CONTENTS:

Math Functions

Math Functions

What is the Order of Precedents?

Order of Precedents for Math

```
()  Parentheses
**  Exponentiation
*   Multiplication
/   Division
+   Addition
-   Subtraction
```

Mathematics can be used within your Syntax. But, like any math problems, it has an Order of Precedents.

What is the Answer to this Math Question?

```
SELECT (2 + 4) * 5
```

What is the Answer?

30

This is an example of mathematics taking place with the Order of Precedents being respected.

What is the Answer to this Math Question?

```
SELECT 2 + 4 / 5
```

What is the Answer?

2

The answer to this example is 2. This is because, when a decimal isn't used within the equation, the system will NOT allow for a decimal to be part of the answer set.

Instead of 2.8, it drops the decimal altogether and just gives you a 2.

What is the Answer to this Math Question?

```
SELECT 2 + 4.0 / 5
```

What is the Answer?

2.8

In order to see the decimal in your answer set, you must have at least one decimal in your equation.

OLAP Functions

OLAP Functions

On-Line Analytical Processing (OLAP) or Ordered Analytics

```
SELECT Product_ID
       ,Sale_Date
       ,Daily_Sales
       ,CSUM(Daily_Sales, Sale_Date)   AS "CSum"
FROM   Sales_Table ;
```

1 Sort the Answer Set first by Sale_Date

2 Calculate the CSUM column second now that the data is sorted

OLAP is often called Ordered Analytics because the first thing every OLAP does before any calculating is SORT all the rows. The query above sorts by Sale_Date!

Cumulative Sum (CSUM) Command and how OLAP Works

```
SELECT   Product_ID, Sale_Date, Daily_Sales
         ,CSUM(Daily_Sales, Sale_Date)   AS "CSum"
FROM     Sales_Table ;
```

Product_ID	Sale_Date	Daily_Sales	CSUM
1000	2000-09-28	48850.40	
2000	2000-09-28	41888.88	
3000	2000-09-28	61301.77	
1000	2000-09-29	54500.22	
2000	2000-09-29	48000.00	
3000	2000-09-29	34509.13	
1000	2000-09-30	36000.07	
2000	2000-09-30	49850.03	
3000	2000-09-30	43868.86	
1000	2000-10-01	40200.43	
2000	2000-10-01	54850.29	
3000	2000-10-01	28000.00	
1000	2000-10-02	32800.50	
2000	2000-10-02	36021.93	
3000	2000-10-02	19678.94	

Not all rows are displayed in this answer set

1 Sort the Answer Set first by Sale_Date, but Don't do any CSUM Calculations yet!

OLAP always sorts first, and then it is in a position to calculate starting with the first sorted row and continuing to the last sorted row, thus calculating all Daily_Sales.

OLAP Commands always Sort (ORDER BY) in the Command

```
SELECT   Product_ID, Sale_Date, Daily_Sales
         ,CSUM(Daily_Sales, Sale_Date)   AS "CSum"
FROM     Sales_Table ;
```

Not all rows are displayed in this answer set

Product_ID	Sale_Date	Daily_Sales	CSUM
1000	2000-09-28	48850.40	48850.40
2000	2000-09-28	41888.88	90739.28
3000	2000-09-28	61301.77	152041.05
1000	2000-09-29	54500.22	206541.27
2000	2000-09-29	48000.00	254541.27
3000	2000-09-29	34509.13	289050.40
1000	2000-09-30	36000.07	325050.41
2000	2000-09-30	49850.03	374900.50
3000	2000-09-30	43868.86	418769.36
1000	2000-10-01	40200.43	458969.79
2000	2000-10-01	54850.29	513820.08
3000	2000-10-01	28000.00	541820.08
1000	2000-10-02	32800.50	574620.58

2

Calculate the CSUM starting with the first sorted row and going to the last

Once the data is sorted by Sale_Date, then phase 2 is ready. The OLAP calculation can be performed on the sorted data. On Day 1, we made 48850.40! Add the next row's Daily_Sales to get a Cumulative Sum (CSUM); to get 90739.28!

Calculate the Cumulative Sum (CSUM) after Sorting the Data

```
SELECT  Product_ID, Sale_Date, Daily_Sales,
        CSUM(Daily_Sales, Sale_Date)     AS "CSUM"
FROM    Sales_Table WHERE Product_ID BETWEEN 1000 and 2000
```

Not all rows are displayed in this answer set

Product_ID	Sale_Date	Daily_Sales	CSUM
2000	2000-09-28	41888.88	41888.88
1000	2000-09-28	48850.40	90739.28
2000	2000-09-29	48000.00	138739.28
1000	2000-09-29	54500.22	193239.50
1000	2000-09-30	36000.07	229239.57
2000	2000-09-30	49850.03	279089.60
1000	2000-10-01	40200.43	319290.03
2000	2000-10-01	54850.29	374140.32
1000	2000-10-02	32800.50	406940.82
2000	2000-10-02	36021.93	442962.75
1000	2000-10-03	64300.00	507262.75

This is our first OLAP known as a CSUM. Right now, the syntax wants to see the cumulative sum of the Daily_Sales sorted by Sale_Date. The first thing the above query does before calculating is SORT all the rows on Sale_Date.

The OLAP Major Sort Key

```
SELECT  Product_ID, Sale_Date, Daily_Sales,
        CSUM(Daily_Sales, Sale_Date)   AS "CSUM"
FROM    Sales_Table WHERE Product_ID BETWEEN 1000 and 2000 ;
```

Not all rows are displayed in this answer set

Product_ID	Sale_Date	Daily_Sales	CSUM
2000	2000-09-28	41888.88	41888.88
1000	2000-09-28	48850.40	90739.28
2000	2000-09-29	48000.00	138739.28
1000	2000-09-29	54500.22	193239.50
1000	2000-09-30	36000.07	229239.57
2000	2000-09-30	49850.03	279089.60
1000	2000-10-01	40200.43	319290.03
2000	2000-10-01	54850.29	374140.32
1000	2000-10-02	32800.50	406940.82
2000	2000-10-02	36021.93	442962.75

1000	2000-10-03	64300.00	507262.75

In a CSUM, the second column listed is always the major SORT KEY. The SORT KEY in the above query is Sale_Date. Notice again the answer set is sorted by this. After the sort has finished, the CSUM is calculated starting with the first sorted row till the end.

The OLAP Major Sort Key and the Minor Sort Key(s)

```
SELECT Product_ID , Sale_Date, Daily_Sales,
       CSUM(Daily_Sales, Product_ID, Sale_Date)   AS "CSum"
FROM  Sales_Table;
```

Major Sort Key

Minor Sort Key

Not all rows are displayed in this answer set

Product_ID	Sale_Date	Daily_Sales	CSUM
1000	2000-09-28	48850.40	48850.40
1000	2000-09-29	54500.22	103350.62
1000	2000-09-30	36000.07	139350.69
1000	2000-10-01	40200.43	179551.12
1000	2000-10-02	32800.50	213251.62
1000	2000-10-03	64300.00	276651.62
1000	2000-10-04	54553.10	331204.72
2000	2000-09-28	41888.88	370093.60
2000	2000-09-29	48000.00	421093.60
2000	2000-09-30	49850.03	470943.63
2000	2000-10-01	54850.29	525793.92

Product_ID is the MAJOR sort key, and Sale_Date is the MINOR Sort key above.

Troubleshooting OLAP – My Data isn't coming back correct

```
SELECT Product_ID, Sale_Date, Daily_Sales,
       CSUM(Daily_Sales, Sale_Date)  AS "CSUM"
FROM Sales_Table WHERE Product_ID BETWEEN 1000 and 2000
ORDER BY Daily_Sales;
```

Don't place an ORDER BY at the end!

Product_ID	Sale_Date	Daily_Sales	CSUM
1000	2000-10-02	32800.50	212351.62
2000	2000-10-04	32800.50	637816.53
1000	2000-09-30	36000.07	139350.69
2000	2000-10-02	36021.93	561581.85
1000	2000-10-01	40200.43	179551.12
2000	2000-09-28	41888.88	373093.60
2000	2000-10-03	43200.18	605016.03
2000	2000-09-29	48000.00	421093.40
1000	2000-09-28	48850.40	48850.40
2000	2000-09-30	49850.03	470943.63
1000	2000-09-29	54500.22	103350.62
1000	2000-10-04	54553.10	331204.72
2000	2000-10-01	54850.29	525793.92
1000	2000-10-03	64300.00	276651.62

The first thing every OLAP does is SORT. That means you should NEVER put an ORDER BY at the end. It will mess up the ENTIRE result set.

GROUP BY in Teradata OLAP Syntax Resets on the Group

```
SELECT Product_ID,Sale_Date, Daily_Sales
      ,CSUM(Daily_Sales, Product_ID, Sale_Date)  AS "CSum"
FROM Sales_Table

GROUP BY Product_ID;
```

Product_ID	Sale_Date	Daily_Sales	CSUM
1000	2000-09-28	48850.40	48850.40
1000	2000-09-29	54500.22	103350.62
1000	2000-09-30	36000.07	139350.69
1000	2000-10-01	40200.43	179551.12
1000	2000-10-02	32800.50	212351.62
1000	2000-10-03	64300.00	276651.62
1000	2000-10-04	54553.10	331204.72
2000	2000-09-28	41888.88	41888.88
2000	2000-09-29	48000.00	89888.88
2000	2000-09-30	49850.03	139738.91
2000	2000-10-01	54850.29	194589.20

Reset now!

Not all rows displayed in answer set

The GROUP BY Statement causes the CSUM to start over (reset) on its calculating the cumulative sum of the Daily_Sales each time it runs into a NEW Product_ID.

CSUM the Number 1 to get a Sequential Number

```
SELECT Product_ID , Sale_Date, Daily_Sales,
       CSUM(Daily_Sales, Product_ID, Sale_Date)   AS "CSum",
       CSUM(1, Product_ID, Sale_Date) as "Seq_Number"
                 FROM  Sales_Table ;
```

Product_ID	Sale_Date	Daily_Sales	CSUM	Seq_Number
1000	2000-09-28	48850.40	48850.40	1
1000	2000-09-29	54500.22	103350.62	2
1000	2000-09-30	36000.07	139350.69	3
1000	2000-10-01	40200.43	179551.12	4
1000	2000-10-02	32800.50	212351.62	5
1000	2000-10-03	64300.00	276651.62	6
1000	2000-10-04	54553.10	331204.72	7
2000	2000-09-28	41888.88	373093.60	8
2000	2000-09-29	48000.00	421093.60	9
2000	2000-09-30	49850.03	470943.63	10

Not all rows are displayed in this answer set

With "Seq_Number", because you placed the number 1 in the area where it calculates, it will continuously add 1 to the answer for each row.

A Single GROUP BY Resets each OLAP with Teradata Syntax

```
SELECT Product_ID , Sale_Date, Daily_Sales,
       CSUM(Daily_Sales, Product_ID, Sale_Date)   AS "CSum",
       CSUM(1, Product_ID, Sale_Date) as "Seq_Number"
FROM  Sales_Table GROUP BY Product_ID ;
```

Not all rows are displayed in this answer set

Product_ID	Sale_Date	Daily_Sales	CSUM	Seq_Number
1000	2000-09-28	48850.40	48850.40	1
1000	2000-09-29	54500.22	103350.62	2
1000	2000-09-30	36000.07	139350.69	3
1000	2000-10-01	40200.43	179551.12	4
1000	2000-10-02	32800.50	212351.62	5
1000	2000-10-03	64300.00	276651.62	6
1000	2000-10-04	54553.10	331204.72	7
2000	2000-09-28	41888.88	41888.88	1
2000	2000-09-29	48000.00	89888.88	2
2000	2000-09-30	49850.03	139738.91	3
2000	2000-10-01	54850.29	194589.20	4

What does the GROUP BY Statement cause? Both OLAP Commands to reset!

A Better Choice – The ANSI Version of CSUM

Start on 1st row and continue till the end

```
SELECT Product_ID , Sale_Date, Daily_Sales,
       SUM(Daily_Sales) OVER (ORDER BY  Sale_Date
       ROWS UNBOUNDED PRECEDING)  AS SUMOVER
FROM  Sales_Table
WHERE Product_ID BETWEEN 1000 and 2000 ;
```

Not all rows are displayed in this answer set

Product_ID	Sale_Date	Daily_Sales	SUMOVER
2000	2000-09-28	41888.88	41888.88
1000	2000-09-28	48850.40	90739.28
2000	2000-09-29	48000.00	138739.28
1000	2000-09-29	54500.22	193239.50
1000	2000-09-30	36000.07	229239.57
2000	2000-09-30	49850.03	279089.60
1000	2000-10-01	40200.43	319290.03
2000	2000-10-01	54850.29	374140.32
1000	2000-10-02	32800.50	406940.82
2000	2000-10-02	36021.93	442962.75

This ANSI version of CSUM is SUM () Over. Right now, the syntax wants to see the sum of the Daily_Sales after it is first sorted by Sale_Date. Rows Unbounded Preceding makes this a CSUM. The ANSI Syntax seems difficult, but only at first.

The ANSI Version of CSUM – The Sort Explained

```
SELECT  Product_ID, Sale_Date, Daily_Sales,
        SUM(Daily_Sales) OVER (ORDER BY Sale_Date
        ROWS UNBOUNDED PRECEDING)       AS SUMOVER
FROM Sales_Table WHERE Product_ID BETWEEN 1000 and 2000 ;
```

Product_ID	Sale_Date	Daily_Sales	SUMOVER
2000	2000-09-28	41888.88	41888.88
1000	2000-09-28	48850.40	90739.28
2000	2000-09-29	48000.00	138739.28

Not all rows are displayed in this answer set

1000	2000-09-29	54500.22	193239.50
1000	2000-09-30	36000.07	229239.57
2000	2000-09-30	49850.03	279089.60
1000	2000-10-01	40200.43	319290.03
2000	2000-10-01	54850.29	374140.32
1000	2000-10-02	32800.50	406940.82
2000	2000-10-02	36021.93	442962.75
1000	2000-10-03	64300.00	507262.75
2000	2000-10-03	43200.18	550462.93

The first thing the above query does before calculating is SORT all the rows by Sale_Date. The Sort is located right after the ORDER BY.

The ANSI CSUM – Rows Unbounded Preceding Explained

```
SELECT  Product_ID, Sale_Date, Daily_Sales,
        SUM(Daily_Sales) OVER (ORDER BY Sale_Date
        ROWS UNBOUNDED PRECEDING)  AS SUMOVER
FROM Sales_Table WHERE Product_ID BETWEEN 1000 and 2000 ;
```

Not all rows are displayed in this answer set

Product_ID	Sale_Date	Daily_Sales	SUMOVER
2000	2000-09-28	41888.88	41888.88
1000	2000-09-28	48850.40	90739.28
2000	2000-09-29	48000.00	138739.28
1000	2000-09-29	54500.22	193239.50
1000	2000-09-30	36000.07	229239.57
2000	2000-09-30	49850.03	279089.60
1000	2000-10-01	40200.43	319290.03
2000	2000-10-01	54850.29	374140.32
1000	2000-10-02	32800.50	406940.82
2000	2000-10-02	36021.93	442962.75
1000	2000-10-03	64300.00	507262.75

The keywords Rows Unbounded Preceding determine that this is a CSUM. There are only a few different statements and Rows Unbounded Preceding is the main one. It means start calculating at the beginning row and continue calculating until the last row.

The ANSI CSUM – Making Sense of the Data

```
SELECT  Product_ID, Sale_Date, Daily_Sales,
        SUM(Daily_Sales) OVER (ORDER BY Sale_Date
        ROWS UNBOUNDED PRECEDING)  AS SUMOVER
FROM Sales_Table WHERE Product_ID BETWEEN 1000 and 2000 ;
```

Not all rows are displayed in this answer set

Product_ID	Sale_Date	Daily_Sales	SUMOVER
2000	2000-09-28	41888.88	41888.88
1000	2000-09-28	48850.40	90739.28
2000	2000-09-29	48000.00	138739.28
1000	2000-09-29	54500.22	193239.50
1000	2000-09-30	36000.07	229239.57
2000	2000-09-30	49850.03	279089.60
1000	2000-10-01	40200.43	319290.03
2000	2000-10-01	54850.29	374140.32
1000	2000-10-02	32800.50	406940.82
2000	2000-10-02	36021.93	442962.75
1000	2000-10-03	64300.00	507262.75

The second "SUMOVER" row is 90739.28. That is derived by the first row's Daily_Sales (41888.88) added to the SECOND row's Daily_Sales (48850.40).

The ANSI CSUM – Making Even More Sense of the Data

```
SELECT  Product_ID, Sale_Date, Daily_Sales,
        SUM(Daily_Sales) OVER (ORDER BY Sale_Date
        ROWS UNBOUNDED PRECEDING)  AS SUMOVER
FROM Sales_Table WHERE Product_ID BETWEEN 1000 and 2000 ;
```

Not all rows are displayed in this answer set

Product_ID	Sale_Date	Daily_Sales	SUMOVER
2000	2000-09-28	41888.88	41888.88
1000	2000-09-28	48850.40	90739.28
2000	2000-09-29	48000.00	138739.28
1000	2000-09-29	54500.22	193239.50
1000	2000-09-30	36000.07	229239.57
2000	2000-09-30	49850.03	279089.60
1000	2000-10-01	40200.43	319290.03
2000	2000-10-01	54850.29	374140.32
1000	2000-10-02	32800.50	406940.82
2000	2000-10-02	36021.93	442962.75
1000	2000-10-03	64300.00	507262.75

The third "SUMOVER" row is 138739.28. That is derived by taking the first row's Daily_Sales (41888.88) and adding it to the SECOND row's Daily_Sales (48850.40). Then, you add that total to the THIRD row's Daily_Sales (48000.00).

The ANSI CSUM – The Major and Minor Sort Key(s)

```
SELECT Product_ID, Sale_Date, Daily_Sales,
   SUM(Daily_Sales) OVER (ORDER BY Product_ID, Sale_Date
   ROWS UNBOUNDED PRECEDING)       AS SumOVER
FROM Sales_Table ;
```

Not all rows are displayed in this answer set

Product_ID	Sale_Date	Daily_Sales	SUMOVER
1000	2000-09-28	48850.40	48850.40
1000	2000-09-29	54500.22	103350.62
1000	2000-09-30	36000.07	139350.69
1000	2000-10-01	40200.43	179551.12
1000	2000-10-02	32800.50	213251.62
1000	2000-10-03	64300.00	276651.62
1000	2000-10-04	54553.10	331204.72
2000	2000-09-28	41888.88	373093.60
2000	2000-09-29	48000.00	421093.60
2000	2000-09-30	49850.03	470943.63
2000	2000-10-01	54850.29	525793.92

You can have more than one SORT KEY. In the top query, Product_ID is the MAJOR Sort and Sale_Date is the MINOR Sort.

The ANSI CSUM – Getting a Sequential Number

```
SELECT Product_ID , Sale_Date, Daily_Sales,
       SUM(Daily_Sales) OVER (ORDER BY Product_ID,  Sale_Date
                 ROWS UNBOUNCED PRECEDING) as SUMOVER,
       SUM(1) OVER (ORDER BY Product_ID, Sale_Date
                 ROWS UNBOUNDED PRECEDING)   AS Seq_Number
FROM  Sales_Table ;
```

Not all rows are displayed in this answer set

Product_ID	Sale_Date	Daily_Sales	SUM OVER	Seq_Number
1000	2000-09-28	48850.40	48850.40	1
1000	2000-09-29	54500.22	103350.62	2
1000	2000-09-30	36000.07	139350.69	3
1000	2000-10-01	40200.43	179551.12	4
1000	2000-10-02	32800.50	213251.62	5
1000	2000-10-03	64300.00	276651.62	6
1000	2000-10-04	54553.10	331204.72	7
2000	2000-09-28	41888.88	373093.60	8
2000	2000-09-29	48000.00	421093.60	9
2000	2000-09-30	49850.03	470943.63	10

With "Seq_Number", because you placed the number 1 in the area which calculates the cumulative sum, it'll continuously add 1 to the answer for each row.

Troubleshooting the ANSI OLAP on a GROUP BY

```
SELECT Product_ID, Sale_Date, Daily_Sales,
       SUM(Daily_Sales) OVER (ORDER BY Sale_Date
          ROWS UNBOUNDED PRECEDING) AS SUMOVER
FROM Sales_Table
GROUP BY Product_ID ;
```

Error! Why?

Never GROUP BY in a SUM () Over or with any ANSI Syntax OLAP command. If you want to reset, you use a PARTITION BY Statement, but never a GROUP BY.

The ANSI OLAP – Reset with a PARTITION BY Statement

```
SELECT Product_ID, Sale_Date, Daily_Sales,
       SUM(Daily_Sales) OVER (PARTITION BY Product_ID
           ORDER BY Product_ID, Sale_Date
           ROWS UNBOUNDED PRECEDING) AS SumANSI
FROM Sales_Table ;
```

Not all rows are displayed in this answer set.

Product_ID	Sale_Date	Daily_Sales	SumANSI
1000	2000-09-28	48850.40	48850.40
1000	2000-09-29	54500.22	103350.62
1000	2000-09-30	36000.07	139350.69
1000	2000-10-01	40200.43	179551.12
1000	2000-10-02	32800.50	212351.62
1000	2000-10-03	64300.00	276651.62
1000	2000-10-04	54553.10	331204.72
2000	2000-09-28	41888.88	41888.88
2000	2000-09-29	48000.00	89888.88
2000	2000-09-30	49850.03	139738.91

The PARTITION Statement is how you reset in ANSI. This will cause the SUMANSI to start over (reset) on its calculating for each NEW Product_ID.

PARTITION BY only Resets a Single OLAP not ALL of them

```
SELECT Product_ID, Sale_Date, Daily_Sales,
  SUM(Daily_Sales) OVER (PARTITION BY Product_ID
  ORDER BY Product_ID, Sale_Date
  ROWS UNBOUNDED PRECEDING) AS Subtotal,
     SUM(Daily_Sales) OVER (ORDER BY Product_ID, Sale_Date
     ROWS UNBOUNDED PRECEDING) AS GRANDTotal
FROM Sales_Table ;
```

Not all rows are displayed in this answer set

Product_ID	Sale_Date	Daily_Sales	SubTotal	GRANDTotal
1000	2000-09-28	48850.40	48850.40	48850.40
1000	2000-09-29	54500.22	103350.62	103350.62
1000	2000-09-30	36000.07	139350.69	139350.69
1000	2000-10-01	40200.43	179551.12	179551.12
1000	2000-10-02	32800.50	212351.62	212351.62
1000	2000-10-03	64300.00	276651.62	276651.62
1000	2000-10-04	54553.10	331204.72	331204.72
2000	2000-09-28	41888.88	41888.88	373093.60
2000	2000-09-29	48000.00	89888.88	421093.60
2000	2000-09-30	49850.03	139738.91	470943.63

Above are two OLAP statements. Only one has PARTITION BY, so only it resets.

The Moving SUM (MSUM) and Moving Window

```
SELECT Product_ID , Sale_Date, Daily_Sales,
    MSUM( Daily_Sales, 3, Product_ID, Sale_Date) as Msum3_Rows
                                                  FROM Sales_Table ;
```

Moving SUM | Moving Window | Major and Minor Sort keys

Not all rows are displayed in this answer set

Product_ID	Sale_Date	Daily_Sales	Msum3_Rows
1000	2000-09-28	48850.40	48850.40
1000	2000-09-29	54500.22	103350.62
1000	2000-09-30	36000.07	139350.69
1000	2000-10-01	40200.43	130700.72
1000	2000-10-02	32800.50	109001.00
1000	2000-10-03	64300.00	137300.93
1000	2000-10-04	54553.10	151653.60
2000	2000-09-28	41888.88	160741.98
2000	2000-09-29	48000.00	144441.98
2000	2000-09-30	49850.03	139738.91

This is the Moving Sum (MSUM). It will calculate the Sum of 3 rows because that is the Moving Window. It will read the current row and TWO preceding to find the MSUM of those 3 rows. It will be sorted by Product_ID and Sale_Date first.

How the Moving Sum is calculated

```
SELECT Product_ID, Sale_Date, Daily_Sales,
       MSUM(Daily_Sales, 3, Product_ID, Sale_Date) as MSum3_Rows
FROM Sales_Table
```

Not all rows are displayed in this answer set

Product_ID	Sale_Date	Daily_Sales	MSum3_Rows
1000	2000-09-28	48850.40	48850.40
1000	2000-09-29	54500.22	103350.62
1000	2000-09-30	36000.07	139350.69
1000	2000-10-01	40200.43	130700.72
1000	2000-10-02	32800.50	109001.00
1000	2000-10-03	64300.00	137300.93
1000	2000-10-04	54553.10	151653.60
2000	2000-09-28	41888.88	160741.98
2000	2000-09-29	48000.00	144441.98
2000	2000-09-30	49850.03	139738.91
2000	2000-10-01	54850.29	152700.32
2000	2000-10-02	36021.93	140722.25

With a Moving Window of 3, how is the 139350.69 amount derived in MSum3_Rows, which is in the third row? It is the SUM of 48850.40, 54500.22 and 36000.07. The fourth row has MSum3_Rows equal to 130700.72. That was the Sum of 54500.22, 360000.07 and 40200.43. The MSum is the current row sum plus the previous two.

How the Sort works for Moving SUM (MSUM)

```
SELECT Product_ID , Sale_Date, Daily_Sales,
       MSUM ( Daily_Sales, 3, Product_ID, Sale_Date) AS MSum3_Rows
FROM  Sales_Table
```

Major and Minor Sort keys

Product_ID	Sale_Date	Daily_Sales	MSum3_Rows
1000	2000-09-28	48850.40	48850.40
1000	2000-09-29	54500.22	103350.62

Not all rows are displayed in this answer set

1000	2000-09-30	36000.07	139350.69
1000	2000-10-01	40200.43	130700.72
1000	2000-10-02	32800.50	109001.00
1000	2000-10-03	64300.00	137300.93
1000	2000-10-04	54553.10	151653.60
2000	2000-09-28	41888.88	160741.98
2000	2000-09-29	48000.00	144441.98
2000	2000-09-30	49850.03	139738.91
2000	2000-10-01	54850.29	152700.32
2000	2000-10-02	36021.93	140722.25

The sorting is shown above as first Product_ID and then Sale_Date on Product_ID ties.

GROUP BY in the Moving SUM does a Reset

```
SELECT Product_ID, Sale_Date, Daily_Sales,
       MSUM(Daily_Sales, 3, Product_ID, Sale_Date) AS MSum3_Rows
FROM Sales_Table
GROUP BY Product_ID;
```

Not all rows are displayed in this answer set

Product_ID	Sale_Date	Daily_Sales	MSum3_Rows
1000	2000-09-28	48850.40	48850.40
1000	2000-09-29	54500.22	103350.62
1000	2000-09-30	36000.07	139350.69
1000	2000-10-01	40200.43	130700.72
1000	2000-10-02	32800.50	109001.00
1000	2000-10-03	64300.00	137300.93
1000	2000-10-04	54553.10	151653.60
2000	2000-09-28	41888.88	41888.88
2000	2000-09-29	48000.00	89888.88
2000	2000-09-30	49850.03	139738.91

What does the GROUP BY Product_ID do? It causes a reset on all Product_ID breaks.

Quiz – Can you make the Advanced Calculation in your mind?

```
SELECT Product_ID , Sale_Date, Daily_Sales,
       MSUM( Daily_Sales, 3, Product_ID, Sale_Date) AS MSum3_Rows
FROM   Sales_Table
GROUP BY Product_ID;
```

Not all rows are displayed in this answer set

Product_ID	Sale_Date	Daily_Sales	MSum3_Rows
1000	2000-09-28	48850.40	48850.40
1000	2000-09-29	54500.22	103350.62
1000	2000-09-30	36000.07	139350.69
1000	2000-10-01	40200.43	130700.72
1000	2000-10-02	32800.50	109001.00
1000	2000-10-03	64300.00	137300.93
1000	2000-10-04	54553.10	151653.60
2000	2000-09-28	41888.88	41888.88
2000	2000-09-29	48000.00	89888.88
2000	2000-09-30	49850.03	139738.91
2000	2000-10-01	54850.29	152700.32

How is the 89888.88 derived in the 9th row of the MSum3_Rows?

Answer to Quiz for the Advanced Calculation in your mind?

```
SELECT Product_ID , Sale_Date, Daily_Sales,
   MSUM( Daily_Sales, 3, Product_ID, Sale_Date) AS MSum3_Rows
FROM Sales_Table
GROUP BY Product_ID;
```

Not all rows are displayed in this answer set

Product_ID	Sale_Date	Daily_Sales	MSum3_Rows
1000	2000-09-28	48850.40	48850.40
1000	2000-09-29	54500.22	103350.62
1000	2000-09-30	36000.07	139350.69
1000	2000-10-01	40200.43	130700.72
1000	2000-10-02	32800.50	109001.00
1000	2000-10-03	64300.00	137300.93
1000	2000-10-04	54553.10	151653.60
2000	2000-09-28	41888.88	41888.88
2000	2000-09-29	48000.00	89888.88
2000	2000-09-30	49850.03	139738.91
2000	2000-10-01	54850.29	152700.32

The GROUP BY reset the column to start over when Product_ID went to 2000. The 89888.88 is the sum of 41888.88 and 48000.00.

Quiz - Write that Teradata Moving Average in ANSI Syntax

```
SELECT Product_ID , Sale_Date, Daily_Sales,
   MSUM( Daily_Sales, 3, Product_ID, Sale_Date) AS MSum3
FROM Sales_Table ;
```

Challenge

Can you place another equivalent Moving Sum in the SQL above using ANSI Syntax?

Here is a challenge that almost everyone fails. Can you do it perfectly?

Both the Teradata Moving SUM and ANSI Version

```
SELECT Product_ID , Sale_Date, Daily_Sales,
    MSUM( Daily_Sales, 3, Product_ID, Sale_Date) AS MSum3,
         SUM(Daily_Sales) OVER (ORDER BY Product_ID,
            Sale_Date ROWS 2 Preceding) AS SUM3_ANSI
FROM Sales_Table ;
```

Product_ID	Sale_Date	Daily_Sales	MSum3	Sum3_ANSI
1000	2000-09-28	48850.40	48850.40	48850.40
1000	2000-09-29	54500.22	103350.62	103350.62

Not all rows are displayed in this answer set

1000	2000-09-30	36000.07	139350.69	139350.69
1000	2000-10-01	40200.43	130700.72	130700.72
1000	2000-10-02	32800.50	109001.00	109001.00
1000	2000-10-03	64300.00	137300.93	137300.93
1000	2000-10-04	54553.10	151653.60	151653.60
2000	2000-09-28	41888.88	160741.98	160741.98
2000	2000-09-29	48000.00	144441.98	144441.98
2000	2000-09-30	49850.03	139738.91	139738.91

The MSUM and SUM (Over) commands above are equivalent. Notice the Moving Window of 3 in the Teradata syntax is a 2 in the ANSI version. That is because in ANSI, the moving window is considered the Current Row and 2 preceding.

The ANSI Moving Window is Current Row and Preceding

```
SELECT Product_ID , Sale_Date, Daily_Sales,
       SUM(Daily_Sales) OVER (ORDER BY Product_ID, Sale_Date
                   ROWS 2 Preceding)AS Sum3_ANSI
FROM  Sales_Table ;
```

Moving Window of 3 rows = Calculate the Current Row and 2 rows preceding

Not all rows are displayed in this answer set

Product_ID	Sale_Date	Daily_ tles	Sum3_ANSI
1000	2000-09-28	48850.40	48850.40
1000	2000-09-29	54500.22	51675.31
1000	2000-09-30	36000.07	46450.23
1000	2000-10-01	40200.43	43566.91
1000	2000-10-02	32800.50	36333.67
1000	2000-10-03	64300.00	45788.98
1000	2000-10-04	54553.10	50551.20
2000	2000-09-28	41888.88	53580.66
2000	2000-09-29	48000.00	48147.33
2000	2000-09-30	49850.03	46579.11

The SUM () Over allows you to get the moving SUM of a certain column.

How ANSI Moving Average Handles the Sort

```
SELECT Product_ID , Sale_Date, Daily_Sales,
       SUM(Daily_Sales) OVER (ORDER BY Product_ID, Sale_Date
       ROWS 2 Preceding) AS Sum3_ANSI
FROM  Sales_Table ;
```

Major and Minor Sort keys

Not all rows are displayed in

Product_ID	Sale_Date	Daily_Sales	Sum3_ANSI
1000	2000-09-28	48850.40	48850.40
1000	2000-09-29	54500.22	103350.62
1000	2000-09-30	36000.07	139350.69
1000	2000-10-01	40200.43	130700.72
1000	2000-10-02	32800.50	109001.00
1000	2000-10-03	64300.00	137300.93
1000	2000-10-04	54553.10	151653.60

this answer set

2000	2000-09-28	41888.88	160741.98
2000	2000-09-29	48000.00	144441.98
2000	2000-09-30	49850.03	139738.91
2000	2000-10-01	54850.29	152700.32
2000	2000-10-02	36021.93	140722.25

The SUM OVER places the sort after the ORDER BY.

Quiz - How is that Total Calculated?

```
SELECT Product_ID , Sale_Date, Daily_Sales,
      SUM(Daily_Sales) OVER (ORDER BY Product_ID, Sale_Date
                        ROWS 2 Preceding) AS Sum3_ANSI
FROM  Sales_Table ;
```

Not all rows are displayed in this answer set

Product_ID	Sale_Date	Daily_Sales	Sum3_ANSI
1000	2000-09-28	48850.40	48850.40
1000	2000-09-29	54500.22	103350.62
1000	2000-09-30	36000.07	139350.69
1000	2000-10-01	40200.43	130700.72
1000	2000-10-02	32800.50	109001.00
1000	2000-10-03	64300.00	137300.93
1000	2000-10-04	54553.10	151653.60
2000	2000-09-28	41888.88	160741.98
2000	2000-09-29	48000.00	144441.98
2000	2000-09-30	49850.03	139738.91
2000	2000-10-01	54850.29	152700.32
2000	2000-10-02	36021.93	140722.25

With a Moving Window of 3, how is the 139350.69 amount derived in the Sum3_ANSI column in the third row?

Answer to Quiz - How is that Total Calculated?

```
SELECT Product_ID , Sale_Date, Daily_Sales,
      SUM(Daily_Sales) OVER (ORDER BY Product_ID, Sale_Date
                        ROWS 2 Preceding) AS Sum3_ANSI
FROM  Sales_Table ;
```

Not all rows are displayed in this answer set

Product_ID	Sale_Date	Daily_Sale	Sum3_ANSI
1000	2000-09-28	48850.40	48850.40
1000	2000-09-29	54500.22	103350.62
1000	2000-09-30	36000.07	139350.69
1000	2000-10-01	40200.43	130700.72
1000	2000-10-02	32800.50	109001.00
1000	2000-10-03	64300.00	137300.93
1000	2000-10-04	54553.10	151653.60
2000	2000-09-28	41888.88	160741.98
2000	2000-09-29	48000.00	144441.98
2000	2000-09-30	49850.03	139738.91
2000	2000-10-01	54850.29	152700.32
2000	2000-10-02	36021.93	140722.25

With a Moving Window of 3, how is the 139350.69 amount derived in the Sum3_ANSI column in the third row? It is the sum of 48850.40, 54500.22 and 36000.07. In other words, it is the current row of Daily_Sales plus the previous two rows of Daily_Sales.

Moving SUM every 3-rows Vs. a Continuous Average

```
SELECT Product_ID , Sale_Date, Daily_Sales,
 SUM(Daily_Sales) OVER (ORDER BY Product_ID, Sale_Date
```

```
                   ROWS  2 Preceding) AS SUM3,
  SUM(Daily_Sales) OVER (ORDER BY Product_ID, Sale_Date
                   ROWS UNBOUNDED Preceding) AS Continuous
FROM  Sales_Table;
```

Not all rows are displayed in this answer set

Product_ID	Sale_Date	Daily_Sales	SUM3	Continuous
1000	2000-09-28	48850.40	48850.40	48850.40
1000	2000-09-29	54500.22	103350.62	103350.62
1000	2000-09-30	36000.07	139350.69	139350.69
1000	2000-10-01	40200.43	130700.72	179551.12
1000	2000-10-02	32800.50	109001.00	212351.62
1000	2000-10-03	64300.00	137300.93	276651.62
1000	2000-10-04	54553.10	151653.60	331204.72
2000	2000-09-28	41888.88	160741.98	373093.60
2000	2000-09-29	48000.00	144441.98	421093.60

The ROWS 2 Preceding gives the MSUM for every 3 rows. The ROWS UNBOUNDED Preceding gives the continuous MSUM.

Partition BY Resets an ANSI OLAP

```
SELECT  Product_ID , Sale_Date, Daily_Sales,
        SUM(Daily_Sales) OVER (ORDER BY Product_ID, Sale_Date
                               ROWS 2 Preceding) AS SUM3,
         SUM(Daily_Sales) OVER (PARTITION BY Product_ID
         ORDER BY Product_ID, Sale_Date
         ROWS UNBOUNDED Preceding) AS Continuous
FROM  Sales_Table;
```

ANSI RESET much Like a GROUP BY

Not all rows are displayed in this answer set

Product_ID	Sale_Date	Daily_Sales	SUM3	Continuous
1000	2000-09-28	48850.40	48850.40	48850.40
1000	2000-09-29	54500.22	103350.62	103350.62
1000	2000-09-30	36000.07	139350.69	139350.69
1000	2000-10-01	40200.43	130700.72	179551.12
1000	2000-10-02	32800.50	109001.00	212351.62
1000	2000-10-03	64300.00	137300.93	276651.62
1000	2000-10-04	54553.10	151653.60	331204.72
2000	2000-09-28	41888.88	160741.98	41888.88
2000	2000-09-29	48000.00	144441.98	89888.88

Use a PARTITION BY Statement to Reset the ANSI OLAP. Notice it only resets the OLAP command containing the Partition By statement, but not the other OLAPs.

The Moving Average (MAVG) and Moving Window

```
SELECT  Product_ID , Sale_Date, Daily_Sales,
     MAVG( Daily_Sales, 3, Product_ID, Sale_Date) AS AVG3_Rows
                                        FROM  Sales_Table
```

Moving Average

Moving Window

Major and Minor Sort keys

Not all rows are displayed in this answer set

Product_ID	Sale_Date	Daily_Sales	AVG3_Rows
1000	2000-09-28	48850.40	48850.40
1000	2000-09-29	54500.22	51675.31
1000	2000-09-30	36000.07	46450.23
1000	2000-10-01	40200.43	43566.91
1000	2000-10-02	32800.50	36333.67
1000	2000-10-03	64300.00	45788.98
1000	2000-10-04	54553.10	50551.20
2000	2000-09-28	41888.88	53580.66
2000	2000-09-29	48000.00	48147.33
2000	2000-09-30	49850.03	46579.11

This is the Moving Average (MAVG). It will calculate the average of 3 rows because that is the Moving Window. It will read the current row and TWO preceding to find the MAVG of those 3 rows. It will be sorted by Product_ID and Sale_Date first.

How the Moving Average is calculated

```
SELECT Product_ID , Sale_Date, Daily_Sales,
    MAVG( Daily_Sales, 3, Product_ID, Sale_Date) AS AVG3_Rows
FROM Sales_Table ;
```

Not all rows are displayed in this answer set

Product_ID	Sale_Date	Daily_Sales	AVG3_Rows
1000	2000-09-28	**48850.40**	48850.40
1000	2000-09-29	**54500.22**	51675.31
1000	2000-09-30	**36000.07**	**46450.23**
1000	2000-10-01	40200.43	43566.91
1000	2000-10-02	32800.50	36333.67
1000	2000-10-03	64300.00	45788.98
1000	2000-10-04	54553.10	50551.20
2000	2000-09-28	41888.88	53580.66
2000	2000-09-29	48000.00	48147.33
2000	2000-09-30	49850.03	46579.11
2000	2000-10-01	54850.29	50900.11
2000	2000-10-02	36021.93	46907.42

With a Moving Window of 3, how is the 46450.23 amount derived in the AVG3_Rows column in the third row? It is the AVG of 48850.40, 54500.22 and 36000.07! The fourth row has AVG3_Rows equal to 43566.91. That was the average of 54500.22, 360000.07 and 40200.43. The calculation is on the current row and the two before.

How the Sort works for Moving Average (MAVG)

```
SELECT  Product_ID , Sale_Date, Daily_Sales,
      MAVG( Daily_Sales, 3, Product_ID, Sale_Date) AS AVG3_Rows
FROM  Sales_Table
```

Major and Minor
Sort keys

Not all rows are displayed in

Product_ID	Sale_Date	Daily_Sales	AVG3_Rows
1000	2000-09-28	48850.40	48850.40
1000	2000-09-29	54500.22	51675.31
1000	2000-09-30	36000.07	46450.23
1000	2000-10-01	40200.43	43566.91
1000	2000-10-02	32800.50	36333.67
1000	2000-10-03	64300.00	45788.98
1000	2000-10-04	54553.10	50551.20

this answer set

2000	2000-09-28	41888.88	53580.66
2000	2000-09-29	48000.00	48147.33
2000	2000-09-30	49850.03	46579.11
2000	2000-10-01	54850.29	50900.11
2000	2000-10-02	36021.93	46907.42

The sorting is show above.

GROUP BY in the Moving Average does a Reset

```
SELECT Product_ID , Sale_Date, Daily_Sales,
       MAVG( Daily_Sales, 3, Product_ID, Sale_Date) AS AVG3_Rows
FROM  Sales_Table
GROUP BY Product_ID;
```

Not all rows are displayed in this answer set

Product_ID	Sale_Date	Daily_Sales	AVG3_Rows
1000	2000-09-28	48850.40	48850.40
1000	2000-09-29	54500.22	51675.31
1000	2000-09-30	36000.07	46450.23
1000	2000-10-01	40200.43	43566.91
1000	2000-10-02	32800.50	36333.67
1000	2000-10-03	64300.00	45788.98
1000	2000-10-04	54553.10	50551.20
2000	2000-09-28	41888.88	41888.88
2000	2000-09-29	48000.00	44944.44
2000	2000-09-30	49850.03	46579.64

What does the GROUP BY Product_ID do? It causes a reset on all Product_ID breaks.

Quiz - Can you make the Advanced Calculation in your mind?

```
SELECT Product_ID , Sale_Date, Daily_Sales,
       MAVG( Daily_Sales, 3, Product_ID, Sale_Date) AS AVG3_Rows
FROM  Sales_Table
GROUP BY Product_ID;
```

Not all rows are displayed in this answer set

Product_ID	Sale_Date	Daily_Sales	AVG3_Rows
1000	2000-09-28	48850.40	48850.40
1000	2000-09-29	54500.22	51675.31
1000	2000-09-30	36000.07	46450.23
1000	2000-10-01	40200.43	43566.91
1000	2000-10-02	32800.50	36333.67
1000	2000-10-03	64300.00	45788.98
1000	2000-10-04	54553.10	50551.20
2000	2000-09-28	41888.88	41888.88
2000	2000-09-29	48000.00	44944.44
2000	2000-09-30	49850.03	46579.64

How is the 44944.44 derived in the 9^{th} row of the AVG_for_3_Rows?

Answer to Quiz for the Advanced Calculation in your mind?

```
SELECT Product_ID , Sale_Date, Daily_Sales,
       MAVG( Daily_Sales, 3, Product_ID, Sale_Date) AS AVG3_Rows
FROM  Sales_Table
GROUP BY Product_ID;
```

Not all rows are displayed in this answer set

Product_ID	Sale_Date	Daily_Sales	AVG3_Rows
1000	2000-09-28	48850.40	48850.40
1000	2000-09-29	54500.22	51675.31
1000	2000-09-30	36000.07	46450.23
1000	2000-10-01	40200.43	43566.91
1000	2000-10-02	32800.50	36333.67
1000	2000-10-03	64300.00	45766.98
1000	2000-10-04	54553.10	50551.20
2000	2000-09-28	41888.88	41888.88
2000	2000-09-29	48000.00	44944.44
2000	2000-09-30	49850.03	46579.64

AVG of 41888.88 and 48000.00

Notice, there are only two calculations although this has a moving window of 3. That is because the GROUP BY caused the MAVG to reset when Product_ID 2000 came.

Quiz - Write that Teradata Moving Average in ANSI Syntax

```
SELECT Product_ID , Sale_Date, Daily_Sales,
    MAVG( Daily_Sales, 3, Product_ID, Sale_Date) AS AVG3_Rows
FROM Sales_Table ;
```

Challenge

Can you place another equivalent Moving Average in the SQL above using ANSI Syntax?

Here is a challenge that almost everyone fails. Can you do it perfectly?

Both the Teradata Moving Average and ANSI Version

```
SELECT Product_ID , Sale_Date, Daily_Sales,
    MAVG( Daily_Sales, 3, Product_ID, Sale_Date) AS AVG_3,
        AVG(Daily_Sales) OVER (ORDER BY Product_ID,
           Sale_Date ROWS 2 Preceding) AS AVG_3_ANSI
FROM Sales_Table ;
```

Product_ID	Sale_Date	Daily_Sales	AVG_3	AVG_3_ANSI
1000	2000-09-28	48850.40	48850.40	48850.40
1000	2000-09-29	54500.22	51675.31	51675.31

Not all rows are displayed in this answer set

1000	2000-09-30	36000.07	46450.23	46450.23
1000	2000-10-01	40200.43	43566.91	43566.91
1000	2000-10-02	32800.50	36333.67	36333.67
1000	2000-10-03	64300.00	45788.98	45788.98
1000	2000-10-04	54553.10	50551.20	50551.20
2000	2000-09-28	41888.88	53580.66	53580.66
2000	2000-09-29	48000.00	48147.33	48147.33
2000	2000-09-30	49850.03	46579.11	46579.11

The MAVG and AVG(Over) commands above are equivalent. Notice the Moving Window of 3 in the Teradata syntax, and that it is a 2 in the ANSI version. That is because in ANSI it is considered the Current Row and 2 preceding.

The ANSI Moving Window is Current Row and Preceding

```
SELECT Product_ID , Sale_Date, Daily_Sales,
       AVG(Daily_Sales) OVER (ORDER BY Product_ID, Sale_Date
                                 ROWS 2 Preceding)AS AVG_3_ANSI
FROM  Sales_Table ;
```

Moving Window of 3 rows = Calculate the Current Row and 2 rows preceding

Not all rows are displayed in this answer set

Product_ID	Sale_Date	Daily_Sales	AVG_3_ANSI
1000	2000-09-28	48850.40	48850.40
1000	2000-09-29	54500.22	51675.31
1000	2000-09-30	36000.07	46450.23
1000	2000-10-01	40200.43	43566.91
1000	2000-10-02	32800.50	36333.67
1000	2000-10-03	64300.00	45788.98
1000	2000-10-04	54553.10	50551.20
2000	2000-09-28	41888.88	53580.66
2000	2000-09-29	48000.00	48147.33
2000	2000-09-30	49850.03	46579.11

The AVG () Over allows you to do is to get the moving average of a certain column. The Rows 2 Preceding is a moving window of 3 in ANSI.

How ANSI Moving Average Handles the Sort

```
SELECT Product_ID , Sale_Date, Daily_Sales,
       AVG(Daily_Sales) OVER (ORDER BY Product_ID, Sale_Date
       ROWS 2 Preceding)AS AVG_3_ANSI
FROM  Sales_Table ;
```

Major and Minor Sort keys

Not all rows are displayed in

Product_ID	Sale_Date	Daily_Sales	AVG_3_ANSI
1000	2000-09-28	48850.40	48850.40
1000	2000-09-29	54500.22	51675.31
1000	2000-09-30	36000.07	46450.23
1000	2000-10-01	40200.43	43566.91
1000	2000-10-02	32800.50	36333.67
1000	2000-10-03	64300.00	45788.98
1000	2000-10-04	54553.10	50551.20

this answer set

2000	2000-09-28	41888.88	53580.66
2000	2000-09-29	48000.00	48147.33
2000	2000-09-30	49850.03	46579.11
2000	2000-10-01	54850.29	50900.11
2000	2000-10-02	36021.93	46907.42

Much like the SUM OVER Command, the Average OVER places the sort keys via the ORDER BY keywords.

Quiz - How is that Total Calculated?

```
SELECT Product_ID , Sale_Date, Daily_Sales,
      AVG(Daily_Sales) OVER (ORDER BY Product_ID, Sale_Date
                        ROWS 2 Preceding) AS AVG_3_ANSI
FROM Sales_Table ;
```

Not all rows are displayed in this answer set

Poduct_ID	Sale_Date	Daily_Sales	AVG_3_ANSI
1000	2000-09-28	48850.40	48850.40
1000	2000-09-29	54500.22	51675.31
1000	2000-09-30	36000.07	46450.23
1000	2000-10-01	40200.43	43566.91
1000	2000-10-02	32800.50	36333.67
1000	2000-10-03	64300.00	45788.98
1000	2000-10-04	54553.10	50551.20
2000	2000-09-28	41888.88	53580.66
2000	2000-09-29	48000.00	48147.33
2000	2000-09-30	49850.03	46579.11
2000	2000-10-01	54850.29	50900.11
2000	2000-10-02	36021.93	46907.42

With a Moving Window of 3, how is the 46450.23 amount derived in the AVG_3_ANSI column in the third row?

Answer to Quiz - How is that Total Calculated?

```
SELECT Product_ID , Sale_Date, Daily_Sales,
      AVG(Daily_Sales) OVER (ORDER BY Product_ID, Sale_Date
                        ROWS 2 Preceding) AS AVG_3_ANSI
FROM Sales_Table ;
```

Not all rows are displayed in this answer set

Product_ID	Sale_Date	Daily_Sales	AVG_3_ANSI
1000	2000-09-28	48850.40	48850.40
1000	2000-09-29	54500.22	51675.31
1000	2000-09-30	36000.07	46450.23
1000	2000-10-01	40200.43	43566.91
1000	2000-10-02	32800.50	36333.67
1000	2000-10-03	64300.00	45788.98
1000	2000-10-04	54553.10	50551.20
2000	2000-09-28	41888.88	53580.66
2000	2000-09-29	48000.00	48147.33
2000	2000-09-30	49850.03	46579.11
2000	2000-10-01	54850.29	50900.11
2000	2000-10-02	36021.93	46907.42

AVG of 48850.40, 54500.22, and 36000.07

Quiz - How is that 4th Row Calculated?

```
SELECT Product_ID , Sale_Date, Daily_Sales,
      AVG(Daily_Sales) OVER (ORDER BY Product_ID, Sale_Date
                        ROWS 2 Preceding) AS AVG_3_ANSI
FROM Sales_Table ;
```

Not all rows are displayed in this answer set

Product_ID	Sale_Date	Daily_Sales	AVG_3_ANSI
1000	2000-09-28	48850.40	48850.40
1000	2000-09-29	54500.22	51675.31
1000	2000-09-30	36000.07	46450.23
1000	2000-10-01	40200.43	43566.91
1000	2000-10-02	32800.50	36333.67
1000	2000-10-03	64300.00	45788.98
1000	2000-10-04	54553.10	50551.20
2000	2000-09-28	41888.88	53580.66
2000	2000-09-29	48000.00	48147.33
2000	2000-09-30	49850.03	46579.11
2000	2000-10-01	54850.29	50900.11
2000	2000-10-02	36021.93	46907.42

AVG of 54500.22, 36000.07, and 40200.43

With a Moving Window of 3, how is the 43566.91 amount derived in the AVG_3_ANSI column in the fourth row? Because of the ANSI syntax, it is derived as the current row plus Rows 2 Preceding.

Moving Average every 3-rows Vs. a Continuous Average

```
SELECT Product_ID , Sale_Date, Daily_Sales,
 AVG(Daily_Sales) OVER (ORDER BY Product_ID, Sale_Date
                   ROWS 2 Preceding) AS AVG3,
 AVG(Daily_Sales) OVER (ORDER BY Product_ID, Sale_Date
                   ROWS UNBOUNDED Preceding) AS Continuous
FROM Sales_Table;
```

Not all rows are displayed in this answer set

Product_ID	Sale_Date	Daily_Sales	AVG3	Continuous
1000	2000-09-24	44450.40	44450.40	44450.40
1000	2000-09-29	54500.22	51675.31	51675.31
1000	2000-09-30	36000.07	46450.23	46450.23
1000	2000-10-01	40200.43	43566.91	44887.78
1000	2000-10-02	32800.50	36333.67	42470.32
1000	2000-10-03	64300.00	45744.94	46104.60
1000	2000-10-04	54553.10	50551.20	47314.96
2000	2000-09-28	41888.88	53580.66	46636.70
2000	2000-09-29	48000.00	48147.33	46788.18

The ROWS 2 Preceding gives the MAVG for every 3 rows. The ROWS UNBOUNDED Preceding gives the continuous MAVG.

Partition BY Resets an ANSI OLAP

```
SELECT  Product_ID , Sale_Date, Daily_Sales,
        AVG(Daily_Sales) OVER (ORDER BY Product_ID, Sale_Date
                        ROWS 2 Preceding) AS AVG3,
        AVG(Daily_Sales) OVER (PARTITION BY Product_ID
          ORDER BY Product_ID, Sale_Date
          ROWS UNBOUNDED Preceding) AS Continuous
FROM  Sales_Table;
```

ANSI RESET much Like a GROUP BY

Not all rows are displayed in this answer set

Product_ID	Sale_Date	Daily_Sales	AVG3	Continuous
1000	2000-09-28	48800.40	48850.40	48800.40
1000	2000-09-29	04000.22	51675.31	51675.31
1000	2000-09-30	36000.07	46400.23	46400.23
1000	2000-10-01	40200.43	43566.91	44887.78
1000	2000-10-02	32800.00	36333.67	42470.32
1000	2000-10-03	64300.00	40788.98	46108.60
1000	2000-10-04	04003.10	00001.20	47314.96
2000	2000-09-28	41888.88	53580.66	41888.88
2000	2000-09-29	48000.00	48147.33	44944.44

Use a PARTITION BY Statement to Reset the ANSI OLAP. The Partition By statement only resets the column using the statement. Notice that only Continuous resets.

The Moving Difference (MDIFF)

```
SELECT  Product_ID , Sale_Date, Daily_Sales,
    MDIFF(Daily_Sales, 4, Product_ID, Sale_Date) as "MDiff"
            FROM Sales_Table ;
```

Moving Difference

Not all rows are displayed in this answer set

Product_ID	Sale_Date	Daily_Sales	MDiff
1000	2000-09-28	48850.40	?
1000	2000-09-29	54500.22	?
1000	2000-09-30	36000.07	?
1000	2000-10-01	40200.43	?
1000	2000-10-02	32800.50	-16049.90
1000	2000-10-03	64300.00	9799.78
1000	2000-10-04	54553.10	18553.03
2000	2000-09-28	41888.88	1688.45
2000	2000-09-29	48000.00	15199.50
2000	2000-09-30	49850.03	-14449.97
2000	2000-10-01	54850.29	297.19
2000	2000-10-02	36021.93	-5866.95
2000	2000-10-03	43200.18	-4799.82
2000	2000-10-04	32800.50	-17049.53

This is the Moving Difference (MDIFF). What this does is calculate the difference between the current row and only the 4th row preceding.

Moving Difference (MDIFF) Visual

```
SEL�CT P=ociuct ID, Sale Date, Daily Sales,
    MD:?P(:Jai:y Sales,-�, Product Io, Sale_Date) as "M:>iff"
FROM Sales_Table ;
```

Product ID	Sale_Date	Daily Sales	MDiff
1000	2000-09-28	48850.40	?
1000	2000-09-29	54500.22	?
1000	2000-09-30	36000.07	?
1000	2000-10-01	40200.43	?
1000	2000-10-02	32800.50	-16049.90
1000	2000-10-03	64300.00	9799.78
1000	2000-10-04	54553.10	18553.03
2000	2000-09-28	41888.88	1688.45
2000	2000-09-29	48000.00	15199.50
2000	2000-09-30	49850.03	-H4�9.9
2000	2000-10-01	54850.29	297.1.
2000	2000-10-02	36021.93	-5866.95
2000	2000-10-03	43200.18	-4799.83
2000	2000-10-04	32800.50	-17049.53

Not all rows are displayed in this answer set

How much more did we make for Product ID 1000 on 2000-10-03 versus Product ID 1000 which was 4 rows earlier on 2000-09-29?

Moving Difference using ANSI Syntax

```
SELEC� P=pduct ID, Sale Date, Daily Sales,
    Daily Sales - SUM(Daily Sales)
    OVER (ORDER BY Product ID ASC, Sale Date ASC
    ROWS BETWEEN 4 PRECEDING AND 4 PRECEDING) AS "MDiff_ANSI"
FROM Sales_Table ;
```

Product ID	Sale_Date	Daily Sales	MDiff_ANSI
1000	2000-09-28	48850.40	?
1000	2000-09-29	54500.22	?
1000	2000-09-30	36000.07	?
1000	2000-10-01	40200.43	?
1000	2000-10-02	32800.50	-16049.90
1000	2000-10-03	64300.00	9799.78
1000	2000-10-04	54553.10	18553.03
2000	2000-09-28	41888.88	1688.45
2000	2000-09-29	48000.00	15199.50
2000	2000-09-30	49850.03	-14449.97
2000	2000-10-01	54850.29	297.19
2000	2000-10-02	36021.93	-5866.95
2000	2000-10-03	43200.18	-4799.82
2000	2000-10-04	32800.50	-17049.53

Not all rows are displayed in this answer set

This is how you do an MDiff using the ANSI Syntax with a moving window of 4.

Moving Difference using ANSI Syntax with Partition By

```
SE�ECT Product ID, Sale Date (Format 'yyyy-mm-dd'), Daily sa:es,
    Daily Sales - SUM(Daily Sales)
    OVER (PARTITION BY Product_ID
    ORDER BY Product ID ASC, Sale Date ASC
    ROWS BETWEEN 4 PRECEDING AND 4 PRECEDING) AS "MDiff_JllisI"
FROM Sales_Tab:e;
```

Product ID	Sale_Date	Daily Sales	MDiff_ANSI
1000	2000-09-28	43850.�0	?
1000	2000-09-29	5q500.22	?
1000	2000-09-30	36000.07	?
1000	2000-10-01	40200.43	?
[illegible]	[illegible]		

Not all rows are displayed in this answer set

1000	2000-10-02	32800.50	-16049.90
1000	2000-10-03	64300.00	9799.78
1000	2000-10-04	54553.10	18553.03
2000	2000-09-28	41888.88	?
2000	2000-09-29	48000.00	?
2000	2000-09-30	49850.03	?
2000	2000-10-01	54850.29	?
2000	2000-10-02	36021.93	-5866.95
2000	2000-10-03	43200.18	-4799.82
2000	2000-10-04	32800.50	-17049.53

Wow! This is how you do an MDiff using the ANSI Syntax with a moving window of 4 and with a PARTITION BY statement.

Trouble Shooting the Moving Difference (MDIFF)

```
SELECT Product_ID , Sale_Date, Daily_Sales,
 MDIFF(Daily_Sales, 7, Product_ID, Sale_Date) as Compare2Rows
FROM Sales_Table
GROUP BY Product_ID ;
```

Not all rows are displayed in this answer set

Product_ID	Sale_Date	Daily_Sales	Compare2Rows
1000	2000-09-28	48850.40	?
1000	2000-09-29	54500.22	?
1000	2000-09-30	36000.07	?
1000	2000-10-01	40200.43	?
1000	2000-10-02	32800.50	?
1000	2000-10-03	64300.00	?
1000	2000-10-04	54553.10	?
2000	2000-09-28	41888.88	?
2000	2000-09-29	48000.00	?
2000	2000-09-30	49850.03	?
2000	2000-10-01	54850.29	?
2000	2000-10-02	36021.93	?
2000	2000-10-03	43200.18	?
2000	2000-10-04	32800.50	?

Do you notice that column Compare2Rows did not produce any data? That is because the GROUP BY Reset before it could get 7 records to find the MDIFF.

Using the RESET WHEN Option in Teradata (V13)

```
SELECT Product_ID ,Sale_Date ,Daily_Sales
   ,ROW_NUMBER() OVER (PARTITION BY Product_ID ORDER BY Sale_Date
    RESET WHEN Daily_Sales <= SUM(Daily_Sales)
      OVER (PARTITION BY Product_ID ORDER BY Sale_Date
       ROWS BETWEEN 1 PRECEDING AND 1 PRECEDING)) -1 as Increases
FROM Sales_Table WHERE Product_ID Between 1000 and 2000 ;
```

Product_ID	Sale_Date	Daily_Sales	Increases
1000	09/24/2000	44450.40	0
1000	09/29/2000	54500.22	1
1000	09/30/2000	36000.07	0
1000	10/01/2000	40200.43	1
1000	10/02/2000	32400.50	0
1000	10/03/2000	64300.00	1
1000	10/04/2000	54553.10	0
2000	09/24/2000	41444.44	0
2000	09/29/2000	48000.00	1
2000	09/30/2000	49850.03	2
2000	10/01/2000	54850.29	3
2000	10/02/2000	36021.93	0
2000	10/03/2000	43200.18	1
2000	10/04/2000	32800.50	0

This query finds how many consecutive days the Daily_Sales increases per Product_ID.

How Many Months per Product_ID has Revenue Increased?

```
SELECT Product_ID, Month(Sale_Date) as Mo, sum(Daily_Sales) as Monthly_Sum,
 ROW_NUMBER() over (PARTITION BY Product_ID ORDER BY Mo
  RESET WHEN sum(Daily_Sales) <=
   SUM(sum(Daily_Sales)) over (PARTITION BY Product_ID ORDER BY Mo
    ROWS BETWEEN 1 PRECEDING AND 1 PRECEDING)) -1 as Balance_Increase
FROM Sales_Table
GROUP BY Product_ID, Mo;
```

Product_ID	Mo	Monthly_Sum	Balance_Increase
1000	9	139350.69	0
1000	10	191454.03	1
2000	9	139738.91	0
2000	10	166872.90	1
3000	9	139679.76	0
3000	10	84908.06	0

This query finds how many months per Product_ID the revenue has increased.

The RANK Command

```
SELECT Product_ID ,Sale_Date , Daily_Sales,
            RANK(Daily_Sales) AS "Rank"
FROM Sales_Table WHERE Product_ID IN (1000, 2000) ;
```

Product_ID	Sale_Date	Daily_Sales	Rank
1000	2000-10-03	64300.00	1
2000	2000-10-01	04800.29	2
1000	2000-10-04	04003.10	3
1000	2000-09-29	04000.22	4
2000	2000-09-30	49800.03	5
1000	2000-09-28	48800.40	6
2000	2000-09-29	48000.00	7
2000	2000-10-03	43200.18	8
2000	2000-09-28	41888.88	9
1000	2000-10-01	40200.43	10
2000	2000-10-02	36021.93	11
1000	2000-09-30	36000.07	12
2000	2000-10-04	32800.50	13
1000	2000-10-02	32800.00	13

This is the RANK. In this example, it will rank your Daily_Sales from greatest to least. The default for this type of RANK is to sort DESC.

How to get Rank to Sort in Ascending Order

```
SELECT Product_ID ,Sale_Date , Daily_Sales,
            RANK(Daily_Sales ASC) AS "Rank"
FROM Sales_Table WHERE Product_ID IN (1000, 2000) ;
```

Product_ID	Sale_Date	Daily_Sales	Rank
1000	2000-10-02	32800.50	1
2000	2000-10-04	32800.50	1
1000	2000-09-30	36000.07	3
2000	2000-10-02	36021.93	4
1000	2000-10-01	40200.43	5
2000	2000-09-28	41888.88	6
2000	2000-10-03	43200.18	7
2000	2000-09-29	48000.00	8
1000	2000-09-28	48859.40	9
2000	2000-09-30	49850.03	10
1000	2000-09-29	54500.22	11
1000	2000-10-04	54553.10	12
2000	2000-10-01	54850.29	13

1000	2000-10-03	64300.00	14

This RANK query sorts in ascending mode.

Two ways to get Rank to Sort in Ascending Order

```
SELECT Product_ID ,Sale_Date , Daily_Sales,
               RANK(Daily_Sales ASC) AS Rank1,
              RANK(-Daily_Sales) AS Rank2
FROM Sales_Table WHERE Product_ID IN (1000, 2000) ;
```

Product_ID	Sale_Date	Daily_Sales	Rank1	Rank2
1000	2000-10-02	32800.50	1	1
2000	2000-10-04	32800.50	1	1
1000	2000-09-30	36000.07	3	3
2000	2000-10-02	36021.93	4	4
1000	2000-10-01	40200.43	5	5
2000	2000-09-28	41888.88	6	6
2000	2000-10-03	43200.18	7	7
2000	2000-09-29	48000.00	8	8
1000	2000-09-28	48859.40	9	9
2000	2000-09-30	49850.03	10	10
1000	2000-09-29	54500.22	11	11
1000	2000-10-04	54553.10	12	12
2000	2000-10-01	54850.29	13	13
1000	2000-10-03	64300.00	14	14

A minus sign or keyword ASC will sort Both RANK in ascending mode.

RANK using ANSI Syntax Defaults to Ascending Order

```
SELECT Product_ID ,Sale_Date , Daily_Sales,
       RANK() OVER(ORDER BY Daily_Sales) AS Rank1
FROM Sales_Table
WHERE Product_ID IN (1000, 2000) ;
```

Not au rows are displayed in this answer set

Product_ID	Sale_Date	Daily_Sales	Rank1
1000	10/02/2000	32800.50	1
2000	10/0.;/2000	32800.50	1
1000	09/30/2000	36000.07	3
2000	10/02/2000	36021.93	4
1000	10/01/2000	40200.43	5
2000	09/28/2000	41888.88	6
2000	10/03/2000	43200.18	7
2000	09/29/2000	48000.00	8
1000	09/28/2000	48859.40	9
2000	09/30/2000	49850.03	10
1000	09/29/2000	54500.22	11
1000	10/04/2000	54553.10	12
2000	10/01/2000	54830.29	13

This is the RANK() OVER. It provides a rank for your queries. Notice how you do not place anything within the () after the word RANK. Default Sort is ASC.

Getting RANK using ANSI Syntax to Sort in DESC Order

```
SELECT Product_ID ,Sale_Date , Daily_Sales,
RANK() OVER (ORDER BY Daily_Sales DESC) AS Rank1
FROM Sales_Table
WHERE Product_ID IN (1000, 2000)
;
```

Product_ID	Sale_Date	Daily_Sales	Rank1
[illegible]	10/03/2000	64300.00	1
1000	10/01/2000	54850.29	[illegible]
2000	10/04/2000	54553.10	[illegible]
1000	09/29/2000	54500.22	[illegible]
2000	09/30/2000	49850.03	[illegible]
1000	09/28/2000	48850.40	[illegible]
2000	09/29/2000	48000.00	[illegible]
2000	10/03/2000	,3200.18	[illegible]
2000	09/28/2000	41888.88	[illegible]
1000	10/01/2000	,0200.43	[illegible]
2000	10/02/2000	36021.93	[illegible]
1000	09/30/2000	36000.07	[illegible]
2000	10/04/2000	3::800.50	[illegible]
1000	10/02/2000	32800.50	[illegible]

4 5 6 7 8 9 10 11 12 13 13

Is the query above in ASC mode or DESC mode for sorting?

RANK() OVER and PARTITION BY

```
SELECT Product_ID ,Sale_Date, Daily_Sales,
       RANK() OVER (PARTITION BY Product_ID
              ORDER BY Daily_Sales DESC) AS Rank1
FROM Sales_Table
WHERE Product_ID IN (1000, 2000)
```

Product_ID	Sale_Date	Daily_Sales	Rank1
1000	10/03/2000	6,300.00	1
1000	10/04/2000	5,553.10	2
1000	09/29/2000	54500.22	3
1000	09/28/2000	48850.40	4
1000	10/01/2000	40200.43	5
1000	09/30/2000	36000.07	6
1000	10/02/2000	32800.50	7
[illegible]	10/01/2000	54850.29	l
[illegible]	09/30/2000	49850.03	[illegible]
[illegible]	09/29/2000	48000.00	3
[illegible]	10/03/2000	43200.18	4
[illegible]	09/28/2000	41888.88	5
[illegible]	10/02/2000	36021.93	6
[illegible]	10/04/2000	32800.50	7

What does the PARTITION Statement in the RANK() OVER do? It resets the rank.

RANK () OVER and QUALIFY

```
SELECT Product_ID ,Sale_Date , Daily_Sales,
         RANK() OVER (ORDER BY Daily_Sales DESC) AS Rank1
FROM Sales_Table
WHERE Product_ID IN (1000, 2000)
QUALIFY Rank1 < 7
```

Product_ID	Sale_Date	Daily_Sales	Rank1
1000	10/03/2000	64300.00	1
2000	10/01/2000	54850.29	2
1000	10/04/2000	54553.10	3
1000	09/29/2000	54500.22	4
2000	09/30/2000	49850.03	5
1000	09/28/2000	48850.40	6

The QUALIFY statement limits rows once the Rank's been calculated.

RANK () OVER and PARTITION BY with a QUALIFY

```
SELECT Product_ID ,Sale_Date , Daily_Sales,
         RANK() OVER (PARTITION BY Product_ID
                      ORDER BY Daily_Sales DESC) AS Rank1
FROM Sales_Table
```

```
WHERE Product_ID IN (1000, 2000)
QUALIFY Rank1 < 4
```

Product_ID	Sale_Date	Daily_Sales	Rank1
1000	2000-10-03	65000.07	1
1000	2000-10-04	54553.10	2
1000	2000-09-29	54500.22	3
2000	2000-10-01	54850.29	1
2000	2000-09-30	49850.03	2
2000	2000-09-29	48000.00	3

What does the PARTITION Statement in the RANK() OVER do? It resets the rank. The QUALIFY statement limits rows once the Rank's been calculated.

QUALIFY and WHERE

```
SELECT Product_ID ,Sale_Date , Daily_Sales,
       RANK(Daily_Sales ASC) AS Rank1,
       RANK(-Daily_Sales) AS Rank2
FROM Sales_Table
WHERE Product_ID IN (1000, 2000)
QUALIFY Rank(-Daily_Sales) < 6 ;
```

Product_ID	Sale_Date	Daily_Sales	Rank1	Rank2
1000	2000-10-02	32800.50	1	1
2000	2000-10-04	32800.50	1	1
1000	2000-09-30	36000.07	3	3
2000	2000-10-02	36021.93	4	4
1000	2000-10-01	40200.43	5	5

The WHERE statement is performed first, so it limits the rows calculated. Then, the QUALIFY takes the calculated rows and limits the returning rows. QUALIFY is to OLAP what HAVING is to Aggregates. Both limit after the calculations. Notice that our Rank1 and Rank2 are exactly the same because we sorted them both the same.

Quiz - How can you simplify the QUALIFY Statement

```
SELECT Product_ID ,Sale_Date , Daily_Sales,
       RANK(Daily_Sales ASC) AS Rank1,
       RANK(-Daily_Sales) AS Rank2
FROM Sales_Table
WHERE Product_ID IN (1000, 2000)
QUALIFY Rank(-Daily_Sales) < 6 ;
```

Product_ID	Sale_Date	Daily_Sales	Rank1	Rank2
1000	2000-10-02	32800.50	1	1
2000	2000-10-04	32800.50	1	1
1000	2000-09-30	36000.07	3	3
2000	2000-10-02	36021.93	4	4
1000	2000-10-01	40200.43	5	5

How can you improve the QUALIFY Statement above for simplicity?

Answer to Quiz -Can you simplify the QUALIFY Statement

```
SELECT Product_ID ,Sale_Date , Daily_Sales,
       RANK(Daily_Sales ASC) AS Rank1,
       RANK(-Daily_Sales)    AS Rank2
FROM Sales_Table
WHERE Product_ID IN (1000, 2000)
QUALIFY Rank2 < 6 ;
```

Product_ID	Sale_Date	Daily_Sales	Rank1	Rank2
1000	2000-10-02	32800.50	1	1
2000	2000-10-04	32800.50	1	1
1000	2000-09-30	36000.07	3	3
2000	2000-10-02	36021.93	4	4
1000	2000-10-01	40200.43	5	5

QUALIFY Rank2 < 6 (Use the Alias)

The QUALIFY Statement without Ties

```
SELECT Product_ID ,Sale_Date , Daily_Sales,
            RANK(Daily_Sales) AS Rank1
FROM Sales_Table
WHERE Product_ID IN (1000, 2000)
QUALIFY Rank1 < 6 ;
```

Product_ID	Sale_Date	Daily_Sales	Rank1
1000	2000-10-03	64300.00	1
2000	2000-10-01	54850.29	2
1000	2000-10-04	54553.10	3
1000	2000-09-29	54500.22	4
2000	2000-09-30	49850.03	5

A QUALIFY < 6 will provide a result that is 5 rows. Notice there are NO ties, yet! This is merely because we have no ties, but turn to the next page and we will have.

The QUALIFY Statement with Ties

```
SELECT Product_ID ,Sale_Date , Daily_Sales,
       RANK(Daily_Sales ASC) AS Rank1
FROM Sales_Table
WHERE Product_ID IN (1000, 2000)
QUALIFY Rank1 < 6 ;
```

Product_ID	Sale_Date	Daily_Sales	Rank1
1000	2000-10-02	32800.50	1
2000	2000-10-04	32800.50	1
1000	2000-09-30	36000.07	3
2000	2000-10-02	36021.93	4
1000	2000-10-01	40200.43	5

A QUALIFY < 6 will provide a result that is five rows. Notice there are Ties! This is because in ASC mode there are two matches within our first five rows.

The QUALIFY Statement with Ties Brings back Extra Rows

```
SELECT Product_ID ,Sale_Date , Daily_Sales,
            RANK(Daily_Sales ASC) AS Rank1
FROM Sales_Table
WHERE Product_ID IN (1000, 2000)
QUALIFY Rank1 < 2 ;
```

Product_ID	Sale_Date	Daily_Sales	Rank1
1000	2000-10-02	32800.50	1
2000	2000-10-04	32800.50	1

A QUALIFY < 2 will provide more rows than 1 because of the Ties!

Mixing Sort Order for QUALIFY Statement

```
SELECT Product_ID ,Sale_Date , Daily_Sales,
        RANK(Daily_Sales) AS Rank1          DESC Mode
FROM   Sales_Table
WHERE Product_ID IN (1000, 2000)
QUALIFY RANK (Daily_Sales ASC) < 6 ;        ASC Mode
```

Product_ID	Sale_Date	Daily_Sales	Rank1
1000	2000-10-02	32800.50	13
2000	2000-10-04	32800.50	13
1000	2000-09-30	36000.07	12
2000	2000-10-02	36021.93	11
1000	2000-10-01	40200.43	10

Look at the Rankings and the Daily_Sales. This data come out odd because Rank1 is DESC by default (using this Syntax), and the QUALIFY specifies ASC mode.

Quiz - What Caused the RANK to Reset?

```
SELECT Product_ID ,Sale_Date , Daily_Sales,
            RANK(Daily_Sales) AS Rank1
FROM Sales_Table
WHERE Product_ID IN (1000, 2000)
GROUP BY Product_ID
QUALIFY Rank1 < 4 ;
```

Product_ID	Sale_Date	Daily_Sales	Rank1
1000	2000-10-03	64300.00	1
1000	2000-10-04	54553.10	2
1000	2000-09-29	54500.22	3
2000	2000-10-01	54850.29	1
2000	2000-09-30	49850.03	2
2000	2000-09-29	48000.00	3

What caused the data to reset the column Rank1?

Answer to Quiz - What Caused the RANK to Reset?

```
SELECT Product_ID ,Sale_Date , Daily_Sales,
              RANK(Daily_Sales) AS Rank1
FROM Sales_Table
WHERE Product_ID IN (1000, 2000)
GROUP BY Product_ID
QUALIFY Rank1 < 4 ;
```

Product_ID	Sale_Date	Daily_Sales	Rank1
1000	2000-10-03	64300.00	1
1000	2000-10-04	54553.10	2
1000	2000-09-29	54500.22	3
2000	2000-10-01	54850.29	1
2000	2000-09-30	49850.03	2
2000	2000-09-29	48000.00	3

GROUP BY

What caused the data to reset the column Rank1? It is the GROUP BY statement.

Quiz - Name those Sort Orders

RANK() OVER (ORDER BY Daily_Sales) AS ANSI_Rank

Is the default above ASC or DESC?

RANK(Daily_Sales) AS NON_ANSI_Rank

Is the default above ASC or DESC?

Answer the questions above.

Answer to Quiz - Name those Sort Orders

RANK() OVER (ORDER BY Daily_Sales) AS ANSI_Rank

Is the default above ASC or DESC?

Defaults to ASC

RANK(Daily_Sales) AS NON_ANSI_Rank

Is the default above ASC or DESC?

Defaults to DESC

Please note that, by default, these different syntaxes sort completely opposite.

PERCENT_RANK () OVER

```
SELECT Product_ID ,Sale_Date , Daily_Sales,
       PERCENT_RANK() OVER (PARTITION BY PRODUCT_ID
       ORDER BY Daily_Sales DESC) AS PercentRank1
FROM Sales_Table WHERE Product_ID in (1000, 2000) ;
```

Product_ID	Sale_Date	Daily_Sales	PercentRank1
1000	2000-10-03	64300.00	0.000000
1000	2000-10-04	54553.10	0.166667
1000	2000-09-29	54500.22	0.333333
1000	2000-09-28	48850.40	0.500000
1000	2000-10-01	40200.43	0.666667
1000	2000-09-30	36000.07	0.833333
1000	2000-10-02	32800.50	1.000000
2000	2000-10-01	54850.29	0.000000
2000	2000-09-30	49850.03	0.166667
2000	2000-09-29	48000.00	0.333333
2000	2000-10-03	43200.18	0.500000
2000	2000-09-28	41888.88	0.666667
2000	2000-10-02	36021.93	0.833333
2000	2000-10-04	32800.50	1.000000

7 Rows in Calculation for 1000 Product_ID

7 Rows in Calculation for 2000 Product_ID

We now have added a Partition statement which resets on Product_ID. So, this produces 7 rows for each of our Product_IDs.

PERCENT_RANK () OVER with 14 rows in Calculation

```
SELECT Product_ID ,Sale_Date , Daily_Sales,
       PERCENT_RANK() OVER ( ORDER BY Daily_Sales DESC) AS PercentRank1
FROM Sales_Table WHERE Product_ID IN (1000, 2000) ;
```

Product_ID	Sale_Date	Daily_Sales	PercentRank1
1000	2000-10-03	64300.00	0.000000
2000	2000-10-01	54850.29	0.076923
1000	2000-10-04	54553.10	0.153846
1000	2000-09-29	54500.22	0.230769
2000	2000-09-30	49850.03	0.307692
1000	2000-09-28	48850.40	0.384615
2000	2000-09-29	48000.00	0.461538
2000	2000-10-03	43200.18	0.538462
2000	2000-09-28	41888.88	0.615385
1000	2000-10-01	40200.43	0.692308
2000	2000-10-02	36021.93	0.769231
1000	2000-09-30	36000.07	0.846154
2000	2000-10-04	32800.50	0.923077
1000	2000-10-02	32800.50	0.923077

14 Rows in Calculation for both the 1000 and 2000 Product_IDs

Percent_Rank is just like RANK. However, it gives you the Rank as a percent, but only as a percent of all the other rows up to 100%.

PERCENT_RANK () OVER with 21 rows in Calculation

```
SELECT Product_ID ,Sale_Date , Daily_Sales,
     PERCENT_RANK() OVER ( ORDER BY Daily_Sales DESC) AS PercentRank1
FROM Sales_Table ;
```

Not all rows are displayed in this answer set

Product_ID	Sale_Date	Daily_Sales	PercentRank1
1000	2000-10-03	64300.00	0.000000
3000	2000-09-28	61301.77	0.050000
2000	2000-10-01	54850.29	0.100000
1000	2000-10-04	54553.10	0.150000
1000	2000-09-29	54500.22	0.200000
2000	2000-09-30	49850.03	0.250000
1000	2000-09-28	48850.40	0.300000
2000	2000-09-29	48000.00	0.350000
3000	2000-09-30	43868.86	0.400000
2000	2000-10-03	43200.18	0.450000
2000	2000-09-28	41888.88	0.500000
1000	2000-10-01	40200.43	0.550000
2000	2000-10-02	36021.93	0.600000
1000	2000-09-30	36000.07	0.650000

21 Rows in Calculation for all of the Product_IDs

Percent_Rank is just like RANK. However, it gives you the Rank as a percent, but only as a percent of all the other rows up to 100%.

Quiz - What Cause the Product_ID to Reset

```
SELECT Product_ID ,Sale_Date , Daily_Sales,
       PERCENT_RANK() OVER (PARTITION BY PRODUCT_ID
        ORDER BY Daily_Sales DESC) AS PercentRank1
FROM Sales_Table WHERE Product_ID in (1000, 2000) ;
```

Product_ID	Sale_Date	Daily_Sales	PercentRank1
1000	2000-10-03	64300.00	0.000000
1000	2000-10-04	54553.10	0.166667
1000	2000-09-29	54500.22	0.333333
1000	2000-09-28	48850.40	0.500000
1000	2000-10-01	40200.43	0.666667
1000	2000-09-30	36000.07	0.833333
1000	2000-10-02	32800.50	1.000000
2000	2000-10-01	54850.29	0.000000
2000	2000-09-30	49850.03	0.166667
2000	2000-09-29	48000.00	0.333333
2000	2000-10-03	43200.18	0.500000
2000	2000-09-28	41888.88	0.666667
2000	2000-10-02	36021.93	0.833333

2000	2000-10-04	32800.50	1.000000

What caused the Product_IDs to be sorted?

Answer to Quiz - What Causes the Product_ID to Reset

```
SELECT Product_ID ,Sale_Date , Daily_Sales,
       PERCENT_RANK() OVER (PARTITION BY PRODUCT_ID
        ORDER BY Daily_Sales DESC) AS PercentRank1
FROM Sales_Table WHERE Product_ID in (1000, 2000) ;
```

Product_ID	Sale_Date	Daily_Sales	PercentRank1
1000	2000-10-03	64300.00	0.000000
1000	2000-10-04	54553.10	0.166667
1000	2000-09-29	54500.22	0.333333
1000	2000-09-28	48850.40	0.500000
1000	2000-10-01	40200.43	0.666667
1000	2000-09-30	36000.07	0.833333
1000	2000-10-02	32800.50	1.000000
2000	2000-10-01	54850.29	0.000000
2000	2000-09-30	49850.03	0.166667
2000	2000-09-29	48000.00	0.333333
2000	2000-10-03	43200.18	0.500000
2000	2000-09-28	41888.88	0.666667
2000	2000-10-02	36021.93	0.833333
2000	2000-10-04	32800.50	1.000000

PARTITION BY caused the data to be sorted!

Answer to Quiz - What Causes the Product_ID to Reset

```
SELECT Product_ID ,Sale_Date , Daily_Sales,
      PERCENT_RANK() OVER (PARTITION BY PRODUCT_ID
       ORDER BY Daily_Sales DESC) AS PercentRank1
FROM Sales_Table WHERE Product_ID in (1000, 2000) ;
```

Product_ID	Sale_Date	Daily_Sales	PercentRank1
1000	2000-10-03	64300.00	0.000000
1000	2000-10-04	54553.10	0.166667
1000	2000-09-29	54500.22	0.333333
1000	2000-09-28	48850.40	0.500000
1000	2000-10-01	40200.43	0.666667
1000	2000-09-30	36000.07	0.833333
1000	2000-10-02	32800.50	1.000000
2000	2000-10-01	54850.29	0.000000
2000	2000-09-30	49850.03	0.166667
2000	2000-09-29	48000.00	0.333333
2000	2000-10-03	43200.18	0.500000
2000	2000-09-28	41888.88	0.666667
2000	2000-10-02	36021.93	0.833333
2000	2000-10-04	32800.50	1.000000

What caused the Product_IDs to be sorted? It was the PARTITION BY statement.

COUNT OVER for a Sequential Number

```
SELECT Product_ID ,Sale_Date , Daily_Sales,
        COUNT(*) OVER (ORDER BY Product_ID, Sale_Date
        ROWS UNBOUNDED PRECEDING) AS Seq_Number
FROM Sales_Table WHERE Product_ID IN (1000, 2000) ;
```

Product_ID	Sale_Date	Daily_Sales	Seq_Number
1000	2000-09-28	48850.40	1
1000	2000-09-29	54500.22	2
1000	2000-09-30	36000.01	3

Not all rows are displayed in this answer set

1000	2000-10-01	40200.43	4
1000	2000-10-02	32800.50	5
1000	2000-10-03	64300.00	6
1000	2000-10-04	54553.10	7
2000	2000-09-28	41888.88	8
2000	2000-09-29	48000.00	9
2000	2000-09-30	49850.03	10
2000	2000-10-01	54850.29	11

This is the COUNT OVER. It will provide a sequential number starting at 1. The Keyword(s) ROWS UNBOUNDED PRECEDING causes Seq_Number to start at the beginning and increase sequentially to the end.

Troubleshooting COUNT OVER

```
SELECT Product_ID ,Sale_Date , Daily_Sales,
       COUNT(*) OVER (ORDER BY Product_ID, Sale_Date) AS No_Seq
FROM Sales_Table WHERE Product_ID IN (1000, 2000) ;
```

Rows Unbounded Preceding is missing in this statement.

Product_ID	Sale_Date	Daily_Sales	No_Seq
1000	2000-09-28	48850.40	14
1000	2000-09-29	54500.22	14
1000	2000-09-30	36000.07	14
1000	2000-10-01	40200.43	14
1000	2000-10-02	32800.50	14
1000	2000-10-03	64300.00	14
1000	2000-10-04	54553.10	14
2000	2000-09-28	41888.88	14
2000	2000-09-29	48000.00	14
2000	2000-09-30	49850.03	14
2000	2000-10-01	54850.29	14
2000	2000-10-02	36021.93	14
2000	2000-10-03	43200.18	14
2000	2000-10-04	32800.50	14

14 rows came back

When you don't have a ROWS UNBOUNDED PRECEDING, No_Seq gets a value of 14 on every row because 14 is the FINAL COUNT NUMBER.

Quiz - What caused the COUNT OVER to Reset?

Product_ID	Sale_Date	Daily_Sales	StartOver
1000	2000-09-28	48850.40	1
1000	2000-09-29	54500.22	2
1000	2000-09-30	36000.07	3
1000	2000-10-01	40200.43	4
1000	2000-10-02	32800.50	5
1000	2000-10-03	64300.00	6
1000	2000-10-04	54553.10	7
2000	2000-09-28	41888.88	1
2000	2000-09-29	48000.00	2
2000	2000-09-30	49850.03	3
2000	2000-10-01	54850.29	4
2000	2000-10-02	36021.93	5
2000	2000-10-03	43200.18	6
2000	2000-10-04	32800.50	7

What Keyword(s) caused the column StartOver to reset?

Answer to Quiz - What caused the COUNT OVER to Reset?

```
SELECT Product_ID ,Sale_Date , Daily_Sales,
       COUNT(*) OVER (PARTITION BY Product_ID
                      ORDER BY Product_ID, Sale_Date
              ROWS UNBOUNDED PRECEDING) AS StartOver
```

```
FROM Sales_Table WHERE Product_ID IN (1000, 2000) ;
```

Product_ID	Sale_Date	Daily_Sales	StartOver
1000	2000-09-28	48850.40	1
1000	2000-09-29	54500.22	2
1000	2000-09-30	36000.0V	3
1000	2000-10-01	40200.43	4
1000	2000-10-02	32800.50	5
1000	2000-10-03	64300.00	6
1000	2000-10-04	54553.10	7
2000	2000-09-28	41888.88	1
2000	2000-09-29	48000.00	2
2000	2000-09-30	49850.03	3
2000	2000-10-01	54850.29	4
2000	2000-10-02	36021.93	5
2000	2000-10-03	43200.18	6
2000	2000-10-04	32800.50	7

What Keyword(s) caused StartOver to reset? It is the PARTITION BY statement.

The MAX OVER Command

```
SELECT Product_ID ,Sale_Date , Daily_Sales,
     MAX(Daily_Sales) OVER (ORDER BY Product_ID, Sale_Date
      ROWS UNBOUNDED PRECEDING) AS MaxOver
FROM Sales_Table WHERE Product_ID IN (1000, 2000) ;
```

Product_ID	Sale_Date	Daily_Sales	MaxOver
1000	2000-09-28	48850.40	48850.40
1000	2000-09-29	54500.22	54500.22
1000	2000-09-30	36000.07	54500.22
1000	2000-10-01	40200.43	54500.22
1000	2000-10-02	32800.50	54500.22
1000	2000-10-03	64300.00	64300.00
1000	2000-10-04	54553.10	64300.00
2000	2000-09-28	41888.88	64300.00
2000	2000-09-29	48000.00	64300.00
2000	2000-09-30	49850.03	64300.00
2000	2000-10-01	54850.29	64300.00
2000	2000-10-02	36021.93	64300.00
2000	2000-10-03	43200.18	64300.00
2000	2000-10-04	32800.50	64300.00

After the sort, the Max () Over shows the Max Value up to that point. With each new max, a new number is max.

MAX OVER with PARTITION BY Reset

```
SELECT Product_ID ,Sale_Date , Daily_Sales,
        MAX(Daily_Sales) OVER (PARTITION BY Product_ID
                               ORDER BY Product_ID, Sale_Date
                     ROWS UNBOUNDED PRECEDING) AS MaxOver
FROM Sales_Table WHERE Product_ID IN (1000, 2000) ;
```

Not all rows are displayed in this answer set

Product_ID	Sale_Date	Daily_Sales	MaxOver
1000	2000-09-28	48850.40	48850.40
1000	2000-09-29	54500.22	54500.22
1000	2000-09-30	36000.07	54500.22
1000	2000-10-01	40200.43	54500.22
1000	2000-10-02	32800.50	54500.22
1000	2000-10-03	64300.00	64300.00
1000	2000-10-04	54553.10	64300.00
2000	2000-09-28	41888.88	41888.88
2000	2000-09-29	48000.00	48000.00
2000	2000-09-30	49850.03	49850.03
2000	2000-10-01	54850.29	54850.29

The largest value is 64300.00 in the column MaxOver. Once it was evaluated, it did not continue until the end because of the PARTITION BY reset.

Troubleshooting MAX OVER

```
SELECT Product_ID ,Sale_Date , Daily_Sales,
   MAX(Daily_Sales) OVER (ORDER BY Product_ID, Sale_Date ) AS MaxOver
FROM Sales_Table WHERE Product_ID IN (1000, 2000) ;
```

Rows Unbounded Preceding is missing in this statement

Not all rows are displayed in this answer set

Product_ID	Sale_Date	Daily_Sales	MaxOver
1000	2000-09-28	48850.40	64300.00
1000	2000-09-29	54500.22	64300.00
1000	2000-09-30	36000.07	64300.00
1000	2000-10-01	40200.43	64300.00
1000	2000-10-02	32800.50	64300.00
1000	2000-10-03	64300.00	64300.00
1000	2000-10-04	54553.10	64300.00
2000	2000-09-28	41888.88	64300.00
2000	2000-09-29	48000.00	64300.00
2000	2000-09-30	49850.03	64300.00

You can also use MAX as an OLAP. 64300.00 came back in MaxOver because that was the MAX value for Daily_Sales in this Answer Set. Notice that it doesn't have a ROWS UNBOUNDED PRECEDING.

The MIN OVER Command

```
SELECT Product_ID, Sale_Date ,Daily_Sales
     ,MIN(Daily_Sales) OVER(ORDER BY Product_ID, Sale_Date
              ROWS UNBOUNDED PRECEDING) AS MinOver
FROM Sales_Table WHERE Product_ID IN (1000, 2000) ;
```

Product_ID	Sale_Date	Daily_Sales	MinOver
1000	2000-09-28	48850.40	48850.40
1000	2000-09-29	54500.22	48850.40
1000	2000-09-30	36000.07	36000.07
1000	2000-10-01	40200.43	36000.07
1000	2000-10-02	32800.50	32800.50
1000	2000-10-03	64300.00	32800.50
1000	2000-10-04	54553.10	32800.50
2000	2000-09-28	41888.88	32800.50
2000	2000-09-29	48000.00	32800.50
2000	2000-09-30	49850.03	32800.50
2000	2000-10-01	54850.29	32800.50
2000	2000-10-02	36021.93	32800.50
2000	2000-10-03	43200.18	32800.50
2000	2000-10-04	32800.50	32800.50

After the sort, the MIN () Over shows the Min Value up to that point. With each new Min, that new Min appears.

Troubleshooting MIN OVER

```
SELECT Product_ID ,Sale_Date , Daily_Sales,
     MIN(Daily_Sales) OVER (ORDER BY Product_ID, Sale_Date ) AS MinOver
FROM Sales_Table WHERE Product_ID IN (1000, 2000) ;
```

Rows Unbounded Preceding is missing in this statement.

Product_ID	Sale_Date	Daily_Sales	MinOver
1000	2000-09-28	48850.40	32800.50

Not all rows are displayed in this answer set

1000	2000-09-29	54500.22	3:000.50
1000	2000-09-30	36000.07	3:000.50
1000	2000-10-01	qo200.43	3'.::800
1000	2000-10-02	3:300.50	0.5 0
1000	2000-10-03	64300.00	3'.::800
1000	2000-10-04	54553.10	0.5 0
2000	2000-09-28	41888.88	3:800.50
2000	2000-09-29	48000.00	3'.::800
2000	2000-09-30	q9950.03	0.5 0
2000	2000-10-01	54850.29	32800.50

Min only displayed 32800.50 because there is NOT a ROWS UNBOUNDED PRECEDING statement. So, it found the lowest Daily_Sales and repeated it

Finding a Value of a Column in the Next Row with MIN

```
SELECT Product IO ,Sale Date, Daily Sales,
       M:N(Daily_Sales)-OVER (PARTITION BY Produet_ID
           ORDER B� Product ID, Sale Date
           ROWS BETWEEN 1 Follow�ng ;nd 1 Following) AS NextSale
FROM Sales_Table W�ERE ?roduct_ID IN (1000, 2000) ;
```

:?roduct ID	Sale Date	Dailv Sales	NextSale
1000	2000-09-28	48850.40	54500.22
1000	2000-09-29	5q500.22	36000.07
1000	2000-09-30	36000.07	40200.43
1000	2000-10-01	40200.q3	32800.50
1000	2000-10-02	32800.50	64300.00
1000	2000-10-03	64300.00	54553.10
1000	2000-10-04	54553.10	?
2000	2000-09-28	41888.88	48000.00
2000	2000-09-29	48000.00	49850.03
2000	2000-09-30	49850.03	54850.29
2000	2000-10-01	54850.29	36021.93
2000	2000-10-02	36021.93	43200.18
2000	2000-10-03	43200.18	32800.50
2000	2000-10-04	32800.50	?

The above example finds the value of a column in the next row for Daily_Sales. Notice it is partitioned, so there is a Null value at the end of each Product_ID.

Finding a Value of a Date in the Next Row with MIN

```
SELECT Product IO ,Sale Date ,     Daily Sa:es,
       MIN(Sal; Date) OVER (PART:TION BY P�oduct
       :D  ORDER BY Product :o,  Sale       -
           Date
           ROWS BETWEEN 1 co:lowing ;nd 1  Fol:ow�ng) AS
           NextDate
FROM Sales_Table WHERE Product_I� IN (1000, 1000)
;
```

ID	[illegible]	[illegible]	[illegible]
1000	2000-09-28	48850.40	091:91:000
1000	2000-09-29	54500.22	091301:000
1000	2000-09-30	36000.07	101011:000
1000	2000-10-01	40200.43	101021:000
1000	2000-10-02	32800.50	101031:000
1000	2000-10-03	64300.00	101041:000
1000	:000-10-04	54553.10	?
2000	2000-09-28	41888.88	091:91:000
2000	2000-09-29	48000.00	091301:000
2000	2000-09-30	49850.03	101011:000
2000	2000-10-01	54850.29	101021:000
2000	2000-10-02	36021.93	101031:000
2000	2000-10-03	43200.18	101041:000
2000	2000-10-04	32800.50	?

The above example finds the value of a column in the next row for Sale_Date. Notice it is partitioned, so there is a Null value at the end of each Product_ID.

Finding Gaps between Dates

```
SELECT  Product  Id ,Sale Date
       ,M!N(Sal;_Jate) OVER (PARTITION BY Product_Id ORDER BY Sale Date
                       ROWS BETWEEN 1 F-OLLOWHIG JI_ND U1"BOUt;DED
                        E"O::.LOWING) AS jate Of Next Row
       ,Date Of Next Row - Sale Date �s jays To Next Row
FROM Sales Table-WHERE ?roduct_!D Between 1000 a;d 2000;
```

Product_ID	Sale_Date	Date_Of_Next_Row	Da�s_To_Next_Row
1000	10/04/2000	?	?
1000	10/03/2000	10/04/2000	I
1000	10/02/2000	10/03/2000	1
1000	10/01/2000	10/02/2000	I
1000	09/30/2000	10/01/2000	I
1000	09/�9/2000	09/30/2000	1
1000	09/�8/2000	09/29/2000	1
2000	10/04/2000	7	?
2000	10/03/2000	10/04/2000	1
2000	10/02/2000	10/03/2000	1
2000	10/01/2000	10/02/2000	1
2000	09/30/2000	10/01/2000	1
2000	09/29/2000	09/30/2000	1
2000	09/28/2000	09/29/2000	1

The above query finds gaps in dates.

The CSUM for Each Product_ID for the First 3 Days

```
SELECT ROW NUMBER() OVER (PARTIIION BY Product_ID ORDER BY Sale_Date) As Row_Nbr
, Product Id, Sale Date
, MIN(Sal;_Date) OVER (PARTIIION BY Product ID OR::>ER BY Sale Date ROWS BETWEEN 1
                                   FOLLOW:NG A.�D 1 F-0::.LOW:NG) As
                                   Next_Start_Dt
,
, Daily Sales
  SUM(D�ily_Sales) OVER (PARTITION BY Product ID ORDER BY Sale Date
                        ROWS UNBOUNDED PRECE::>I�G) As To Date-Revenue
FROM Sales Table
QUALIFY Row Nbr <= 3
```

Row_Nbr	Product_ID	Sale_Date	Next_Start_	Dail�_Sales	To_Date_Revenue
1	1000	09/28/2000	09/29/2000	48850.40	�8850.40
2	1000	09/29/2000	09/30/2000	5,500.22	103350.62
3	1000	09/30/2000	10/01/2000	36000.07	139350.69
1	2000	09/28/2000	09/29/2000	41888.88	,1888.88
2	2000	09/29/2000	09/30/2000	48000.00	89888.88
3	2000	09/30/2000	10/01/2000	49850.03	139738.91
1	3000	09/28/2000	09/29/2000	61301.77	61301.77
2	3000	09/29/2000	09/30/2000	3,509.13	95810.90
3	3000	09/30/2000	10/01/2000	43868.86	139679.76

The above example shows the cumulative SUM for the Daily_Sales for the first three days for each of our Product_IDs.

Quiz - Fill in the Blank

```
SELECT Product ID ,Sale Date , Daily Sales,
      MIN(Daily_Sales)-OVER (P�.RTITION BY Product ID
                            ORDER BY ?roduct_!J,-Sale_Jace
                            ROWS UNBOUNDED PRECEDING) AS MinOVer
FROM Sales_Table WHERE Producc_ID IN (1000, 2000) ;
```

Product_ID	Sale Pats	Daily sale;:s	MinOv5r
1000	2000-09-28	48850.40	48850.40
1000	2000-09-29	54500.22	48850.40
1000	2000-09-30	36000.07	36000.07

1000	2000-10-01	40200.43	36000.07
1000	2000-10-02	32800.50	32800.50
1000	2000-10-03	64300.00	32800.50
1000	2000-10-04	54553.10	32800.50
2000	2000-09-28	41888.88	41888.88
2000	2000-09-29	48000.00	41888.88
2000	2000-09-30	49850.03	41888.88
2000	2000-10-01	54850.29	41888.88
2000	2000-10-02	36021.93	36021.93
2000	2000-10-03	43200.18	
2000	2000-10-04	32800.50	

The last two answers (MinOver) are blank, so you can fill in the blank.

Answer to Quiz - Fill in the Blank

```
SELECT Product_ID ,Sale_Date , Daily_Sales,
       MIN(Daily_Sales) OVER (PARTITION BY Product_ID
                        ORDER BY Product_ID, Sale_Date
                        ROWS UNBOUNDED PRECEDING) AS MinOver
FROM Sales_Table WHERE Product_ID IN (1000, 2000) ;
```

Product_ID	Sale_Date	Daily_Sales	MinOver
1000	2000-09-28	48850.40	48850.40
1000	2000-09-29	54500.22	48850.40
1000	2000-09-30	36000.07	36000.07
1000	2000-10-01	40200.43	36000.07
1000	2000-10-02	32800.50	32800.50
1000	2000-10-03	64300.00	32800.50
1000	2000-10-04	54553.10	32800.50
2000	2000-09-28	41888.88	41888.88
2000	2000-09-29	48000.00	41888.88
2000	2000-09-30	49850.03	41888.88
2000	2000-10-01	54850.29	41888.88
2000	2000-10-02	36021.93	36021.93
2000	2000-10-03	43200.18	36021.93
2000	2000-10-04	32800.50	32800.50

The Row_Number Command

```
SELECT Product_ID ,Sale_Date , Daily_Sales,
      ROW_NUMBER() OVER (ORDER BY Product_ID, Sale_Date) AS Seq_Number
FROM Sales_Table WHERE Product_ID IN (1000, 2000) ;
```

Product_ID	Sale_Date	Daily_Sales	Seq_Number
1000	2000-09-28	48850.40	1
1000	2000-09-29	54500.22	2
1000	2000-09-30	36000.01	3
1000	2000-10-01	40200.43	4
1000	2000-10-02	32800.50	5
1000	2000-10-03	64300.00	6
1000	2000-10-04	54553.10	7
2000	2000-09-28	41888.88	8
2000	2000-09-29	48000.00	9
2000	2000-09-30	49850.03	10
2000	2000-10-01	54850.29	11

Not all rows are displayed in this answer set

The ROW_NUMBER () Keyword(s) caused Seq_Number to increase sequentially. Notice that this does NOT have a Rows Unbounded Preceding and it still works!

Quiz - How did the Row_Number Reset?

```
SELECT Product_ID ,Sale_Date , Daily_Sales,
       ROW_NUMBER() OVER (PARTITION BY Product_ID
         ORDER BY Product_ID, Sale_Date ) AS StartOver

FROM Sales_Table WHERE Product_ID IN (1000, 2000) ;
```

Product_ID	Sale_Date	Daily_Sales	StartOver
1000	2000-09-28	48850.40	1
1000	2000-09-29	54500.22	2
1000	2000-09-30	36000.07	3
1000	2000-10-01	40200.43	4
1000	2000-10-02	32800.50	5
1000	2000-10-03	64300.00	6
1000	2000-10-04	54553.10	7
2000	2000-09-28	41888.88	1
2000	2000-09-29	48000.00	2
2000	2000-09-30	49850.03	3
2000	2000-10-01	54850.29	4
2000	2000-10-02	36021.93	5
2000	2000-10-03	43200.18	6
2000	2000-10-04	32800.50	7

What Keyword(s) caused StartOver to reset?

Quiz - How did the Row_Number Reset?

```
SELECT Product_ID ,Sale_Date , Daily_Sales,
       ROW_NUMBER() OVER (PARTITION BY Product_ID
        ORDER BY Product_ID, Sale_Date ) AS StartOver
FROM Sales_Table WHERE Product_ID IN (1000, 2000) ;
```

Product_ID	Sale_Date	Daily_Sales	StartOver
1000	2000-09-28	48850.40	1
1000	2000-09-29	54500.22	2
1000	2000-09-30	36000.07	3
1000	2000-10-01	40200.43	4
1000	2000-10-02	32800.50	5
1000	2000-10-03	64300.00	6
1000	2000-10-04	54553.10	7
2000	2000-09-28	41888.88	1
2000	2000-09-29	48000.00	2
2000	2000-09-30	49850.03	3
2000	2000-10-01	54850.29	4
2000	2000-10-02	36021.93	5
2000	2000-10-03	43200.18	6
2000	2000-10-04	32800.50	7

What Keyword(s) caused StartOver to reset? It is the PARTITION BY statement.

Row_Number with Qualify to get the Typical Rows per Value

```
SELECT Counter AS "Typical Rows per Value"
FROM
      (SELECT Product_ID, COUNT(*)
        FROM Sales_Table GROUP BY 1) AS TeraTom (Col1, Counter),

        (SELECT COUNT(DISTINCT(Product_ID))
           FROM Sales_Table) AS Derived2 (num_rows)

QUALIFY ROW_NUMBER () OVER
        (ORDER BY TeraTom.Col1) = Derived2.num_rows /2 ;
```

Typical Rows Per Value
7

The query above retrieved the typical rows per value for the column Product_ID.

A Second Typical Rows per Value Query on Sale_Date

```
SELECT Counter AS "Typical Rows per Sale_Date"
FROM
      (SELECT Sale_Date, COUNT(*)
        FROM Sales_Table GROUP BY 1) AS TeraTom (Coll, Counter),

          (SELECT COUNT(DISTINCT(Sale_Date))
            FROM Sales_Table) AS Derived2 (num_rows)

QUALIFY ROW_NUMBER () OVER
          (ORDER BY TeraTom.Coll) = Derived2.num_rows /2 ;
```

Typical Rows Per Sale_Date
3

The query above retrieved the typical rows per value for the column Sale_Date.

Testing Your Knowledge

```
SELECT Product_ID , Sale_Date, Daily_Sales,
       CSUM(Daily_Sales, Product_ID, Sale_Date) AS "CSum"
FROM Sales_Table
WHERE Product_ID BETWEEN 1000 and 2000
GROUP BY Product_ID ;
```

This is the CSUM. However, what we want to see is the Sum()Over ANSI version. Use the information in the CSUM, and convert this to the equivalent Sum()Over.

Testing Your Knowledge

```
SELECT Product_ID , Sale_Date, Daily_Sales,
       CSUM(Daily_Sales, Product_ID, Sale_Date) AS "CSum"
FROM Sales_Table
WHERE Product_ID BETWEEN 1000 and 2000
GROUP BY Product_ID ;
```

```
SELECT Product_ID , Sale_Date, Daily_Sales,
SUM(Daily_Sales) OVER (PARTITION BY Product_ID
                       ORDER BY Product_ID, Sale_Date
                       ROWS UNBOUNDED PRECEDING) AS SumANSI
FROM Sales_Table
WHERE Product_ID BETWEEN 1000 and 2000 ;
```

Both statements are exactly the same except the bottom example uses ANSI syntax.

Testing Your Knowledge

```
SELECT Product_ID , Sale_Date, Daily_Sales,
 MAVG( Daily_Sales, 3, Product_ID, Sale_Date) AS AVG_for_3_Rows
FROM Sales_Table WHERE Product_ID BETWEEN 1000 and 2000 ;
```

Write the equivalent to the SQL above using ANSI Syntax such as AVG () Over.

Testing Your Knowledge

```
SELECT Product_ID , Sale_Date, Daily_Sales,
MAVG( Daily_Sales, 3, Product_ID, Sale_Date) AS AVG_for_3_Rows
FROM Sales_Table WHERE Product_ID BETWEEN 1000 and 2000 ;
SELECT Product_ID , Sale_Date, Daily_Sales,
  AVG(Daily_Sales) OVER (ORDER BY Product_ID, Sale_Date
                         ROWS 2 Preceding) AS AVG_3_ANSI
FROM Sales_Table WHERE Product_ID BETWEEN 1000 and 2000 ;
```

The SQL above is equivalent except the bottom example uses ANSI Syntax.

Testing Your Knowledge

```
SELECT Product_ID ,Sale_Date , Daily_Sales,
         RANK(Daily_Sales) AS "Rank"
FROM Sales_Table
WHERE Product_ID IN (1000, 2000) ;
```

This is the Rank. However, what we want to see is the RANK () Over. Use the information in the Rank to make it the Rank () Over.

Testing Your Knowledge

```
        SELECT Product_ID ,Sale_Date , Daily_Sales,
                 RANK(Daily_Sales) AS "Rank"
        FROM Sales_Table
        WHERE Product_ID IN (1000, 2000) ;
SELECT Product_ID ,Sale_Date , Daily_Sales,
        RANK() OVER (ORDER BY Daily_Sales DESC) AS Rank1
FROM Sales_Table
WHERE Product_ID IN (1000, 2000)
```

The SQL above is equivalent except the bottom example uses ANSI Syntax. Also, notice the sort key. DESC is the default in the top example.

Substrings and Positioning Functions

Substrings and Positioning Functions

The CHARACTERS Command Counts Characters

Employee_Table

Employee_No	Dept_No	Last_Name	First_Name	Salary
2000000	?	Jones	Squiggy	32800.50
1000234	10	Smythe	Richard	32800.00
1232578	100	Chambers	Mandee	48850.00
1324657	200	Coffing	Billy	41888.88
1333454	200	Smith	John	48000.00
2312225	300	Larkins	Loraine	40200.00
1256349	400	Harrison	Herbert	54500.00
2341218	400	Reilly	William	36000.00
1121334	400	Strickling	Cletus	54500.00

VARCHAR

```
SELECT First_Name
       ,CHARACTERS(First_Name) AS Lnth
FROM   Employee_Table
WHERE  CHARACTERS (First_Name) < 7
ORDER BY 1;
```

Answer Set

First_Name	Lnth
Billy	5
Cletus	6
John	4
Mandee	6

The CHARACTERS command counts the number of characters. If 'Tom' was in the Employee_Table, his length would be 3.

The CHARACTERS Command – Spaces can Count too

Employee_Table

Employee_No	Dept_No	Last_Name	First_Name	Salary
2000000	?	Jones	Squiggy	32800.50
1000234	10	Smythe	Richard	32800.00
1232578	100	Chambers	Mandee	48850.00
1324657	200	Coffing	Billy	41888.88
1333454	200	Smith	John	48000.00
2312225	300	Larkins	Loraine	40200.00
1256349	400	Harrison	Herbert	54500.00
2341218	400	Reilly	William	36000.00
1121334	400	Strickling	Cletus	54500.00

Answer Set

```
SELECT'T o m'AS First_Name
       ,CHARACTERS('T o m')AS Lnth
```

First_Name	Lnth
T o m	5

If ' To m' was in the Employee_Table, his length would be 5. Yes, spaces do count as characters.

The CHARACTERS Command and Char (20) Data

CHAR (20)

```
SELECT Last_Name
       ,CHARACTERS(Last_Name) AS Lnth
FROM   Employee_Table
ORDER BY 1;
```

Last_Name	Lnth
Chambers	20
Coffing	20
Harrison	20
Jones	20
Larkins	20
Reilly	20
Smith	20
Smythe	20
Strickling	20

The CHARACTERS command brings back a length of 20 every time for a Char (20) data type because of the spaces. Turn the page and we will explain further.

Troubleshooting the CHARACTERS Command

Last_Name is a CHAR(20) fixed length field.

CHAR(20)

```
SELECT Last_Name
,CHARACTERS(Last_Name) AS Lnth
FROM   Employee_Table;
```

Last_Name as a Char(20)

Jones _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ Spaces
Hanson _ _ _ _ _ _ _ _ _ _ _ _ _ _
McRoberts _ _ _ _ _ _ _ _ _ _ _
Johnson _ _ _ _ _ _ _ _ _ _ _ _ _

Last_Name	Lnth
Chambers	20
Coffing	20
Harrison	20
Jones	20
Larkins	20
Reilly	20
Smith	20
Smythe	20
Strickling	20

When it comes to Characters, 20 would be the length of each and every name. That is because it has been set as a CHAR (20) in the table create syntax.

TRIM for Troubleshooting the CHARACTERS Command

Last_Name is a CHAR(20) fixed length field.

Trim Command

```
SELECT Last_Name
       ,CHARACTERS(TRIM(Last_Name) ) AS C_Length
FROM   Employee_Table
ORDER BY 1 ;
```

Last_Name	Lnth
Chambers	8
Coffing	7

```
Hare:ison     8
Jones         5
Larkins       7
Reilly        6
Smith         5
Smythe        6
Strickling   !0
```

The TRIM command will trim off any spaces before and after the Last_name.

CHARACTERS and CHARACTER_LENGTH equivalent

```
SELECT First_Name
,C:iAR.�CTERS(F�rs� Name) AS
C_tengt
FROM Em lo ee Table;
```

Query2

```
SE�ECT First Name
,C:iA.RACTER_Leng�h(�irs�_Name) AS
C_Lengt
FROM Em lo ee Tab:e;
```

These two queries will get you the SAME EXACT answer set in your report

OCTET_LENGTH

Query 1

```
SELECT First Name
,CH�..RACTER(First_Name) AS
C_Lengt
FROM Em lo ee Table;
```

Query2

```
SE�ECT First Name
,C:iARACTE� Leng�h(First Name) AS
C_Lengt
PROM Em lo-ee Table ;
```

Query3

```
SELECT First Name
,Octet �ength(Pirst Name) AS C_Lengt
FROM Ein lo ee Table-;
```

You can also use the OCTET LENGTH command. These three queries get the same exact answer sets! Query 2 and 3 are ANSI Standard.

The TRIM Command trims both Leading and Trailing Spaces

Query 1

```
SELECT Last Name
,Trirn(Last Name) AS No_Space
FROM Em lo ee Table ;
```

Query2

```
SELECT Last_Name
,Trim(Both from Last_Name)AS No_Spaces
FROM Employee_Table ;
```

Both queries trim both the leading and trailing spaces from Last_Name.

Trim and Trailing is Case Sensitive

VARCHAR

```
SELECT First_Name,
Trim(trailing 'Y' from First_Name)  AS No_Y
FROM   Employee_Table
ORDER BY 1;
```

'Billy' and 'Squiggy' does not TRIM the trailing 'y' because it was after a capitol 'Y'

First_Name	No_Y
Billy	Billy
Clems	Cletus
Herbert	Herbert
John	John
Loraine	Loraine
Mandee	Mandee
Richard	Richard
Squiggy	Squiggy
William	William

For LEADING and TRAILNG, it IS case sensitive.

Trim and Trailing works if Case right

VARCHAR

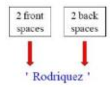
```
SELECT First_Name,
Trim(trailing 'y' from First_Name) AS No_Y
FROM   Employee_Table
ORDER BY 1;
```

```
First_Name No_Y
Billy      Bill
Clems      Cletus
Herbert    Herbert
John       John
Loraine    Loraine
Mandee     Mandee
Richard    Richard
Squiggy    Squigg
William    William
```

For LEADING and TRAILNG, it IS case sensitive.

Trim Combined with the CHARACTERS Command

```
SELECT' Rodriquez '
      ,Characters(Trim(' Rodriquez ')) AS No_Spaces ;
```

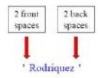


' Rodriquez '	No_Spaces
Rodriquez	9

This will allow for the character count to only be 9 because both the leading and trailing spaces have been cut.

How to TRIM only the Trailing Spaces

```
SELECT' Rodriquez '
      ,Characters(Trim(Trailing FROM ' Rodriquez ')) AS Front_Spaces ;
```



' Rodriquez '	Front_Spaces
Rodriquez	11

The TRAILING FROM Command allows you to only TRIM the spaces behind the Last_Name. Now, we will still get a character count of 11 because we are only cutting off the trailing spaces and not the beginning spaces.

How to TRIM Trailing Letters

VARCHAR

```
SELECT First_Name
       ,Trim(trailing 'y' from First_Name) AS No_Y
       ,Last_Name
       ,Trim(trailing 'g' from (trim (Last_Name))) AS No_G
FROM   Employee_Table ;
```

CHAR(20)

First_Name	No_Y	Last_Name	No_G
Squiggy	Squigg	Jones	Jones
John	John	Smith	Smith
Richard	Richard	Smythe	Smythe
Herbert	Herbert	Harrison	Harrison
Mandee	Mandee	Chambers	Chambers
Cletus	Cletus	Strickling	Stricklin
William	William	Reilly	Reilly

Billy	Bill	Coffing	Coffin
Loraine	Loraine	Larkins	Larkins

The above example removed the trailing 'y' from the First_Name and the trailing 'g' from the Last_Name. Remember that this is case sensitive.

How to TRIM Trailing Letters and use CHARACTER_Length

VARCHAR

CHAR(20)

```
SELECT First_Name,
       Trim(trailing 'y' from First_Name)  AS No_Y
       ,Last_Name
       ,Character_Length (Trim(trailing 'g' from (trim (Last_Name))) ) as No2
FROM   Employee_Table ;
```

First_Name	No_Y	Last_Name	No_2
Squiggy	Squigg	Jones	5
John	John	Smith	5
Richard	Richard	Smythe	6
Herbert	Herbert	Harrison	8
Mandee	Mandee	Chambers	8
Cletus	Cletus	Strickling	9
William	William	Reilly	6
Billy	Bill	Coffing	6
Loraine	Loraine	Larkins	7

Notice that the length (column No2) for Coffing is six and not seven. Notice the length for Strickling is 9 and not 10. This is because the trailing 'g' was trimmed before calculating the length. We trimmed the last name and then trimmed the trailing 'g'.

The SUBSTRING Command

```
SELECT First_Name,
SUBSTRING (First_Name FROM 2 for 3)  AS Quiz
FROM   Employee_Table ;
```

Start in Position 2

Go for 3 Positions

First_Name	Quiz
Squiggy	qui
John	ohn
Richard	ich
Herbert	erb
Mandee	and
Cletus	let
William	ill
Billy	ill
Loraine	ora

This is a SUBSTRING. The substring is passed two parameters. They are the starting position of the string and the number of positions to return (from the starting position). The above example will start in position 2 and go for 3 positions!

How SUBSTRING Works with NO ENDING POSITION

```
SELECT First_Name,
SUBSTRING (First_Name FROM 2)  AS GoToEnd
FROM   Employee_Table ;
```

Start in
Position 2

First_Name	GoToEnd
Squiggy	quiggy
John	ohn
Richard	ichard
Herbert	erbert
Mandee	andee
Cletus	letus
William	illiam
Billy	illy
Loraine	oraine

If you don't tell the Substring the end position, it will go all the way to the end.

Using SUBSTRING to move Backwards

```
SELECT First_Name,
SUBSTRING (First_Name FROM 0 For 6)  AS Before1
FROM   Employee_Table ;
```

Start in
Position 0
one space
before

First_Name	Before1
Squiggy	Squig
John	John
Richard	Richa
Herbert	Herbe
Mandee	Mande
Cletus	Cletu
William	Willi
Billy	Billy
Loraine	Lorai

A starting position of zero moves one space in front of the beginning. Notice that our FOR Length is 6 so 'Squiggy' turns into ' Squig'. The point being made here is that both the starting position and ending positions can move backwards, which will come in handy as you see other examples.

How SUBSTRING Works with a Starting Position of -1

```
SELECT First_Name,
SUBSTRING (First_Name FROM -1 For 3)  AS Before2
FROM   Employee_Table ;
```

Start in
Position -1
two spaces
before

First_Name	Before2
Squiggy	S
John	J
Richard	R
Herbert	H
Mandee	M
Cletus	C
William	W
Billy	B
Loraine	L

A starting position of -1 moves two spaces in front of the beginning character. Notice that our FOR Length is 3, so each name delivers only the first initial. The point being made here is that both the starting position and ending positions can move backwards, which will come in handy as you see other examples.

How SUBSTRING Works with an Ending Position of 0

```
SELECT First_Name,
SUBSTRING (First_Name FROM 3 For 0)  AS WhatsUp
FROM   Employee_Table ;
```

Start in
Position -1
two spaces
before

```
F-rs'C Name   v;t:.a":.sUF
Sqtaggy
John
Richard
Herbert
.
Mandee
Cletus
William
Billy
!.ora:.ne
```

In our example above, we start in position 3, but we go for zero positions so nothing is delivered in the column. That is what's up!

An Example using SUBSTRING, TRIM and CHAR Together

CHAR(20)

```
SELECT Last _Name
       .SUBSTRING(La�t_Name FRO:t'1
         CHARACTERS( TRI'.\1 {TRAILING FROM Last_Name)) ·I FOR 2) AS Letters
FROM Employee_Table:
```

```
Lsi:i!.      ,�t:t�:
HAm�         � es
Jones
Smi-ch         th
Smythe         he
Harr:.so       on
n              zs
Chambers       ng
Stricklin,i    ly
Reil:y         ng
Coffing        ns
:,ark:.ns
```

The SOL above brings back the last two letters of each Last_Name, even though the last names are of different length. We first trimmed the spaces off of Last_Name. Then we counted the characters in the Last_Name. Then we subtracted two from the Last_Name character length, and then passed it to our substring as the starting position. Since we didn't give an ending position in our substring it defaulted to the end.

SUBSTRING and SUBSTR are equal, but use differe,nt syntax

Query 1

```
SELECT First Name,
SUBS7RING(Fi;st Name FRO� 2 for3) AS
Qui
FROM Em lo cc 7ablc ;
```

Query2

```
SE�ECT First. Name,
SUBSTR(First.-Narne, 2 ,3) AS
FROM Emp lo y e; Table,
```

Both queries above are going to yield the same results! SUBSTR is just a different way of doing a substring. Both have two parameters in starting position and number of character length.

The POSITION Command finds a Letters Position

```
SE�ECT Last
�ane ,?os��:.or-('e' n Last Nane) AS Find
     ,Pos��ion('f' n Last=Nane) AS Find The
?ROM �-ployee ;
Tab:e
```

c is in 4t1i position

e is2r.d position in name

LastName	Find_Tue_E	Find_Tue_F
Jones	4	
Smith	0	0
Smythe	6	0
Harrison	0	0
Chambe	6	0
rs	0	0
Strickling	2	0
Reilly	0	3
Coffing	0	0
Larkins		

Nofisin the name

l"fi.s in 3rd position

This is the position counter. What it will do is tell you what position a letter is on. Why did Jones have a 4 in the result set? The 'e' was in the 4th position. Why did Smith get a zero for both columns? There is no 'e' in Smith and no 'f in Smith. If there are two 'fs' only the first occurrence is reported.

The POSITION Command is brilliant with SUBSTRING

```
SE:..ECT Dept No
       ,Jepartment Name as Jepty
      ,SU3STR(JeptJ,1 ,   ?OSIT:OK('  '   IN Jcpartreent_Nane) -1) as
      Wordl
FROM Department Table;
```

```
SE:..ECT Dept_Xo
       ,Jepa=tment Name as Jepty
      ,SUBSTR:NG(J;pty ?ROM 1 ;-OR ?OSITION('  '   IN Jepart�ent_Na.ne) -1) as
      Wo=dl
FROM Department Table;
```

```
Dept No  Depty                    ____
  200    Research and Develop     Research
  100    Marketing                �a=ket�n
                                  g
  400    Custome= Suppo=t         Custocer
  300    Sales                    sa:es
  500    Suman Resou=ces          Eun.an
```

What was the starting position of the Substring in the above query? It was one. The ending position (FOR length) was calculated to look for the first space and then subtract 1. So for "Research and Develop" the starting position was one and for 9-1 = 8.

Quiz -Name that SUBSTRING Starting and For Length

```
SE�ECT Dept_No
  ,Department �a�e as Depty
,SUBSTRIXG(Jepty PROM 1 c-OR POSlT:ON('  '   IN Jepartoent_Nane) -1) as Wo=dl
FROM Department Table;
```

```
Dept No  Depty                    Wordl
  200    Research and �evelop     Research
  100    Marketing                Marketing
  400    Customer Support         Custome=
  300    Sales                    Sales
  500    Human Resources          :3.uman
Marketing (FRO!- __ E'O __ )
Research and Jevelop (FROM __ FOR
Sales (FROM __ FOR __ )
Customer Support (;RO!- __ FOR    )
Human Resources (PROM� POR __ -i-
```

Fill in the number for the FROM and the FOR numbers above for each row. Next page!

The POSITION Command is brilliant with SUBSTRING

```
SELECT Dept_Xo
       ,�epartment Name as �epty
      ,SUBSTR(Jepty,l ,   ?OSlT:ON('  '   IN Depart�ent_Naoe) -1) as
      Wordl
FROM Department_Table;
```

```
SELECT Dept_No
      ,Department_Name as Depty
 ,SUBSTRING(Depty FROM 1 FOR POSITION(' ' IN Department_Name) -1) as Word1
FROM Department_Table;
```

```
Dept_No  Depty                 Word1
   200   Research and Develop  Research
   100   Marketing             Marketing
   400   Customer Support      Customer
   300   Sales                 Sales
   500   Human Resources       Human
```

What was the starting position of the Substring in the above query? It was one. The ending position (FOR length) was calculated to look for the first space and then subtract 1. So for "Research and Develop" the starting position was one and for 9-1 = 8.

Quiz – Name that SUBSTRING Starting and For Length

```
SELECT Dept_No
  ,Department_Name as Depty
,SUBSTRING(Depty FROM 1 FOR POSITION(' ' IN Department_Name) -1) as Word1
FROM Department_Table;
```

```
Dept_No  Depty                 Word1
   200   Research and Develop  Research
   100   Marketing             Marketing
   400   Customer Support      Customer
   300   Sales                 Sales

   500   Human Resources       Human
```

```
Marketing (FROM__ FOR__ )
Research and Develop (FROM__ FOR__ )
Sales (FROM__ FOR __)
Customer Support (FROM__ FOR__ )
Human Resources (FROM__ FOR__ )
```

Fill in the number for the FROM and the FOR numbers above for each row. Next page!

Answer to Quiz – Name that Starting and For Length

```
SELECT Dept_No
  ,Department_Name as Depty
,SUBSTRING(Depty FROM 1 FOR POSITION(' ' IN Department_Name) -1) as Word1
FROM Department_Table;
```

```
Dept_No  Depty                 Word1
   200   Research and Develop  Research
   100   Marketing             Marketing
   400   Customer Support      Customer
   300   Sales                 Sales
   500   Human Resources       Human
```

The FOR Length is calculated by finding the length up to the first SPACE and then subtracting 1.

```
Marketing (FROM 1 FOR 9 )
Research and Develop (FROM 1 FOR 8 )
Sales (FROM 1 FOR 5)
Customer Support (FROM 1 FOR 8 )
Human Resources (FROM 1 FOR 5 )
```

The FOR was calculated in the POSITION Subquery.

Answer to Quiz – Name that Starting and For Length

```
SELECT Dept_No
  ,Department_Name as Depty
,SUBSTRING(Depty FROM 1 FOR POSITION(' ' IN Department_Name) -1) as Word1
FROM Department_Table;
```

Dept_No	Depty	Word1
200	Research and Develop	Research
100	Marketing	Marketing
400	Customer Support	Customer
300	Sales	Sales
500	Human Resources	Human

The FOR Length is calculated by finding the length up to the first SPACE and then subtracting 1.

```
Marketing (FROM 1 FOR 9 )
Research and Develop (FROM 1 FOR 8 )
Sales (FROM 1 FOR 5)
Customer Support (FROM 1 FOR 8 )
Human Resources (FROM 1 FOR 5 )
```

The FOR was calculated in the POSITION Subquery.

Using the SUBSTRING to Find the Second Word On

```
SELECT DISTINCT Department_Name as Dept_Name
,SUBSTRING(Department_Name FROM
 POSITION(' ' IN Department_Name) +1) as Word2
 FROM Department_Table
 WHERE POSITION(' ' IN trim(Department_Name)) >0;
```

Dept_Name	Word2

```
Customer Support      Support
Human Resources       Resources
Research and Develop  and Develop
```

Notice we only had three rows come back. That is because our WHERE looks for only Department_Name that has multiple words. Then notice that our starting position of the Substring is a subquery that looks for the first space. Then it adds 1 to the starting position and we have a starting position for the 2nd word. We don't give a FOR length parameter, so it goes to the end.

Quiz – Why did only one Row Return

```
   SELECT Department_Name
     ,SUBSTRING(Department_Name from
           POSITION(' ' IN Department_Name) + 1 +
           POSITION(' ' IN SUBSTRING(Department_Name
     FROM POSITION(' ' IN Department_Name) + 1))) as Third_Word
FROM Department_Table
WHERE POSITION(' ' IN
  TRIM(Substring(Department_Name from
        POSITION(' ' in Department_Name) + 1)))> 0
```

```
Dept_Name             Third_Word
Research and Develop  Develop
```

Why did only one row come back?

Answer to Quiz – Why Did only one Row Return

```
   SELECT Department_Name
     ,SUBSTRING(Department_Name from
           POSITION(' ' IN Department_Name) + 1 +
           POSITION(' ' IN SUBSTRING(Department_Name
     FROM POSITION(' ' IN Department_Name) + 1))) as Third_Word
FROM Department_Table
WHERE POSITION(' ' IN
 TRIM(Substring(Department_Name from
       POSITION(' ' in Department_Name) + 1)))> 0
```

Dept_Name	Third_Word
Research and Develop	Develop

It has 3 words

Why did only one row come back? It's the Only Department Name with three words. The SUBSTRING and the WHERE clause both look for the first space, and if they find it then look for the second space. If they find that, add 1 to it and their Starting Position is the third word. There is no FOR position so it defaults to "go to the end".

Concatenation

```
SELECT First_Name
      ,Last_Name
      ,First_Name
       ||' '
       || Last_Name as Full_Name
FROM Employee_Table
WHERE First_Name = 'Squiggy'
```

Two Pipe Symbols together (no space) mean concatenate

First Name	Last Name	Full Name
Squiggy	Jones	Squigay Jones

See those II? Those represent concatenation. That allows you to combine multiple columns into one column. The II (Pipe Symbol) on your keyboard is just above the ENTER key. Don't put a space in between, but just put two Pipe Symbols together.

Concatenation and SUBSTRING

In this example, we have combined the first name, then a single space, and then the last name to get a new column called 'Full name', like Squiggy Jones.

```
SELECT First_Name
      ,Last_Name
      ,Substring(First_Name, 1, 1) || '. ' || Last_Name
       as Full_Name
FROM Employee_Table
WHERE First_Name = 'Squiggy'
```

A Period (.) and a space

First Name	Last Name	Full Name
Squiggy	Jones	S. Jones

Of the three items being concatenated together, what is the first item of concatenation in the example above? It is the first initial of the First_Name. Then we concatenated a literal space and a period. Then we concatenated the Last_Name. Notice that the report shows only three columns.

Four Concatenations Together

```
SELECT First_Name
      ,Last_Name
      ,TRIM(Last_Name) || ' ' || Substring(First_Name, 1, 1) || '.' AS Last_Name_1st
FROM Employee_Table
WHERE First_Name = 'Squiggy' ;
```

VARCHAR

First Name	Last Name	Last_Name_1st
Squiggy	Jones	Jones S.

Why did we TRIM the Last_Name? To get rid of the spaces or the output would have looked odd. How many items are being concatenated in the example above? There are 4 items concatenated. We start with the Last_Name (after we trim it), then we have a single space, then we have the First Initial of the First Name, and then we have a Period.

Troubleshooting Concatenation

```
SELECT First_Name,
      , Last_Name
      , TRIM (Last_Name) | | ' ' | | Substring (First_Name, 1, 1) || AS Last_Name_1st
FROM Employee_Table
WHERE First_Name = 'Squiggy'
```

ERROR

What happened above to cause the error? Can you see it? The Pipe Symbols II have a space between them like 1 1, when it should be 11. It is a tough one to spot so be careful.

Temporal Tables Create Functions

Temporal Tables Create Functions

Three types of Temporal Tables

The Three types of Temporal Tables are:

1. Valid Time Temporal Tables
2. Transaction Time Temporal Tables
3. Bi-Temporal Tables containing both Valid Time and Transaction Time.

Temporal Tables use a Valid Time or Transaction Time or combine both Valid Time and Transaction Time to form Bi-Temporal tables.

CREATING a Bi-Temporal Table

```
CREATE MULTISET TABLE Property_Owners
( Cust_No          INTEGER
 ,Prop_No          INTEGER
 ,Prop_Val_Time    PERIOD (DATE) NOT NULL as VALIDTIME
 ,Prop_Tran_Time   PERIOD (TIMESTAMP(6) with TIME ZONE)
                       NOT NULL as TRANSACTIONTIME
) PRIMARY INDEX(Prop_No) ;
```

This is a Bi-Temporal Table because one column is aliased VALIDTIME, and another column is aliased TRANSACTIONTIME. This makes this table a Bi-Temporal Table.

PERIOD Data Types

```
CREATE MULTISET TABLE Property_Owners
( Cust_No         INTEGER
 ,Prop_No         INTEGER
 ,Prop_Val_Time  PERIOD(DATE) NOT NULL as VALIDTIME
 ,Prop_Tran_Time PERIOD(TIMESTAMP(6) with TIME ZONE)
                      NOT NULL as TRANSACTIONTIME
) PRIMARY INDEX(Prop_No) ;
```

A Period Data Type means a beginning and ending date or Timestamp:

2011-01-01 , 9999-12-31
Or
2011-01-01 , 2012-06-30
Or
2011-01-01 08:09:290000-05:00, 9999-12-31 23:59:59.999999+00:00

A new data type PERIOD has been introduced. This means two dates (begin and end date), or it could be two Timestamps (begin and ending Timestamp).

Bi-Temporal Data Type Standards

```
CREATE MULTISET TABLE Property_Owners
(
 Cust_No         INTEGER
,Prop_No         INTEGER
,Prop_Val_Time  PERIOD(DATE) NOT NULL as VALIDTIME
,Prop_Tran_Time PERIOD(TIMESTAMP(6) with TIME ZONE)
                      NOT NULL as TRANSACTIONTIME
)
PRIMARY INDEX(Prop_No) ;
```

What PERIOD Data Types do ValidTime and TransactionTime require?

- ValidTime can be either a date or a Timestamp
- TransactionTime must be a Timestamp written exactly as above!

The example above is perfect for your PERIOD Data type for TRANSACTIONTIME. You have options for the VALIDTIME, as it can be either a Date or Timestamp.

Bi-Temporal Example – Tera-Tom buys!

```
CREATE MULTISET TABLE Property_Owners
(
 Cust_No         INTEGER
,Prop_No         INTEGER
,Prop_Val_Time  PERIOD(DATE) NOT NULL as VALIDTIME
,Prop_Tran_Time PERIOD(TIMESTAMP(6) with TIME ZONE)
                      NOT NULL as TRANSACTIONTIME
)
PRIMARY INDEX(Prop_No) ;
```

```
INSERT INTO PROPERTY_OWNERS
       (Cust_No, Prop_No)
        VALUES (1, 100) ;
```

On January 1, 2011, Tera-Tom buys property 100 which is beach front property. Tera-Tom is Cust_No 1 in your table and number 1 in your heart.

A Look at the Temporal Results

```
INSERT INTO PROPERTY_OWNERS
  (Cust_No, Prop_No)
    VALUES (1, 100) ;
```

Below, is what the table looks like internally.

TransactionTime should be displayed as Timestamp but not enough room here.

Property_Owners

Cust_No	Prop_No	ValidTime	TransactionTime
1	100	2011-01-01, 9999-12-31	2011-01-01 , 9999-12-31

On January 1, 2011, Tera-Tom buys property 100 and this is what the Bi-Temporal table looks like. Notice the 9999-12-31 dates. That means this is an OPEN Date.

Bi-Temporal Example - Tera-Tom Sells!

```
UPDATE Property_Owners
SET Cust_No= 2
WHERE Prop_No = 100 ;
```

How will the table below change after the UPDATE?

TransactionTime should be displayed as Timestamp, but not enough room here.

Property_Owners

Cust_No	Prop_No	ValidTime	TransactionTime
1	100	2011-01-01, 9999-12-31	2011-01-01 , 9999-12-31

On January 1, 2011 Tera-Tom buys property 100, and then Tera-Tom sells to Socrates (Cust_No 2) on February 14th, 2011.

Bi-Temporal Example - How the data looks!

Property_Owners Before Update

Cust_No	Prop_No	ValidTime	TransactionTime
1	100	2011-01-01, 9999-12-31	2011-01-01 , 9999-12-31

Property_Owners After Update

Cust_No	Prop_No	ValidTime	TransactionTime
1	100	2011-01-01, 9999-12-31	2011-01-01 , 2011-02-14
1	100	2011-01-01, 2011-02-14	2011-02-14 , 9999-12-31
2	100	2011-02-14, 9999-12-31	2011-02-14 , 9999-12-31

Here is how the new table looks like with three rows. In the bottom table example, there is only 1-row that is still open. Do you know which one? The last one!

Normal SQL for Bi-Temporal Tables

Property_Owners Table

Cust_No	Prop_No	ValidTime	TransactionTime
1	100	2011-01-01, 9999-12-31	2011-01-01 , 2011-02-14
1	100	2011-01-01, 2011-02-14	2011-02-14 , 9999-12-31
2	100	2011-02-14, 9999-12-31	2011-02-14 , 9999-12-31

```
SELECT *   FROM Property_Owners ;
```

Normal SQL

Cust_No	Prop_No
2	100

Shows only open rows

It is special SQL that allows Bi-Temporal tables to work so effectively. You will see a wide variety of SQL Keywords before the real SQL starts. The first is normal SQL.

NONSEQUENCED SQL for Temporal Tables

Property_Owners Table

Cust_No	Prop_No	ValidTime	TransactionTime
1	100	2011-01-01, 9999-12-31	2011-01-01 , 2011-02-14
1	100	2011-01-01, 2011-02-14	2011-02-14 , 9999-12-31
2	100	2011-02-14, 9999-12-31	2011-02-14 , 9999-12-31

Nonsequenced Keyword

```
NONSEQUENCED VALIDTIME
SELECT *   FROM Property_Owners ;
```

Cust_No	Prop_No	Prop_Val_Time
1	100	2011-01-01 , 2011-02-14
2	100	2011-02-14 , 9999-12-31

Shows all customers

It is special SQL that allows Bi-Temporal tables to work so effectively. Here is a look at the keyword NONSEQUENCED. This brings back all customers.

AS OF SQL for Temporal Tables

Property_Owners Table

Cust_No	Prop_No	ValidTime	TransactionTime
1	100	2011-01-01, 9999-12-31	2011-01-01 , 2011-02-14
1	100	2011-01-01, 2011-02-14	2011-02-14 , 9999-12-31
2	100	2011-02-14, 9999-12-31	2011-02-14 , 9999-12-31

```
VALIDTIME AS OF DATE '2011-01-30'
SELECT *   FROM Property_Owners ;
```

AS OF SQL

Cust_No	Prop_No
1	100

Point of view as of Date

It is special SQL that allows Bi-Temporal tables to work so effectively. The VALIDTIME AS OF DATE '2011-01-30' keywords report the state of Property_Owners on that exact date.

NONSEQUENCED for Both

Cust_No	Prop_No	ValidTime	TransactionTime
1	100	2011-01-01, 9999-12-31	2011-01-01 , 2011-02-14
1	100	2011-01-01, 2011-02-14	2011-02-14 , 9999-12-31
2	100	2011-02-14, 9999-12-31	2011-02-14 , 9999-12-31

```
NONSEQUENCED VALIDTIME
AND NONSEQUENCED TRANSACTIONTIME
SELECT* FROM Property_Owners ;
```

```
Cus_No    Prop_No    Prop_Val_Time                Prop_Tran_Time
  1         100      2011-01-01 , 2011-02-14      2011-01-01 , 2011-02-14
  1         100      2011-01-01 , 2011-02-14      2011-02-14 , 9999-12-31
  2         100      2011-02-14 , 9999-12-31      2011-02-14 , 9999-12-31
```

It is special SQL that allows Bi-Temporal tables to work so effectively. Above, are NONSEQUENCED VALIDTIME and NONSEQUENCED TRANSACTIONTIME

Creating Views for Temporal Tables

```
CREATE VIEW SQL01.Prop_As_Is
AS
Locking row for access
CURRENT VALIDTIME
SELECT Cust_No
      ,Prop_No
BEGIN(Prop_Val_Time) AS Beg_Val_Time,
END(Prop_Val_Time) AS End_Val_Time,
FROM Property_Owners;
```

```
CREATE VIEW SQL01.Prop_As_Was
AS
Locking row for access
NONSEQUENCED VALIDTIME
SELECT Cust_No
      ,Prop_No
BEGIN(Prop_Val_Time) AS Beg_Val_Time,
END(Prop_Val_Time) AS End_Val_Time,
FROM Property_Owners;
```

```
SELECT * FROM SQL01.Prop_As_Is ;
```

```
SELECT * FROM SQL01.Prop_As_Was ;
```

You can create views that will allow users to see the way things are or the way things were. Above, are two excellent examples.

Bi-Temporal Example – Socrates is DELETED!

```
DELETE FROM Property_Owners
WHERE Prop_No = 100 ;
```

How will the table change below after the DELETE?

Property_Owners Before DELETE

Cust_No	Prop_No	ValidTime	TransactionTime
1	100	2011-01-01, 9999-12-31	2011-01-01 , 2011-02-14
1	100	2011-01-01, 2011-02-14	2011-02-14 , 9999-12-31
2	100	2011-02-14, 9999-12-31	2011-02-14 , 9999-12-31

On April Fool's Day, April 1, 2011, Socrates sells the property but through another Mortgage company, so, since the mortgage company no longer owns the property, Socrates is DELETED. How will the table look after the Delete?

Bi-Temporal Results – Socrates is DELETED

Property_Owners Before DELETE on April 1st

Cust_No	Prop_No	ValidTime	TransactionTime
1	100	2011-01-01, 9999-12-31	2011-01-01 , 2011-02-14
1	100	2011-01-01, 2011-02-14	2011-02-14 , 9999-12-31
2	100	2011-02-14, 9999-12-31	2011-02-14 , 9999-12-31

Property_Owners AFTER DELETE on April 1st

Cust_No	Prop_No	ValidTime	TransactionTime

1	100	2011-01-01, 9999-12-31	2011-01-01, 2011-02-14
1	100	2011-01-01, 2011-02-14	2011-02-14, 9999-12-31
2	100	2011-02-14, 9999-12-31	2011-02-14, 2011-04-01
2	100	2011-02-14, 2011-04-01	2011-04-01, 9999-12-31

Here is the table and it has no Open Rows. The bold red shows why the row is closed.

Temporary Tables

Temporary Tables

Temporary Tables

There may be times when an existing production database table does not provide precisely what you need. Sometimes, a particular query might need summarized or aggregated data. At other times, a small number of rows, from a very large table or data for a specific organization, are required to find an answer.

In a data warehouse with millions of rows, it might take too long to locate, derive or mathematically calculate the data needed. This is especially true when it is needed more than once per day. So, a view might not be the best solution or a view does not exist and you do not have the privilege to create one and both a view and derived table take too long. Any of these conditions prevent the ability to complete the request.

In the past, temporary tables have been created and used to help SQL run faster or be more efficient. They are extremely useful for solving problems that require stored "temporary" results or which require multiple SQL steps. They are also great for holding aggregated or summarized data.

Most databases lose speed when they have to:

- Read every row in a very large table (full table scan)
- Perform several aggregations
- Perform several data type conversions
- Join rows together from multiple tables
- Sort data

Temporary tables are often useful in a de-normalization effort. This might be done to make certain queries execute faster. Other times it is done to make the SQL easier to write, especially when using tools that generate SQL. However, these temporary tables are real tables and require manual operations to create, populate, and maintain them.

As a result, better name for these temporary tables might be interim or temporal tables. They exist for a specific period of time and when no longer needed, they are dropped to free up the disk space. During the interim time, they provide a valuable service. However, if the data in the original tables changes, the interim tables must be repopulated to reflect that change. This adds a level of difficulty or complexity regarding their use.

Temporary Table Choices

There are three types of temporary tables available within Teradata. All of which have advantages over traditional temporary tables.

Derived tables are always local to a single SQL request. They are built dynamically using an additional SELECT within the query. The rows of the derived table are stored in spool and discarded as soon as the query finishes. The DD has no knowledge of derived tables. Therefore, no extra privileges are necessary. Its space comes from the users spool space.

Volatile Temporary tables are local to a session rather than a specific query. This means that the table may be used repeatedly within a user session. That is the major difference between volatile temporary tables (multiple use) and derived tables (single use). Like a derived, a volatile temporary table is materialized in spool space. However, it is not discarded until the session ends or when the user manually drops it. The DD has no knowledge of volatile temporary tables. They are often simply called, volatile tables; no extra privileges are required to use them either. Its space comes from the users spool space. New with V2R6.1 a Volatile table can also be a partitioned table (a PPI table). You still can NOT COLLECT STATISTICS on a Volatile Table in Teradata V12, but that will change in Teradata V13.

Global Temporary tables are local to a session, like volatile tables. However, they are known in the DD where a permanent definition is kept. Global temporary tables are materialized within a session in a new type of database area called temporary space. Also like volatile tables, they are discarded at the end of the session or when the user manually requests the table to be dropped. They are often called, global tables. Its space comes from a new type of space called temporary space. New with V2R6.1 a Global Temporary table can also be a partitioned table (a PPI table). You can also COLLECT STATISTICS on a Global Temporary Table

Derived Tables

Derived tables were introduced into Teradata with V2R2. The creation of the derived table is local to the SQL statement and available only for a single request. However, a request may contain multiple derived tables. Once these tables are defined and populated, they may be joined or manipulated just like any other table. Derived tables become an alternative to creating views or the use of interim tables.

Derived tables are very useful. However, since they only exist for the duration of a single request, they may not be a practical solution if the rows are needed for multiple, follow-up queries needing the same data. The derived table is materialized in spool space, used and dropped automatically at the end of the query. Since it is entirely in spool, it only requires the user to have enough spool space. Since there is no DD involvement, special privileges are not required.

The process of deriving a table is much like deriving column data. They are both done dynamically in an SQL statement. The main difference is that column data is normally derived in the SELECT list, but derived tables are defined in the FROM. A derived table is created dynamically by referring to it in the FROM portion of a SELECT, UPDATE or DELETE. Like all tables, it needs a table name, one or more column names and data rows. All of these requirements are established in the FROM clause of an SQL statement.

The following shows how to use the syntax for creating a derived table:

```
SELECT <column-name> [.., <column-name> ]
FROM   ( SELECT <column-name>  [ AS <alias-name> ]
         [..,<column -name> ]   FROM <table-name> )
     <Derived-table-name> [ ( <alias-name> [..,<alias-name> ] ) ];
```

In the above syntax, everything after the first FROM is used to dynamically name the derived table with its columns and populate it with a SELECT. The SELECT is in parentheses and looks like a subquery. However, subqueries are written in the WHERE clause and this is in the FROM. This SELECT is used to populate the table like an INSERT/SELECT for a real table, but without the INSERT.

The derived table and its columns must have valid names. If desired, the derived table column names can default to the actual column names in the SELECT from a real table. Otherwise, they can be alias names established using AS in the SELECT of the derived table, or specified in the parentheses after the name of the derived table, like in a CREATE VIEW. Using this technique is our preference. It makes the names easy to find because they are all physically close together and does not require a search through the entire SELECT list to find them.

These columns receive their data type from the columns listed in the SELECT from a real table. Their respective data types are established as a result of the sequence that the columns appear in the SELECT list. If a different data type is required, the CAST can be used to make the adjustment.

The following is a simple example using a derived table named TeraTom with a column alias called avgsal and its data value is obtained using the AVG aggregation:

```
SELECT  *
FROM    (SELECT AVG(salary) FROM Employee_table)  TeraTom(avgsal) ;
```

1 Row Returned

avgsal
46782.15

Once the derived table has been materialized and populated, the actual SQL statement reads its rows from the derived table, just like any other table. Although this derived table and its SELECT are simplified, it can be any valid SELECT and therefore can use any of the SQL constructs such as: inner and outer joins, one or more set operators, subqueries and correlated subqueries, aggregates and OLAP functions. Like a view, it cannot contain an ORDER BY, a WITH, or a WITH BY. However, these operations can still be requested in the main query, just not in the SELECT for the derived table.

The best thing about a derived table is that the user is not required to have CREATE TABLE privileges and after its use. A derived table is automatically "dropped" to "clean up" after itself. However, since it is dropped the data rows are not available for a second SELECT operation. When these rows are needed in more than a single SELECT, a derived table may not be as efficient as a volatile or global temporary table.

The next example uses the same derived table named TeraTom to join against the Employee table to find all the employees who make more than the average salary:

```
SELECT Last_name
       ,Salary
       ,Avgsal
FROM (SELECT AVG(salary) FROM Employee_table) AS TeraTom(avgsal)
     INNER JOIN Employee_table
           ON  avgsal < salary;
```

Now that avgsal is a defined column(in the derived table), it can be selected for display as well as being compared to determine which rows to return.

You must name your Derived Table, but you can name the columns in the Derived Table in different ways. Here is the same query as above written differently.

```
SELECT  Last_name
        ,Salary
        ,Avgsal
FROM   (SELECT AVG(salary) as avgsal FROM Employee_table) AS TeraTom
       INNER JOIN Employee_table
                 ON  avgsal < salary;
```

You can have multiple columns in the derived table.

```
SELECT  Last_name
        ,Salary
        ,Avgsal
        ,Maxsal
FROM (SELECT AVG(salary) as avgsal, Max(Salary) as maxsal FROM
Employee_table) AS TeraTom
     INNER JOIN Employee_table
           ON  avgsal < salary;
```

The above examples only had one row in the derived table, but our next query will find out who is making more than the average salary within their own department. The derived table we build will have one row for each department.

```
SELECT Last_name
       ,Dept_No
       ,Salary
       ,Avgsal
FROM (SELECT Dept_No as Depty, AVG(salary) as avgsal FROM Employee_table
       GROUP BY 1) AS TeraTom
      NNER JOIN Employee_table
           ON  Dept_No = Depty
           WHERE Salary > Avgsal ;
```

I will write the exact same query again, but moving things around. Can you see the differences?

```
SELECT Last_name
       ,Dept_No
       ,Salary
       ,Avgsal
FROM   Employee_Table
INNER JOIN
        (SELECT Dept_No, AVG(salary) FROM Employee_table
         GROUP BY 1) AS TeraTom (Depty, Avgsal)
ON  Dept_No = Depty
WHERE Salary > Avgsal ;
```

The differences between the two above queries is that I changed the order of the two tables and put Employee_Table first and then joined it with TeraTom. I also defined the columns in my derived table after defining TeraTom. Doesn't matter, but you will see Derived Tables written many different ways, but now you will recognize them all.

This example uses a derived table:

```
SELECT   Product_ID AS Product,   Cal_yr
         ,Sep_sales    AS  September_sales
         ,Oct_sales  AS  October_sales
         ,Nov_sales  AS  November_sales
FROM (SELECT Product_ID ,EXTRACT(YEAR FROM Sale_date) AS Cal_Yr
          ,SUM(CASE ((Sale_date/100) MOD 100)
                    WHEN 9  THEN  Daily_Sales
```

```
                  ELSE 0
              END)  AS Sep_sales
          ,SUM(CASE ((Sale_date/100) MOD 100)
                  WHEN 10  THEN  Daily_Sales
                  ELSE 0
              END)  AS Oct_sales
          ,SUM(CASE ((Sale_date/100) MOD 100)
                  WHEN 11  THEN  Daily_Sales
                  ELSE 0
              END)  AS Nov_sales
        FROM   Sales_table
        WHERE Sale_date BETWEEN 1000901 AND 1001130
        GROUP BY 1,2)
    DT_Month_Sum_Sales
```

/* The Derived table above is called DT_Month_Sum_Sales and gets its column names from the alias names of the above SELECT in parentheses*/
WHERE Cal_Yr = 2000
ORDER BY 1 ;

3 Rows Returned

Product ID	Cal Yr	September_sales	October_sales	November_sales
1000	2000	139350.69	191854.03	0
2000	2000	139738.90	166872.90	0
3000	2000	139679.76	84908.06	0

The next SELECT is rather involved; it builds My_Derived_Tbl as a derived table:

```
SELECT       Derived_Col1
             ,Derived_Col2
             ,Payment_date
             ,Payment_amount
/* The Derived table definition starts below  */
FROM     (SELECT OthT1_Col1, OthT2_Col2, OthT1_Col3
             FROM Oth_Tbl_1  AS OT1
             INNER JOIN Oth_Tbl_2  AS OT2
             ON  OT1.Col3 = OT2.Col3
/* The correlated subquery to populate the Derived table starts below */
         WHERE OT1.Sale_date =  (SELECT  MAX(Purchase_Date)
                                 FROM Sales_Tbl
                                 WHERE  OT1.OthT1_Col3 = Sales_Product )
        My_Derived_Tbl   ( Derived_Col1, Derived_Col2, Derived_Col3 )
/*  The Derived table definition ends here    */
        RIGHT OUTER JOIN   Payment_Tbl  AS  PT
          ON Derived_Col3 = Payment_Col5
/*  The correlated subquery for the main SELECT starts below */
 WHERE  Payment_Date = (SELECT MAX(Payment_Date) FROM Payment_Tbl
                        WHERE Payment_Tbl.Account_Nbr=PT.Account_Nbr);
```

The derived table is created using an INNER JOIN and a Correlated Subquery. The main SELECT then uses the derived table as the outer table to process an OUTER JOIN. It is joined with the Payment table and uses a Correlated Subquery to make sure that only the latest payment is accessed for each account.

Whether your requirements are straightforward or complex, derived tables provide an ad hoc method to create a "table" with data rows and use them one time in an SQL statement without needing a real table to store them.

Derived Tables Using a Non-Recursive WITH

Starting with V2R6 there is a second way to create and use **Derived tables**. It is very similar to the original technique begun in release V2R3, with one main difference. That difference is that they are no longer defined in the FROM, like previously. Instead, they are defined at the beginning of the query using a WITH as seen in the following syntax:

```
WITH <Derived-table-name> [ ( <column-alias-name> [..,<column-alias-name> ]
   AS  (SELECT <column-name>  [ AS <column-alias-name> ]
         [..,<column-name> [ AS <column-alias-name> ]  ]
```

```
        FROM <table-name> )
SELECT <column-alias-name> [..., <column-alias-name> ]
FROM   <Derived-table-name> ;
```

The primary advantage to this technique is that now a derived table can be referenced multiple times without being required to code the entire SELECT over again. It now has a table name that is known throughout the query.

The derived table example that was used earlier is rewritten below using the WITH:

```
WITH DT(avgsal) AS (SELECT AVG(salary) FROM Employee_table)
SELECT Last_name
       ,Salary
       ,Avgsal
FROM  DT INNER JOIN Employee_table
           ON avgsal < salary;
```

5 Row Returned

Last name	Salary	Avgsal
Chambers	48850.00	46782.15
Smythe	64300.00	46782.15
Smith	48000.00	46782.15
Harrison	54500.00	46782.15
Strickling	54500.00	46782.15

The power of the WITH is that it makes the table globally available:

```
WITH DT(deptno, maxsal) AS
      (SELECT dept_no, MAX(salary) FROM Employee_table
       GROUP BY 1)
SELECT Last_name
       ,deptno
       ,Salary
       ,Maxsal
FROM  DT  INNER JOIN Employee_table
            ON  dept_no=deptno
WHERE avgsal < salary  and
      Deptno IN (SELECT deptno FROM DT JOIN Department_table
                 ON deptno=dept_no and
   /* deptno is not correlated, DT is globally available inside the subquery  */
                       department name= 'Customer support');
```

2 Row Returned

Last name	Deptno	Salary	Maxsal
Harrison	400	54500.00	54500.00
Strickling	400	54500.00	54500.00

Derived Tables Using a Recursive WITH

The other new variation of a Derived table in V2R6 is that of a recursive **Derived table**. This form begins much like the non-recursive WITH. The major difference is that normally a Set Operator (Chapter 11) is also used and one of the SELECT statements within the derived table references the derived table name. The recursive nature is that the table is joined multiple times as long as rows matched in the previous iteration.

The syntax is shown here:

```
WITH RECURSIVE <Derived-table-name [ ( <column-alias-name> [...,<column-
alias-name ]
      AS   (SELECT <column-name  [ AS <derived-column-name> ]
           [... ,<column-name> [ AS <derived-column-name> ] ]
            FROM <table-name>
           [WHERE <column-name> <comparison-operator> <value-or-value-list> ]
         ( UNION | INTERSECT | EXCEPT | MINUS   [ ALL ] )
            SELECT <column-name>  [ AS <column-alias-name>]
```

```
            [... ,<column-name> [ AS <column-alias-name> ]]
             FROM <Derived-table-name>
             WHERE <column-name. = <derived-column-name> )
SELECT <column-alias-name> [..., <column-alias-name>]
FROM   <Derived-table-name> ;
```

In effect what this does internally is a self-join of the table(s) referenced in the first SELECT following the AS. It is for building output such as bill of material, organizational structure and any output that benefits from a hierarchical arrangement.

The following shows an example of this form of query and the EXPLAIN:

```
EXPLAIN WITH RECURSIVE Assembly_dt (partno,assemblyname,subpart,depth) AS
       (select part_no, part_desc,subpart_no, 0 from parts_vt
          where part_no IN (select partno from BOM_table)
         UNION ALL
        select subpart_no, assemblyname, subpart,depth+1
        from parts_vt as p join Assembly_dt as a on a.partno=p.subpart_no
        where depth < 3)
select partno, assembl,'count 1',count(*),depth from Assembly_dt
group by depth
order by 4 ;
```

The first thing to notice is the zero in the SELECT before the UNION establishes a column with an initial value of zero. This literal is associated with the column name of depth as defined in the WITH RECURSIVE clause. The rows selected here are placed into a SPOOL file.

Once these rows are in SPOOL, the iteration begins. At this point Teradata is joining these initial rows back to the table. As these rows are joined, it selects the depth+1 column in the second select and sets up the recursive nature as it "loops" through the join processing and adds one more in each consecutive iteration. The iteration occurs because this same select references the Assembly_dt as established in the WITH. The important thing is to select the subpart so that it is all that is joined on the next iteration or recursion.

The depth column provides not only an indication of the iteration in which two rows are joined, but also it can be used as a means to stop the iterations that continue over and over again until no rows are joined or the depth comparison is no longer true. The explain below reveals the nature of this processing.

Explanation

1) First, we do an all-AMPs RETRIEVE step from PLS.parts_vt by way of an all-rows scan with no residual conditions into Spool 3 (all_amps), which is built locally on the AMPs.

The size of Spool 3 is estimated with high confidence to be 1 row. The estimated time for this step is 0.03 seconds.

2) Next, we do an all-AMPs RETRIEVE step from Spool 3 by way of an all-rows scan into Spool 2 (all_amps), which is built locally on the AMPs. The size of Spool 2 is estimated with no confidence to be 1 row. The estimated time for this step is 0.04 seconds.

3) We execute the following steps in parallel.

1) We do an all-AMPs RETRIEVE step from PLS.parts_vt by way of an all-rows scan with a condition of ("NOT (PLS.parts_vt.subpart_no IS NULL)") into Spool 4 (all_amps), which is duplicated on all AMPs. The size of Spool 4 is estimated with no confidence to be

2 rows. The estimated time for this step is 0.01 seconds.

2) We do an all-AMPs RETRIEVE step from Spool 3 (Last Use) by way of an all-rows scan with a condition of ("DEPTH < 3") into Spool 5 (all_amps), which is built locally on the AMPs. The size of Spool 5 is estimated with no confidence to be 1 row. The estimated time for this step is 0.01 seconds.

4) We do an all-AMPs JOIN step from Spool 4 (Last Use) by way of an all-rows scan, which is joined to Spool 5 (Last Use) by way of an all-rows scan. Spool 4 and Spool 5 are joined using a single partition hash join, with a join condition of ("PARTNO = subpart_no"). The result goes into Spool 6 (all_amps), which is built locally on the AMPs. The size of Spool 6 is estimated with no confidence to be 1 row. The estimated time for this step is 0.04 seconds.

5) We do an all-AMPs RETRIEVE step from Spool 6 (Last Use) by way of an all-rows scan into Spool 3 (all_amps), which is built locally on the AMPs. The size of Spool 3 is estimated with no confidence to be 2 rows. The estimated time for this step is 0.04 seconds.

If one or more rows are inserted into spool 3, then go to step 2.

6) We do an all-AMPs SUM step to aggregate from Spool 2 (Last Use) by way of an all-rows scan, and the grouping identifier in field 5. Aggregate Intermediate Results are computed globally, then placed in Spool 9. The size of Spool 9 is estimated with no

confidence to be 46 rows.

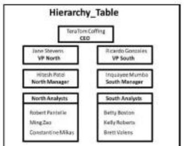
The estimated time for this step is 0.05 seconds.

7) We do an all-AMPs RETRIEVE step from Spool 9 (Last Use) by way of an all-rows scan into Spool 7 (all_amps), which is built locally on the AMPs. Then we do a SORT to order sort key Spool 7 by the in spool field1. The size of Spool 7 is estimated with no confidence to be 46 rows. The estimated time for this step is 0.04 seconds.

8) Finally, we send out an END TRANSACTION step to all AMPs involved in processing the request.

-> The contents of Spool 7 are sent back to the user as the result of statement 1. The total estimated time is 0.24 seconds.

Notice in the above EXPLAIN output that the last line of step 5 (bolded above) indicates that if at least one row is joined in an iteration and inserted into spool 3, the processing will go back to step 2 and loop through the same logic again. This will continue until no new rows are joined and inserted or the optional depth comparison is true. This is the recursive nature of the join. It is normally a good idea to have a depth value checked to prevent a run-away join.



The Hierarchy Table is an Employee_Table.

Each Employee has an Employee_No.

Each Employee has a Mgr_Employee_No except TeraTom, who is the CEO so he has a NULL Value for Mgr_Employee_No).

```
WITH RECURSIVE
 Hierarchy_DT (Emp,Dept,FirstN, LastN,Sal, Pos_Name, Mgr, DEPTH) AS
 (select Employee_No,Dept_No,First_Name, Last_Name,Salary, Position_Name,
Mgr_Employee_No, 0
 FROM   Hierarchy_Table
  where Mgr_Employee_No is NULL
```

UNION ALL

```
SELECT Employee_No, Dept_No, First_Name, Last_Name, Salary, Position_Name,
Mgr_Employee_No, DEPTH+1
 from Hierarchy_DT
     INNER JOIN
     Hierarchy_Table
     on Emp= Mgr_Employee_No
  )
select Emp, Dept, FirstN, LastN, Sal, Pos_Name, Mgr, DEPTH
    FROM Hierarchy_DT
       order by DEPTH, Mgr ;
```

11 Rows Returned

Emp	Dept	FirstN	LastN	Sal	Pos_Name	Mgr	DEPTH

1	100	TeraTom	Coffing	250000.00	CEO	?	0
20	200	Ricardo	Gonzales	175000.00	VP South	1	1
10	100	Jane	Stevens	175000.00	VP North	1	1
100	100	Hitesh	Patel	90000.00	North Manager	10	2
200	200	Inquayee	Mumba	90000.00	Sorth Manager	20	2
5000	100	Robert	Pantelle	70000.00	Analyst North	100	3
3000	100	Ming	Zao	70000.00	Analyst North	100	3
1000	100	Constantine	Mikas	70000.00	Analyst North	100	3
4000	200	Kelly	Roberts	70000.00	Analyst South	200	3
6000	200	Betty	Boston	70000.00	Analyst South	200	3
2000	200	Brett	Valens	70000.00	Analyst South	200	3

The query above uses a Recursive Derived Table. First, we will define the Recursive Derived table and call it Hierarchy_DT. We will populate it from the Hierarchy_Table with one row to start by only loading rows where the Mgr_Employee_No is NULL. TeraTom is the only row loaded (Depth 0) because he doesn't have a manager (NULL).

Then we will continue to UNION ALL the Recursive Derived Table with a JOIN of the Recursive Derived Table to the Hierarchy_Table ON EMP = Mgr_Employee_No. The second pass (DEPTH 1) the two VPs are placed in the Recursive Derived Table because they report to TeraTom. On the third pass (Depth 2) the two Managers are placed in the Recursive Derived Table because they report to the VPs. Then on the fourth pass (Depth 3) the Analysts are placed in the Recursive Derived Table because they report to the respective Managers. Then on the fifth pass the system doesn't find anyone else so it knows it is time to end. At the end we Select * from the Recursive Derived Table!

FROM TABLE UDF Tables

The FROM TABLE UDF Table is also introduced in release V2R6 of Teradata. It facilitates the use of a User Defined Function (UDF) to create and return a row to a FROM in an SQL request.

The table and column definition is created in a user defined function. The TABLE option can only appear once in a FROM and cannot be part of a join operation.

Syntax for using FROM TABLE:

```
FROM TABLE ( <UDF-function-name> ( <parameter-list> ) )
```

The expression list is set up to match with the RETURNS TABLE clause of the CREATE FUNCTION statement. Therefore the number of expressions must match those in the UDF, they are assigned on a positional basis from first to last and they override the names used in the UDF. If the expression list is omitted, the names come from the UDF.

Since this book does not address C or C++ programming there is not an example of this capability. It is here to serve as an introduction only.

Volatile Temporary Tables

Volatile tables were introduced in release V2R3 of Teradata. They have two characteristics in common with derived tables. They are materialized in spool and are unknown in the DD. However, unlike a derived table, a volatile table may be used in more than one SQL statement throughout the life of a session. This feature allows other follow-up queries to utilize the same rows in the temporary table without requiring them to be established again. This ability to use the rows multiple times is their biggest advantage over derived tables.

A volatile table may be dropped manually at any time when it is no longer needed. If it is not dropped manually, it will be dropped automatically at the end of the user session. Through release V2R5 user could materialize up to a maximum of 64 volatile tables at a time. In V2R6 that has increased to 1000. Each volatile table requires its own CREATE statement. Unlike a real table with its definition stored in the DD, the volatile table name and column definitions are stored only in cache memory of the Parsing Engine. Since the rows of a volatile table are stored in spool and do not have DD entries, they do not survive a system restart. That is why they are called volatile.

The syntax to create a volatile table follows:

```
CREATE [ {SET | MULTISET} ] VOLATILE TABLE <table-name> [ , {LOG | NO LOG}
<Teradata-table-options> ]
```

```
( <column-name>    <data-type>
[  , <column-name>    <data-type>
[  , <column-name>    <data-type>  ]  )
[ [ UNIQUE ] PRIMARY INDEX (<column-list>) ]
[ ON COMMIT ( PRESERVE | DELETE )  ROWS  ] ;
```

The **LOG** option indicates the desire for standard transaction logging of "before images" in the transient journal. Without journaling, maintenance activities can be much faster. However, be aware that without journaling, there is no transaction recovery available. LOG is the default, but unlike real tables it can be turned off, by specifying: NO LOG.

The second table option regards the retention of rows that are inserted into a volatile table. The default value is **ON COMMIT DELETE ROWS**. It specifies that at the end of a transaction, the table rows should be deleted. Although this approach seems unusual, it is actually the default required by the ANSI standard. It is appropriate in situations where a table is materialized only to produce rows and the rows are not needed after the transaction completes. Remember, in ANSI mode, all SQL is considered part of a single transaction until it fails or the user does a COMMIT WORK command.

The **ON COMMIT PRESERVE ROWS** option provides the more normal situation where the table rows are kept after the end of the transaction. If the rows are going to be needed for other queries in other transactions, use this option or the table will be empty. Since each SQL request is a transaction in Teradata mode, this is the commonly used option to make rows stay in the volatile table for continued use.

Without DD entries, the following options are NOT available with volatile tables:

- Permanent Journaling
- Referential Integrity
- CHECK constraints
- Column compression
- Column default values
- Column titles
- Named indexes

Volatile tables must have names that are unique within the user's session. They are qualified by the user-id of the session, either explicitly or implicitly. A volatile table cannot exist in a database; it can only materialize in a user's session and area.

The fact that a volatile table exists only to a user's session implies a hidden consequence. No other user may access rows in someone else's volatile table. Furthermore, since it is local to a session, the same user cannot access the rows of their own "volatile table" from another session, only in the original session. Instead, another session must run the same create volatile table command to obtain an instance of it and another SELECT to populate it with the same rows if they are needed in a second session.

Although this might sound bad, it provides greater flexibility. It allows for a situation where the same "table" is used to process different requests by storing completely different rows. On the other hand, it means that a volatile table may not be the best solution when multiple sessions or multiple users need access to the same rows on a frequent basis.

In the original release, volatile tables could not have secondary index definitions because they are in SPOOL. Today, that is no longer the case. A Volatile table can have both USI and NUSI index definitions.

The following examples show how to create, populate, and run queries using a volatile table:

```
CREATE VOLATILE TABLE  Dept_Aggreg_vt , NO LOG
( Dept_no        Integer
  ,Sum_Salary    Decimal(10,2)
  ,Avg_Salary    Decimal(7,2)
  ,Max_Salary    Decimal(7,2)
  ,Min_Salary    Decimal(7,2)
  ,Cnt_Salary    Integer )
ON COMMIT PRESERVE ROWS ;
```

The definition is built in the PE's cache memory. This is the only place that it resides, not in the DD.

The next INSERT/SELECT populates the volatile table created above with one data row per department that has at least one employee in it:

```
INSERT INTO Dept_Aggreg_vt
      SELECT  Dept_no
              ,SUM(Salary)
              , AVG(Salary)
              ,MAX(Salary)
              ,MIN(Salary)
              ,COUNT(Salary)
      FROM  Employee_Table
      GROUP BY  Dept_no ;
```

Now that the volatile table exists in the cache memory of the PE and it contains data rows, it is ready for use in a variety of SQL statements:

```
SELECT * FROM Dept_Aggreg_vt ORDER BY 1;
```

6 Rows Returned

Dept_no	Sum_Salary	Avg_Salary	Max_Salary	Min_Salary	Cnt_Salary
?	32800.50	32800.50	32800.50	32800.50	1
10	64300.00	64300.00	64300.00	64300.00	1
100	48850.00	48850.00	48850.00	48850.00	1
200	89888.88	44944.44	48000.00	41888.88	2
300	40200.00	40200.00	40200.00	40200.00	1
400	145000.00	48333.33	54000.00	36000.00	3

The same rows are still available for another SELECT:

```
SELECT  Department_Name
        ,Avg_Salary
        ,Max_Salary
        ,Min_Salary
FROM   Dept_Aggreg_vt AS VT INNER JOIN   Department_Table D
      ON    VT.dept_no = D.dept_no
WHERE  Cnt_Salary > 1 ;
```

2 Rows Returned

Department_Name	Avg_Salary	Max_Salary	Min_Salary
Research and Development	44944.44	48000.00	41888.88
Customer Support	48333.33	54000.00	36000.00

Whenever a single user needs data rows and they are needed more than once in a session, the volatile table is a better solution than the derived table. Then, as the user logs off, the table definition and spool space are automatically deleted.

Since no DD entry is available for a volatile table, they will not be seen with a HELP USER command. The only way to see how many and which volatile tables exist is to use the following command:

```
HELP VOLATILE TABLE ;
```

1 Row Returned

Session Id	Table Name	Table Id	Protection	Creator Name	Commit Option	Transaction Log
1010	my_vt	10C0C40000	N	MIKEL	P	Y

The main disadvantage of a volatile table is that it must be created via the CREATE VOLATILE TABLE statement every time a new session is established. This situation can be overcome using a global temporary table.

Global Temporary Tables

Global Temporary Tables were also introduced in release V2R3 of Teradata. Their table and column definition is stored in the DD, unlike volatile tables. The first SQL DML statement to access a global temporary table, typically an INSERT/SELECT, materializes the table. They are often called global tables.

Like volatile tables, global tables are local to a session. The materialized instance of the table is not shareable with other sessions. Also like volatile tables, the global table instance may be dropped explicitly at any time or it is dropped automatically at the end of the session. However, the definition remains in the dictionary for future materialized instances of the same table. At the same time, the materialized instance or base definition may be dropped with an explicit DROP command, like any table.

The only privilege required to use a global table is the DML privilege necessary to materialize the table, usually an INSERT/SELECT. Once it is materialized, no other privileges are checked.

A special type of space called "Temporary space" is used for global temporary tables. Like Permanent space, Temporary space is preserved during a system restart and thus, global temporary tables are able to survive a system restart.

These global tables are created using the CREATE GLOBAL TEMPORARY TABLE command. Unlike the volatile table, this CREATE stores the base definition of the table in the DD and is only executed once per database. Like volatile tables, the table defaults are to LOG transactions and ON COMMIT DELETE ROWS. Up to 32 materialized instances of a global temporary table may exist for a single user.

Once the table is accessed by a DML command, such as the INSERT/SELECT, the table is considered materialized and a row is entered into a DD table called DBC.Temptables. An administrator may SELECT from this table to determine the users with global tables materialized and how many global tables exist.

Deleting all rows from a global table does not de-materialize the table. The instance of the table must be dropped or the session must be ended for the definition of the materialized table to be discarded.

The syntax to create a global temporary table follows:

```
CREATE [ {SET | MULTISET} ] GLOBAL TEMPORARY TABLE  <table-name>
  [ { LOG | NO LOG } <Teradata-table-options> ]
( <column-name>    <data-type>
[  , <column-name>   <data-type>  ]
[ [ UNIQUE ] PRIMARY INDEX (<column-list>) ]
[ ON COMMIT { PRESERVE | DELETE }  ROWS  ]
```

GLOBAL Temporary Table Examples

This series of commands show how to create, insert, and select from a global temporary table:

```
CREATE GLOBAL TEMPORARY TABLE  Dept_Aggreg_gt
 ( Dept_no          Integer
  ,Sum_Salary       Decimal(10,2)
  ,Avg_Salary       Decimal(7,2)
  ,Max_Salary       Decimal(7,2)
  ,Min_Salary       Decimal(7,2)
  ,Cnt_Salary       Integer )
ON COMMIT PRESERVE ROWS ;
```

The next INSERT will create one data row per department that has at least one employee in it:

```
INSERT INTO Dept_Aggreg_gt
      SELECT  Dept_no ,SUM(Salary) , AVG(Salary) ,MAX(Salary) ,MIN(Salary)
              ,COUNT(Salary)
      FROM  Employee_Table  GROUP BY  Dept_no ;
```

Now that the global temporary table exists in the DD and it contains data rows, it is ready for use in a variety of SQL statements like the following:

```
SELECT * FROM Dept_Aggreg_gt
ORDER BY 1;
```

6 Rows Returned

Dept_no	Sum_Salary	Avg_Salary	Max_Salary	Min_Salary	Cnt_Salary
?	32800.50	32800.50	32800.50	32800.50	1

10	64300.00	64300.00	64300.00	64300.00	1
100	48850.00	48850.00	48850.00	48850.00	1
200	89888.88	44944.44	48000.00	41888.88	2
300	40200.00	40200.00	40200.00	40200.00	1
400	145000.00	48333.33	54000.00	36000.00	3

It can immediately be used by other SELECT operations:

```
SELECT   Department_Name
        ,Avg_Salary
        ,Max_Salary
        ,Min_Salary
FROM Dept_Aggreg_gt AS GT  INNER JOIN   Department_Table  D
ON GT.dept_no = D.dept_no
WHERE Cnt_Salary > 1;
```

2 Rows Returned

Department_Name	Avg_Salary	Max_Salary	Min_Salary
Research and Development	44944.44	48000.00	41888.88
Customer Support	48333.33	54000.00	36000.00

At this point, it is probably obvious that these examples are the same as those used for the volatile table except for the fact that the table name ends with "gt" instead of "vt." Volatile tables and global temporary tables are very much interchangeable from the user perspective. The biggest advantage to using the global temporary table lies in the fact that the table never needs to be created a second time. All the user needs to do is reference it with an INSERT/SELECT and it is automatically materialized with rows.

Therefore, when multiple users need the same definition, it is better to store it one time and give all users the INSERT privilege on it. It is the standard definition available to all users without requiring each user to run a CREATE statement and overcomes the main disadvantage of a volatile table. However, no user can access or disturb rows belonging to another user. They can only access their own rows due to each user session owning a different instance of the table.

Since the global temporary table's definition is stored in the DD, it may be altered using the ALTER command. It can change any attributes of the table, like real tables. Additionally, for extra flexibility, a materialized instance of the table may be altered without affecting the base definition or other user's materialized instance. Talk about flexibility.

This advantage means that a user is not restricted to having an identical definition as all other users. By using the ALTER TEMPORARY TABLE statement, the user can fine-tune the table for their specific needs, session by session.

Since a global temporary table can be altered and is not in spool space, this means that within an instance, it can take advantage of the following operations:

- Add / Drop columns
- Add / Drop attributes
- Create / Drop indices
- Collect Statistics

As an example, if someone did not wish to use the LOG option for his or her instance, the next ALTER could be used:

```
ALTER TEMPORARY TABLE Dept_Aggreg_gt NO LOG;
```

Therefore, care should be taken to insure that not all users have ALTER privileges on the base table definition in the DD. Otherwise, accidentally omitting the word "temporary" alters the base definition and no one has the LOG option as seen below:

```
ALTER TABLE Dept_Aggreg_gt NO LOG;
```

Likewise, the same consideration should be used when defining and collecting Statistics on the stored definition versus the materialized instance. The following defines which statistics to collect on the table definition:

```
COLLECT STATISTICS ON Dept_Aggreg_gt index (Dept_no);
```

However, when this is executed there are no rows in the table and therefore no rows to evaluate and no statistics to store. So, why bother? The reason is that once an instance is materialized all a user needs to do is collect statistics at the table level after inserting their rows into their temporary instance of the table.

The following COLLECT specifies the importance of the word TEMPORARY to denote the instance and not the base definition:

```
COLLECT TEMPORARY STATISTICS on Dept_Aggreg_gt;
```

The above statement collects all statistics for rows in the temporary table, as defined by the base table. However, a user might wish to collect statistics on a column not originally defined for the table, such as Max_Salary. To accomplish this collection operation, the user could execute the next statement:

```
COLLECT TEMPORARY STATISTICS on Dept_Aggreg_gt COLUMN Max_Salary;
```

As a reminder, each instance can only be accessed by a single user and furthermore, only within a single session for that user. Like the volatile table, the same user cannot access rows from their own temporary table from a different session.

Also like a volatile table, a global table releases its temporary space and the instance when the user logs off. If the user wishes to manually drop the instance, use the following command:

```
DROP TEMPORARY TABLE Dept_Aggreg_gt;
```

Again, the word TEMPORARY is very important because without it:

```
DROP TABLE Dept_Aggreg_gt;
```

Will drop the base definition and cause problems for other users. Privileges should be established to prevent a user from accidentally dropping a global table definition.

With that being said, there might come a time when it is desired to drop the base definition. If the above DROP TABLE is executed, it will work unless a user has a materialized instance. One materialized instance is enough to cause the statement to fail. As an alternative, an ALL option can be added, as seen in the next statement, in an attempt to drop the definition:

```
DROP TABLE Dept_Aggreg_gt ALL;
```

This works as long as a user is not in the middle of a transaction. Otherwise, the only option is to wait until the user's transaction completes and then execute the DROP again.

The above format for a Global table indicates the ability to define a primary index as either unique or non-unique. Additionally, since the definition is in the data dictionary, placing a UNIQUE constraint on one or more columns would also make the first unique column a UPI. This logic is the same for a real table.

General Practices for Temporary use Tables

The following are guidelines to consider when determining which type of "temporary" table to use. Most of the criteria are based on the number of users needing access to the data. The second issue is related to the frequency of use.

Multiple user access to a table:	Temporal or interim table (short-term real table)
Single user access to a table,	
- Single ad hoc SQL use table:	Derived table
- Multiple SQL use table:	Volatile or Global temporary table
- Standardized, multiple SQL use table:	Global temporary table

Use these guidelines to decide which type of table to use based on the needs of the user.