SUMMARY: WINDOWS FUNCTIONS-I

SESSION OVERVIEW:

By the end of this session, you will be able to:

- Understand the fundamental window analytical functions in a database system.
- Understand the purpose and functionality of each window analytic function.

KEY TOPICS AND EXAMPLES:

Understanding the fundamental window analytical functions:

What are Windows Functions?

Window functions are SQL operations that perform a calculation across a set of rows that are related to the current row. Unlike aggregate functions, they do not cause rows to become grouped into a single output row — the rows retain their separate identities. Window functions can perform a calculation across a set of rows that are related to the current row. They are called window functions because they perform a calculation across a “window” of rows. For instance, you might want to calculate a running total of sales or find out the highest score in a group.

They are particularly useful in data analysis for computing aggregates like running totals, moving averages, ranking, and cumulative sums, which are crucial for gaining insights into patterns and trends within your data.

Industry Uses of Window Functions:

- **Financial Services:**
  - **Moving Averages:** Calculate moving averages of stock prices over a specified time period to identify trends.
  - **Ranking Transactions**: Rank transactions by amount within each account to identify the largest or smallest transactions.
- **Sales and Marketing:**
  - **Running Totals:** Compute cumulative sales totals to date for each salesperson or region.
  - **Percentile Ranks:** Determine the percentile rank of each product based on sales to identify top-performing products.
- **Healthcare:**
  - **Patient Readmissions:** Calculate the time difference between hospital admissions to track patient readmissions.
  - **Rank Patients:** Rank patients based on the number of visits or treatments received in a given period.
- **Telecommunications:**
  - **Session Analysis:** Compute the duration of each call session and the cumulative

duration of all sessions for a user.
    - **Data Usage**: Calculate the rank of users based on their data usage to identify heavy data users.
- **Retail:**
    - **Customer Segmentation:** Rank customers by their total spend to identify high-value customers.
    - **Sales Trends**: Calculate the difference in sales between consecutive months to identify sales trends.
- **Human Resources:**
    - **Employee Performance**: Rank employees based on performance metrics within each department.
    - **Salary Analysis:** Compute the difference between an employee's salary and the average salary within their department.

When should you use the Windows function:

Imagine that you have some building blocks, and each building block represents some data. Your task requires you to look at certain groups of blocks or to make new blocks depending on the existing blocks that you have.

- **You Want to Compare Blocks Without Mixing Them Up**
  Imagine you want to see if one block is taller than the blocks right next to it. A Window Function lets you look at each block and its neighbors without mixing them all up, so you can easily compare them.

- **You Want to Count or Add Up Blocks in a Row**
  If you want to count how many blocks TOTAL you have in a column or add up their numbers, a Window Function can do that for you, looking at each block one by one and keeping a running total. It can help you find a running average of those blocks as well!

- **You Want to Find the Biggest or Smallest Block in a Section**
  Let's say you have your blocks sorted in rows by color, and you want to find the biggest block in each row. A Window Function helps you look at each row separately and pick out the biggest block in each one.

- **You Want to Give Blocks a Score or a Rank**
  If you want to give each block a score or a rank based on its size or color, a Window Function can do that too. It looks at all the blocks, sorts them how you want, and then gives each one a number to show its rank in the overall set of blocks.

- **You Want to See How Blocks Compare to Their Friends**
  Maybe you want to see if a block is taller than the average height of the blocks around it. A Window Function can look at a block and its buddies, calculate the average height, and then tell you how that block compares.

Here is the basic syntax of the Windows functions which we will help you understand how Windows

functions work. All the concepts mentioned in the syntax will be covered in detail in the upcoming section of the session.

```
SELECT
  window_function() OVER(
       PARTITION BY partition_expression
       ORDER BY order_expression
       window_frame_extent
  ) AS window_column_alias
  FROM table_name
```

Components:

1. **Window Function:** The functions being used are ROW_NUMBER(), RANK(), DENSE_RANK(), SUM(), AVG(), etc.
2. **OVER Clause:** Specifies the window for the function. This is required for a window function to work.
3. **PARTITION BY (optional):** If you want to perform your calculations on specific chunks (groups) of your data, this is how you tell SQL to divide things up. If no PARTITION BY is specified, the function treats all rows of the query result set as a single partition. It works similarly to the GROUP BY clause, but while GROUP BY aggregates the data, PARTITION BY doesn't, it just groups the data for the purpose of the window function.
4. **ORDER BY (optional):** Defines the logical order of rows within each partition to which the window function is applied.
5. **Frame Clause (optional):** Specifies the subset of rows within the partition to be considered for the window function. The default frame is RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW for functions like SUM() and AVG().

Now, we will move into understanding the above syntax using some of the examples.

NOTE: Initially we will introduce a small sample dataset where we will understand the concepts and then we will move towards performing similar concepts in our dataset which we introduced at the beginning of the session.

EXAMPLES: (Dataset)

NOTE: Here as we haven't introduced all the Windows functions and their use cases, thus we perform the initial set of examples around the aggregate functions which we have already discussed in our previous session. So, the discussion will be more about how aggregate functions work as Windows functions.

Keeping in mind all the components of the syntax let's proceed ahead.

Let's imagine that we have some simple Sales Data and line items for this sales data.

1. **Calculate the running total for the sales:**

   First, we need to understand what running total means before jumping onto writing and discussing the query for it.

   So, a **running total** is a way to keep track of the sum of values as you go along, updating the total each time you add a new value. Think of it like adding money to a savings jar over time and keeping a record of how much you have after each addition.

   Here is the way to do so using Windows functions using the sum aggregate function.

```
SELECT SaleID, Salesperson, SaleAmount, SaleDate,
  SUM(SaleAmount) OVER (ORDER BY SaleDate) AS RunningTotal
FROM Sales_sample;
```

Output:

Result Grid | Filter Rows: | Export: | Wrap Cell Content

SaleID	Salesperson	SaleAmount	SaleDate	RunningTotal
1	Alice	300	2023-01-01	300
2	Bob	150	2023-01-02	450
3	Alice	200	2023-01-03	650
4	Charlie	250	2023-01-04	900
5	Bob	300	2023-01-05	1200
6	Alice	100	2023-01-06	1300
7	Charlie	350	2023-01-07	1650
8	Alice	450	2023-01-08	2100
9	Bob	200	2023-01-09	2300
10	Charlie	400	2023-01-10	2700
11	Alice	150	2023-01-11	2850
12	Bob	250	2023-01-12	3100
13	Charlie	300	2023-01-13	3400
14	Alice	350	2023-01-14	3750
15	Bob	100	2023-01-15	3850

Result 1

2. **Calculating the cumulative total sales by salesperson:**

   Again jumping onto writing the query we must understand the general concept behind the cumulative totals.

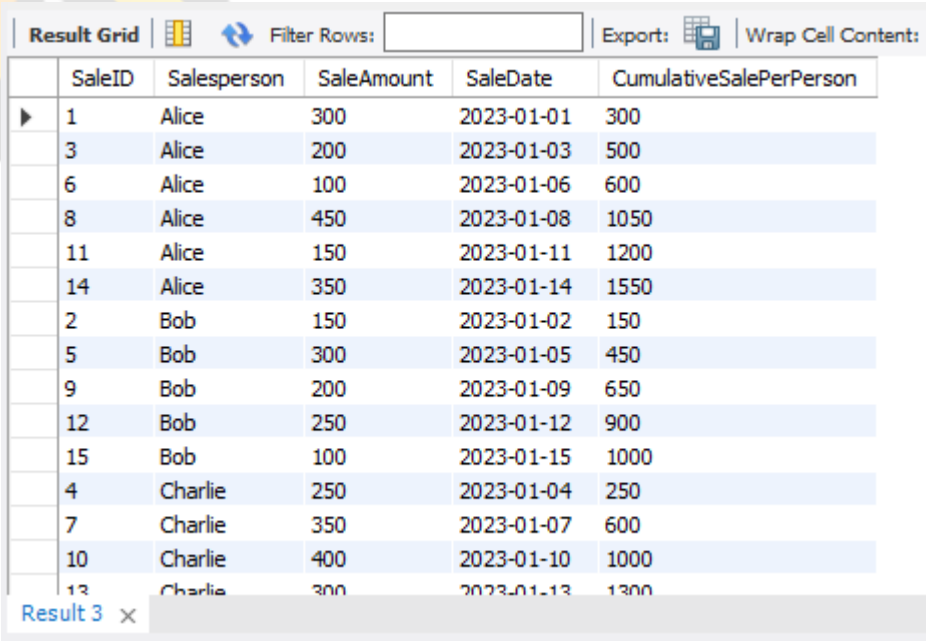
   A **cumulative total** is the same as a running total. It represents the sum of a series of values over time, where each value is added to the previous total to get the new total. It shows the aggregate amount accumulated up to each point in a series.

   Now that we know what cumulative total is, let's discuss why we need to calculate the cumulative total of a particular parameter. So here, we are using a sales dataset that indicates sales made by different salesperson which becomes important at times

for the company to check the details according to the salesperson. How much sales they have made during a particular period.

```
SELECT SaleID, Salesperson, SaleAmount, SaleDate,
  SUM(SaleAmount) OVER (PARTITION BY Salesperson ORDER BY SaleDate) AS
CumulativeSalePerPerson
FROM Sales_sample;
```

Output:



Result Grid | Filter Rows: | Export: | Wrap Cell Content:

SaleID	Salesperson	SaleAmount	SaleDate	CumulativeSalePerPerson
1	Alice	300	2023-01-01	300
3	Alice	200	2023-01-03	500
6	Alice	100	2023-01-06	600
8	Alice	450	2023-01-08	1050
11	Alice	150	2023-01-11	1200
14	Alice	350	2023-01-14	1550
2	Bob	150	2023-01-02	150
5	Bob	300	2023-01-05	450
9	Bob	200	2023-01-09	650
12	Bob	250	2023-01-12	900
15	Bob	100	2023-01-15	1000
4	Charlie	250	2023-01-04	250
7	Charlie	350	2023-01-07	600
10	Charlie	400	2023-01-10	1000
13	Charlie	300	2023-01-13	1300

Result 3

3. **Ranking Sales by Sales Amount:**

   The ranking of sales will help us rank the sales of the salesperson and will help us identify the salesperson with the highest sales made for the present year. (accordingly, the parameters can be set.

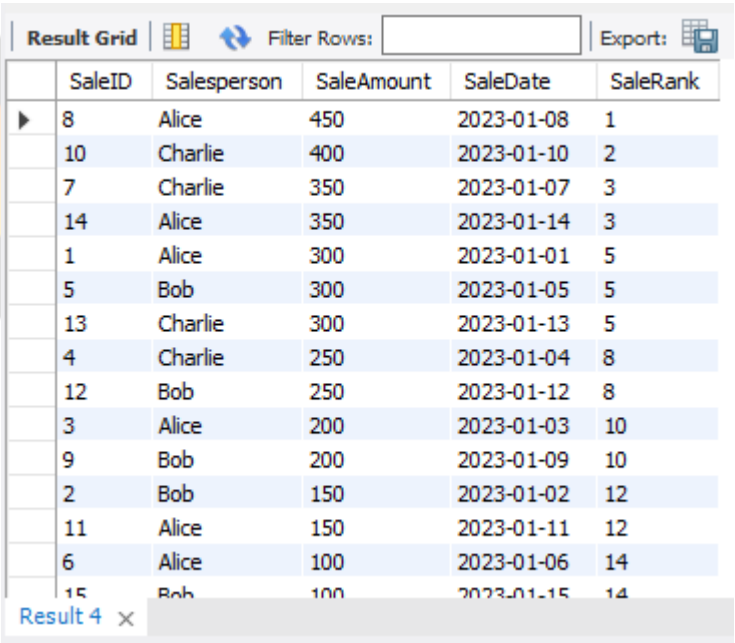
   Here, we will use the RANK() function which will help us rank the salesperson (in this context), decide the rank of the students in a class, etc.
   The rank () function decides the rank according to the specified column and allows the same rank to the person who has the same value in the specified column.

   Let's see how to rank the salesperson in the example on which we have been working. This will clear the concept of how the rank() function works.

   NOTE: In the second half of this session we will introduce some of the Windows functions which is majorly used in the industry. There we will again come across this concept.

```
SELECT SaleID, Salesperson, SaleAmount, SaleDate,
  RANK() OVER (ORDER BY SaleAmount DESC) AS SaleRank
FROM Sales_sample;
```

Output:



Result Grid | Filter Rows: | Export:

SaleID	Salesperson	SaleAmount	SaleDate	SaleRank
8	Alice	450	2023-01-08	1
10	Charlie	400	2023-01-10	2
7	Charlie	350	2023-01-07	3
14	Alice	350	2023-01-14	3
1	Alice	300	2023-01-01	5
5	Bob	300	2023-01-05	5
13	Charlie	300	2023-01-13	5
4	Charlie	250	2023-01-04	8
12	Bob	250	2023-01-12	8
3	Alice	200	2023-01-03	10
9	Bob	200	2023-01-09	10
2	Bob	150	2023-01-02	12
11	Alice	150	2023-01-11	12
6	Alice	100	2023-01-06	14
15	Bob	100	2023-01-15	14

Result 4

In the above output, we can see Charlie and Alice in the third and fourth rows have the same rank because the saleamount is the same. (In the upcoming section we will understand other concepts of handling these issues if you do not want to allot the same rank to the person with the same marks or saleamount)

4. **Moving Average (3-Day) of SalesAmount:**

   Let's discuss the concepts of moving average and then we will move on to the query part.

   A moving average (3 days) is a method used to smooth out short-term fluctuations and highlight longer-term trends or cycles. It calculates the average of a subset of data points over a specific period, which in this case is 3 days.

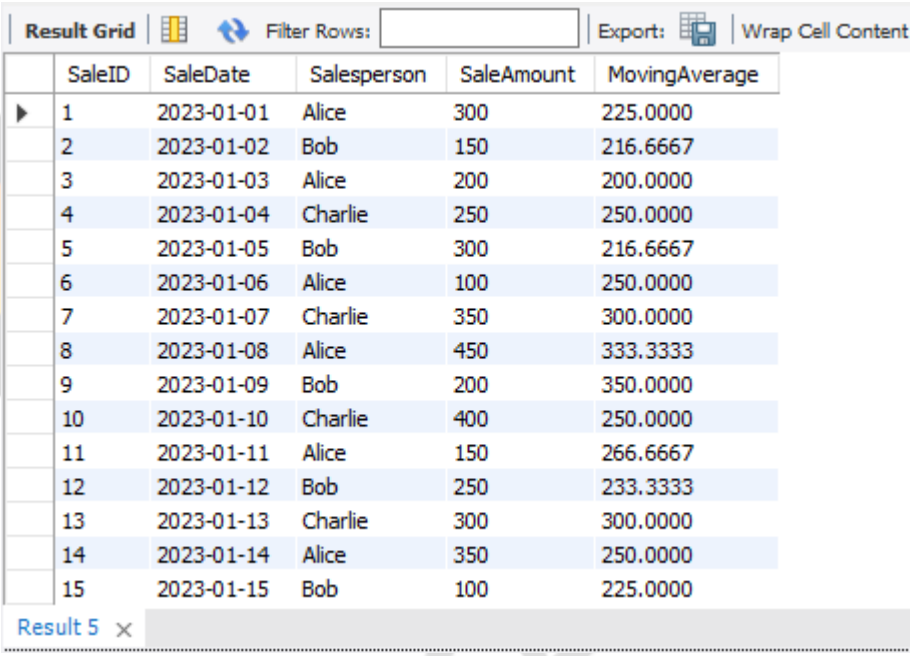
   This example will give you a better understanding of how moving averages work and give you a clear picture of how we can get the moving average value.

   Here we simply use the avg() aggregate function using some extra conditions.

```
SELECT SaleID, SaleDate, Salesperson, SaleAmount,
       AVG(SaleAmount) OVER (
                   ORDER BY SaleDate
                   ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING
                           ) AS MovingAverage
```

```
FROM Sales_sample;
```

Output:



Result Grid | Filter Rows: | Export: | Wrap Cell Content

SaleID	SaleDate	Salesperson	SaleAmount	MovingAverage
1	2023-01-01	Alice	300	225.0000
2	2023-01-02	Bob	150	216.6667
3	2023-01-03	Alice	200	200.0000
4	2023-01-04	Charlie	250	250.0000
5	2023-01-05	Bob	300	216.6667
6	2023-01-06	Alice	100	250.0000
7	2023-01-07	Charlie	350	300.0000
8	2023-01-08	Alice	450	333.3333
9	2023-01-09	Bob	200	350.0000
10	2023-01-10	Charlie	400	250.0000
11	2023-01-11	Alice	150	266.6667
12	2023-01-12	Bob	250	233.3333
13	2023-01-13	Charlie	300	300.0000
14	2023-01-14	Alice	350	250.0000
15	2023-01-15	Bob	100	225.0000

Result 5

For the simplicity of this example, we've used a 3-day WINDOW (3-day Moving Average), but it could just as easily have been a 7-day (Weekly MA), 30-day (Monthly), or any period of time you decide to look at.

Understanding the purpose and functionality of each window analytic function:

1. **Aggregate Windows functions:**

   We have already studied a set of aggregation functions in one of our previous sessions which included some functions like avg(), sum(), min(), max(), and count(). The aggregate windows function also contains these functions. But you might think how is the aggregation functions and the aggregate windows functions different from each other. Here is how:

   Difference between aggregate functions and aggregate windows function:

   The main difference between Window Functions and GROUP BY Aggregate Functions is that while an Aggregate Function returns a single result per group of rows (like the SUM or AVG of a group), a Window Function will return a result for every row, often in relation to other rows in the window (like the running total at each row).

   Here's an **analogy** to understand the difference:

   Imagine you are analyzing the performance of a football team over a season. You want to know the total number of goals scored by the team in each match.

If we use the **aggregation functions**, you get the total goals scored in each match, but you don't see the individual contributions of each player in those matches.

Now, imagine you want to analyze the performance of individual players in each match, while also keeping track of the total goals scored in each match.

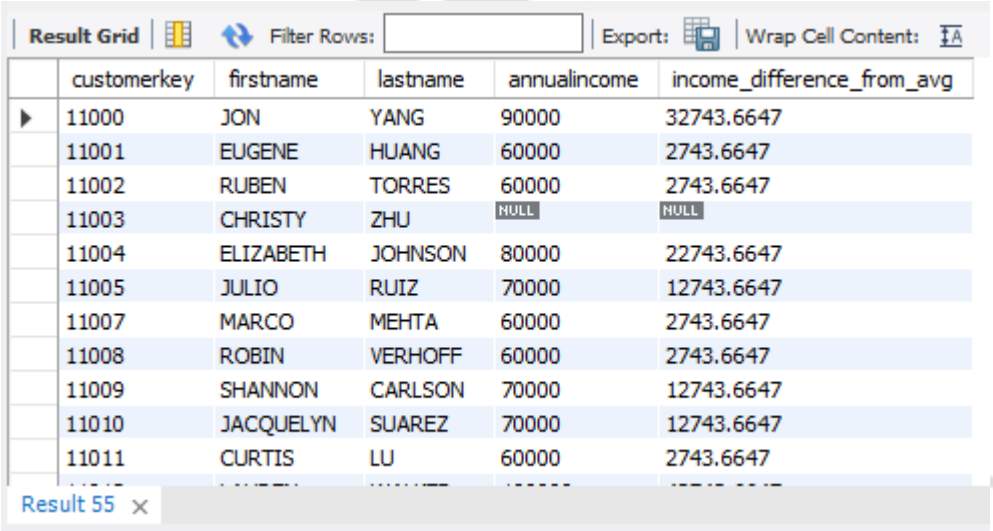
This will be accomplished by **aggregate window functions** where each player sees their own goals and the total goals scored by the team in each match, retaining the details of each player's contribution.

Let's look into some of the examples related to aggregate windows functions:

Example 1:

```
SELECT customerkey, firstname, lastname, annualincome,
    annualincome - AVG(annualincome) OVER () AS
income_difference_from_avg
FROM customers;
```

Output:

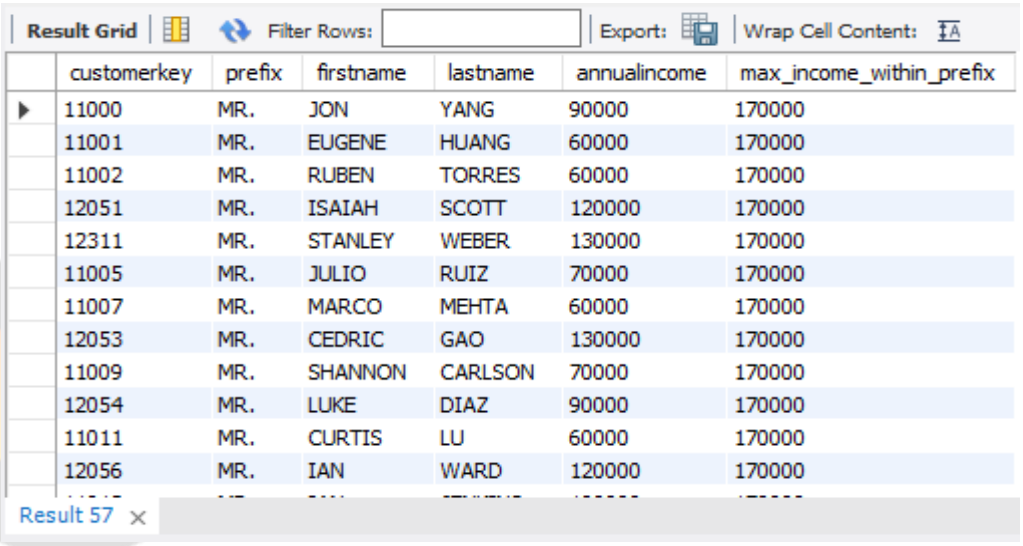


customerkey	firstname	lastname	annualincome	income_difference_from_avg
11000	JON	YANG	90000	32743.6647
11001	EUGENE	HUANG	60000	2743.6647
11002	RUBEN	TORRES	60000	2743.6647
11003	CHRISTY	ZHU	NULL	NULL
11004	ELIZABETH	JOHNSON	80000	22743.6647
11005	JULIO	RUIZ	70000	12743.6647
11007	MARCO	MEHTA	60000	2743.6647
11008	ROBIN	VERHOFF	60000	2743.6647
11009	SHANNON	CARLSON	70000	12743.6647
11010	JACQUELYN	SUAREZ	70000	12743.6647
11011	CURTIS	LU	60000	2743.6647

Example 2:

```
SELECT customerkey, prefix, firstname, lastname, annualincome,
    MAX(annualincome) OVER (PARTITION BY prefix) AS
max_income_within_prefix
FROM customers
where prefix is not null;
```

Output:



Result Grid | Filter Rows: | Export: | Wrap Cell Content:

customerkey	prefix	firstname	lastname	annualincome	max_income_within_prefix
11000	MR.	JON	YANG	90000	170000
11001	MR.	EUGENE	HUANG	60000	170000
11002	MR.	RUBEN	TORRES	60000	170000
12051	MR.	ISAIAH	SCOTT	120000	170000
12311	MR.	STANLEY	WEBER	130000	170000
11005	MR.	JULIO	RUIZ	70000	170000
11007	MR.	MARCO	MEHTA	60000	170000
12053	MR.	CEDRIC	GAO	130000	170000
11009	MR.	SHANNON	CARLSON	70000	170000
12054	MR.	LUKE	DIAZ	90000	170000
11011	MR.	CURTIS	LU	60000	170000
12056	MR.	IAN	WARD	120000	170000

Result 57

Example 3:

```
SELECT customerkey, firstname, lastname, annualincome,
    (annualincome * 100.0 / SUM(annualincome) OVER ()) AS
income_percentage
FROM customers;
```

Output:

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

customerkey	firstname	lastname	annualincome	income_percentage
11000	JON	YANG	90000	0.07660
11001	EUGENE	HUANG	60000	0.05107
11002	RUBEN	TORRES	60000	0.05107
11003	CHRISTY	ZHU	NULL	NULL
11004	ELIZABETH	JOHNSON	80000	0.06809
11005	JULIO	RUIZ	70000	0.05958
11007	MARCO	MEHTA	60000	0.05107
11008	ROBIN	VERHOFF	60000	0.05107
11009	SHANNON	CARLSON	70000	0.05958
11010	JACQUELYN	SUAREZ	70000	0.05958
11011	CURTIS	LU	60000	0.05107
11012	LAUREN	WALKER	100000	0.08511

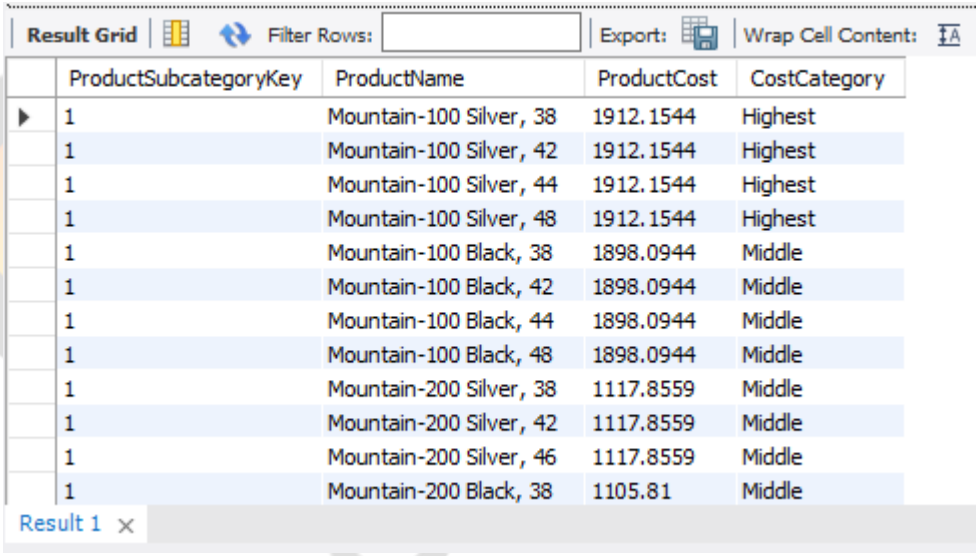
Result 58

Example 4: (Using CASE WHEN statement in window function)

```
SELECT ProductSubcategoryKey, ProductName, ProductCost,
       CASE
           WHEN ProductCost = MAX(ProductCost) OVER (PARTITION BY
ProductSubcategoryKey) THEN 'Highest'
           WHEN ProductCost = MIN(ProductCost) OVER (PARTITION BY
ProductSubcategoryKey) THEN 'Lowest'
           ELSE 'Middle'
```

```
        END AS CostCategory
FROM products;
```

Output:



Result Grid | Filter Rows: | Export: | Wrap Cell Content:

ProductSubcategoryKey	ProductName	ProductCost	CostCategory
1	Mountain-100 Silver, 38	1912.1544	Highest
1	Mountain-100 Silver, 42	1912.1544	Highest
1	Mountain-100 Silver, 44	1912.1544	Highest
1	Mountain-100 Silver, 48	1912.1544	Highest
1	Mountain-100 Black, 38	1898.0944	Middle
1	Mountain-100 Black, 42	1898.0944	Middle
1	Mountain-100 Black, 44	1898.0944	Middle
1	Mountain-100 Black, 48	1898.0944	Middle
1	Mountain-200 Silver, 38	1117.8559	Middle
1	Mountain-200 Silver, 42	1117.8559	Middle
1	Mountain-200 Silver, 46	1117.8559	Middle
1	Mountain-200 Black, 38	1105.81	Middle

Result 1

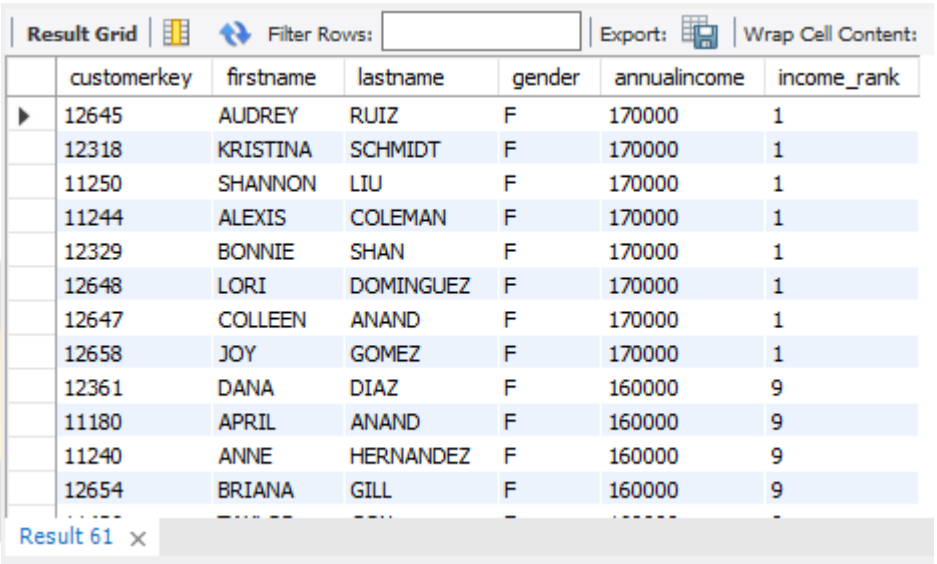
2. **Ranking functions:**

- **RANK()**: This function assigns a unique rank to each distinct row within the partition of a result set. The ranks are assigned in the order specified in the ORDER BY clause of the OVER() clause. If two or more rows tie for a rank, each tied row receives the same rank, and the next rank(s) are skipped.

Example 1:

```
SELECT customerkey, firstname, lastname, gender, annualincome,
    RANK() OVER (partition by gender ORDER BY annualincome DESC) AS
income_rank
FROM customers;
```

Output:



Result Grid | Filter Rows: | Export: | Wrap Cell Content:

customerkey	firstname	lastname	gender	annualincome	income_rank
12645	AUDREY	RUIZ	F	170000	1
12318	KRISTINA	SCHMIDT	F	170000	1
11250	SHANNON	LIU	F	170000	1
11244	ALEXIS	COLEMAN	F	170000	1
12329	BONNIE	SHAN	F	170000	1
12648	LORI	DOMINGUEZ	F	170000	1
12647	COLLEEN	ANAND	F	170000	1
12658	JOY	GOMEZ	F	170000	1
12361	DANA	DIAZ	F	160000	9
11180	APRIL	ANAND	F	160000	9
11240	ANNE	HERNANDEZ	F	160000	9
12654	BRIANA	GILL	F	160000	9

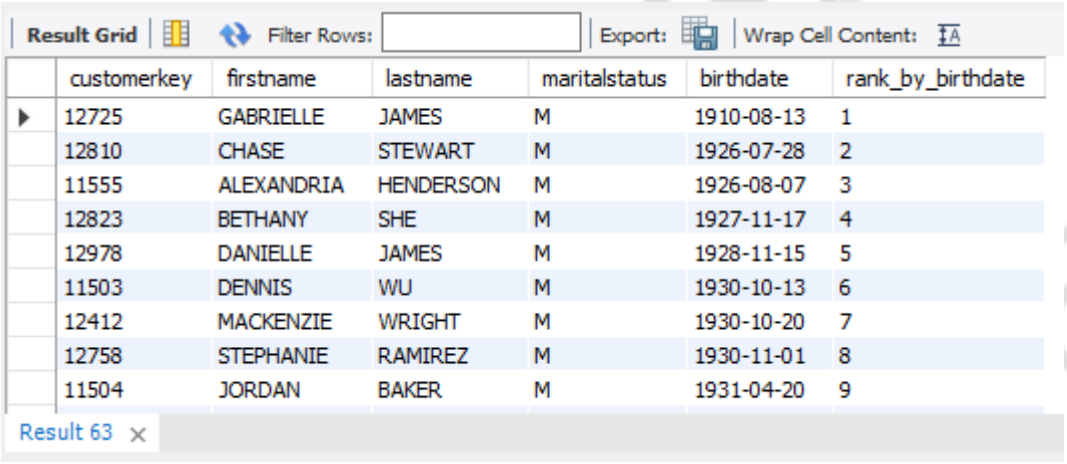
Result 61

Here we can observe that the first rank has been allotted to 8 customers as they have the same annualincome and the next person in the list has been allotted rank 9. It identifies the number of same entries and the next number is allotted according to the number of entries. At times, this ranking system might create doubt while analyzing data. To remove this ambiguity we will be using DENSE Rank() function. (we will discuss about it in the nxt section of this session)

Example 2:

```
SELECT customerkey,firstname,lastname,maritalstatus, birthdate,
    RANK() OVER (PARTITION BY maritalstatus ORDER BY birthdate) AS
rank_by_birthdate
FROM customers;
```

Output:



Result Grid | Filter Rows: | Export: | Wrap Cell Content:

customerkey	firstname	lastname	maritalstatus	birthdate	rank_by_birthdate
12725	GABRIELLE	JAMES	M	1910-08-13	1
12810	CHASE	STEWART	M	1926-07-28	2
11555	ALEXANDRIA	HENDERSON	M	1926-08-07	3
12823	BETHANY	SHE	M	1927-11-17	4
12978	DANIELLE	JAMES	M	1928-11-15	5
11503	DENNIS	WU	M	1930-10-13	6
12412	MACKENZIE	WRIGHT	M	1930-10-20	7
12758	STEPHANIE	RAMIREZ	M	1930-11-01	8
11504	JORDAN	BAKER	M	1931-04-20	9

Result 63

Example 3:

```
SELECT
    ProductSubcategoryKey,
    ProductName,
    ProductCost,
```

```
    RANK() OVER (PARTITION BY ProductSubcategoryKey ORDER BY ProductCost
DESC) AS ProductRank,
    CASE
        WHEN RANK() OVER (PARTITION BY ProductSubcategoryKey ORDER BY
ProductCost DESC) = 1 THEN 'Top'
        WHEN RANK() OVER (PARTITION BY ProductSubcategoryKey ORDER BY
ProductCost DESC) <= 3 THEN 'Middle'
        ELSE 'Bottom'
    END AS CostCategory
FROM
    products;
```

Output:

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

ProductSubcategoryKey	ProductName	ProductCost	ProductRank	CostCategory
1	Mountain-100 Silver, 38	1912.1544	1	Top
1	Mountain-100 Silver, 42	1912.1544	1	Top
1	Mountain-100 Silver, 44	1912.1544	1	Top
1	Mountain-100 Silver, 48	1912.1544	1	Top
1	Mountain-100 Black, 38	1898.0944	5	Bottom
1	Mountain-100 Black, 42	1898.0944	5	Bottom
1	Mountain-100 Black, 44	1898.0944	5	Bottom
1	Mountain-100 Black, 48	1898.0944	5	Bottom
1	Mountain-200 Silver, 38	1117.8559	9	Bottom
1	Mountain-200 Silver, 42	1117.8559	9	Bottom
1	Mountain-200 Silver, 46	1117.8559	9	Bottom
1	Mountain-200 Black, 38	1105.81	12	Bottom

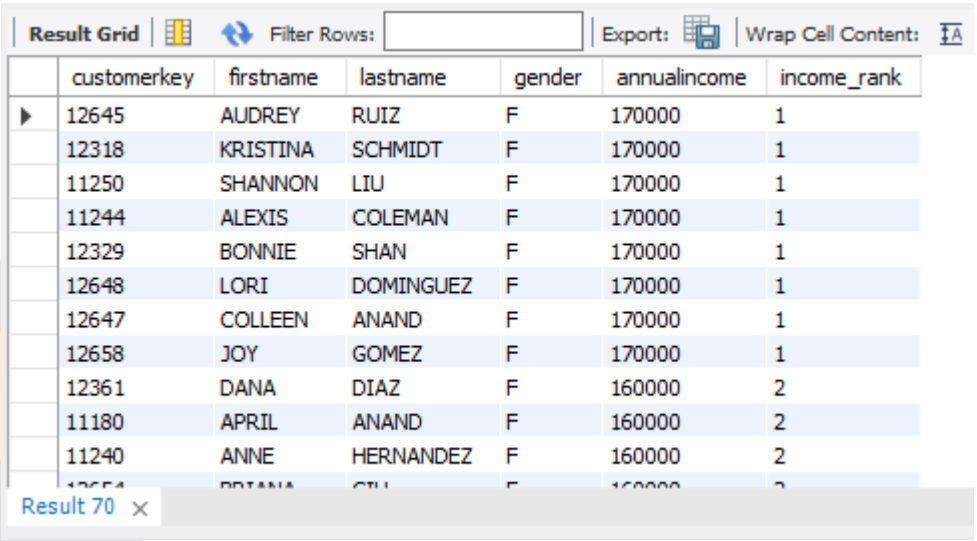
Result 2

- **DENSE_RANK():** This function works similarly to RANK(), but when two or more rows tie for a rank, the next rank is not skipped. So if you have three items at rank 2, the next rank listed would be 3.

Example 1:

```
SELECT customerkey, firstname, lastname, gender, annualincome,
    dense_RANK() OVER (partition by gender ORDER BY annualincome DESC)
AS income_rank
FROM customers;
```

Output:



Result Grid | Filter Rows: | Export: | Wrap Cell Content:

customerkey	firstname	lastname	gender	annualincome	income_rank
12645	AUDREY	RUIZ	F	170000	1
12318	KRISTINA	SCHMIDT	F	170000	1
11250	SHANNON	LIU	F	170000	1
11244	ALEXIS	COLEMAN	F	170000	1
12329	BONNIE	SHAN	F	170000	1
12648	LORI	DOMINGUEZ	F	170000	1
12647	COLLEEN	ANAND	F	170000	1
12658	JOY	GOMEZ	F	170000	1
12361	DANA	DIAZ	F	160000	2
11180	APRIL	ANAND	F	160000	2
11240	ANNE	HERNANDEZ	F	160000	2

Result 70

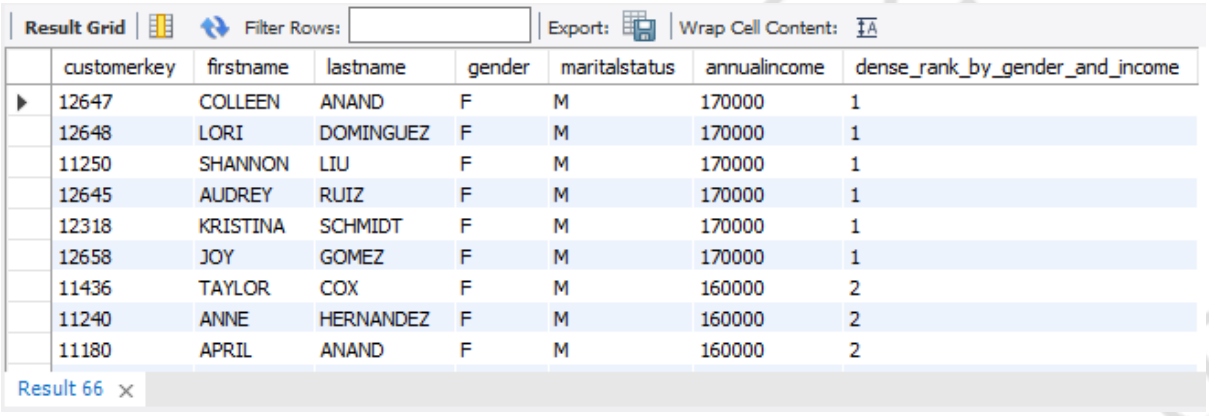
If you notice this particular example is similar to one of the examples we have done in the rank() function part. So the ambiguity that we were talking about in that section will be fixed using the dense_rank() function. This function helps us maintain continuity while ranking the data on your desired column.

In the output above we can see that the top-rank holders are 8 in number and the next rank allotted is not 9, rather it is 2 which follows the desired ranking.

Example 2:

```
SELECT customerkey, firstname, lastname, gender, maritalstatus,
annualincome,
    DENSE_RANK() OVER (PARTITION BY gender, maritalstatus ORDER BY
annualincome DESC) AS dense_rank_by_gender_and_income
FROM customers;
```

Output:



Result Grid | Filter Rows: | Export: | Wrap Cell Content:

customerkey	firstname	lastname	gender	maritalstatus	annualincome	dense_rank_by_gender_and_income
12647	COLLEEN	ANAND	F	M	170000	1
12648	LORI	DOMINGUEZ	F	M	170000	1
11250	SHANNON	LIU	F	M	170000	1
12645	AUDREY	RUIZ	F	M	170000	1
12318	KRISTINA	SCHMIDT	F	M	170000	1
12658	JOY	GOMEZ	F	M	170000	1
11436	TAYLOR	COX	F	M	160000	2
11240	ANNE	HERNANDEZ	F	M	160000	2
11180	APRIL	ANAND	F	M	160000	2

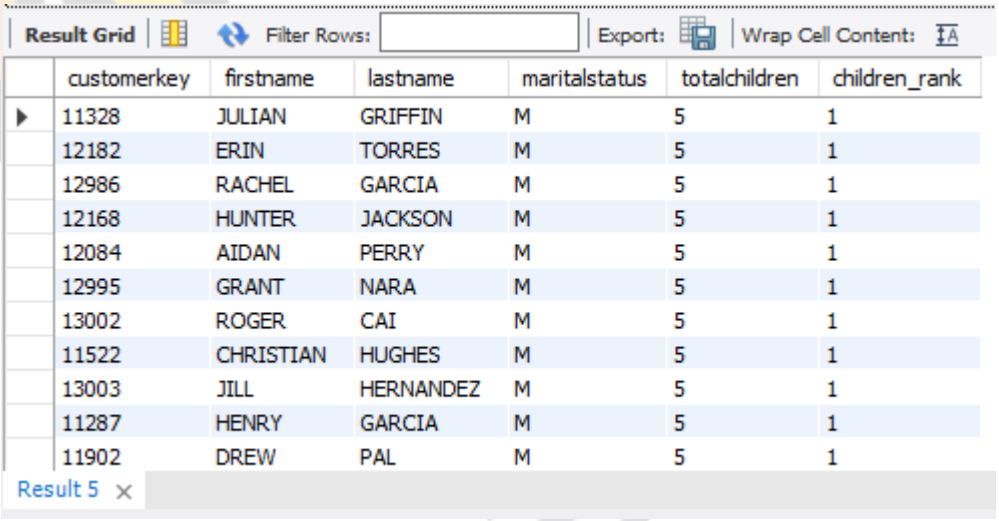
Result 66

Example 3:

```
SELECT
    customerkey,
    firstname,
    lastname,
```

```
    maritalstatus,
    totalchildrens,
    DENSE_RANK() OVER (PARTITION BY maritalstatus ORDER BY
totalchildrens DESC) AS children_rank
FROM
    customers;
```

Output:



Result Grid | Filter Rows: | Export: | Wrap Cell Content:

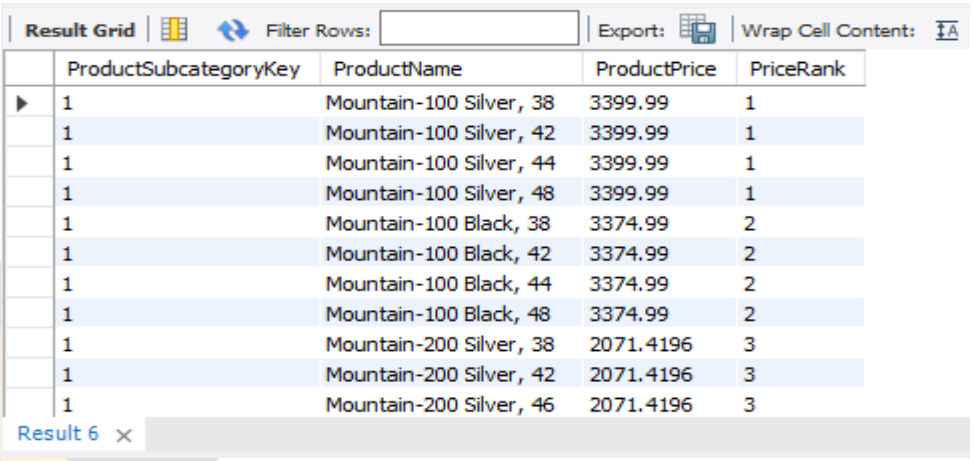
customerkey	firstname	lastname	maritalstatus	totalchildren	children_rank
11328	JULIAN	GRIFFIN	M	5	1
12182	ERIN	TORRES	M	5	1
12986	RACHEL	GARCIA	M	5	1
12168	HUNTER	JACKSON	M	5	1
12084	AIDAN	PERRY	M	5	1
12995	GRANT	NARA	M	5	1
13002	ROGER	CAI	M	5	1
11522	CHRISTIAN	HUGHES	M	5	1
13003	JILL	HERNANDEZ	M	5	1
11287	HENRY	GARCIA	M	5	1
11902	DREW	PAL	M	5	1

Result 5

Example 4:

```
SELECT
    ProductSubcategoryKey,
    ProductName,
    COALESCE(ProductPrice, 0) AS ProductPrice,
    DENSE_RANK() OVER (PARTITION BY ProductSubcategoryKey ORDER BY
COALESCE(ProductPrice, 0) DESC) AS PriceRank
FROM
    products;
```

Output:



Result Grid | Filter Rows: | Export: | Wrap Cell Content:

ProductSubcategoryKey	ProductName	ProductPrice	PriceRank
1	Mountain-100 Silver, 38	3399.99	1
1	Mountain-100 Silver, 42	3399.99	1
1	Mountain-100 Silver, 44	3399.99	1
1	Mountain-100 Silver, 48	3399.99	1
1	Mountain-100 Black, 38	3374.99	2
1	Mountain-100 Black, 42	3374.99	2
1	Mountain-100 Black, 44	3374.99	2
1	Mountain-100 Black, 48	3374.99	2
1	Mountain-200 Silver, 38	2071.4196	3
1	Mountain-200 Silver, 42	2071.4196	3
1	Mountain-200 Silver, 46	2071.4196	3

Result 6

- **ROW_NUMBER():** This function assigns a unique row number to each row within the partition, regardless of duplicates. If there are duplicate values in the ordered set, it will still assign different row numbers to each row.

Example 1:

```
SELECT customerkey, firstname, lastname, gender, totalchildren,
    ROW_NUMBER() OVER (PARTITION BY gender ORDER BY totalchildren DESC)
AS row_num_by_children
FROM customers;
```

Output:

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

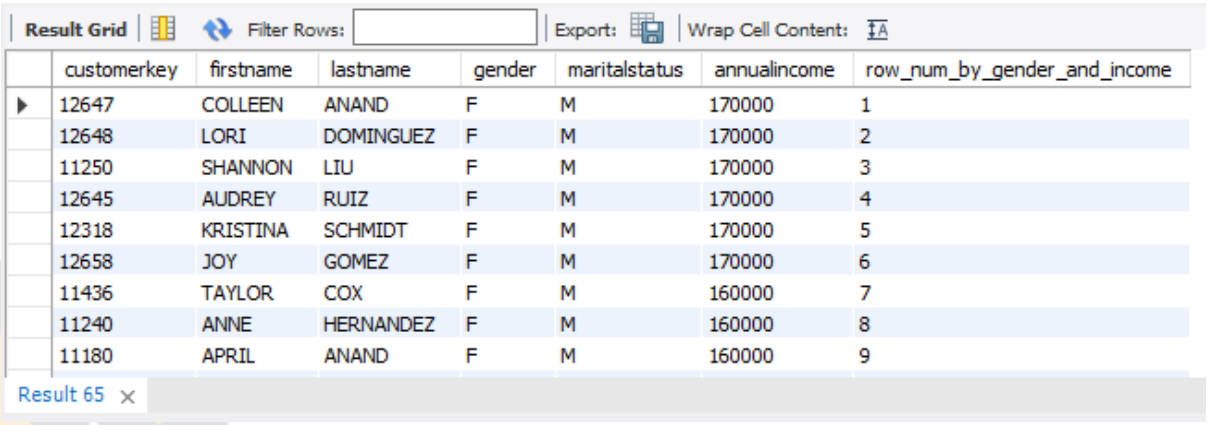
customerkey	firstname	lastname	gender	totalchildren	row_num_by_children
11884	LATOYA	SHEN	F	5	1
12183	EMILY	ROBINSON	F	5	2
11202	ALEXIA	PRICE	F	5	3
11883	HANNAH	ANDERSON	F	5	4
11004	ELIZABETH	JOHNSON	F	5	5
12787	KATELYN	COOK	F	5	6
12663	MORGAN	ADAMS	F	5	7
12149	BAILEY	SCOTT	F	5	8
12008	SHARON	YUAN	F	5	9

Result 64

Example 2:

```
SELECT customerkey, firstname, lastname, gender, maritalstatus,
annualincome,
    ROW_NUMBER() OVER (PARTITION BY gender, maritalstatus ORDER BY
annualincome DESC) AS row_num_by_gender_and_income
FROM customers;
```

Output:



Result Grid | Filter Rows: | Export: | Wrap Cell Content:

customerkey	firstname	lastname	gender	maritalstatus	annualincome	row_num_by_gender_and_income
12647	COLLEEN	ANAND	F	M	170000	1
12648	LORI	DOMINGUEZ	F	M	170000	2
11250	SHANNON	LIU	F	M	170000	3
12645	AUDREY	RUIZ	F	M	170000	4
12318	KRISTINA	SCHMIDT	F	M	170000	5
12658	JOY	GOMEZ	F	M	170000	6
11436	TAYLOR	COX	F	M	160000	7
11240	ANNE	HERNANDEZ	F	M	160000	8
11180	APRIL	ANAND	F	M	160000	9

Result 65

When to use which kind of ranking function:

1. **RANK():**
   Use the RANK() function when you want to assign ranks to rows within a partition, and you want to handle ties by giving the same rank to tied rows, leaving gaps in the ranking sequence for the next distinct value.

   Imagine you have a group of runners in a race, and you want to rank them based on their finishing times. If multiple runners finish at the exact same time, they would be considered tied. In this scenario, the RANK() function would assign the same rank to all the tied runners, and then it would skip the next rank value(s) in the sequence to account for the tie.

2. **DENSE_RANK():**
   Use the DENSE_RANK() function when you want to assign ranks to rows within a partition, and you want to handle ties by giving the same rank to tied rows, but without leaving gaps in the ranking sequence.

   In contrast to the previous example in RANK(), if you don't want any gaps in the ranking sequence when handling ties, you would use the DENSE_RANK() function instead.

3. **ROW_NUMBER:**
   Use the ROW_NUMBER() function when you want to assign a unique sequential number to each row within a partition, even if there are ties. The row numbers are assigned based on the ordering specified, and ties are handled by assigning the same row number to tied rows.

   For instance, if you have a dataset of orders and want to assign a unique sequential number to each order within a customer, even if multiple orders have the same order date, you would use ROW_NUMBER(). Tied orders (with the same order date) within a customer will be assigned the same row number.

Lastly, we can have a section that will help us understand how Windows functions are helpful to us. Let’s look into the example.

Without window functions:

```
SELECT t1.order_id, t1.order_date, t1.order_amount,
SUM(t2.order_amount) AS RunningTotal
FROM orders t1
JOIN orders t2 ON t1.order_date >= t2.order_date
GROUP BY t1.order_id, t1.order_date, t1.order_amount
ORDER BY t1.order_date;
```

With window functions:

```
SELECT order_id, order_date, order_amount,
SUM(order_amount) OVER (ORDER BY order_date) AS RunningTotal
FROM orders;
```

NOTE: The window functions module hasn't yet been covered completely. In the next session, we will continue with some of the important window functions and deep dive into critical queries which will help us generate different sets of insights using Windows functions.