Variables:

Diagram

Description automatically generated

Local:

Graphical user interface, text, application, chat or text message

Description automatically generated

A picture containing diagram

Description automatically generated

Graphical user interface, text, application, email

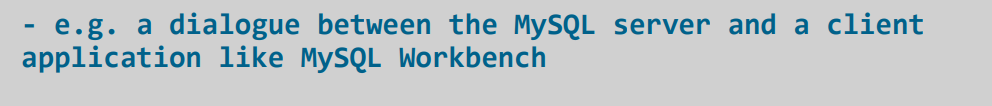
Description automatically generated

Graphical user interface, text, application, email

Description automatically generated

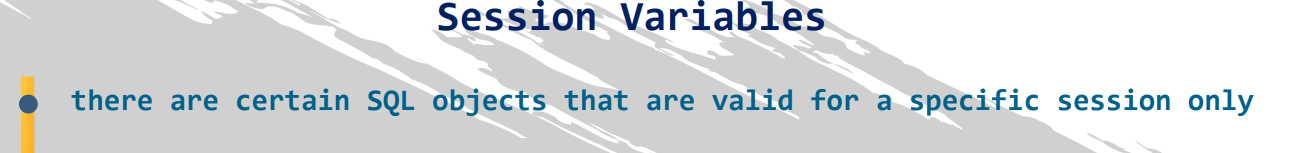
Graphical user interface, text, application

Description automatically generated



Timeline

Description automatically generated



Diagram

Description automatically generated

Graphical user interface, text, application

Description automatically generated

Graphical user interface, application, Word

Description automatically generated

Graphical user interface, text, application, email

Description automatically generated

Graphical user interface, text, application

Description automatically generated

Text, timeline

Description automatically generated

Table

Description automatically generated with medium confidence

**set global max\_connections =1000;**

**set @@global.max\_connections = 1;**

**User-Defined vs System Variables**

user-defined: variables that can be set by the user manually

system: variables that are pre-defined on our system – the MySQL server

set session max\_connections=1000;

Error:

**18:07:10 set session max\_connections=1000 Error Code: 1229. Variable 'max\_connections' is a GLOBAL variable and should be set with SET GLOBAL 0.000 sec**

Graphical user interface, text, application, email

Description automatically generated

**The Case Statement:**

Graphical user interface, text, application

Description automatically generated

**Example\_01:**

select emp\_no,first\_name,last\_name,

case

when gender='M' then 'Male'

else 'Female'

end as gender

from employees;

**Example-02:**

select emp\_no,first\_name,last\_name,

case gender

when 'M' then 'Male'

else 'Female'

end as gender

from

employees;

select \* from employees;

**Example-03:**

select emp\_no,first\_name,last\_name,

if(gender='M','Male','Female') as gender

from employees;

**Window Functions:**

The table presents the available window functions in MySQL along with a brief explanation of each:

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text, application, email

Description automatically generated

Table

Description automatically generated

* When running this query, which uses GROUP BY with the aggregate function SUM(), the number of rows in the result table is less than in the original input table.
* This is because the rows are treated as the operands in the addition operation. It returns one summary row for each car type.

**SELECT**

**Car,**

**SUM(Sold)**

**FROM CARS\_SOLD**

**GROUP BY Car;**

Table

Description automatically generated

On the other hand, the number of rows in the input and output tables is the same when we use a window function:

**SELECT**

**Car,**

**Country,**

**Sold,**

**SUM(Sold)**

**OVER (PARTITION BY Car) as SoldBrand**

**FROM CARS\_SOLD;**

Table

Description automatically generated

* The summary is returned in the SoldBrand column, where the total value of sold cars per brand is presented.
* Thus, the returned number of records equals the total number of rows in the input table.
* You may notice that the PARTITION BY clause in this example plays the same role as the GROUP BY clause in the previous query: it groups rows that have the same values in the Car column.

use employees;

create table CARS\_SOLD(

sale\_id int not null auto\_increment,

Car varchar(100) not null,

country varchar(100) not null,

sold int not null, primary key(sale\_id));

desc cars\_sold;

insert into cars\_sold values(1,'Audi','Germany',120),(2,'Audi','USA',110),(3,'Audi','Japan',100),(4,'BMW','Germany',250),(5,'BMW','USA',200),

(6,'BMW','Japan',200),(7,'Ford','Germany',260),(8,'Ford','USA',300),(9,'Ford','Japan',200);

select \* from cars\_sold;

When running this query, which uses GROUP BY with the aggregate function SUM(), the number of rows in the result table is less than in the original input table.

This is because the rows are treated as the operands in the addition operation. It returns one summary row for each car type.

SELECT

Car,

SUM(Sold)

FROM CARS\_SOLD

GROUP BY Car;

On the other hand, the number of rows in the input and output tables is the same when we use a window function:

SELECT

Car,

Country,

Sold,

SUM(Sold)

OVER (PARTITION BY Car) as SoldBrand

FROM CARS\_SOLD;

The summary is returned in the SoldBrand column, where the total value of sold cars per brand is presented. Thus, the returned number of records equals the total number of rows in the input table.

You may notice that the PARTITION BY clause in this example plays the same role as the GROUP BY clause in the previous query: it groups rows that have the same values in the Car column.

Table:

desc continents;

insert into continents values(1,'America'),(2,'Europe'),(3,'Asia');

select \* from continents;

Table:

**COUNTRIES**

**desc countries;**

**insert into countries values(1,'USA',1,90),(2,'Germany',2,95),(3,'France',2,80),(4,'Japan',3,60),(5,'Italy',2,75),(6,'Sweden',2,85);**

**select \* from countries;**

Table:

**Car\_makers**

desc car\_makers;

insert into car\_makers values(2,'Volkswagen', 2015, 2, 70000),(3, 'BMW', 2015, 2, 100000),(5, 'Ford Motor Company', 2015, 1, 80000),

(7, 'Citroen', 2015, 3, 50000),(8, 'Nissan Motors', 2017, 4, 50000),(9, 'Fiat', 2017, 5, 40000),

(11, 'Honda', 2017, 4, 40000),(13, 'Daimler Benz', 2017, 2, 110000),(14, 'Opel', 2017, 2, 65000),

(15, 'Peugeot', 2019, 3, 55000),(16, 'Renault', 2019, 3, 60000),(19, 'Toyota', 2019, 4, 45000),

(21, 'Volvo', 2019, 6, 75000);

**Example 1: The Ranking Window Functions**

* **RANK() Window Function:**
* **The RANK() window function assigns a ranking value to each row within the defined partition. It reinitializes the rank to start from 1 when the partition is switched.**
* **RANK() skips sequence numbers if the row value is repeated, i.e. the same rank is given to rows with the same values.**

**SELECT**

**Maker,**

**Price,**

**RANK()**

**OVER (ORDER BY Price) as RankValue**

**FROM CAR\_MAKERS;**

Table

Description automatically generated

**Explanation:**

* In this example, we are ordering the result using the Price column and treating all table rows as one partition.
* The first row is assigned to the rank value 1. In the second row, the rank value is 1 again because the Price column value is the same for the first and second row.
* In the third row, the rank value is 3 (because of the skipped number due to the repeated Price column values); the value is 3 because this is the third row. In the fourth row, the rank value is 4 as expected.
* Once again, two rows share the same Price value, and thus the rank 4 is repeated. This principle is applied throughout all the rows in the example.

Example-02:

* Next, let’s look at an example of the RANK() window function that partitions the result by the ProductionYear column and then orders each partition according to the Country column:

Table

Description automatically generated

We placed the PARTITION BY clause inside the OVER() clause. This divides the result into 3 groups:

* Group 1 with a ProductionYear of 2015.
* Group 2 with a ProductionYear of 2017.
* Group 3 with a ProductionYear of 2019.

Similar partitioning by the ProductionYear column takes place in the upcoming examples of this article.

In this example, each group (partition) has its rank values started from 1. The rank values are assigned using the same working principle as described in the previous example.

**DENSE\_RANK() Window Function:**

The DENSE\_RANK() window function is very similar to the RANK() function. The only difference is that it does not skip any numbers in the rank sequence.

SELECT

Maker,

ProductionYear,

Price,

DENSE\_RANK()

OVER (PARTITION BY ProductionYear ORDER BY Price)

as DenseRankValue

FROM CAR\_MAKERS;

Table

Description automatically generated

* In the seventh row of the above output table, the DenseRankValue column value is 2 – although it is in the third row of its partition.
* The DENSE\_RANK() function does not skip the rank value 2 , which was not used in the second row due to row value repetitions.
* On the other hand, RANK() would use the rank value 3 in the third row of the red partition.

**ROW\_NUMBER() Window Function**

* The ROW\_NUMBER() window function does exactly what its name says. It assigns a row number to each record within the partition; it reinitializes row numbers to start from 1 when the partition is switched.
* First, let’s look at an example that does not use the PARTITION BY clause within the OVER() clause. Please note that although the OVER() clause is empty, you still have to define it with the window function:

**SELECT**

**CountryName,**

**Continent,**

**ROW\_NUMBER() OVER () as RowNumberValue**

**FROM COUNTRIES;**

Table

Description automatically generated

* As we do not partition the result set, the output is very straightforward. The rows are numbered from 1 to 6.
* Once we partition the output table, there will be more than one set of row numbers. Please note that the ORDER BY clause is optional in both of these examples.

Table

Description automatically generated

Notice that rows are numbered from 1 for each of the partitioned groups.

**PERCENT\_RANK() Window Function:**

The PERCENT\_RANK() window function returns the value from 0 to 1 (both inclusive), which indicates the percentage of current partition rows with a value less than the current row value.

This will become clearer with an example. Have a look:

SELECT

Maker,

ProductionYear,

Price,

PERCENT\_RANK()

OVER (PARTITION BY ProductionYear ORDER BY Price)

as PercentValue

FROM CAR\_MAKERS;

Table

Description automatically generated

* Let’s go through the output table starting from the first row. In the first partition, there is no Price column value that is smaller than 50000. Hence, the PercentValue column value is 0, i.e. 0% of the current partition’s Price values are smaller than the current row’s Price value.
* In the case of the second row, 1/3 of the Price values in the current partition are smaller than 70000. So, the PercentValue value of this row is 0.333(or 33%).
* In the last row of this partition, we see that all the Price values are smaller than 100000. Thus, the PercentValue value is 1, i.e. 100% of the current partition’s Price values are smaller than the current Price value.
* You can apply the same working principle to go through the remaining two partitions of the above example.

**Example 2: The Value Window Functions**

**LEAD() Window Function**

* The LEAD() window function allows us to look up the values of subsequent rows in the current partition.
* It is commonly used to calculate the difference between current and following row values.
* First, let’s look at the example that simply uses the LEAD() function to output the value of the row after the current one:

**SELECT**

**Maker,**

**ProductionYear,**

**Price,**

**TotalCountrySales,**

**LEAD(TotalCountrySales, 1, 0)**

**OVER (PARTITION BY ProductionYear ORDER BY Price)**

**as LeadValue**

**FROM CAR\_MAKERS cm JOIN COUNTRIES c**

**ON cm.Country = c.CountryID;**

Table

Description automatically generated

**The LEAD() window function takes three arguments:**

* The column value to be returned – in this case, it is the TotalCountrySales column.
* The row number (relative to the current row) to be looked up – here, LEAD() looks up the value of the row immediately after the current row.
* A default value that’s returned if there is no subsequent row – in the current example, the LeadValue in the fourth row is 0 because there are no more rows in this partition.
* In the output, the first row has a LeadValue equal to 95 – the TotalCountrySales of the second row. The second row has a LeadValue of 90, the TotalCountrySales of the third row, and so on. As I already mentioned, the LeadValue of the fourth row is 0 (the default value we defined); although there are subsequent rows in the table, the fourth row is the last row in the current partition. This process is repeated for each partition of the resulted table.

Now let’s try to calculate the TotalCountrySales difference between the current and subsequent rows.

SELECT

Maker,

ProductionYear,

Price,

TotalCountrySales,

LEAD(TotalCountrySales , 1, 0)

OVER (PARTITION BY ProductionYear ORDER BY Price)

as LeadValue,

TotalCountrySales - (LEAD(TotalCountrySales , 1, 0)

OVER (PARTITION BY ProductionYear ORDER BY Price))

as Difference

FROM CAR\_MAKERS cm JOIN COUNTRIES c

ON cm.Country = c.CountryID;

A screenshot of a computer

Description automatically generated with low confidence

* Except for the Difference column, this example is similar to the previous one. This column stores the difference between the TotalCountrySales value of the current row (column TotalCountrySales) and the TotalCountrySales value of the next row (column LeadValue).
* Please note that the LEAD() window function is not limited to getting the value from the immediate next row. Let’s see an example where LEAD() is used to get a value from the third row after the current row. In this example, we do not partition the result table.

SELECT

Maker,

ProductionYear,

Price,

LEAD(TotalCountrySales, 3)

OVER (ORDER BY Price)

as LeadValue

FROM CAR\_MAKERS cm JOIN COUNTRIES c

ON cm.Country = c.CountryID;

A picture containing text, indoor, white, screenshot

Description automatically generated

* Starting from the first row of the result table, each row gets a value of the 3rd row following it. Please note that the last 3 rows values are null because there is no value for the 3rd row following them.

**LAG() Window Function**

The LAG() window function is the opposite of the LEAD() function. It returns the value from a previous row.

Let’s see it in action. The following example returns a LagValue and calculates the difference between the current row and the previous row:

**SELECT**

**Maker,**

**ProductionYear,**

**Price,**

**TotalCountrySales,**

**LAG(TotalCountrySales, 1, 0)**

**OVER (PARTITION BY ProductionYear ORDER BY Price)**

**as LagValue,**

**TotalCountrySales - (LAG(TotalCountrySales, 1, 0)**

**OVER (PARTITION BY ProductionYear ORDER BY Price))**

**as Difference**

**FROM CAR\_MAKERS cm JOIN COUNTRIES c**

**ON cm.Country = c.CountryID;**

A screenshot of a computer

Description automatically generated with medium confidence

* The example should speak for itself, as it is analogous to the example used for LEAD(). Here the values are looked up from the previous row; this is opposite of the LEAD() function, where the values are looked up from the subsequent row.
* Like LEAD(), the LAG() window function can fetch the value of any row preceding it – not just the immediately preceding row.

**FIRST\_VALUE() Window Functions**

This window function outputs the first value of the current partition. This will be clear after looking at the example below.

**SELECT**

**Maker,**

**ProductionYear,**

**Price,**

**TotalCountrySales,**

**FIRST\_VALUE(TotalCountrySales)**

**OVER (PARTITION BY ProductionYear ORDER BY Price)**

**as FirstValue**

**FROM CAR\_MAKERS cm JOIN COUNTRIES c**

**ON cm.Country = c.CountryID;**

A screenshot of a computer

Description automatically generated with medium confidence

* Let’s analyze the first partition in this output table.
* The FirstValue column contains an 80 for all rows; this is the TotalCountrySales column value of the first row in this partition.

**Example 3: Using Aggregate Functions with OVER()**

Our next example demonstrates the usage of the aggregate functions SUM(), COUNT(), and AVG() with the OVER() clause. Using these functions with OVER() turns them into window functions.

Please note that the OVER() clause is mandatory for all window functions, regardless of whether it is empty or not.

**SELECT**

**Maker,**

**ProductionYear,**

**Country,**

**TotalCountrySales,**

**SUM(TotalCountrySales)**

**OVER (PARTITION BY ProductionYear) as SumValue,**

**COUNT(TotalCountrySales)**

**OVER (PARTITION BY ProductionYear) as CountValue,**

**AVG(TotalCountrySales)**

**OVER (PARTITION BY ProductionYear) as AvgValue**

**FROM CAR\_MAKERS cm JOIN COUNTRIES c**

**ON cm.Country = c.CountryID;**

Calendar

Description automatically generated with medium confidence

* In the above example, the sum, count, and average values are calculated using the values in the current partition. It is also worth mentioning that they are calculated for each partition separately.
* Let’s analyze the first partition to see what’s going on:
  + The SumValue column stores the sum of all TotalCountrySales values in the current partition (i.e. 95+95+90+80=360).
  + The CountValue column stores the number of records in the current partition (i.e. 4).
  + The AvgValue column stores the average of TotalCountrySales column values in the current partition (i.e. (95+95+90+80)/4=90).

**Example 4: NTILE() and CUME\_DIST() Window Functions**

* The NTILE() window function divides partition rows into buckets. It takes the parameter n that the user places in the function brackets and creates that number of buckets. Thus, n groups/buckets are created for each partition and the appropriate bucket number is assigned to each row.
* Like a few of the other concepts we’ve talked about, this will be clearer after you see an example. Suppose you want to divide car makers into two groups for each production year. This is the query you’d use:

**SELECT**

**Maker,**

**ProductionYear,**

**NTILE(2)**

**OVER (PARTITION BY ProductionYear ORDER BY Maker)**

**as NtileValue**

**FROM CAR\_MAKERS cm JOIN COUNTRIES c**

**ON cm.Country = c.CountryID;**



* The column NtileValue contains the bucket number – which is either 1 or 2 – for each row in each partition.
* The NTILE window function is used to divide the result table rows into groups. In the example above, we partition the result table into 3 partitions and each partition’s rows are divided into 2 groups because the query uses NTILE(2).

**The CUME\_DIST() Window Function**

* The CUME\_DIST() window function – as its name indicates – calculates the cumulative distribution value for each row in a partition.
* Cumulative distribution is the number of rows that have a value that’s less than or equal to the current row value divided by the number of all rows within the partition.
* The example below will help clear up this concept:

**SELECT**

**Maker,**

**ProductionYear,**

**Country,**

**TotalCountrySales,**

**CUME\_DIST()**

**OVER (PARTITION BY ProductionYear ORDER BY TotalCountrySales) as CumeDistValue**

**FROM CAR\_MAKERS cm JOIN COUNTRIES c**

**ON cm.Country = c.CountryID;**



* Let’s analyse the first partition as an example. It includes TotalCountrySales column values of 80, 90, 95, and 95.
* The CumeDistValue column values of the first four rows can be explained as follows:
  + The first row has a TotalCountrySales column value of 80. This is the smallest TotalCountrySales value in the current partition; hence, 1/4=0.25.
  + The second row has a TotalCountrySales value of 90. There are two values less than or equal to 90; hence, 2/4=0.5.
  + The third and fourth rows have a TotalCountrySales value of 95. This is the highest in this partition, so 4/4=1.
* The CUME\_DIST window function returns the cumulative distribution value. It is used mostly for data analysis.

**Cursors:**

In MySQL, a cursor allows row-by-row processing of the result sets.

A cursor is used for the result set and returned from a query.

By using a cursor, you can iterate, or step through the results of a query and perform certain operations on each row.

The cursor allows you to iterate through the result set and then perform the additional processing only on the rows that require it.

A cursor contains the data in a loop. Cursors may be different from SQL commands that operate on all the rows in the returned by a query at one time.

MySQL cursor has three types of properties.

**1. Read Only**

The data in the underlying table cannot be modified via a cursor.

**2. Non\_Scrollable**

Only rows can be retrieved in the order specified by the SELECT statement. In the reverse order, users can not retrieve records. Additionally, in the result set, users cannot skip rows or jump to a particular row.

**3. Asensitive**

An asensitive cursor is used to points the actual data, whereas a temporary copy of the data is used by an insensitive cursor used. An asensitive cursor performs faster than an insensitive cursor because it does not have to make a temporary copy of data.

**Working with MySQL Cursors**

There are some steps we must follow while using the MySQL Cursors, let’s see.

* Declare a Cursor
* Open a Cursor
* Fetch the Cursor
* Close the Cursor

1. Declaration of a Cursor

To declare a cursor, you must use the DECLARE statement. With the help of the variables, we need conditions and handlers to declare a cursor before we can use it.

First of all, we will give the cursor a name, this is how we will refer to it later in the procedure. We can have more than one cursor in a single procedure so it’s necessary to give it a name that will in some way tell us what it’s doing.

We then need to specify the select statement we want to associate with the cursor. The SQL statement can be any valid SQL statement and it is possible to use a dynamic where clause using variable or parameters as we have seen previously.

Syntax

DECLARE <cursor\_name> CURSOR FOR <select\_statement>;

**2. Open a Cursor**

For opening a cursor we must use the open statement. If we want to fetch rows from it, then you must have to open the cursor.

**Syntax**

OPEN <cursor\_name>;

3. Fetch the Cursor

When we have to retrieve the next row from the cursor and move the cursor to the next row then you need to fetch the cursor. If any row exists, then the below statement fetches the next row and cursor pointer moves ahead to the next row.

**Syntax**

FETCH <cursor\_name> INTO <variable\_list>;

**4. Close the Cursor**

This statement closes the open cursor, and it will deactivate the cursor and release the memory. By this statement we can close the previously opened cursor.

If it is not closed explicitly then a cursor is closed at the end of compound statement in which that was declared.

**Syntax**

CLOSE <cursor\_name>;

**Example\_01:**

use employees;

DELIMITER $$

drop function if exists FindSiteID;

CREATE FUNCTION FindSiteID (name\_in VARCHAR(50)) RETURNS INT

no sql

BEGIN

DECLARE done INT DEFAULT FALSE;

DECLARE DID INT DEFAULT 0;

DECLARE c1 CURSOR FOR

SELECT emp\_no

FROM employees

WHERE first\_name = name\_in;

DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = TRUE;

OPEN c1;

FETCH c1 INTO DID;

CLOSE c1;

RETURN DID;

END$$

DELIMITER ;

select FindSiteID('Mary');

**Example 2:**

CREATE TABLE GetVatsaCursor(

C\_ID INT PRIMARY KEY AUTO\_INCREMENT,

c\_name VARCHAR(50),

c\_address VARCHAR(200)

);

CREATE TABLE Vbackupdata(

C\_ID INT,

c\_name VARCHAR(50),

c\_address VARCHAR(200)

);

INSERT INTO GetVatsaCursor(c\_name, c\_address) VALUES('Test', '132, Vatsa Colony'),

('Admin', '133, Vatsa Colony'),

('Shareef', '134, Vatsa Colony'),

('Vedant', '135, Vatsa Colony'),

('Harish', '136, Vatsa Colony'),

('Sahoo', '137, Vatsa Colony'),

('Wasim', '138, Vatsa Colony'),

('Baharathi', '139, Vatsa Colony'),

('Srinithi', '140, Vatsa Colony');

SELECT \* FROM GetVatsaCursor;

SELECT \* FROM Vbackupdata;

delimiter $$

CREATE PROCEDURE firstCurs()

BEGIN

DECLARE d INT DEFAULT 0;

DECLARE c\_id INT;

DECLARE c\_name, c\_address VARCHAR(20);

DECLARE Get\_cur CURSOR FOR SELECT \* FROM GetVatsaCursor;

DECLARE CONTINUE HANDLER FOR SQLSTATE '02000'

SET d = 1;

DECLARE CONTINUE HANDLER FOR SQLSTATE '23000'

SET d = 1;

OPEN Get\_cur;

lbl: LOOP

IF d = 1 THEN

LEAVE lbl;

END IF;

IF NOT d = 1 THEN

FETCH Get\_cur INTO c\_id, c\_name, c\_address;

INSERT INTO Vbackupdata VALUES(c\_id, c\_name, c\_address);

END IF;

END LOOP;

CLOSE Get\_cur;

END$$

delimiter ;

call firstCurs();

SELECT \* FROM Vbackupdata;