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>...}]

)

 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>...}]
* )

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.

he 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

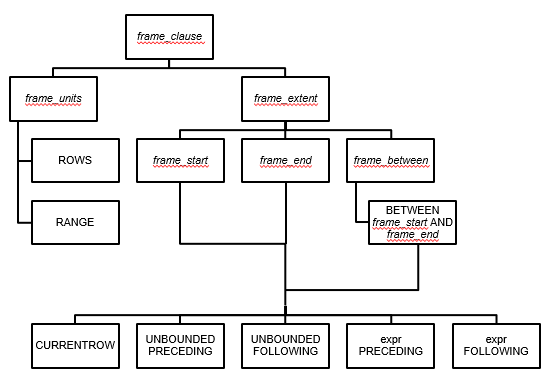
**Frames - I**

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](http://www.investopedia.com/articles/active-trading/052014/how-use-moving-average-buy-stocks.asp)

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.



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.

LEAD AND LAG Functions

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'.

* SUMMARY   
    
    
  **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>...}]
* )

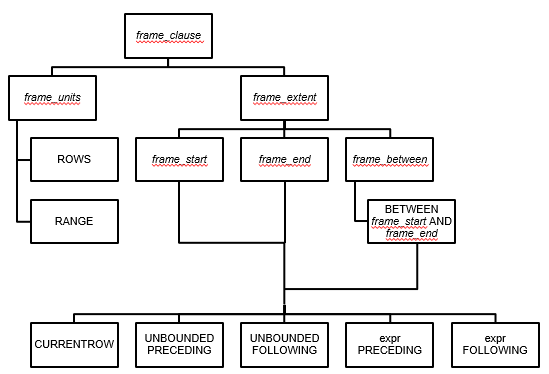
**Named windows:** A named window makes it easier to define and reuse multiple window functions. The syntax for a named window is as follows:

* 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.



Structure of a Frame

**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)

**CREATE** **FUNCTION** function\_name(func\_parameter1, func\_parameter2, ...)

**RETURN** datatype [**characteristics**]

/\* func\_body \*/

**BEGIN**

<**SQL** Statements>

**RETURN** expression;

**END** ; $$

**DELIMITER**

**DELIMITER** $$

**CREATE** **PROCEDURE** Procedure\_name (<Paramter List>)

**BEGIN**

<**SQL** Statements>

**END** $$

**DELIMITER** ;

**CALL** Procedure\_name;

| **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. |

* 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).

**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:**

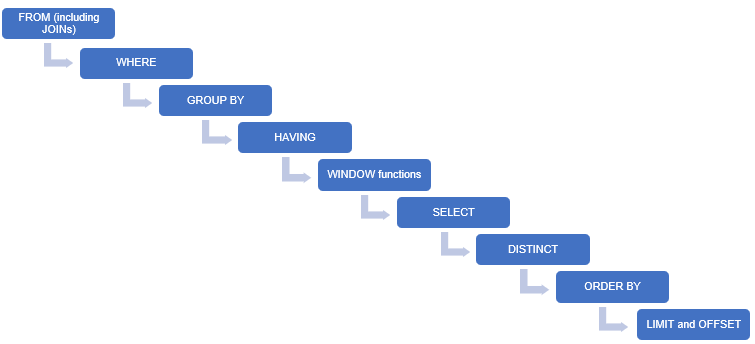
* **ALTER** **TABLE** **table\_name**
* **DROP** **INDEX** index\_name;

| **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. |

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

Order of execution”



* 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.