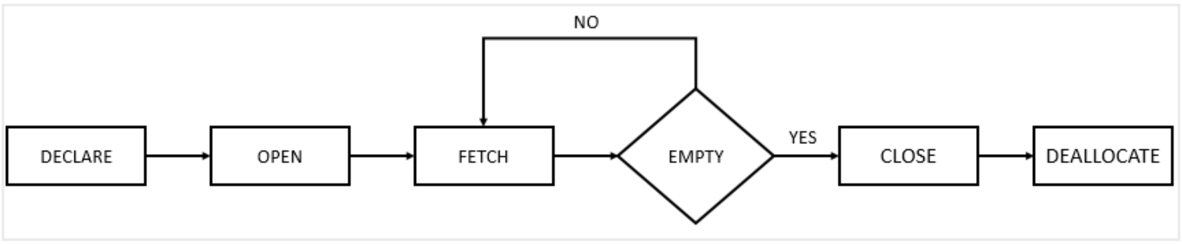
While using either a UDF or a stored procedure in MySQL, it is important to remember that it is not portable across different database engines.

With this, you have come to the end of this session. In this session, you learnt about some of the advanced SQL concepts. So, let's summarise the various topics and the syntax used for these concepts in the final segment.

<u>Some important readout:-</u>

SQL Server cursor life cycle

These are steps for using a cursor:



First, declare a cursor.

```
DECLARE cursor_name CURSOR
   FOR select_statement;
```

Code language: SQL (Structured Query Language) (sql)

To declare a cursor, you specify its name after the DECLARE keyword with the CURSOR data type and provide a SELECT statement that defines the result set for the cursor.
Next, open and populate the cursor by executing the SELECT statement:

```
OPEN cursor_name;
```

Code language: SQL (Structured Query Language) (sql)

Then, fetch a row from the cursor into one or more variables:

```
FETCH NEXT FROM cursor INTO variable_list;
```

Code language: SQL (Structured Query Language) (sql)

SQL Server provides the @@FETCHSTATUS function that returns the status of the last cursor FETCH statement executed against the cursor;
If @@FETCHSTATUS returns 0, meaning the FETCH statement was successful. You can use the WHILE statement to fetch all rows from the cursor as shown in the following code:

```
WHILE @@FETCH_STATUS = 0
   BEGIN
      FETCH NEXT FROM cursor_name;
   END;
```

Code language: SQL (Structured Query Language) (sql)

After that, close the cursor:

```
CLOSE cursor_name;
```

Code language: SQL (Structured Query Language) (sql)

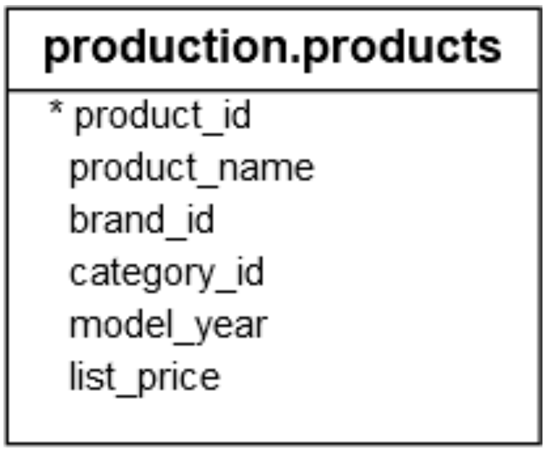
Finally, deallocate the cursor:

```
DEALLOCATE cursor_name;
```

Code language: SQL (Structured Query Language) (sql)

SQL Server cursor example

We'll use the prodution.products table from the sample database to show you how to use a cursor:



First, declare two variables to hold product name and list price, and a cursor to hold the result of a query that retrieves product name and list price from the production.products table:

```
DECLARE 
   @product_name VARCHAR(MAX), 
   @list_price   DECIMAL;

DECLARE cursor_product CURSOR
FOR SELECT 
      product_name, 
      list_price
   FROM 
      production.products;
```

Code language: SQL (Structured Query Language) (sql)

Next, open the cursor:

```
OPEN cursor_product;
```

Code language: SQL (Structured Query Language) (sql)

Then, fetch each row from the cursor and print out the product name and list price:

```
FETCH NEXT FROM cursor_product INTO 
   @product_name, 
   @list_price;

WHILE @@FETCH_STATUS = 0
   BEGIN
      PRINT @product_name + CAST(@list_price AS varchar);
```

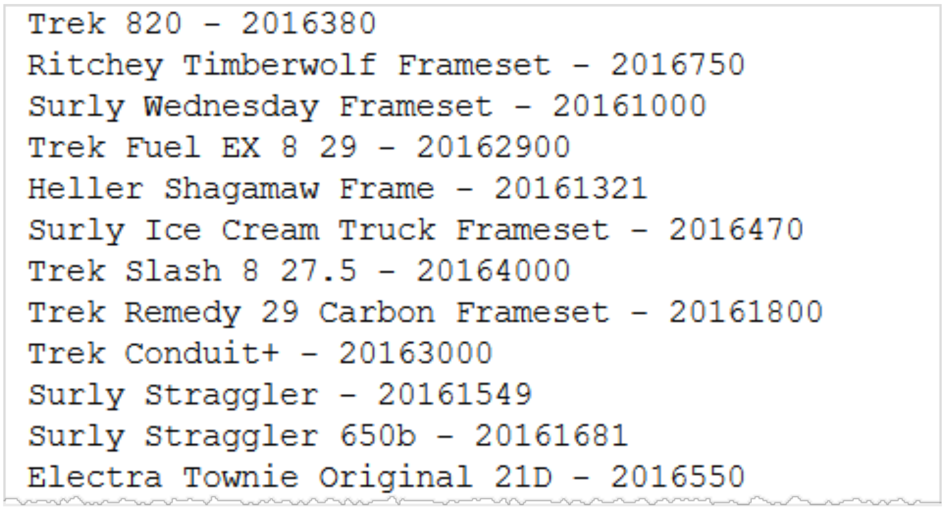
```
        FETCH NEXT FROM cursor_product INTO
            @product_name,
            @list_price;
    END;
```
Code language: SQL (Structured Query Language) (sql)

After that, close the cursor:
```
CLOSE cursor_product;
```
Code language: SQL (Structured Query Language) (sql)

Finally, deallocate the cursor to release it.
```
DEALLOCATE cursor_product;
```
Code language: SQL (Structured Query Language) (sql)

The following code snippets put everything together:
```
DECLARE
    @product_name VARCHAR(MAX),
    @list_price   DECIMAL;

DECLARE cursor_product CURSOR
FOR SELECT
        product_name,
        list_price
    FROM
        production.products;

OPEN cursor_product;

FETCH NEXT FROM cursor_product INTO
    @product_name,
    @list_price;

WHILE @@FETCH_STATUS = 0
    BEGIN
        PRINT @product_name + CAST(@list_price AS varchar);
        FETCH NEXT FROM cursor_product INTO
            @product_name,
            @list_price;
    END;

CLOSE cursor_product;

DEALLOCATE cursor_product;
```
Code language: SQL (Structured Query Language) (sql)

Here is the partial output:



```
Trek 820 - 2016380
Ritchey Timberwolf Frameset - 2016750
Surly Wednesday Frameset - 20161000
Trek Fuel EX 8 29 - 20162900
Heller Shagamaw Frame - 20161321
Surly Ice Cream Truck Frameset - 2016470
Trek Slash 8 27.5 - 20164000
Trek Remedy 29 Carbon Frameset - 20161800
Trek Conduit+ - 20163000
Surly Straggler - 20161549
Surly Straggler 650b - 20161681
Electra Townie Original 21D - 2016550
```

In practice, you will rarely use the cursor to process a result set in a row-by-row manner.
In this tutorial, you have learned how to use the SQL Server cursor to process a result set, each row at a time.

Summary:-

In this session, you learnt about case statements, stored routines and cursors in SQL. Let's watch the upcoming video to revisit the topics that were covered in this session one by one.

The topics that were covered in this session can be summarised as follows:-

Case statements: Case statements are used to classify data values into different groups according to the given criteria. The syntax of a case statement is as follows:

- **CASE**
- **WHEN** condition1 **THEN** result1
- **WHEN** condition2 **THEN** result2
- .
- .
- **WHEN** conditionN **THEN** resultN
- **ELSE result**
- **END AS column_name**;

UDFs: UDFs are used to create and reuse certain pieces of

functionality in SQL. The syntax of a UDF is as follows:

- **DELIMITER** $$
-
- **CREATE FUNCTION** function_name(func_parameter1, func_parameter2, ...)
-   **RETURN** datatype [**characteristics**]
- /*   func_body   */
-   **BEGIN**
-     <**SQL** Statements>
-     **RETURN** expression;
- **END** $$
-
- **DELIMITER** ;

- **CALL** function_name;

Stored procedures: Stored procedures are also used to reuse some required functionality in SQL. The syntax of a stored procedure is as follows:

- **DELIMITER** $$
-
- **CREATE PROCEDURE** Procedure_name (<Paramter List>)
- **BEGIN**
-   <**SQL** Statements>
- **END** $$
-
- **DELIMITER** ;

- **CALL** Procedure_name;

UDFs vs stored procedures: The differences between UDFs and stored procedures are summarised in the table given below.

UDF	Stored Procedure

1. It supports only the input parameter, not the output.	1. It supports input, output and input-output parameters.
2. It cannot call a stored procedure.	2. It can call a UDF.
3. It can be called using any SELECT statement.	3. It can be called using only a CALL statement.
4. It must return a value.	4. It need not return a value.
5. Only the 'select' operation is allowed.	5. All database operations are allowed.

Cursors: A cursor is used to individually process each row that is returned in a query.

Module 3 : Query Optimisation and Best Practices

Introduction:-

So far, you have learnt how to use different commands and tools in SQL. There is, however, one marked difference between the data sets that we used and the ones that are used by the likes of Amazon and Uber. The difference, of course, is the size of data.

We were using a small database for simplicity and ease of understanding. However, real-life data sets, such as those used by Amazon and Uber, would have millions or even billions of rows. Hence, although a 'select' operation on our dummy 'Market' table takes less than a fraction of a second, a similar query when applied on a table with millions of rows, for example, a 'customer' or 'product' table for Amazon, might take hours if not optimised properly.

In this session

You will understand the importance of query optimisation and learn about the various tips and techniques used in order to save runtime and the required computational power. You will also learn about some of the best practices that you should follow while writing SQL code.

Best Practices - I:-

Suppose you want to buy a new car. Now, once you enter a showroom, a salesperson will explain the features of various cars to you, including each model's cost, mileage and so on. They will probably also mention an optimum range of speed that you should maintain while driving. But why is this important? Doing so ensures that the car continues to give the best possible mileage.

Similarly, you should follow best practices while writing any SQL code. This makes your code more readable for the people working on it and helps in maximising the output for your company. Let's watch the upcoming video to understand the importance of following best practices while writing queries.

Now that you have understood why it is important to follow best practices, let's watch the next video to understand some of the guidelines that you should adhere to.

In the next segment, you will learn about some more best practices in SQL. You will also learn how to use the SQL Formatter, which cleans your code for you.

Best Practices - II:-

In the previous segment, you understood the importance of following best practices while writing queries in SQL. Now, let's watch the next video to learn about some of the guidelines that you should follow while writing queries.

In this video, you learnt about some of the important best practices, which are as follows:

- Comment your code by using a hyphen **(-)** for a single line and **(/* ... */)** for multiple lines of code.
- Always use table aliases when your query involves more than one source table.
- Assign simple and descriptive names to columns and tables.
- Write SQL keywords in upper case and the names of columns, tables and variables in lower case.
- Always use column names in the 'order by' clause, instead of numbers.
- Maintain the right indentation for different sections of a query.
- Use new lines for different sections of a query.
- Use a new line for each column name.
- Use the SQL Formatter or the MySQL Workbench Beautification tool (Ctrl+B).

As Shreyas mentioned in the previous video, you can beautify your code by using either the MySQL Workbench Beautification tool or the SQL Formatter (or another similar online formatter). Now, let's watch the next video to learn more about the SQL Formatter.

As you learnt in this video, the SQL Formatter is a handy online tool that enables you to customise the formatting of your code. Make sure you are comfortable using the tool and make it a practice to use it while writing a query.

In the next segment, you will learn the concept of indexing and also understand how it optimises query execution.

The link to the SQL Formatter is provided below.

- Instant SQL Formatter

Indexing - I:-

In the previous segment, you learnt about some of the important best practices that you should follow while writing queries and understood the uses of the SQL Formatter. ***Now, 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 value(s) directly, instead of going through the entire table.*** This prevents the query engine from looking up values one by one; instead, it returns the exact value that you want right away.

In the upcoming video, Professor Ramanathan will explain why indexing is required for querying extremely large data sets.

As explained in this video, indexing is necessary for querying extremely large data sets. A primary key is an index because it helps in identifying each record in a table uniquely. Note that you cannot actually see an index, as it is an internal construct used in database engines to speed up queries.

In the next segment, you will learn the syntax for creating, adding and dropping an index in your query. You can learn more about indexing and any other concept(s) by referring to the MySQL documentation. Refer to the link below to understand what the documentation says about indexing.

- How MySQL Uses Indexes

Indexing - II:-

In the previous segment, you understood the importance of indexing in SQL. Now, let's watch the upcoming video to learn how indexing can be implemented in MySQL using the 'where' clause.

So, as you learnt in this video, MySQL has a specific DDL statement called CREATE INDEX, which is used for specifying the attribute on which you want to create an index. Generally, you would not have the permission to create such indices on a database. In such cases, you can ask the Database Administrator (DBA) to create the index for you.

In the next video, you will learn the syntaxes for creating, adding and dropping an index. You will also learn how to implement these commands to optimise your queries.

Let’s reiterate the commands used for creating, adding and

dropping an index in MySQL Workbench.

The command for creating an index is as follows:

- **CREATE INDEX** index_name
- **ON table_name** (column_1, column_2, ...);

The command for adding an index is as follows:

- **ALTER TABLE table_name**
- **ADD INDEX** index_name(column_1, column_2, ...);

The command for dropping an index is as follows:

- **ALTER TABLE table_name**
- **DROP INDEX** index_name;

Answer the questions given below to test your understanding of indexing in SQL.

There are two types of indexing. Learn about what these are and how they differ in the next segment.

Clustered vs Non-Clustered Indexing:-

As you learnt in the previous segments, indexing is an important aspect of optimising queries. You can speed up query processing by applying filters on an indexed column. In the upcoming video, you will learn about the two types of indices.

In this video, you learnt that there are two types of indices: clustered and non-clustered. The major differences between these are summarised in the table given below.

Clustered Index	Non-Clustered Index
1. This is mostly the primary key of the table.	1. This is a combination of one or more columns of the table.
2. It is present within the table.	2. The unique list of keys is present outside the table.
3. It does not require a separate mapping.	3. The external table points to different sections of the main table.
4. It is relatively faster.	4. It is relatively slower.

In the next segment, you will learn about the order in which the various parts of an SQL query are executed in the database engine. This will enable you to write the most optimal query for a given problem statement.

Order of Query Execution - I:-

Suppose you are an aspiring singer who wants to make it in the music industry. In order to start your journey, you register to enter the reality TV show *Indian Idol*. You find yourself waiting in the line for hours with thousands of other contestants. After interacting with some of them, you realise that most of the contestants have no sense of tune, rhythm or music in general.

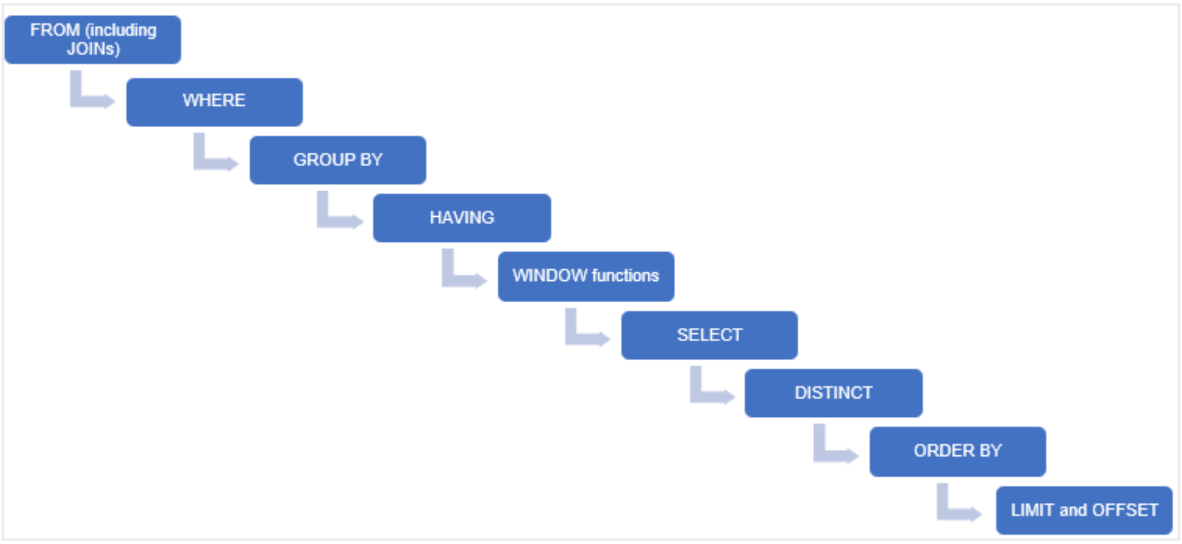
Can you think of a way in which you can make it easier for the judges to filter the applications? Think about it, and watch the next video to find out what Shreyas thinks is a feasible solution to this problem.

In this video, Shreyas explained one of the methods to filter the applications. In the module 'Database Design and Introduction to SQL', you learnt about the different aspects of a query, which appear in a particular order. The order in which the various SQL statements appear in a query is as follows:

1. SELECT
2. FROM
3. [JOIN]
4. WHERE
5. GROUP BY
6. HAVING
7. WINDOW
8. ORDER BY

However, the order in which the various statements are executed by the database engine is not the same. In the upcoming video, you will learn about the order of query execution, which will help you understand the need for reducing your data set as much as possible before querying it.

In this video, you learnt about the order in which the different SQL statements are executed in a query, which is depicted in the diagram given below.



Some of the important points that you should keep in mind while writing a query are as follows:

- Use inner joins wherever possible to avoid having any unnecessary rows in the resultant table.
- Apply all the required filters to get only the required data values from multiple tables.
- Index the columns that are frequently used in the WHERE clause.
- Avoid using DISTINCT while using the GROUP BY clause, as it slows down query processing.
- Avoid using SELECT * as much as possible. Select only the required columns.
- Use the ORDER BY clause only if it is absolutely necessary, as it is processed late in a query.
- Avoid using LIMIT and OFFSET as much as possible. Instead, apply appropriate filters using the WHERE clause.

Now that you know the order of query execution in SQL, let's see a practical example of how this can be useful. In the next segment, you will reduce the run time of a query by applying some of the optimisation techniques that you have learnt so far.

Order of Query Execution - II:-

Do you remember the problem statement where you wrote a query to determine the top ten customers from the 'market_star' schema using different rank functions? Let's revisit it and try to optimise the query. In the upcoming video, Shreyas will attempt to optimise the query using some optimisation techniques. Watch the upcoming video to understand how this changes the query execution time.

You must have noticed that you obtained the exact same result as you did earlier. However, the execution time of the unoptimised and the optimised queries are 141 ms and 15 ms, respectively. The order of magnitude is

$141/15 = 9.4.$

This is obviously a huge improvement and will only become more pronounced in a real-life data set, which will be much larger than what you are dealing with in this module.

Do you recall the concepts of joins and nested queries that you learnt in basic SQL? In the next segment, you will see how they compare with each other in terms of performance.

Joins vs Nested Queries:-

As you learnt previously, nested queries and joins are used for retrieving data from multiple tables. But is one of them more efficient than the other, especially in the case of large data sets? Well, it depends on the query processor. Query processors run optimising operations on your queries to ensure that the runtime is as low as possible. You will learn more about this in the upcoming video.

As you learnt in this video, executing a statement with the 'join' clause creates a join index, which is an internal indexing structure. This makes it more efficient than a nested query. However, a nested query would perform better than a join while querying data from a distributed database.

In a distributed database, tables are stored in different locations instead of a local system. In this case, a nested query would perform better than a join, as we can extract relevant information from different tables located in different computers. We can then merge the values in order to obtain the desired result. In the case of a join, we would need to create a large table from the existing tables, and filtering this large table would require comparatively more time.

With this, we have come to the end of this session. You learnt about various concepts in this session, so let's summarise the topics and the syntax for writing optimised SQL queries in the final segment.

To learn more about the performance of joins in comparison with nested queries in different scenarios, you can refer to the link provided below.

- Join operation vs Nested query in DBMS

Summary:-

In this session, you learnt about some of the best practices that should be followed while writing SQL code. You also understood the importance of query optimisation and learnt about the factors that you need to consider while optimising queries. These factors are indexing, the order of query execution and the relative performance of joins as compared with that of nested queries for different scenarios. Let's watch the upcoming video to revisit the topics that were covered in this session one by one.

In this video, Shreyas briefly revisited the topics that were covered in this session, which can be summarised as follows:

Best practices: Some of the best practices that you should remember while writing an SQL query are as follows:

- Comment your code using a hyphen **'-'** for a single line and **'/* ... */'** for multiple lines of code.
- Always use table aliases when your query involves more than one source table.
- Assign simple and descriptive names to columns and tables.
- Write SQL keywords in upper case and the names of columns, tables and variables in lower case.
- Always use column names in the 'order by' clause instead of numbers.
- Maintain the right indentation for different sections of a query.
- Use new lines for different sections of a query.
- Use a new line for each column name.
- Use the SQL Formatter or the MySQL Workbench Beautification tool (Ctrl+B) to clean your code.

Indexing: Indexing is an effective way to optimise query execution, as it selects the required data values instead of processing the entire table. The syntaxes for creating, adding and dropping an index are as follows:

- **CREATE INDEX** index_name
- **ON table_name** (column_1, column_2, ...);

- **ALTER TABLE table_name**
- **ADD INDEX** index_name(column_1, column_2, ...);

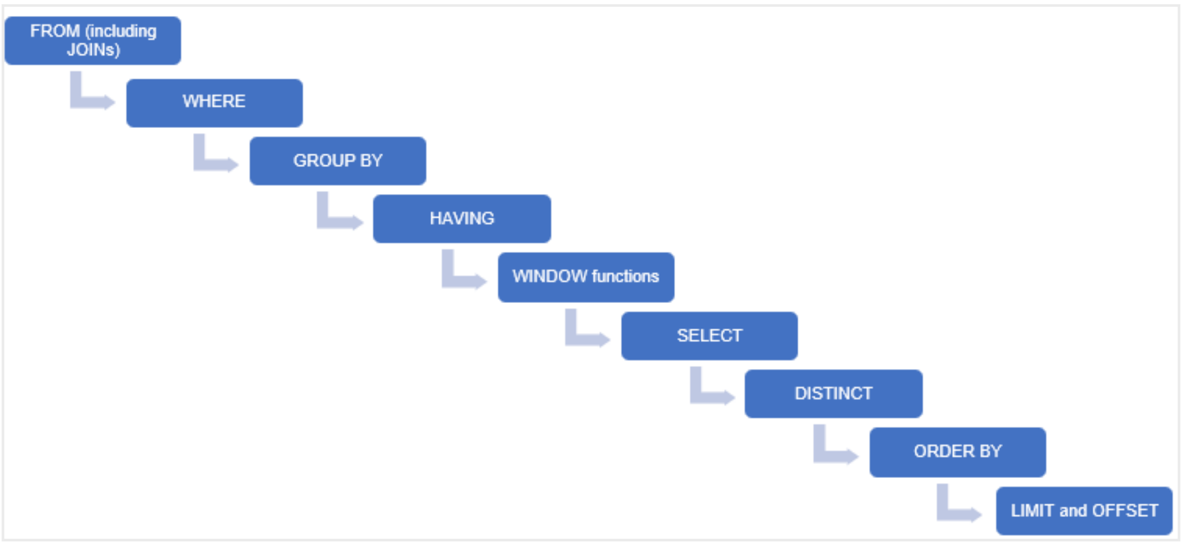
- **ALTER TABLE table_name**
- **DROP INDEX** index_name;

Clustered vs non-clustered indexing: The major differences

between clustered and non-clustered indexing are summarised in the table given below.

Clustered Index	Non-Clustered Index
1. This is mostly the primary key of the table.	1. It is a combination of one or more columns of the table.
2. It is present within the table.	2. The unique list of keys is present outside the table.
3. It does not require a separate mapping.	3. The external table points to different sections of the main table.
4. It is relatively faster.	4. It is relatively slower.

Order of query execution: The order in which the different SQL statements are executed in a query is depicted in the diagram given below.



Query optimisation techniques: The points that you should remember while writing a query are as follows:

- Use inner joins wherever possible to avoid having any unnecessary rows in the resultant table.
- Apply all the required filters to get only the required data values from multiple tables.
- Index the columns that are frequently used in the WHERE clause.
- Avoid using DISTINCT while using the GROUP BY clause, as it slows down query processing.
- Avoid using SELECT * as much as possible. Select only the required columns.
- Use the ORDER BY clause only if it is absolutely necessary, as it is processed late in a query.

- Avoid using LIMIT and OFFSET as much as possible. Instead, apply appropriate filters using the WHERE clause.

Joins vs nested queries: Executing a statement with the 'join' clause creates a join index, which is an internal indexing structure. This makes it more efficient than a nested query. However, a nested query would perform better than a join while querying data from a distributed database.

Module 4 : Problem-Solving Using SQL

Introduction:-

By now, you must be familiar with the advanced concepts of SQL. You went through a few examples where you implemented them by writing queries. In this session, you will apply your knowledge of these concepts to solve a business problem and identify the most profitable customers.

In this session

You will perform profitability analysis on the 'market star' schema that you have been working on throughout this module. You will learn how to solve three business problems by using the concepts that you have learnt so far in the basic and advanced SQL modules.

Profitability Analysis - I:-

In this segment, you will solve a business problem using SQL. In the first problem, you will identify the profitable product categories in the 'market star' schema. Let's watch the upcoming video to get an understanding of the problem statement and learn about the metrics that can be used for solving it.

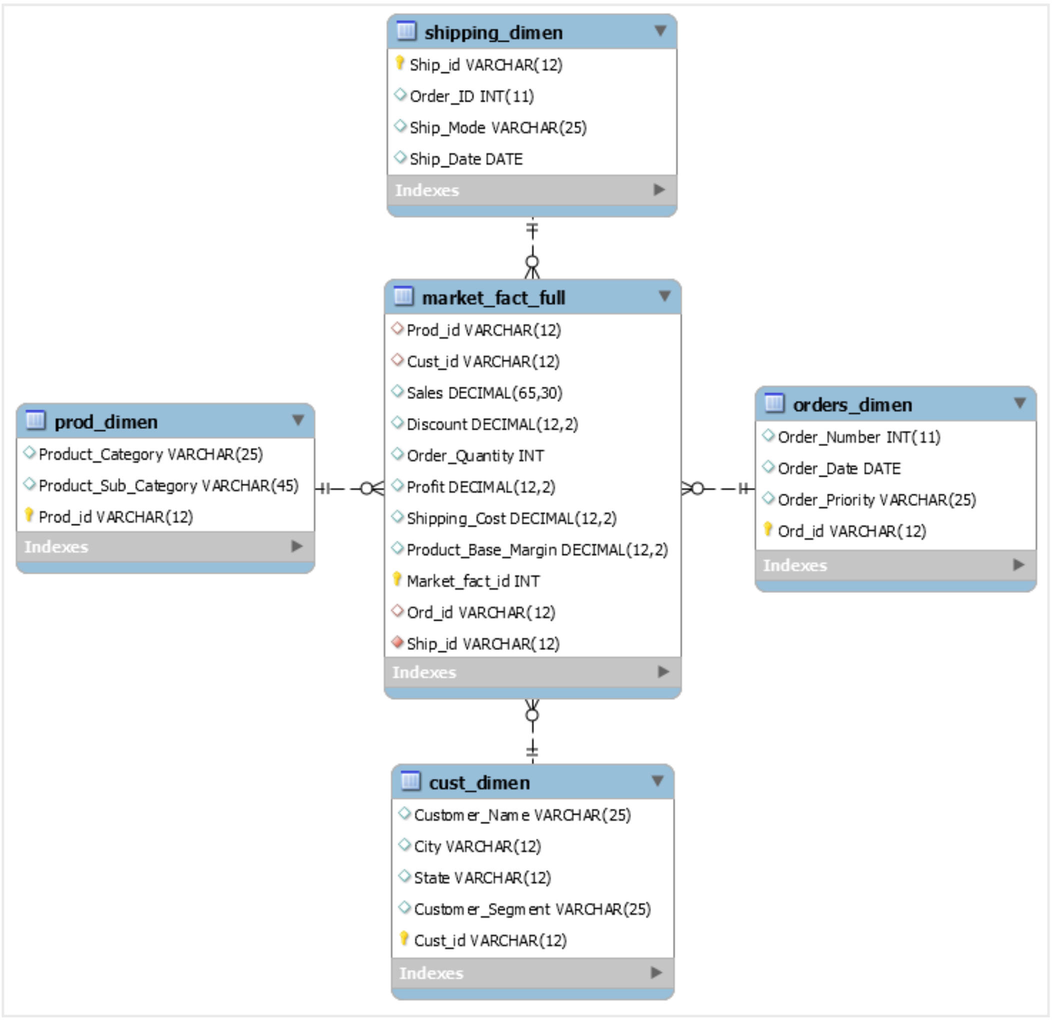
In this video, you were introduced to the problem statement, the required metrics and required tables, which can be summarised as follows:

Problem statement: Identify the sustainable (profitable) product categories so that the growth team can capitalise on them to increase sales.

Metrics: Some of the metrics that can be used for performing the profitability analysis are as follows:
- Profits per product category
- Profits per product subcategory
- Average profit per order
- Average profit percentage per order

Tables: The tables that are required for solving this problem are as follows:
- 'market_fact_full'
- 'prod_dimen'
- 'orders_dimen'

The ERD for the 'market star' schema is given below for your reference.



Although you have used the schema throughout this module, make sure you are familiar with its tables, columns and the relationships between its tables. In the next segment, you will start solving the problem statement along with Shreyas.

Profitability Analysis - II:-

Let's start solving the business problem by calculating the metrics that we discussed in the previous segment. In the next video, you will write the queries that are required to find the profits for all the product categories and subcategories. Which of the metrics (if any) discussed in the previous segment are actually useful? Let's watch the video and find out.

As you learnt in this video, all the product categories report good profits. Hence, they do not contribute to the solution, as you cannot identify the reasons for any loss that the company is facing. However, the profit per product category is still an important metric that can be used for creating business reports. The profit per product subcategory is a useful metric, as it shows the subcategories that report heavy losses.

Before calculating the third metric, let's first find out if there is any relation between the Order IDs table and the Order Numbers table.

As Shreyas explained in this video, the Order IDs table and the Order Numbers table do not have a one-to-one relation. Why does this matter? It proves that there are customers who have ordered products more than once. It is important to identify frequent customers so that you can offer discount coupons to them in exchange for their loyalty.

In the next segment, you will calculate the remaining metrics and determine their importance in the profitability analysis.

Profitability Analysis - III:-

You have already calculated the profit for each category and subcategory. In this segment, you will calculate the third metric: average profit per order. Let's watch the next video and find out more about this.

You observed that the average profit per order for furniture products is low (₹75.23) compared with that of the other product categories.

In the next video, let's compute the fourth and last metric: average profit percentage per order. While coding along with Shreyas, you must have observed that he examined each table to check whether there are any other metrics that could be useful for the analysis.

You observed that the average profit percentage per order for furniture products is quite low (2.27%) compared to the other product categories. Such low values of the average profit and profit percentage per order for furniture show that these products are not doing well. Their sale should ideally be stopped or the company should come up with a robust plan to deal with this issue.

You have now calculated all the required metrics that were specified in the problem statement. Notice how you derived additional insights by calculating the total number of orders for each product category. While performing any analysis on a data set, you should always look for additional metrics that can help you understand the trends in the data set.

In the next segment, you will solve another problem. This will help you to identify the most profitable customers for the company.

Profitable Customers - I:-

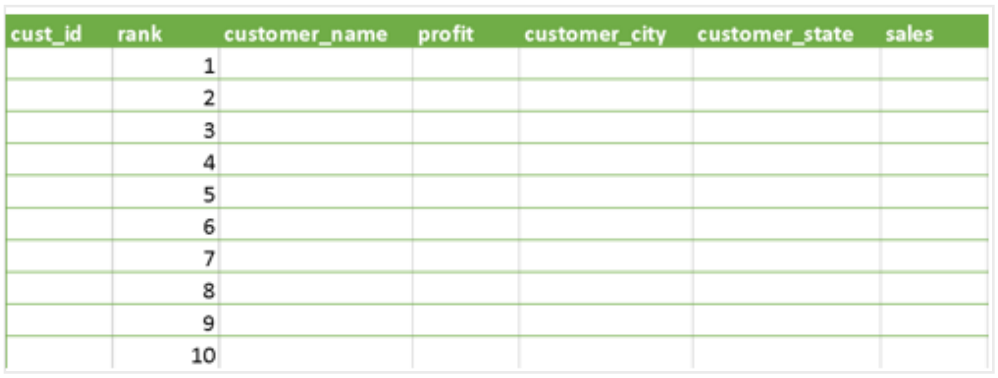
Now that you have completed the profitability analysis for product categories and subcategories, it is time to solve the second problem statement. Let's watch the upcoming video and learn more about it.

As you learnt in this video, there is a major difference between this problem statement and the one you solved in the previous segment. You already have the expected output table that contains the required columns. Sometimes, interviewers test whether you can write the exact query that is required for extracting the expected output.

The problem statement, the expected output format and the required tables can be summarised as follows:

Problem statement: Extract the details of the top ten customers in the expected output format.

Expected output format: The expected output format is given in the table below.

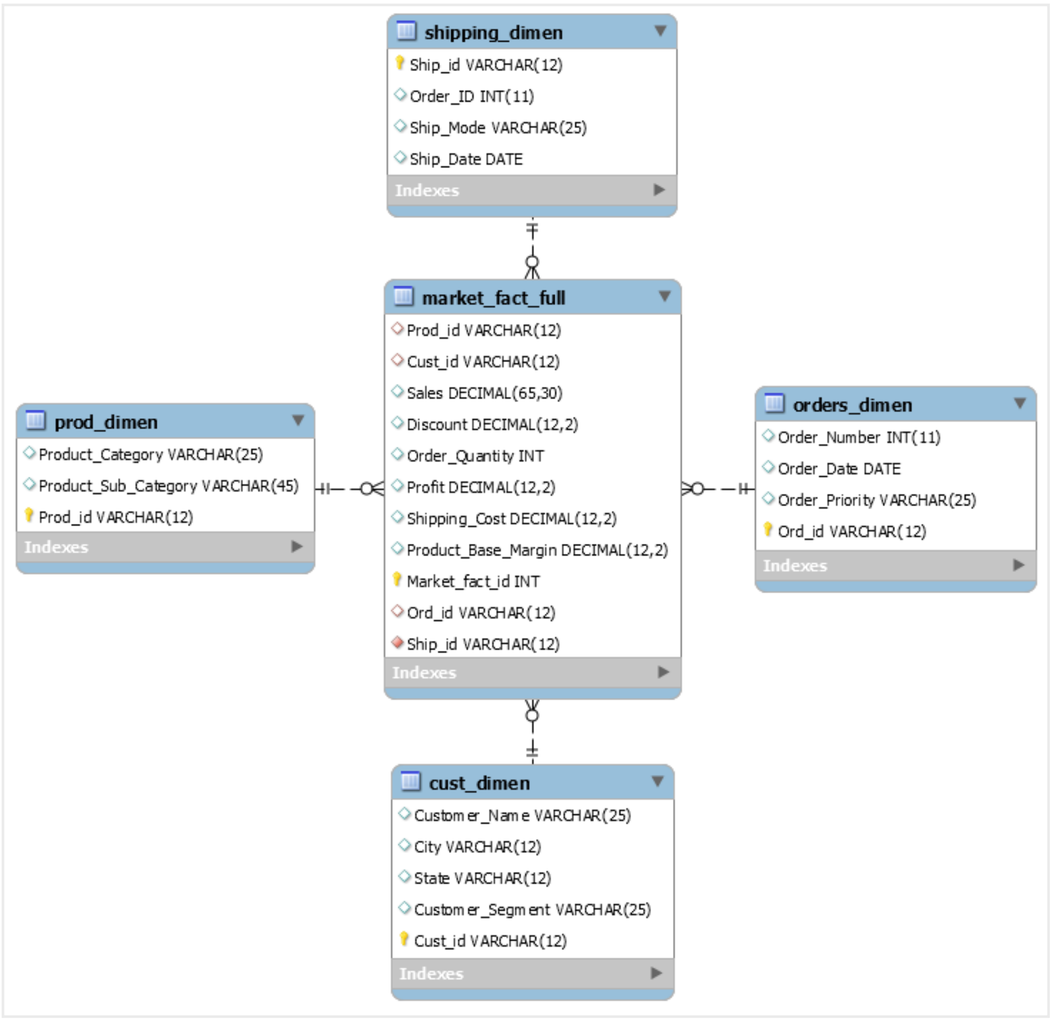


cust_id	rank	customer_name	profit	customer_city	customer_state	sales
	1					
	2					
	3					
	4					
	5					
	6					
	7					
	8					
	9					
	10					

Tables: The tables that are required for solving this problem are as follows:

- 'cust_dimen'
- 'market_fact_full'

The ERD for the 'market star' schema is given below for your reference.



In the next segment, you will learn how to identify ten of the most profitable customers and extract their details in the required format.

Profitable Customers - II:-

Now that you are well-versed with the problem statement and the format of the expected output table, in the upcoming video, Shreyas will walk you through the approach and the code that you will use to solve this problem statement.

After extracting the details of the most profitable customers in the required format, you can draw further insights into the city or state to which they belong. This can help you identify some of the major regions where your products are selling the most.

Customers Without Orders - I:-

In the previous segment, you performed profitability analysis as well as identified the most profitable customers. In this segment, you will be introduced to the final problem statement. Let's watch the upcoming video and learn more about this.

The problem statement, the required columns and the required tables introduced in this video can be summarised as follows:

Problem statement: Extract the required details of the customers who have not placed an order yet.

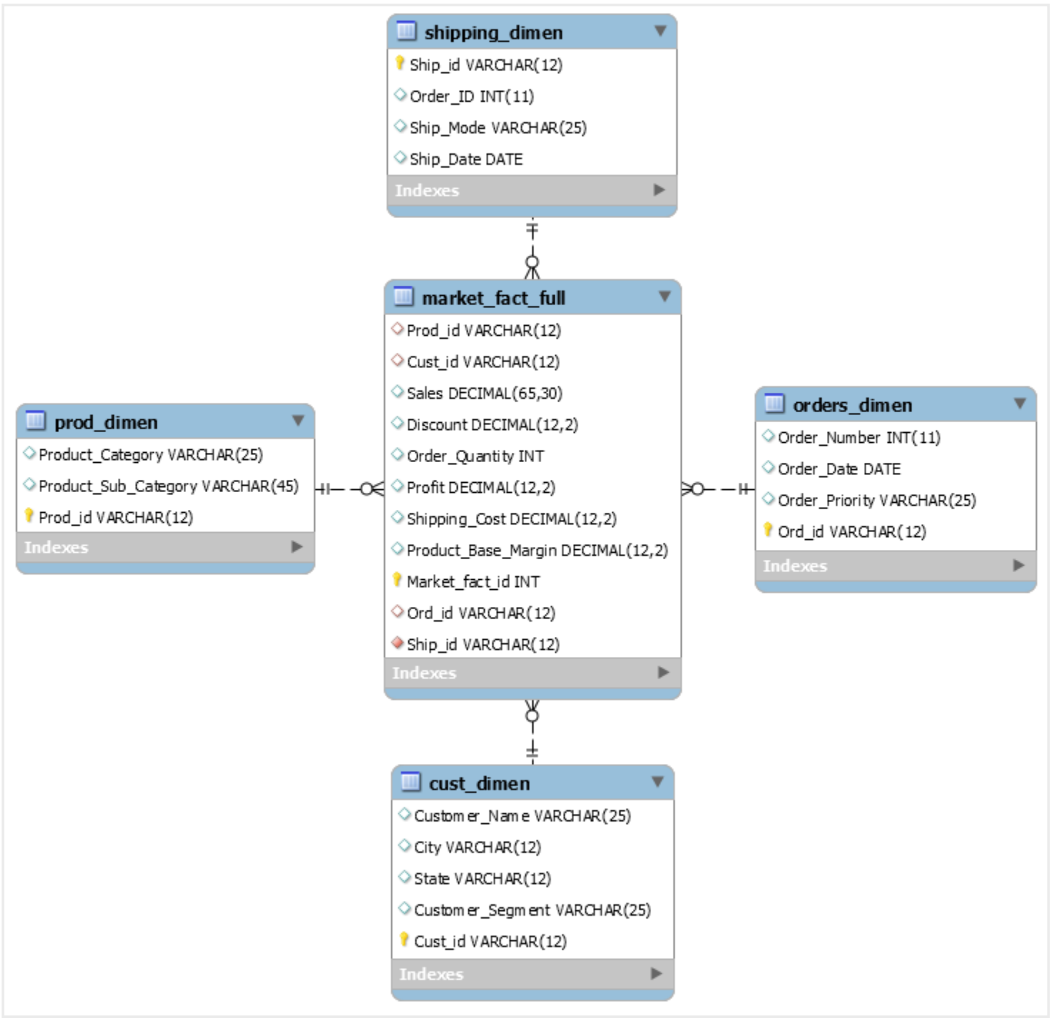
Expected columns: The columns that are required as the output are as follows:

- 'cust_id'
- 'cust_name'
- 'city'
- 'state'
- 'customer_segment'
- A flag to indicate that there is another customer with the exact same name and city but a different customer ID.

Tables: The tables that are required for solving this problem are as follows:

- 'cust_dimen'
- 'market_fact_full'

The ERD for the 'market star' schema is given below for your reference.



You now have an understanding of the problem statement. In the next segment, you will learn how to explore the tables and find out if there are any customers who haven't placed a single order.

Customers Without Orders - II:-

In the previous segment, you were introduced to the second problem statement and also understood the required columns that you need to output. Let's watch the upcoming video and learnt how to extract the details of the customers who have not placed a single order yet.

In this video, you learnt that are no such customers who have not placed a single order. In the next video, you will learn how to solve this problem statement by making slight modifications to it.

In this video, you learnt how to extract the details of the customers who have placed more than one order. In the next segment, you will learn how to extract the 'Fraud Flag' column and understand the importance of this column.

Fraud Detection:-

Suppose you are a senior manager at Domino's Pizza and your job is to manage operations in Mumbai. The company has come up with an attractive offer for new customers: each new customer who signs up can avail a discount of 75% on their first order with a maximum cap of ₹500. Everything seems to be going well in all the areas in Mumbai, except in Chembur. An unexpectedly large number of customers are signing up on a daily basis. What could be the reason for this?

In the upcoming video, Shreyas will continue solving the problem statement to determine if there are any fraud customers. Keep the above scenario in mind as you learn how to detect customer fraud in a company.

As you saw in this video, there were many customers who repeatedly tried to sign up to receive discounts on their orders. Recall the scenario that was described earlier in this segment and try to come up with a possible reason why a large number of customers in Chembur are signing up.

One of the reasons can be that the same customers who signed up earlier are using different mobile numbers to avail the 75% discount on their orders. You could prevent this by extracting the customer IDs that have the same email address, residential address and other such details. In such cases, there is a strong likelihood of fraud customers.

With this, we have come to the end of this session as well as this module. You were introduced to and solved three problem statements in this session. Let's summarise the learnings from this session in the final segment.

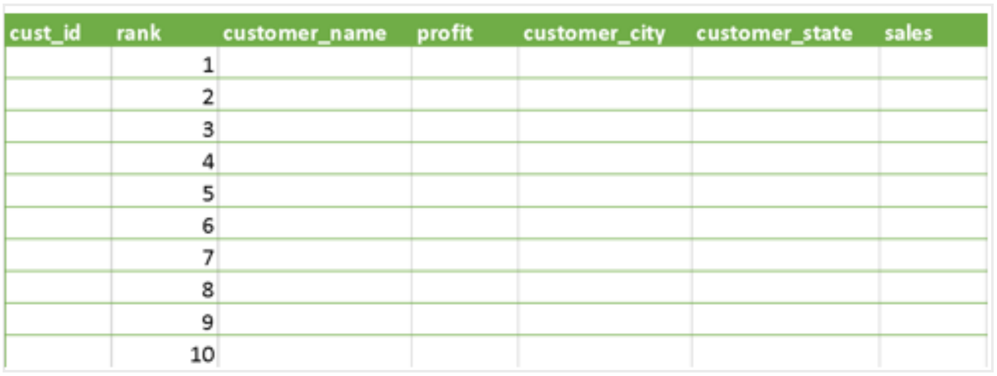
Summary:-

In this session, you solved various business problems using SQL. You might encounter similar problems in your daily job role.

Profitability analysis: You performed a profitability analysis on the 'market star' schema by identifying sustainable/profitable product categories so that the growth team can capitalise on them and increase sales. You calculated the following metrics:

- Profits per product category
- Profits per product subcategory
- Average profit per order
- Average profit percentage per order

Profitable customers: You also identified the ten most profitable customers and extracted their details in the format given below.



cust_id	rank	customer_name	profit	customer_city	customer_state	sales
	1					
	2					
	3					
	4					
	5					
	6					
	7					
	8					
	9					
	10					

Fraud detection: Finally, you found out the details of fraud customers after retrieving their data in the following columns:

- 'cust_id'
- 'cust_name'
- 'city'
- 'state'
- 'customer_segment'
- A flag to indicate that there is another customer with the exact same name and city but a different customer ID.

You must have realised that there are various applications of SQL in the data analytics domain. Hence, it is important for you to have a strong understanding of both basic and advanced SQL concepts to derive useful insights.

You can find the script file used in the demonstrations throughout this session in the link provided below.

Finally, you can download the PPT used throughout this module from the link provided below.