## SQL ideas

|  |  |
| --- | --- |
| to handle huge data- if ur app cant receive huge data received by the qry execution | Then use cursors |
| Generally for auditing purposes, we can happily write sql triggers na,  if any insert into 1 table, then automatically it will insert into another table  or when u want to get update when someone inserts the data into certain table | sql triggers |
|  |  |

### Execution flow

note: -always 1st inner query will be executed then WHERE clause will be executed

## SQL OVER Clause and PARTITION BY

In SQL, the OVER clause is a powerful tool that allows you to perform calculations across a set of rows that are related to the current row. This set of rows is called a "window". The PARTITION BY clause is used to divide the result set into partitions, or groups, and the window function is then applied to each partition independently.

### OVER Clause

They allow you to perform calculations across a set of rows without having to use subqueries or self-joins.

The OVER clause is used in conjunction with window functions, which are functions that perform calculations across a set of rows. Window functions include aggregate functions like SUM(), AVG(), COUNT(), MAX(), and MIN(), as well as ranking functions like ROW\_NUMBER(), RANK(), and DENSE\_RANK().

The basic syntax of the OVER clause is:

SQL

Inside this over clause we can happily write all below expressions

OVER (

[PARTITION BY value\_expression]

[ORDER BY value\_expression [ASC | DESC]]

[ROWS | RANGE frame\_clause]

)

The PARTITION BY clause is optional. If it is not specified, the window function is applied to the entire result set. The ORDER BY clause is also optional. If it is not specified, the order of the rows within each partition is not guaranteed. The ROWS or RANGE clause is used to define the window frame, which is the set of rows that the window function is applied to.

### PARTITION BY Clause

The PARTITION BY clause is used to divide the result set into partitions, or groups, based on the values in one or more columns. The window function is then applied to each partition independently.

For example, the following query calculates the average salary for each department:

SQL

|  |  |
| --- | --- |
| SELECT department, salary,  AVG(salary) OVER (PARTITION BY department) AS average\_salary  FROM  employees;  select country, AVG(salary) over(partition by Country) from person  //Here partition by will happen 1st on the partitioned data avg function will be applied to each partition of data | select Country, AVG(Salary) as sal from person group by Country |

The PARTITION BY clause divides the result set into partitions based on the department column. The AVG(salary) window function is then applied to each partition independently, calculating the average salary for each department.

select name, country, AVG(salary) over(partition by Country) from person ;

### Benefits of Using the OVER Clause and PARTITION BY Clause

the advantage with partition by is – u can mention all columns which are not even mentioned in partition by clause

select name, country, AVG(salary) over(partition by Country) from person ;

here I mentioned all remaining columns which are not mentioned in partition by clause, the same thing is not possible in group by

- in group by u have to mention same column name which are mentioned in group by clause

The OVER clause and PARTITION BY clause offer several benefits:

* They allow you to perform calculations across a set of rows without having to use subqueries or self-joins.
* They can improve the performance of your queries.
* They can make your queries easier to read and understand.

### Examples of Using the OVER Clause and PARTITION BY Clause

Here are some examples of how you can use the OVER clause and PARTITION BY clause:

* Calculate the running total of sales for each month:

SQL

SELECT

month,

sales,

SUM(sales) OVER (ORDER BY month) AS running\_total

FROM

sales\_data;

* Rank employees within each department based on their salary:

SQL

SELECT

employee\_id,

department,

salary,

RANK() OVER (PARTITION BY department ORDER BY salary DESC) AS rank

FROM

employees;

* Calculate the moving average of stock prices:

SQL

SELECT

date,

price,

AVG(price) OVER (ORDER BY date ROWS BETWEEN 2 PRECEDING AND 2 FOLLOWING) AS moving\_average

FROM

stock\_prices;

## order by

SELECT \* FROM aims.person order by country desc , name asc

Order by 2 columns means, 1st it will order by desc and then entire result set will be order by asc

## Functions

### ROW\_NUMBER(), RANK(), DENSE\_RANK()

Row\_number will assign row number for each and every function

ROW\_NUMBER() , RANK(), DENSE\_RANK() - these 3 func will assign rownumber to each row in different fashions

1) The ROW\_NUMBER() function in MySQL assigns a unique number to each row in a result set

it will assign row number to each row eventhough rows are duplicate

when u use row\_number() order by is mandatory

when u have duplicates then use below function

whereas rank(), dense\_rank() will assign row number

ROW\_NUMBER()- it will give unique row number for each and every record eventhough order by col is having duplicate values

RANK()- this will give same rank if both are having same salary- if 2 persons having same mark then both will get same rank-1 but it will skip the rank

and gives rank-3 for next person instead of rank-2

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| select \*,row\_number() over (order by salary desc) as rowNum from Person  Name, Product, Country, Year, Salary, rowNum   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | Peter | Mouse | England | 2016 | 999 | 1 | | Prakash | Transistor | England | 2016 | 999 | 2 | | John | TV | USA | 2016 | 700 | 3 | | Anuradha | keyboard | USA | 2016 | 700 | 4 | | Donald | Laptop | England | 2015 | 600 | 5 | | Charan | Laptop | England | 2016 | 600 | 6 | | Joseph | Laptop | India | 2016 | 500 | 7 | | Stephen | Computer | USA | 2015 | 200 | 8 | | Rama | Sweeper | USA | 2015 | 200 | 9 | | Joseph | Mobile | India | 2015 | 159 | 10 | | Ranga | Router | India | 2015 | 159 | 11 | | Name, Product, Country, Year, Salary, rowNum  select \* , rank() over (order by salary desc) as ranks from Person   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | Peter | Mouse | England | 2016 | 999 | 1 | | Prakash | Transistor | England | 2016 | 999 | 1 | | John | TV | USA | 2016 | 700 | 3 | | Anuradha | keyboard | USA | 2016 | 700 | 3 | | Donald | Laptop | England | 2015 | 600 | 5 | | Charan | Laptop | England | 2016 | 600 | 5 | | Joseph | Laptop | India | 2016 | 500 | 7 | | Stephen | Computer | USA | 2015 | 200 | 8 | | Rama | Sweeper | USA | 2015 | 200 | 8 | | Joseph | Mobile | India | 2015 | 159 | 10 | | Ranga | Router | India | 2015 | 159 | 10 | | select \* , dense\_rank() over (order by salary desc) as ranks from Person  Name, Product, Country, Year, Salary, rowNum   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | Peter | Mouse | England | 2016 | 999 | 1 | | Prakash | Transistor | England | 2016 | 999 | 1 | | John | TV | USA | 2016 | 700 | 2 | | Anuradha | keyboard | USA | 2016 | 700 | 2 | | Donald | Laptop | England | 2015 | 600 | 3 | | Charan | Laptop | England | 2016 | 600 | 3 | | Joseph | Laptop | India | 2016 | 500 | 4 | | Stephen | Computer | USA | 2015 | 200 | 5 | | Rama | Sweeper | USA | 2015 | 200 | 5 | | Joseph | Mobile | India | 2015 | 159 | 6 | | Ranga | Router | India | 2015 | 159 | 6 | |

DESNSE\_RANK() - this wont skip the rank , if 2 persons having same marks then same rank-1 for both, & next rank will be 2nd for next highest mark,

Nth highest salary query

select \* from

(select \*, row\_number() over (partition by Country order by Salary desc) as row\_num from person) personTemp where row\_num=1

select \* from (SELECT Name, Salary , row\_number() over( order by salary desc ) as rowNumber from person ) personTemp where rowNumber=6;

we can give rownumber uniquely to each group- ex:- for cse guys row number 1,2,3,4 , for ece guys 1,2,3,4

while assigning it can give unique number to each groups

ex- if u want to get 2nd high ranker in every dept

here we are assigining row number after partitioning

### Use case to delete duplicate rows using row\_number function

|  |  |
| --- | --- |
| # id, first\_name, last\_name, email, dept  '6', 'Virat', 'Yadav', 'virat@gmail.com', 'CSE'  '6', 'Virat', 'Yadav', 'virat@gmail.com', 'CSE'  '7', 'Prabhas', 'Shirke', 'prabhas@gmail.com', 'ECE'  '7', 'Prabhas', 'Shirke', 'prabhas@gmail.com', 'ECE'  '8', 'Tina', 'Kapoor', 'tina@gmail.com', 'MECH'  '8', 'Tina', 'Kapoor', 'tina@gmail.com', 'MECH'  '9', 'Mona', 'Sharma', 'mona@gmail.com', 'ECE'  '9', 'Mona', 'Sharma', 'mona@gmail.com', 'ECE'  '10', 'Rahul', 'Varma', 'rahul@gmail.com', 'CSE'  '10', 'Rahul', 'Varma', 'rahul@gmail.com', 'CSE'  '11', 'Mohan', 'wood', 'mohan@gmail.com', 'MECH'  '11', 'Mohan', 'wood', 'mohan@gmail.com', 'MECH'  '12', 'radhika', 'chakka', 'Mr@gmail.com', 'ECE'  '12', 'radhika', 'chakka', 'Mr@gmail.com', 'ECE'  '13', 'Nitin', 'marthala', 'chakkan@gmail.com', 'CSE'  '13', 'Nitin', 'marthala', 'chakkan@gmail.com', 'CSE' | Here every row is duplicated once, so we used row\_number with partition by to delete the duplicate rows  after partitioning data formed into groups  select \* , row\_number() over (partition by first\_name ) as row\_num from student  # id, first\_name, last\_name, email, dept, row\_num  '11', 'Mohan', 'wood', 'mohan@gmail.com', 'MECH', '1'  '11', 'Mohan', 'wood', 'mohan@gmail.com', 'MECH', '2'  '9', 'Mona', 'Sharma', 'mona@gmail.com', 'ECE', '1'  '9', 'Mona', 'Sharma', 'mona@gmail.com', 'ECE', '2'  '13', 'Nitin', 'marthala', 'chakkan@gmail.com', 'CSE', '1'  '13', 'Nitin', 'marthala', 'chakkan@gmail.com', 'CSE', '2'  now delete all rows with row\_num as 1  DELETE s1  FROM student s1  INNER JOIN (  SELECT id, -- Include the primary key of your student table  ROW\_NUMBER() OVER (PARTITION BY first\_name ORDER BY id) as row\_num -- Add an ORDER BY clause for consistent results  FROM student  ) s2 ON s1.id = s2.id -- Join on the primary key  WHERE s2.row\_num > 1;  You can't directly delete from a result set generated by ROW\_NUMBER() or similar window functions. |

### Triggers

when an

CREATE TRIGGER tr\_EmployeeSalaryChange

ON Employees

AFTER UPDATE

AS

BEGIN

IF UPDATE(Salary)

BEGIN

INSERT INTO EmployeeSalaryAudit (EmployeeID, OldSalary, NewSalary, ChangeDate)

SELECT i.EmployeeID, d.Salary, i.Salary, GETDATE()

FROM inserted i

INNER JOIN deleted d ON i.EmployeeID = d.EmployeeID;

END

END;

## Joins

Inner join == join – only common rows will be fetched

Right outer join – all data in right side table + common data in left side table

Left outer join – all data in left side table + common data

Full outer join – all data in all tables

Union – will fetch all data but it will not fech duplicates

Union all – it will fetch duplicates also

## Cursors

You have a table of customers, and you want to send a personalized email to each customer. You could write a query to get all the customer data, but then your program would need to hold all that data in memory at once. This might be inefficient if you have a huge number of customers.

**Here's where cursors come in:**

1. **You declare a cursor:** You define a cursor and associate it with a query that retrieves the customer data. The cursor doesn't actually fetch all the data yet.
2. **You open the cursor:** This executes the query and prepares the result set. The cursor now points to the first row of the result set.
3. **You fetch data:** You can fetch the data from the current row that the cursor is pointing to. Your program can then process this data (e.g., send an email).
4. **You move the cursor:** You can move the cursor to the next row in the result set and fetch the data from that row. You repeat this process until you've processed all the rows.
5. **You close the cursor:** This releases the resources associated with the cursor.

**Why use cursors?**

* **Memory efficiency:** Cursors allow you to process large result sets without loading all the data into memory at once.
* **Row-by-row processing:** Cursors provide a way to perform operations on each row of a result set individually, which is useful for tasks like sending personalized emails or updating data based on conditions.
* **Flexibility:** Cursors offer control over how you navigate and process the data in a result set.