Window Functions In SQL

- **Window functions:** These are special SQL functions that perform a calculation across a set of related rows.

- **How it works**: Instead of operating on individual rows, a window function operates on a group or **'window'** of rows that are somehow related to the current row. This allows for complex calculations based on these related rows.

- **Window definition**: The **'window'** in window functions refers to a **set of rows**. The window can be defined using different criteria depending on the requirements of your operation.

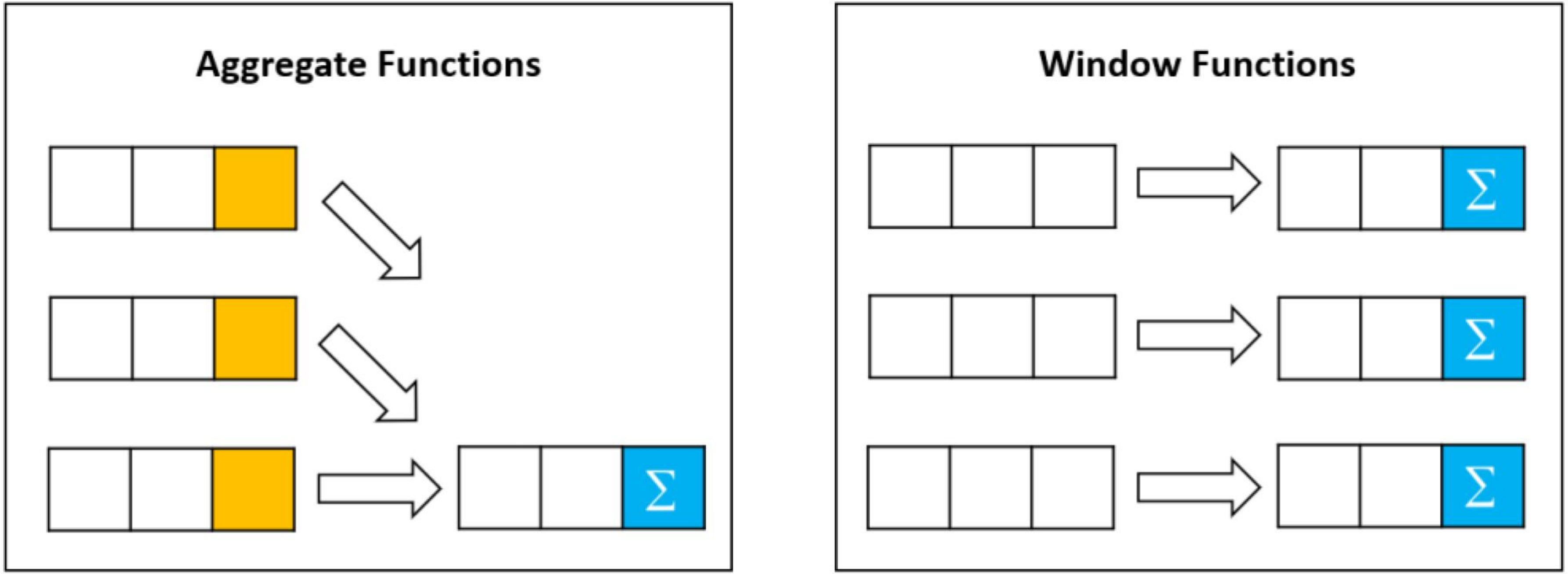
- **Partitions**: By using the **PARTITION BY** clause, you can divide your data into smaller sets or **'partitions'**. The window function will then be applied individually to each partition.

- **Order of rows**: You can specify the order of rows in each partition using the ORDER BY clause. This order influences how some window functions calculate their result.

- **Frames**: The **ROWS/RANGE** clause lets you further narrow down the window by defining a **'frame'** or subset of rows within each partition.

- **Comparison with Aggregate Functions:** Unlike aggregate functions that return a **single result per group**, window functions return a **single result for each row** of the table based on the group of rows defined in the window.

- **Advantage**: Window functions allow for **more complex operations** that need to take into account not just the current row, but also its **'neighbours'** in some way.


Aggregate Functions
Window Functions

Example

	order_id	order_date	customer_name	city	order_amount	grand_total
1	1002	2017-04-02	David Jones	Arington	20000.00	37000.00
2	1007	2017-04-10	Andrew Smith	Arington	15000.00	37000.00
3	1008	2017-04-11	David Brown	Arington	2000.00	37000.00
4	1001	2017-04-01	David Smith	GuildFord	10000.00	50500.00
5	1006	2017-04-06	Paum Smith	GuildFord	25000.00	50500.00
6	1004	2017-04-04	Michael Smith	GuildFord	15000.00	50500.00
7	1010	2017-04-25	Peter Smith	GuildFord	500.00	50500.00
8	1005	2017-04-05	David Williams	Shalford	7000.00	13000.00
9	1003	2017-04-03	John Smith	Shalford	5000.00	13000.00
10	1009	2017-04-20	Robert Smith	Shalford	1000.00	13000.00

Window Function Syntax

```
function_name (column) OVER (
    [PARTITION BY column_name_1, ..., column_name_n]
    [ORDER BY column_name_1 [ASC | DESC], ..., column_name_n [ASC | DESC]]
)
```

- **function_name**: This is the window function you want to use. Examples include ROW_NUMBER(), RANK(), DENSE_RANK(), SUM(), AVG(), and many others.

- (**column**): This is the column that the window function will operate on. For some functions like SUM(salary)

- **OVER** (): This is where you define the window. The parentheses after OVER contain the specifications for the window.

- **PARTITION BY column_name_1, ..., column_name_n**: This clause divides the result set into partitions upon which the window function will operate independently. For example, if you have PARTITION BY salesperson_id, the window function will calculate a result for each salesperson independently.

- **ORDER BY column_name_1 [ASC | DESC], ..., column_name_n [ASC | DESC]**: This clause specifies the order of the rows in each partition. The window function operates on these rows in the order specified. For example, ORDER BY sales_date DESC will make the window function operate on rows with more recent dates first.

Different Types of Window Functions

There are three main categories of window functions in SQL: **Ranking functions**, **Value functions**, and **Aggregate functions**. Here's a brief description and example for each:

Ranking Functions:

- **ROW_NUMBER()**: Assigns a unique row number to each row, ranking start from 1 and keep increasing till the end of last row

```
SELECT Studentname,
       Subject,
       Marks,
       ROW_NUMBER() OVER(ORDER BY Marks desc)
RowNumber
FROM ExamResult;
```

	Studentname	Subject	Marks	RowNumber
1	Isabella	english	90	1
2	Olivia	english	89	2
3	Lily	Science	80	3
4	Lily	english	70	4
5	Isabella	Science	70	5
6	Lily	Maths	65	6
7	Olivia	Science	60	7
8	Olivia	Maths	55	8
9	Isabella	Maths	50	9

- **RANK()**: Assigns a rank to each row. Rows with equal values receive the same rank, with the next row receiving a rank which skips the duplicate rankings.

```
SELECT Studentname,
       Subject,
       Marks,
       RANK() OVER(ORDER BY Marks DESC) Rank
FROM ExamResult
ORDER BY Rank;
```

Results | Messages

	Studentname	Subject	Marks	Rank
1	Isabella	english	90	1
2	Olivia	english	89	2
3	Lily	Science	80	3
4	Lily	english	70	4
5	Isabella	Science	70	4
6	Lily	Maths	65	6
7	Olivia	Science	60	7
8	Olivia	Maths	55	8
9	Isabella	Maths	50	9

- **DENSE_RANK()**: Similar to RANK(), but does not skip rankings if there are duplicates.

```
SELECT Studentname,
       Subject,
       Marks,
       DENSE_RANK() OVER(ORDER BY Marks DESC) Rank
FROM ExamResult
ORDER BY Rank;
```

	Studentname	Subject	Marks	Rank
1	Isabella	english	90	1
2	Olivia	english	89	2
3	Lily	Science	80	3
4	Lily	english	70	4
5	Isabella	Science	70	4
6	Lily	Maths	65	5
7	Olivia	Science	60	6
8	Olivia	Maths	55	7
9	Isabella	Maths	50	8

Similar Rank

Value Functions: These functions perform calculations on the values of the window rows.

- **FIRST_VALUE():** Returns the first value in the window.

```
SELECT
    employee_name,
    department,
    hours,
    FIRST_VALUE(employee_name) OVER (
        PARTITION BY department
        ORDER BY hours
    ) least_over_time
FROM
    overtime;
```

employee_name	department	hours	least_over_time
Diane Murphy	Accounting	37	Diane Murphy
Jeff Firrelli	Accounting	40	Diane Murphy
Mary Patterson	Accounting	74	Diane Murphy
Gerard Bondur	Finance	47	Gerard Bondur
William Patterson	Finance	58	Gerard Bondur
Anthony Bow	Finance	66	Gerard Bondur
Leslie Thompson	IT	88	Leslie Thompson
Leslie Jennings	IT	90	Leslie Thompson
Loui Bondur	Marketing	49	Loui Bondur
Gerard Hernandez	Marketing	66	Loui Bondur
George Vanauf	Marketing	89	Loui Bondur
Steve Patterson	Sales	29	Steve Patterson
Foon Yue Tseng	Sales	65	Steve Patterson
Julie Firrelli	Sales	81	Steve Patterson
Barry Jones	SCM	65	Barry Jones
Pamela Castillo	SCM	96	Barry Jones
Larry Bott	SCM	100	Barry Jones

- **LAST_VALUE():** Returns the last value in the window.

```
SELECT employee_name, department,salary,
LAST_VALUE(employee_name)
 OVER (
     PARTITION BY department ORDER BY
salary
     ) as max_salary
FROM Employee;
```

employee_name	department	salary	max_salary
Vishal	Accounting	40000	Ravi
Ravi	Accounting	60000	Ravi
Nilesh	Finance	55000	Abdul
Sushant	Finance	65000	Abdul
Abdul	Finance	68000	Abdul
Jai	IT	45000	Mohit
Aman	IT	60000	Mohit
Mohit	IT	70000	Mohit

- **LAG():** Returns the value of the previous row.

```
SELECT
      Year,
      Quarter,
      Sales,
      LAG(Sales, 1, 0) OVER(
         PARTITION BY Year
         ORDER BY Year,Quarter ASC)
      AS NextQuarterSales
FROM ProductSales;
```

	Year	Quarter	Sales	NextQuarterSales
1	2017	1	55000.00	0.00
2	2017	2	78000.00	55000.00
3	2017	3	49000.00	78000.00
4	2017	4	32000.00	49000.00
5	2018	1	41000.00	0.00
6	2018	2	8965.00	41000.00
7	2018	3	69874.00	8965.00
8	2018	4	32562.00	69874.00
9	2019	1	87456.00	0.00
10	2019	2	75000.00	87456.00
11	2019	3	96500.00	75000.00
12	2019	4	85236.00	96500.00

1

2

3

- **LEAD():** Returns the value of the next row.

```
SELECT Year,
       Quarter,
       Sales,
       LEAD(Sales, 1, 0) OVER(
       PARTITION BY Year
       ORDER BY Year,Quarter ASC)
       AS NextQuarterSales
FROM ProductSales;
```

	Year	Quarter	Sales	NextQuarterSales
1	2017	1	55000.00	78000.00
2	2017	2	78000.00	49000.00
3	2017	3	49000.00	32000.00
4	2017	4	32000.00	0.00
5	2018	1	41000.00	8965.00
6	2018	2	8965.00	69874.00
7	2018	3	69874.00	32562.00
8	2018	4	32562.00	0.00
9	2019	1	87456.00	75000.00
10	2019	2	75000.00	96500.00
11	2019	3	96500.00	85236.00
12	2019	4	85236.00	0.00

1

2

3

Lead function on PARTITION for Year column

Aggregation Functions: These functions perform calculations on the values of the window rows.

- SUM()
- MIN()
- MAX()
- AVG()

Query editor

```sql
SELECT product, sales, category, SUM(sales)
  OVER (
    PARTITION BY category
    ORDER BY sales
    ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
  ) AS total_purchases
FROM
```

Run | Save query | Save view | Schedule query | More

Query results | SAVE RESULTS | EXPLORE DATA

Query complete (0.2 sec elapsed, 285 B processed)

Job information | Results | JSON | Execution details

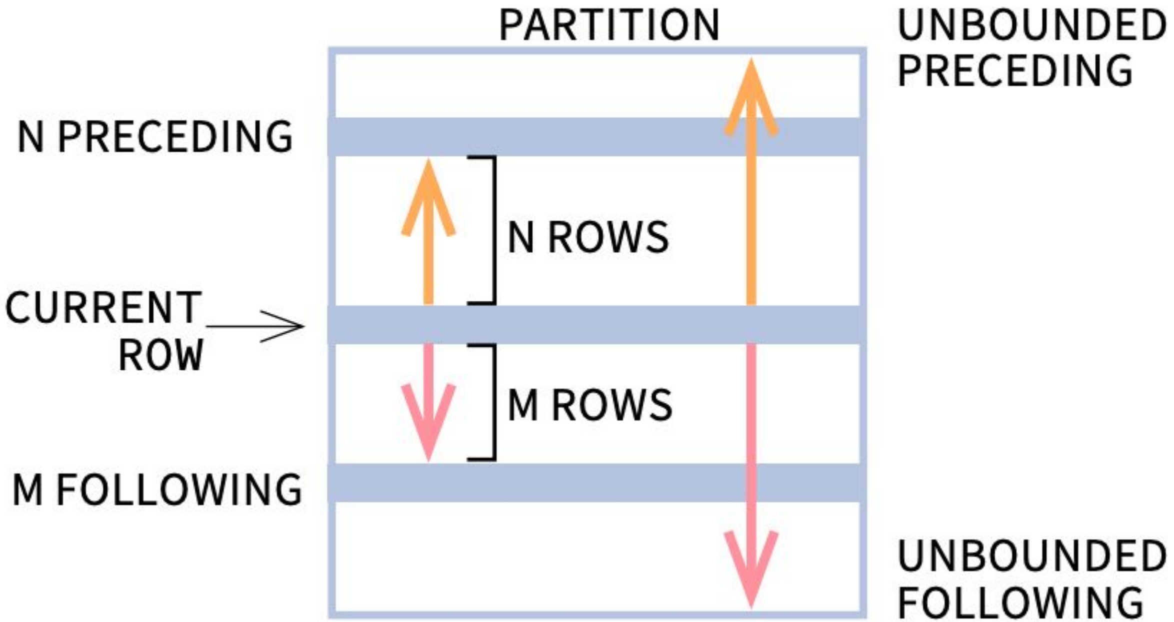
Row	product	sales	category	total_purchases
1	Polo Shirt	2	clothing	2
2	Denim Skirt	4	clothing	6
3	Blue Jeans	8	clothing	14
4	Maxi Skirt	9	clothing	23
5	Long Sleeve T-shirt	44	clothing	67
6	Denim Sneakers	1	footwear	1
7	Crocs	5	footwear	6
8	Long Boots	11	footwear	17
9	Air-Max Sneakers	11	footwear	28

Frame Clause in Window Functions

- The frame clause in window functions defines the **subset of rows ('frame')** used for calculating the result of the function for the current row.
- It's specified within the OVER() clause after PARTITION BY and ORDER BY.
- The frame is defined by two parts: a **start** and an **end**, each relative to the **current row**.
- Generic syntax for a window function with a frame clause:

```
function_name (expression) OVER (
  [PARTITION BY column_name_1, ..., column_name_n]
  [ORDER BY column_name_1 [ASC | DESC], ..., column_name_n [ASC | DESC]]
  [ROWS|RANGE frame_start TO frame_end]
)
```

- The frame start can be:
  - UNBOUNDED PRECEDING (starts at the first row of the partition)
  - N PRECEDING (starts N rows before the current row)
  - CURRENT ROW (starts at the current row)
- The frame end can be:
  - UNBOUNDED FOLLOWING (ends at the last row of the partition)
  - N FOLLOWING (ends N rows after the current row)
  - CURRENT ROW (ends at the current row)
- For **ROWS**, the frame consists of N rows coming before or after the current row.
- For **RANGE**, the frame consists of rows within a certain value range relative to the value in the current row.


PARTITION
UNBOUNDED PRECEDING
N PRECEDING
N ROWS
CURRENT ROW
M ROWS
M FOLLOWING
UNBOUNDED FOLLOWING

ROWS BETWEEN Example

```
SELECT date, revenue,
    SUM(revenue) OVER (
      ORDER BY date
      ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) running_total
FROM sales
ORDER BY date;
```

Input Table

sales		
record_id	date	revenue
1	2021-09-01	1515.45
2	2021-09-02	2345.35
3	2021-09-03	903.99
4	2021-09-04	2158.55
5	2021-09-05	1819.80

Output Table

date	revenue	running_total
2021-09-01	1515.45	1515.45
2021-09-02	2345.35	3860.80
2021-09-03	903.99	4764.79
2021-09-04	2158.55	6923.34
2021-09-05	1819.80	8743.14

RANGE BETWEEN Example

```
SELECT
  shop,
  date,
  revenue_amount,
  MAX(revenue_amount) OVER (
    ORDER BY DATE
    RANGE BETWEEN INTERVAL '3' DAY PRECEDING
   AND INTERVAL '1' DAY FOLLOWING
  ) AS max_revenue
FROM revenue_per_shop;
```

shop	date	revenue_amount	max_revenue
Shop 1	2021-05-01	12,573.25	18,847.54
Shop 2	2021-05-01	11,348.22	18,847.54
Shop 1	2021-05-02	14,388.14	18,847.54
Shop 2	2021-05-02	18,847.54	18,847.54
Shop 1	2021-05-03	9,845.29	18,847.54
Shop 2	2021-05-03	14,574.56	18,847.54
Shop 1	2021-05-04	11,500.63	18,847.54
Shop 2	2021-05-04	16,897.21	18,847.54
Shop 1	2021-05-05	9,634.56	21,489.22
Shop 2	2021-05-05	14,255.87	21,489.22
Shop 1	2021-05-06	11,248.33	21,489.22
Shop 2	2021-05-06	21,489.22	21,489.22
Shop 2	2021-05-07	15,517.22	21,489.22
Shop 1	2021-05-07	14,448.65	21,489.22

Output Table

Common Table Expression

A Common Table Expression (CTE) in SQL is a named temporary result set that exists only within the execution scope of a single SQL statement. Here are some important points to note about CTEs:

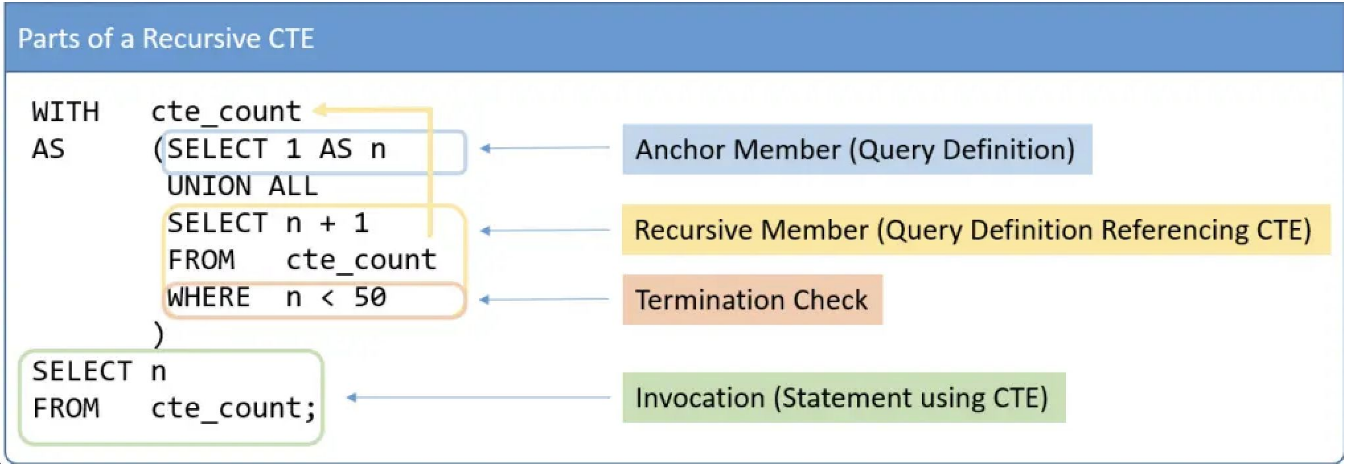
- CTEs can be thought of as alternatives to derived tables, inline views, or subqueries.
- They can be used in SELECT, INSERT, UPDATE, or DELETE statements.
- CTEs help to simplify complex queries, particularly those involving multiple subqueries or recursive queries.
- They make your query more readable and easier to maintain.
- A CTE is defined using the **WITH keyword,** followed by the CTE name and a query. The CTE can then be referred to by its name elsewhere in the query.

Here's a basic example of a CTE:

```
WITH sales_cte AS (
   SELECT sales_person, SUM(sales_amount) as total_sales
   FROM sales_table
   GROUP BY sales_person
)
SELECT sales_person, total_sales
FROM sales_cte
WHERE total_sales > 1000;
```

- **Recursive CTE:** This is a CTE that references itself. In other words, the CTE query definition refers back to the CTE name, creating a loop that ends when a certain condition is met. Recursive CTEs are useful for working with hierarchical or tree-structured data.

  Example:
  ```
  WITH RECURSIVE number_sequence AS (
      SELECT 1 AS number
      UNION ALL
      SELECT number + 1
      FROM number_sequence
      WHERE number < 10
  )
  SELECT * FROM number_sequence;
  ```



Subqueries in SQL

- **IN:** The IN operator allows you to specify multiple values in a WHERE clause. It returns true if a value matches any value in a list.

  SELECT * FROM Orders WHERE ProductName IN ('Apple', 'Banana');

- **NOT IN:** The NOT IN operator excludes the values in the list. It returns true if a value does not match any value in the list.

  SELECT * FROM Orders WHERE ProductName NOT IN ('Apple', 'Banana');

- **ANY:** The ANY operator returns true if any subquery value meets the condition.
- **ALL:** The ALL operator returns true if all subquery value meets the condition.
- **EXISTS:** The EXISTS operator returns true if the subquery returns one or more records.
- **NOT EXISTS:** The NOT EXISTS operator returns true if the subquery returns no records.

Views

A view in SQL is a virtual table based on the result-set of an SQL statement. It contains rows and columns, just like a real table. The fields in a view are fields from one or more real tables in the database.

Here are some key points about views:

- You can add SQL functions, WHERE, and JOIN statements to a view and display the data as if the data were coming from one single table.
- A view always shows up-to-date data. The database engine recreates the data every time a user queries a view.
- Views can be used to encapsulate complex queries, presenting users with a simpler interface to the data.
- They can be used to restrict access to sensitive data in the underlying tables, presenting only non-sensitive data to users.

Syntax to create Views

```
CREATE VIEW View_Products AS
SELECT ProductName, Price
FROM Products
WHERE Price > 30;
```

Employee

EmployeeID	Ename	DeptID	Salary
1001	John	2	4000
1002	Anna	1	3500
1003	James	1	2500
1004	David	2	5000
1005	Mark	2	3000
1006	Steve	3	4500
1007	Alice	3	3500

```
CREATE VIEW emp_view AS
SELECT DeptID, AVG(Salary)
FROM Employee
GROUP BY DeptID;
```

Create View of grouped records on Employee table

emp_view

DeptID	AVG(Salary)
1	3000.00
2	4000.00
3	4250.00

Indexing

Indexing in databases involves creating a data structure that improves the speed of data retrieval operations on a database table. Indexes are used to quickly locate data without having to search every row in a table each time a database table is accessed.

PK

Emp No.	Block Ptr
1001	*
1005	*
1009	*
1013	*
...	...
...	...

Primary Index

PK	Non PK		
Emp ID	Dept No.	Data...	
1001	1		
1002	2		
1003	2		
1004	1		
1005	1		
1006	3		
1007	4		
1008	3		
1009	3		
1010	3		
1011	4		
1012	1		
1013	2		
1014	2		
...	...		
...	...		
...	...		

Main Data File (sorted wrt Emp ID - PK)

Why is Indexing Important?

Indexes are crucial for enhancing the performance of a database by:

1. ***Speeding up Query Execution:*** Indexes reduce the amount of data that needs to be scanned for a query, significantly speeding up data retrieval operations.
2. ***Optimizing Search Operations:*** Indexes help in efficiently searching for records based on the indexed columns.
3. ***Improving Sorting and Filtering***: Indexes assist in sorting and filtering operations by providing a structured way to access data.
4. ***Enhancing Join Performance***: Indexes on join columns improve the performance of join operations between tables.

Advantages of Indexing

1. ***Faster Data Retrieval:*** Indexes make search queries faster by providing a quick way to locate rows in a table.
2. ***Efficient Use of Resources***: Reduced query execution time translates to more efficient use of CPU and memory resources.
3. ***Improved Performance for Large Tables***: Indexes are particularly beneficial for large tables where full table scans would be time-consuming.
4. ***Better Sorting and Filtering***: Indexes can improve the performance of ORDER BY, GROUP BY, and WHERE clauses.

How to Choose the Right Indexing Column

1. ***Primary Key and Unique Constraints***: Always index columns that are primary keys or have unique constraints, as they uniquely identify rows.
2. ***Frequently Used Columns in WHERE Clauses***: Index columns that are frequently used in WHERE clauses to filter data.
3. ***Columns Used in Joins***: Index columns that are used in join conditions to speed up join operations.
4. ***Columns Used in ORDER BY and GROUP BY:*** Index columns that are used in ORDER BY and GROUP BY clauses for faster sorting and grouping.
5. ***Selectivity of the Column***: Choose columns with high selectivity (columns with many unique values) to maximize the performance benefits of the index.

Query Optimizations

- Use Column Names Instead of * in a SELECT Statement

- Avoid including a HAVING clause in SELECT statements

  The HAVING clause is used to filter the rows after all the rows are selected and it is used like a filter. It is quite useless in a SELECT statement. It works by going through the final result table of the query parsing out the rows that don’t meet the HAVING condition.

  Example:
  Original query:
  ```
  SELECT s.cust_id,count(s.cust_id)
  FROM SH.sales s
  GROUP BY s.cust_id
  HAVING s.cust_id != '1660' AND s.cust_id != '2';
  ```

  Improved query:
  ```
  SELECT s.cust_id,count(cust_id)
  FROM SH.sales s
  WHERE s.cust_id != '1660'
  AND s.cust_id !='2'
  GROUP BY s.cust_id;
  ```

Query Optimizations

- Eliminate Unnecessary DISTINCT Conditions

  Considering the case of the following example, the DISTINCT keyword in the original query is unnecessary because the table_name contains the primary key p.ID, which is part of the result set.

  Example:

  Original query:
  SELECT DISTINCT * FROM SH.sales s
  JOIN SH.customers c
  ON s.cust_id= c.cust_id
  WHERE c.cust_marital_status = 'single';

  Improved query:
  SELECT * FROM SH.sales s JOIN
  SH.customers c
  ON s.cust_id = c.cust_id
  WHERE c.cust_marital_status='single';

Query Optimizations

Grow Data Skills

- Consider using an IN predicate when querying an indexed column

  The IN-list predicate can be exploited for indexed retrieval and also, the optimizer can sort the IN-list to match the sort sequence of the index, leading to more efficient retrieval.

  Example:
  Original query:
  ```
  SELECT s.*
  FROM SH.sales s
  WHERE s.prod_id = 14
  OR s.prod_id = 17;
  ```

  Improved query:
  ```
  SELECT s.*
  FROM SH.sales s
  WHERE s.prod_id IN (14, 17);
  ```

Query Optimizations

Grow Data Skills

- Try to use UNION ALL in place of UNION

  The UNION ALL statement is faster than UNION, because UNION ALL statement does not consider duplicate s, and UNION statement does look for duplicates in a table while selection of rows, whether or not they exist.

  Example:
  Original query:

  ```
  SELECT cust_id
  FROM SH.sales
  UNION
  SELECT cust_id
  FROM customers;
  ```

  Improved query:

  ```
  SELECT cust_id
  FROM SH.sales
  UNION ALL
  SELECT cust_id
  FROM customers;
  ```