Hello everyone and welcome to this video. I hope all of you are doing good , here we are back with another MySQL project session and this session is going to be very interesting . We will be getting into few core SQL query concepts that we have. Primarily we will be learning about window functions and the how it helps us.

Firstly we will work with a couple of tables one will be where employee and their salary details will be present and one will be there where different brands of laptop , mobile smartwatch etc details are given and with the help of window function we will solve many real life problem scenarios.

We will perform window functions on a newly created table and the table creation query will look as given

Schema Name – **emp\_table\_window**

CREATE TABLE `emp\_table\_window`.`employee\_details` (

`emp\_id` INT NOT NULL,

`emp\_name` VARCHAR(50) NOT NULL,

`emp\_department` VARCHAR(45) NOT NULL,

`emp\_salary` INT NOT NULL,

PRIMARY KEY (`emp\_id`));

Now suppose , initially we want to find out the max salary drawn by the employees present in the table , we simply can use the max function , which is an aggregate functionality present in SQL , the code will be as below –

SELECT max(emp\_salary) FROM employee\_details;

This will simply fetch us the maximum salary present in the table , to get max salary drawn by each department we can use a group by clause here like below –

SELECT emp\_department ,max(emp\_salary)

FROM employee\_details

GROUP BY emp\_department;

But a simple way to get all the details of the employees alongside their salary department wise , we can use a ‘Over’ clause , this will turn the ‘max’ function from a aggregate function to a window function. Let’s have a look

SELECT \* , max(emp\_salary)

OVER() as max\_salary

FROM employee\_details;

As we can see that the over clause here has created a new window that produced result based on the maximum salary present in the whole table and that max amount is displayed in the new window.

Now in the same query if we want to extract the maximum salary present corresponding to each department, we can simply use a partition by clause inside our over clause argument as below –

SELECT \* , max(emp\_salary)

OVER(PARTITION BY emp\_department) as max\_salary

FROM employee\_details;

As we can see , now the max salary window has a different value for each department as they represent the value corresponding to their maximum salary.

For the same query where we have used max as a parameter, we can use other aggregate functionalities as min , count , average as well –

SELECT \* , min(emp\_salary)

OVER(PARTITION BY emp\_department) as min\_salary

FROM employee\_details;

SELECT \* , COUNT(emp\_salary)

OVER(PARTITION BY emp\_department) as Count\_salary

FROM employee\_details;

SELECT \* , AVG(emp\_salary)

OVER(PARTITION BY emp\_department) as Average\_salary

FROM employee\_details;

Let’s talk about other window functions , starting with row number ,

**Row Number –**

The row number function actually helps to identify each column developed with a newly created row number for them , let’s have a look at the query below –

SELECT \* ,

ROW\_NUMBER() OVER() AS RN

FROM employee\_details ;

As we can see a newly created column called RN is displaying the row numbers developed for each record present in the table , in this table as there is already a column present named ‘emp\_id’ which is also defining each row record , we can compare both of them.

Now if in the same query , we assign a column name in the over clause argument , then specific sequential row numbers will be assigned to each of the record specific to that particular column , suppose if we provide ‘emp\_department’ as an over clause argument then each of the department will have their own specific row numbers. The query will look like as below –

SELECT \* ,

ROW\_NUMBER() OVER(PARTITION BY emp\_department) AS RN

FROM employee\_details ;

As we can see above , each of the records are segregated based on the column name given in the over clause , which is emp\_department in this case.

Now suppose we want to get our data sorted based on the salaries of the employees of each department ,we can add a order by clause inside of our over argument , the query will look as like below –

SELECT \* ,

ROW\_NUMBER()

OVER(

PARTITION BY emp\_department

ORDER BY emp\_salary DESC

)

AS RN

FROM employee\_details ;

As we can see above salaries of each departments are sorted from highest to lowest in order.

Now that highest salaried employees are sorted , we can use it as a sub query to get only the highest paid employees in each department , because in each department their RN will be ‘1’ , The query will look like as below , X is defined here as the result of subquery for which the RN is sorted. –

SELECT \* FROM(SELECT \* ,

ROW\_NUMBER()

OVER(

PARTITION BY emp\_department

ORDER BY emp\_salary DESC )

AS RN

FROM emp\_table\_window.employee\_details) X

WHERE X.RN = 1;

With removing DESC from our query we can as well find the lowest salaried people

SELECT \* FROM(SELECT \* ,

ROW\_NUMBER()

OVER(

PARTITION BY emp\_department

ORDER BY emp\_salary)

AS RN

FROM emp\_table\_window.employee\_details) X

WHERE X.RN = 1;

**Rank Function –**

With the help of rank function we can achieve the ranking of employees based on given arguments , suppose we want to find out the ranking of employees based on their salary above , we can go ahead and use the rank function , but the only difference here than using the row\_number function here is that , in Rank method if our programme finds a duplicate value it issues both of the value the same rank , but while assigning the rank to it’s next value it skips a rank. Let’s have a look below at the query –

SELECT e.\* ,

RANK() OVER(PARTITION BY emp\_department ORDER BY emp\_salary) AS Rnk

FROM employee\_details e;

As you can see in the output Chandan , Palash & Sakshi are from IT department , where Chandan and palash draws the same salary so for both of them rank is given as 1 , but for the next value sakshi’s name was assigned although the rank skipped a value and given as 3.

Now converting this into a subquery we can actually find the top 2 employees in terms of salary drawing , The query will look like as below ,

SELECT \* FROM

(SELECT e.\* ,

RANK() OVER(PARTITION BY emp\_department ORDER BY emp\_salary DESC) AS Rnk

FROM employee\_details e) X

WHERE X.rnk<3;

Now let’s talk about dense rank, It is almost similar to rank function only, the difference that lies here is , it doesn’t skip a value for every duplicate value executed . Let’s take the same example

SELECT e.\* ,

DENSE\_RANK() OVER(PARTITION BY emp\_department ORDER BY emp\_salary) AS D\_Rnk

FROM employee\_details e;

**The Comparison –**

To get a better clarity , let’s run down a query which will pop up three windows , for row\_number , rank and dense\_rank , it’ll give us a better sense of comparison ,

**The Query –**

SELECT e.\* ,

ROW\_NUMBER() OVER(PARTITION BY emp\_department ORDER BY emp\_salary) AS Row\_num,

RANK() OVER(PARTITION BY emp\_department ORDER BY emp\_salary) AS Rnk,

DENSE\_RANK() OVER(PARTITION BY emp\_department ORDER BY emp\_salary) AS Dense\_Rnk

FROM employee\_details e;

As it can clearly be seen , row number gets assigned sequentially irrespective of the fact that the values are duplicate or not , in ranking duplicate values get the same rank but while coming to the next value , it skips one value between , in dense rank , after duplicate values , no value gets skipped while ranking the next value.

**Lag Function –**

This function is basically used to provide a argument for which we want to fetch the previous record , if salary is given here as an argument in Lag , we will be displaying the salary for the previous person in the row records. Let’s have a look at the query –

SELECT e.\* ,

LAG(emp\_salary) OVER (PARTITION BY emp\_department ORDER BY emp\_id) AS previous\_salary

FROM employee\_details e;

The lag function takes two argument into consideration, the first one as given here as 2 denotes how much backwards it needs to go , like 1 previous employee or 2 previous employee etc , and the 0 given here denotes , instead of ‘null’ we would like to display 0.

SELECT e.\* ,

LAG(emp\_salary ,2,0)

OVER (PARTITION BY emp\_department ORDER BY emp\_id) AS previous\_salary

FROM employee\_details e;

**LEAD Function –**

Opposite to the Lag function , the lead function displays the next values , and it also takes two arguments as LAG function itself.

|  |  |
| --- | --- |
| **LEAD Function Without Argument** | **LEAD Function With Argument** |
| SELECT e.\* ,  LEAD(emp\_salary)  OVER (  PARTITION BY emp\_department  ORDER BY emp\_id  )  AS next\_salary  FROM employee\_details e; | SELECT e.\* ,  LEAD(emp\_salary,2,0)  OVER (  PARTITION BY emp\_department  ORDER BY emp\_id  )  AS next\_salary  FROM employee\_details e; |

**Comparison –**

Let’s go ahead and find a comparison between LEAD and LAG here by using both of the query together ,

SELECT e.\* ,

LAG(emp\_salary,1,0) OVER (PARTITION BY emp\_department ORDER BY emp\_id) AS previous\_salary ,

LEAD(emp\_salary,1,0) OVER (PARTITION BY emp\_department ORDER BY emp\_id) AS next\_salary

FROM employee\_details e;

Let’s go ahead and create a use case where based on the previous salary statement we will display if the salary is higher or lower. We will be using LAG functionality alongside with Case statements –

SELECT e.\* ,

LAG(emp\_salary,1,0) OVER (PARTITION BY emp\_department ORDER BY emp\_id) AS previous\_salary ,

CASE

WHEN e.emp\_salary > LAG(emp\_salary) OVER (PARTITION BY emp\_department ORDER BY emp\_id) THEN 'Higher'

WHEN e.emp\_salary < LAG(emp\_salary) OVER (PARTITION BY emp\_department ORDER BY emp\_id) THEN 'Lower'

WHEN e.emp\_salary = LAG(emp\_salary) OVER (PARTITION BY emp\_department ORDER BY emp\_id) THEN 'Same'

END sal\_range

FROM employee\_details e;

Now let’s use another schema called ‘Gadget’ , This contains product category , brand name and values of different products. We will look into few other Window functions now ,

**First\_Value –**

With the help of first\_value window function we will be able to extract the most expensive product inside of each category . The query looks like below , The first\_value query takes one argument in it ,in here it will product\_name will be the argument as stated , we are trying to find out the most expensive product in each category. We will use partition by with the over clause to segregate the table based on the product category,

SELECT p.\* ,

FIRST\_VALUE(Name) OVER(PARTITION BY Category ORDER BY Price DESC) AS most\_expensive

FROM product p;

As we can see that in the newly created most\_expensive , we see the product name , as that’s the argument given inside First\_Value argument , had it been given as Brand we would have seen brand name in the newly created column as below ,

SELECT p.\* ,

FIRST\_VALUE(Brand) OVER(PARTITION BY Category ORDER BY Price DESC) AS most\_expensive

FROM product p;

**Last\_Value –**

Similarly , as the name suggests , we can find out the lest expensive product with the help of last\_value query ,

SELECT p.\* ,

LAST\_VALUE(Name) OVER(PARTITION BY Category ORDER BY Price DESC) AS least\_expensive

FROM product p;

**FRAME Clause –**

But as we can see , this query is not providing us with the proper results , we are not getting a single output for only least expensive product , this is happening due to the frame clause of SQL. , Frame clause is generally defined within the over clause , at the end of it , as of our previous query , it’s placed after the desc keyword. The default frame clause is denoted as , ‘RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW’. As we can see below , it’s fetching us the same result , as even if we don’t write this in our query , it remains the same –

SELECT p.\* ,

LAST\_VALUE(Name) OVER(PARTITION BY Category ORDER BY Price DESC

RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) AS least\_expensive

FROM product p;

Let’s try to understand the Frame clause , each partition that’s getting created , window function is processing the partitions by each record present in it , for the first record unbounded preceding value and current row value remains the same , hence it gets displayed , but from next value as it has a preceding row record and itself is a current row , hence our query is not getting access to all records at once as the range is getting stopped at current row , we can change our query on the ‘current row’ part to get proper execution of ‘Last\_value’.

SELECT p.\* ,

LAST\_VALUE(Name) OVER(PARTITION BY Category ORDER BY Price DESC

RANGE BETWEEN UNBOUNDED PRECEDING AND **UNBOUNDED FOLLOWING**) AS least\_expensive

FROM product p;

**ROWS & RANGE Functionality in Frame Clause** –

Within the Frame clause , we can use the rows /ranges function , the difference lies there is , if there are duplicate values present , with rows function being provided , it’ll execute and display all duplicate row , with separate values (as given in the argument of last\_value clause) while the range function will only consider the last record stored in it and will only display that.

SELECT p.\* ,

LAST\_VALUE(Name)

OVER(PARTITION BY Category ORDER BY Price DESC RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)

AS least\_expensive\_range ,

LAST\_VALUE(Name)

OVER(PARTITION BY Category ORDER BY Price DESC ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)

AS least\_expensive\_row

FROM product p;

As we can see , with a duplicate value in price , in range function we are only getting the last stored record (S5e) while in row function both duplicate values (IPad Mini and S5e is displayed).

With the help of range function , it can also be directed that how many records prior(preceding) and how many records after (following) we want to access and work on –

OVER(PARTITION BY Category ORDER BY Price DESC ROWS BETWEEN 2 PRECEDING AND 2 FOLLOWING)

**Alternate Query Writing for Windows Function** –

In our previous query where we searched for first and last value from our table , we have seen that the over clause write up the exact same , in many window functions it remains the same , so instead of writing the over clause again and again we can take up an alias to write an effective query as below –

SELECT p.\* ,

FIRST\_VALUE(Name) OVER w AS expensive\_product,

LAST\_VALUE(Name) OVER w AS cheap\_product

FROM product p

WINDOW w AS(PARTITION BY Category ORDER BY Price DESC

RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) ;

**nth Value Function** –

As the name suggests , it takes a position as argument alongside of the column name it needs to show the value for , for an example with first value we have seen the most expensive product in each category , with a given position of 2 , we can proceed and find the second most expensive product in the category defined.

SELECT p.\* ,

NTH\_VALUE(Name ,2 ) OVER w AS second\_most\_expensive\_product

FROM product p

WINDOW w AS(PARTITION BY Category ORDER BY Price DESC

RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) ;

**Ntile Function –**

Ntile function basically helps to segregate and put the records in a equal sized buckets based on given arguments , Ntile function itself takes one argument , which is the number of buckets to be created , let’s say for our table we want to create 3 buckets from . expensive , mid range and cheap phones. So our ntile function will take argument as ‘3’ , as 3 buckets are being created. The over clause now will not take any partition by clause as we are not partitioning our data based on products or anything , as we only want the range specification for phones , we will use a where clause for that , and as price is the basis on which we are performing our range calculation , a order by clause will follow in over clause , lets have a look at the query –

SELECT \*,

NTILE(3) OVER (ORDER BY Price DESC) AS 'Buckets'

FROM product

WHERE Category = 'Phone’;

Now if we go ahead with a partition by input inside of the over clause , then we will be able to get the range brackets of phone based on their specific brands ,

SELECT \*,

NTILE(3) OVER (PARTITION BY Brand ORDER BY Price DESC) AS 'Buckets'

FROM product

WHERE Category = 'Phone';

Now , let’s go ahead and combine a case when statement with our query , it’ll display the phone names alongside it’s range , either it’s being expensive , mid range or cheap –

SELECT Name ,

CASE

WHEN X.Buckets = 1 THEN 'Expensive'

WHEN X.Buckets = 2 THEN 'Mid-Range'

ELSE 'Poor'

END AS 'Phone Range'

FROM (

SELECT \*,

NTILE(3)

OVER (PARTITION BY Brand

ORDER BY Price DESC

) AS 'Buckets'

FROM product

WHERE Category = 'Phone') X ;

**Cume\_Dist (Cumulative distribution function) –**

Cume dist function actually gets us the cumulative distribution of the data we have in the table , Suppose from our gadgets table we want to fetch the first 30% of the entire data based on price , to put in simpler words , we would like to find out products , for which prices are comprising of 30% of the entire data. Our query will look like below –

SELECT P.\* ,

ROUND(CUME\_DIST() OVER(ORDER BY price)\*100,2) AS cume\_distribution

FROM gadget\_schema.product P;

As we can see we are able to see the distribution ratio of each product in the cume\_distribution column.

Suppose we are looking for first 30% of the price distribution and the product name , we can use the above query as a subquery to fetch the same –

SELECT Name, cume\_distribution

FROM(

SELECT P.\* ,

ROUND(CUME\_DIST() OVER(ORDER BY price)\*100,2) AS cume\_distribution

FROM gadget\_schema.product P) X

WHERE X.cume\_distribution<=30;

**Percent\_Rank –**

Percent rank function is almost similar to cume\_dist , it finds out the ranking distribution among the result set of the table , suppose we want to find out the price ranking distribution of the products we have in the table , our query will be like below –

SELECT P.\* ,

ROUND(PERCENT\_RANK() OVER(ORDER BY price)\*100,2) AS per\_rank

FROM gadget\_schema.product P;

One insight that we can gather from this is that as shown above , per\_rank column holds a value of 12.12 , which goes to show that it is 12.12% more expensive than all other products in the table.

These are all about window functions , now let’s talk about views and joins a bit , and this time we will be using another database which contains employee data.

When we execute a query , a view gets stored ,,, suppose in the below query , I am trying to find out the details of all male employees from the database –

SELECT \* FROM companydatabase.employee

WHERE sex = 'M';

This particular query is not making any changes to the table itself , it just is showcasing the details as per our requirement , Now suppose a third party client wants to get this data , we do have the option of exporting this into a csv or an excel file and share the same with the client , but suppose they want themselves to run the code to see for themselves each time they want the updates on the table itself. As an organization we can’t share our table structure or the direct query with the client , for such situations , views are actually created.

We can create a view for the above query like this –

CREATE VIEW male\_emp

AS

SELECT \* FROM companydatabase.employee

WHERE sex = 'M';

We have given the name as male\_emp , so each time we want to fetch the result , we can just run a query based on this view name as below and will get the same result –

SELECT \* FROM male\_emp;

As we can see , we are getting the same result by just querying on this view. And now in workbench , we can see that under the views section , we are getting a view named as male\_emp.

The query script for our created view will still be visible under the views tab as below.

We can make changes to the query and modify our requirements here as well , as you can see , now we are filtering for female employees , and modifying the same query –

The entire modified view query will be as below –

DROP VIEW IF EXISTS `companydatabase`.`male\_emp` ;

USE `companydatabase`;

CREATE

OR REPLACE ALGORITHM = UNDEFINED

DEFINER = `root`@`localhost`

SQL SECURITY DEFINER

VIEW `companydatabase`.`female\_emp` AS

SELECT

`companydatabase`.`employee`.`emp\_id` AS `emp\_id`,

`companydatabase`.`employee`.`first\_name` AS `first\_name`,

`companydatabase`.`employee`.`last\_name` AS `last\_name`,

`companydatabase`.`employee`.`birth\_day` AS `birth\_day`,

`companydatabase`.`employee`.`sex` AS `sex`,

`companydatabase`.`employee`.`salary` AS `salary`,

`companydatabase`.`employee`.`super\_id` AS `super\_id`,

`companydatabase`.`employee`.`branch\_id` AS `branch\_id`

FROM

`companydatabase`.`employee`

WHERE

(`companydatabase`.`employee`.`sex` = 'F');

Now we can run the same modified query as below –

SELECT \* FROM female\_emp;

Now there is a catch , above the particular script that we have seen under the views , is accessible to the third party client as well , so how to prevent this is a concern. For this kind of scenarios , we have DCL commands , where we can create new user , give them selective permissions on running the query , revoke , switch their permissions etc. By default the user is “root” , we can create a new user with below syntax in MySQL –

CREATE USER 'raj'@'localhost'IDENTIFIED BY'abcd1234';

As you can see , I have created a new user called raj here.

And now if we go to the administration section below instances , we will be able to see the newly created user shown up there –

We can check the default privileges/access given to the newly created user with below query –

SHOW GRANTS FOR 'raj'@'localhost';

As we can see , no default access is given to the user created. We can grant the user access to the databases and the tables with below query –

GRANT SELECT ON \*.\* TO 'raj'@'localhost';

The first \* denotes the database and the second \* denotes the tables , and now if we execute the show grants query for raj , we’ll get the below result –

Now let’s go ahead and create a view , and then provide the specific access to the created user ‘raj’. The view will of employee details in companydatabase , without the salary –

CREATE VIEW non\_salary\_view

AS

SELECT emp\_id , first\_name , last\_name , birth\_day , sex , super\_id , branch\_id

FROM employee;

CREATE USER 'harry'@'localhost'IDENTIFIED BY'abcd1234';

GRANT SELECT ON non\_salary\_view TO 'harry'@'localhost';

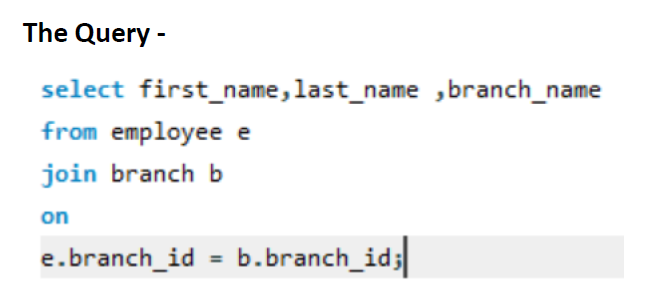
Now let’s talk joins ,

Joins In MySQL –

In MySQL we come across few types of Joins , lets talk about them one by one. First we will talk about ‘Join’/’Inner Join’ syntax

Join/Inner Join –

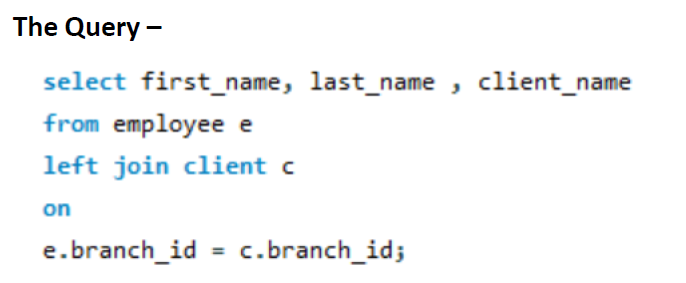
As seen in the image , with the help of inner join as the ‘C’ sectioned part of the image contains , we are able to retrieve information from the common part present in both of the tables and related information in it. We will now run a query performing a inner join between employee table and branch table , and will retrieve information on employee names and branch names , using branch\_id as our connector. Remember in this type of joins only records those have matching values in both the tables are showcased , as here we requested for first\_name , last\_name and branch\_name to be displayed , if any of the column holds not matching values based on their given primary/foreign key , then those records will not be displayed. As if one branch\_id , if present in employee table , but not in branch table , it won’t be displayed in the result.



Left Join –

As the name suggests , this type of join all the records from the left side of the table will be displayed whether they contain matching value in both the tables or not and it’ll also showcase the common records present in both the tables. As per our image , A,C will be the output record in this case.

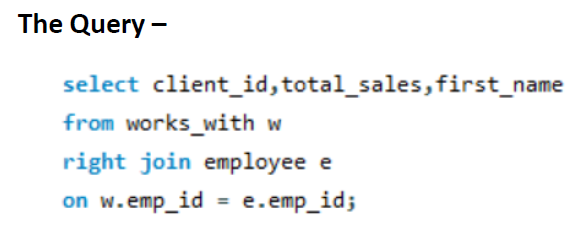
Now let’s run a query trying to join the employee table with the client table and we will see that records for the employee who doesn’t have a matching value in the client name section will appear here as null because we are performing left join and hence all records from the left table will be a part of our result.



Right Join –

As the name suggests , this type of join all the records from the right side of the table will be displayed whether they contain matching value in both the tables or not and it’ll also showcase the common records present in both the tables. As per our image , B,C will be the output record in this case.

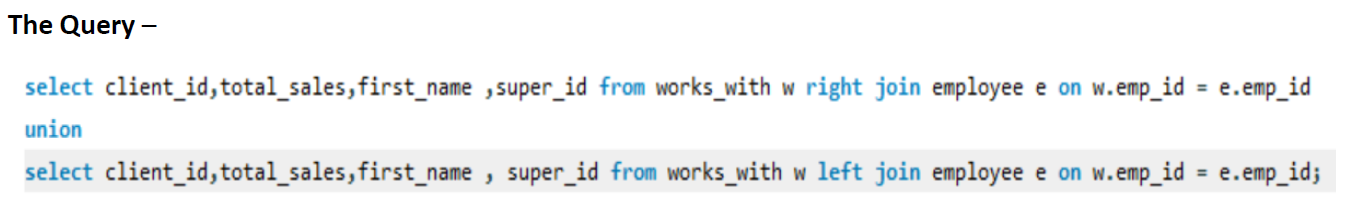
Now let’s run a query trying to join the works\_with table with the employee table and we will see that records for the employee who doesn’t have a matching value in the total\_sales , client\_id section will appear here as null because we are performing right join and hence all records from the right table will be a part of our result.



FULL Outer Join/FULL Join –

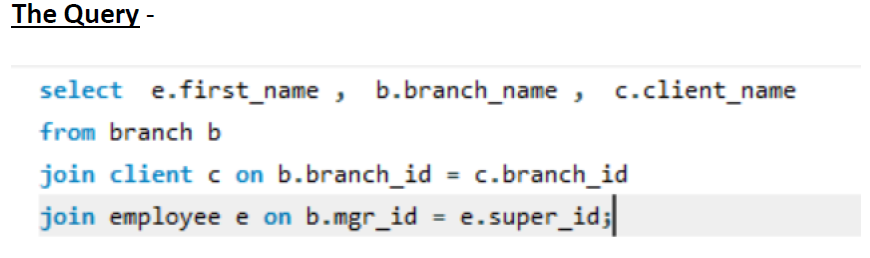
As it can be derived by the name itself , the full outer join/full join executes and displays all records from all tables being joined , as per our image it displays , a full outer join result will consist of A,B,C altogether. But in MySQL , using of full join/full outer join is not permitted directly , so we need to use the union function with a combination of left/right join to perform a outer join here,

Let’s try to have a walkaround on this using our previous query itself and we will be able to see that non matching values from both the tables will be a part of our record will ‘null’ keyed in it.



Multiple Table Join –

We can join multiple tables to extract data from all of the tables , suppose in the below query we are attempting to get the data of employee first name , client name and the branch name , and these data are spread across three tables , employee , client and branch table , so we will be performing a join in all of the tables.



This will be all for today. I hope this session clarifies your doubt on window functions and MySQL joins to a greater extent. Take care.