CONTENTS:

The Quantile Function

The Quantile Function

Overview

"There is flattery in friendship."
- William Shakespeare

The Quantile Function and Syntax

The syntax for the Quantile function:

```
QUANTILE (<partitions>, <column-name> ,<sort-key> [DESC | ASC])
[QUALIFY QUANTILE (<column-name>) {< | > | = | <= | >=} <number-of-rows>]
```

A Quantile is used to divide rows into a number of categories or grouping of roughly the same number of rows in each group. The percentile is the QUANTILE most commonly used in business. This means that the request is based on a value of 100 for the number of partitions. It is also possible to have quartiles (based on 4), tertiles (based on 3) and deciles (based on 10).

By default, both the QUANTILE column and the QUANTILE value itself will be output in ascending sequence. As in some cases, the ORDER BY clause may be used to reorder the output for display. Here, the order of the output does not change the meaning of the output, unlike a summation where the values are being added together and all need to appear in the proper sequence.

A Quantile Example

This example determines the percentile for every row in the Sales table based on the daily sales amount, and sorts it into sequence by the value being categorized, which is Daily_Sales here.

```
SELECT Product_ID, Sale_Date, Daily_Sales
      ,QUANTILE (100, Daily_Sales ) AS "Quantile"
FROM   Sales_Table
WHERE  Product_ID < 3000
AND    Sale_Date > 1000930 ;
```

Product_ID	Sale_Date	Daily_Sales	Quantile
1000	10/02/2000	32800.50	0
2000	10/04/2000	32800.50	0
2000	10/02/2000	36021.93	25
1000	10/01/2000	40200.43	37
2000	10/03/2000	43200.18	50
1000	10/04/2000	54553.10	62
2000	10/01/2000	54850.29	75
1000	10/03/2000	64300.00	87

Notice that the amount of 32800.50 in the first two rows has the same percentile value. They are the same value, and will, therefore, be put into the same partition.

A Quantile Example using DESC Mode

```
SELECT Product_ID ,Sale_Date ,Daily_Sales
      ,QUANTILE(100, Daily_Sales , Sale_Date DESC) AS "Quantile"
FROM  Sales_Table
WHERE Product_ID < 3000
AND   Sale_Date >= 1000930 ;
```

Product_ID	Sale_Date	Daily_Sales	Quantile
1000	10/02/2000	32800.50	0
2000	10/04/2000	32800.50	10
1000	09/30/2000	36000.07	20
2000	10/02/2000	36021.93	30
1000	10/01/2000	40200.43	40

2000	10/03/2000	43200.18	50
2000	09/30/2000	49850.03	60
1000	10/04/2000	54553.10	70
2000	10/01/2000	54850.29	80
1000	10/03/2000	64300.00	90

Notice the first two rows. This is because the Sale date DESC, impacts the first two rows. Why? Since these rows have the same value, it uses the Sale_Date column as a tiebreaker for the sequencing and makes them different from each other. Hence, they are assigned to different values in different partitions.

QUALIFY to find Products in the top Partitions

This example uses a QUALIFY to show only products that sell in the top 60 Percentile

```
SELECT Product_ID, Sale_Date, Daily_Sales
      ,QUANTILE(100, Daily_Sales, Sale_Date ) as "Percentile"
FROM   Sales_Table
QUALIFY "Percentile" >= 60 ;
```

Product_ID	Sale_Date	Daily_Sales	Percentile
2000	09/29/2000	48000.00	61
1000	09/28/2000	48850.40	66
2000	09/30/2000	49850.03	71
1000	09/29/2000	54500.22	76
1000	10/04/2000	54553.10	80
2000	10/01/2000	54850.29	85
3000	09/28/2000	61301.77	90
1000	10/03/2000	64300.00	95

Like the aggregate functions, OLAP functions must read all required rows before performing their operation. Therefore, the WHERE clause cannot be used. Where the aggregates use HAVING, the OLAP functions use QUALIFY. The QUALIFY evaluates the result to determine which ones to return.

QUALIFY to find Products in the top Partitions Sorted DESC

```
SELECT  Product_ID, Sale_Date, Daily_Sales
       ,QUANTILE (100, Daily_Sales, Sale_Date ) as "Percentile"
FROM    Sales_Table
QUALIFY  "Percentile" >= 70
ORDER BY "percentile" DESC ;
```

Product_ID	Sale_Date	Daily_Sales	Percentile
1000	10/03/2000	64300.00	95
3000	09/28/2000	61301.77	90
2000	10/01/2000	54850.29	85
1000	10/04/2000	54553.10	80
1000	09/29/2000	54500.22	76
2000	09/30/2000	49850.03	71

The ORDER BY changes the sequence of the rows being listed, not the meaning of the percentile. The above functions both determined that the highest number in the column is the highest percentile. The data value sequence ascends as the percentile ascends, or descends as the percentile descends. When the sort in the QUANTILE function is changed to ASC, the data value sequence changes to ascend as the percentile descends. The sequence of the percentile does not change. But, the data value sequence is changed to ascend (ASC) instead of the default, which is to descend (DESC).

QUALIFY to find Products in the top Partitions Sorted ASC

```
SELECT  Product_ID ,Sale_Date, Daily_Sales
       ,QUANTILE (100, Daily_Sales ASC, Sale_Date) as "Percentile"
FROM    Sales_Table
QUALIFY"Percentile" >=70 ;
```

Product_ID	Sale_Date	Daily_Sales	Percentile
1000	10/02/2000	32800.50	71
2000	10/04/2000	32800.50	76
3000	10/01/2000	28000.00	80
3000	10/03/2000	21553.79	85
3000	10/02/2000	19678.94	90
3000	10/04/2000	15675.33	95

The example above uses the ASC to cause the data values to go contradictory to the percentile.

QUALIFY to find Products in top Partitions with Tiebreaker

```
SELECT  Product_ID, Sale_Date, Daily_Sales
       ,QUANTILE (100, Daily_Sales ASC, Sale_Date ASC) as "Percentile"
FROM    Sales_Table
QUALIFY"Percentile" >= 70 ;
```

Product_ID	Sale_Date	Daily_Sales	Percentile
2000	10/04/2000	32800.50	71
1000	10/02/2000	32800.50	76
3000	10/01/2000	28000.00	80
3000	10/03/2000	21553.79	85
3000	10/02/2000	19678.94	90
3000	10/04/2000	15675.33	95

The next SELECT modifies the above query to incorporate the sale date as a tiebreaker and reverse the ordering for the two rows with sales of $32,800.50.

Using Tertiles (Partitions of Four)

```
SELECT Product_ID, Sale_Date, Daily_Sales
      ,QUANTILE(4, Daily_Sales , Sale_Date ) AS "Quartiles"
FROM  Sales_Table
WHERE Product_ID in (1000, 2000) ;
```

Product_ID	Sale_Date	Daily_Sales	Quartiles
1000	10/02/2000	32800.50	0
2000	10/04/2000	32800.50	0
1000	09/30/2000	36000.07	0
2000	10/02/2000	36021.93	0
1000	10/01/2000	40200.43	1
2000	09/28/2000	41888.88	1
2000	10/03/2000	43200.18	1
2000	09/29/2000	48000.00	2
1000	09/28/2000	48850.40	2
2000	09/30/2000	49850.03	2
1000	09/29/2000	54500.22	2
1000	10/04/2000	54553.10	3
2000	10/01/2000	54850.29	3
1000	10/03/2000	64300.00	3

Instead of 100, the example above uses a quartile (QUANTILE based on 4 partitions).

How Quantile Works

```
SELECT Product_ID, Sale_Date, Daily_Sales
      ,QUANTILE (4, Daily_Sales , Sale_Date ) AS "Quartiles"
FROM   Sales_Table WHERE Product_ID = 1000;
```

Product_ID	Sale_Date	Daily_Sales	Quartiles
1000	10/02/2000	32800.50	0
1000	09/30/2000	36000.07	0
1000	10/01/2000	40200.43	1

1000	09/28/2000	48850.40	1
1000	09/29/2000	54500.22	2
1000	10/04/2000	54553.10	2
1000	10/03/2000	64300.00	3

Assigning a different value to the <partitions> indicator of the QUANTILE function changes the number of partitions established. Each Quantile partition is assigned a number starting at 0, increasing to a value that is one less than the partition number specified. So, with a Quantile of 4, the partitions are 0 through 3 and for 10, the partitions are assigned 0 through 9. Then, all the rows are distributed as evenly as possible into each partition from highest to lowest values. Normally, extra rows with the lowest value begin back in the lowest numbered partitions.

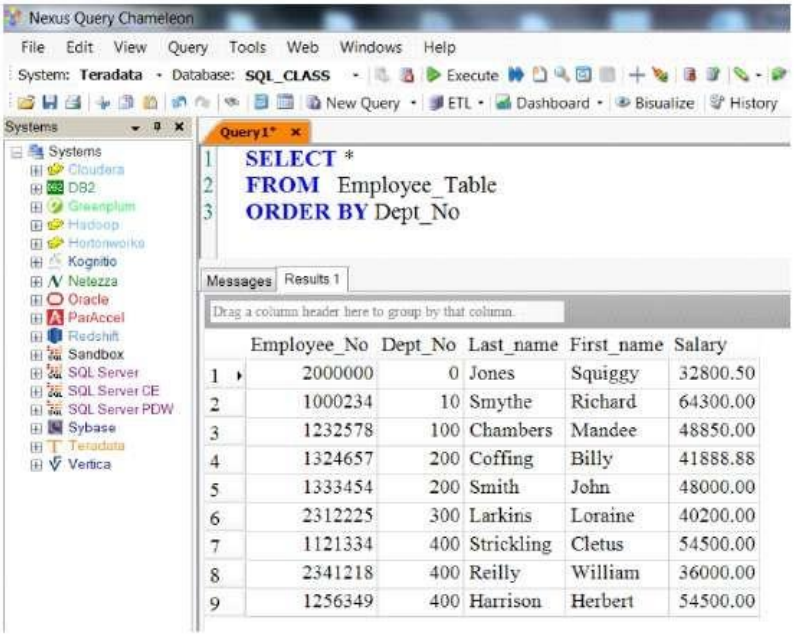
Top SQL Commands Cheat Sheet

Top SQL Commands Cheat Sheet

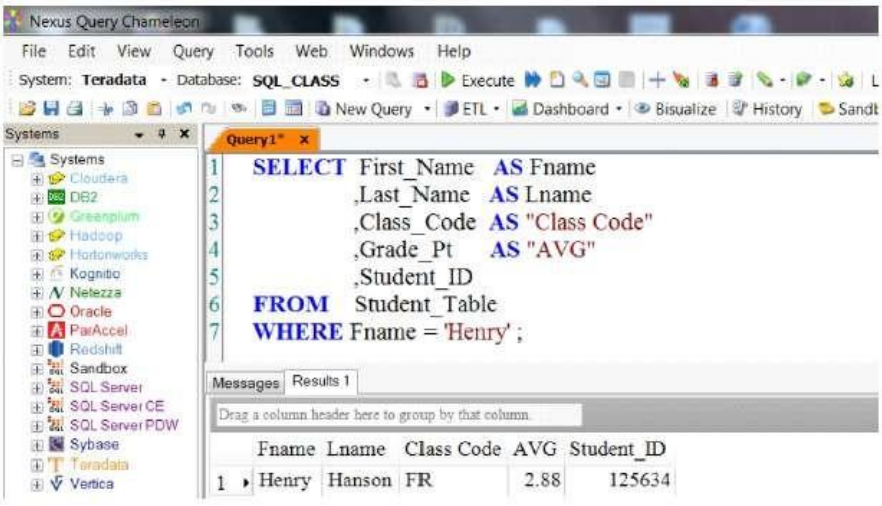
Overview

"You miss 100 percent of the shots you never take."
- Wayne Gretzky

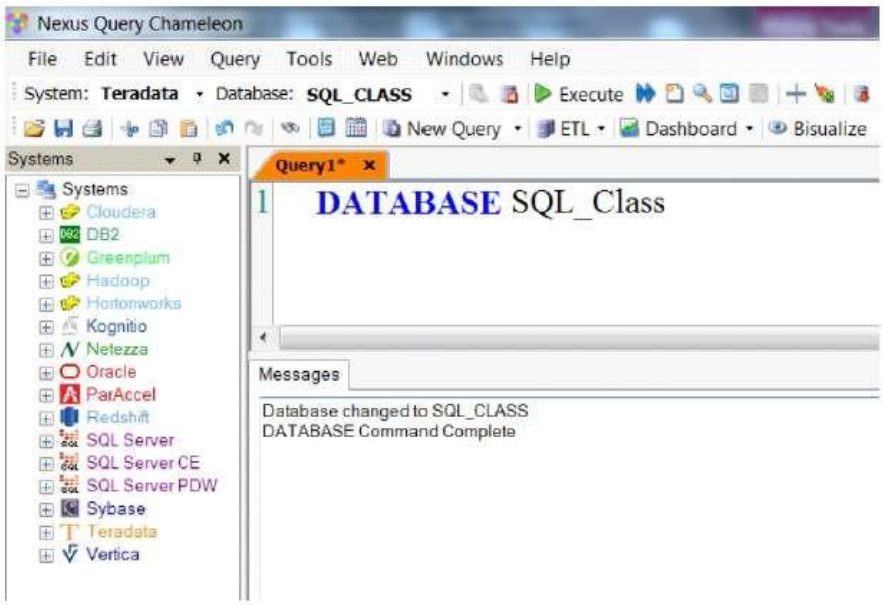
SELECT All Columns from a Table and Sort



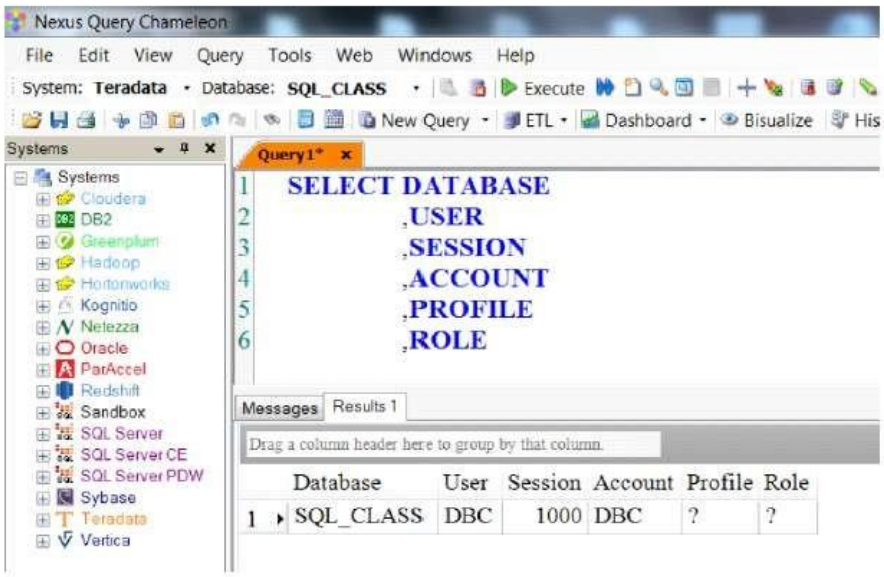
Select Specific Columns and Limiting the Rows



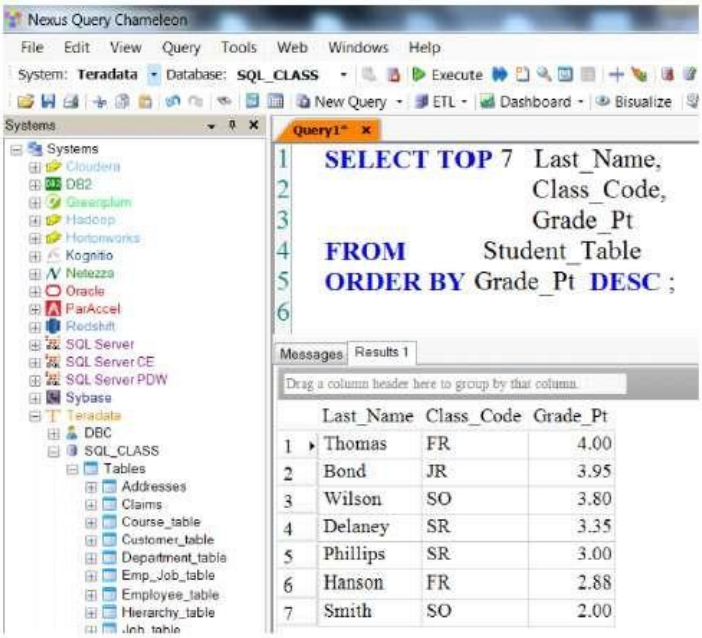
Changing your Default Database



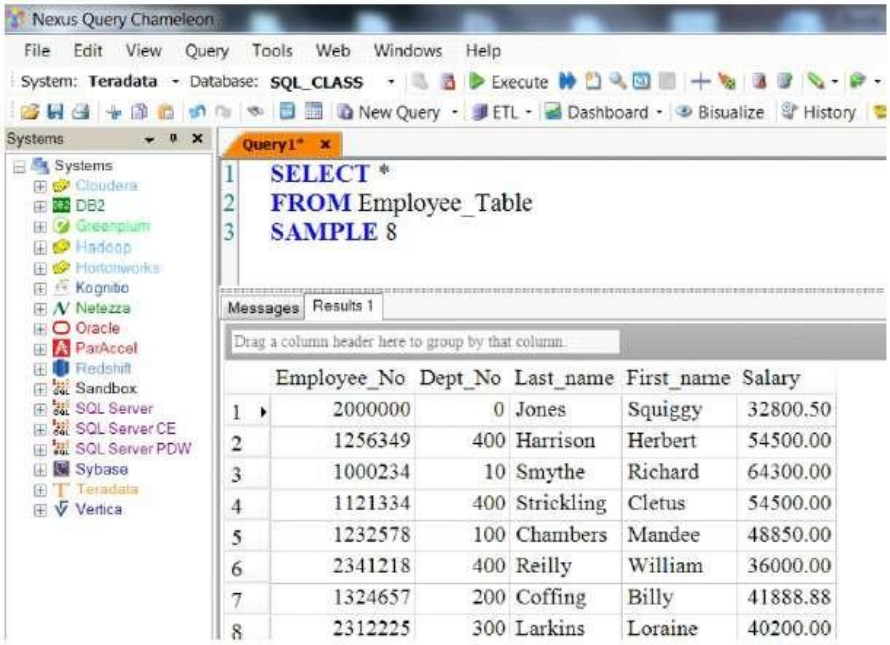
Keywords that describe you



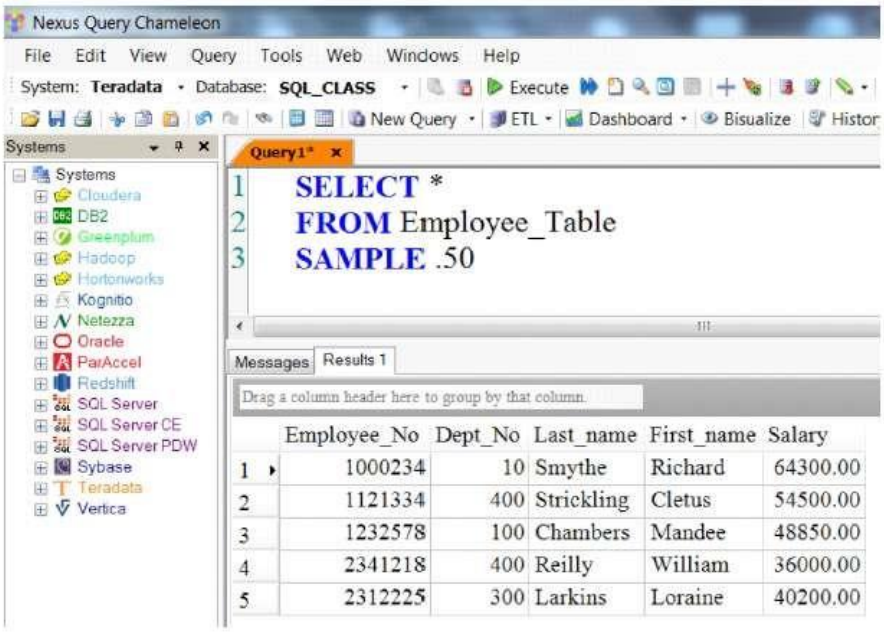
Select TOP Rows in a Rank Order



A Sample number of rows

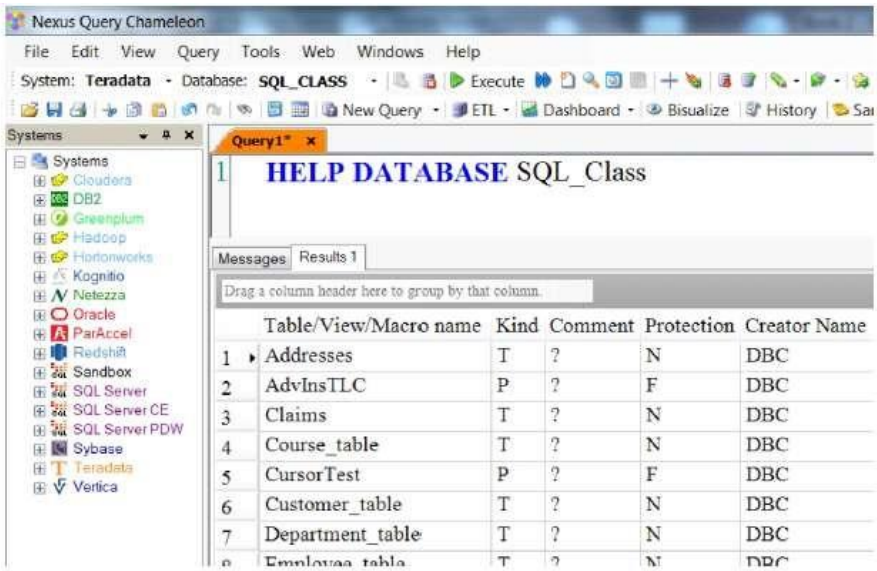


Getting a Sample Percentage of rows



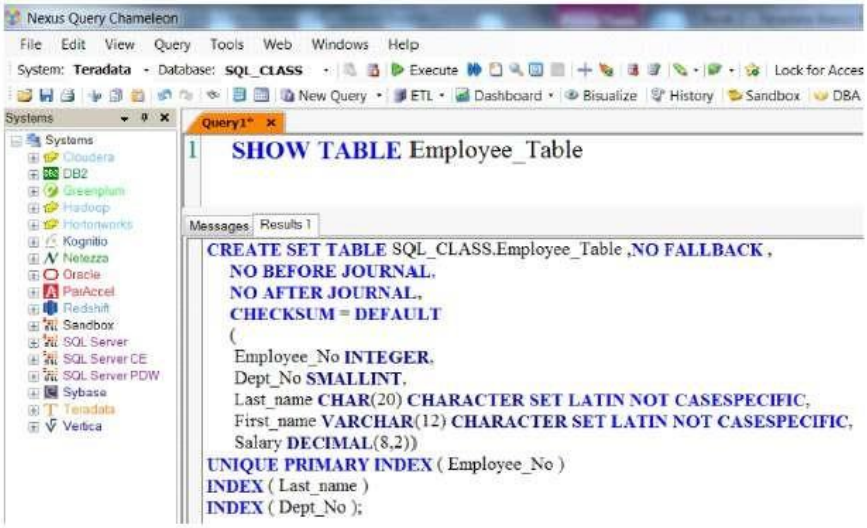
	Employee_No	Dept_No	Last_name	First_name	Salary
1	1000234	10	Smythe	Richard	64300.00
2	1121334	400	Strickling	Cletus	54500.00
3	1232578	100	Chambers	Mandee	48850.00
4	2341218	400	Reilly	William	36000.00
5	2312225	300	Larkins	Loraine	40200.00

Find Information about a Database

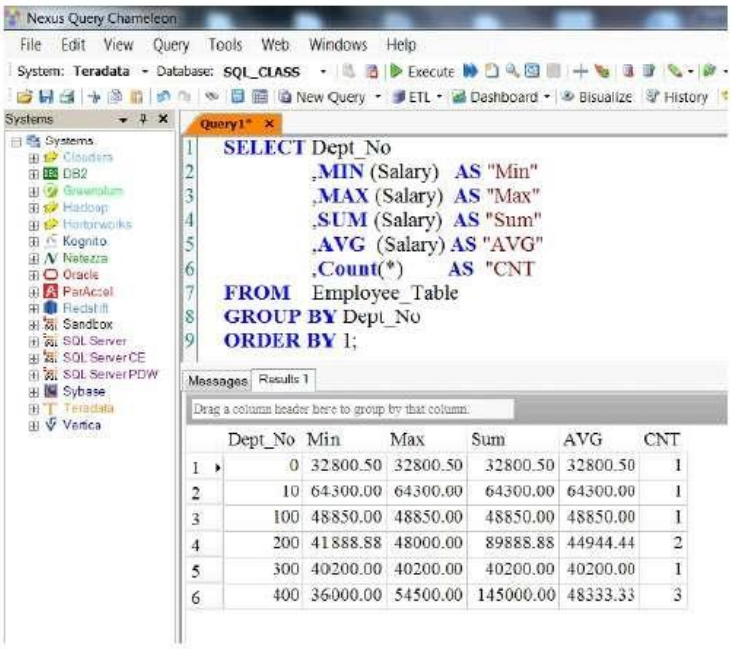


	Table/View/Macro name	Kind	Comment	Protection	Creator Name
1	Addresses	T	?	N	DBC
2	AdvInsTLC	P	?	F	DBC
3	Claims	T	?	N	DBC
4	Course_table	T	?	N	DBC
5	CursorTest	P	?	F	DBC
6	Customer_table	T	?	N	DBC
7	Department_table	T	?	N	DBC
8	Employee_table	T	?	N	DBC

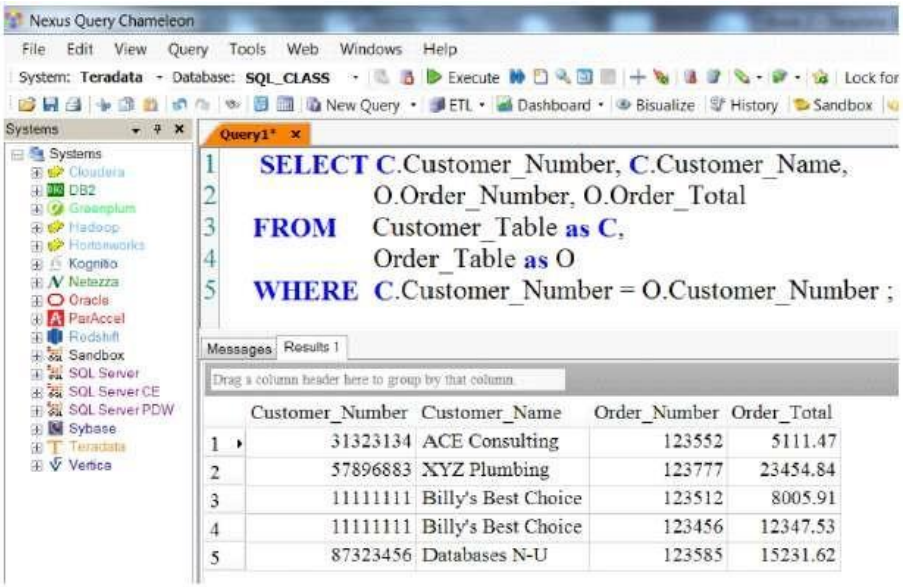
Find information about a Table



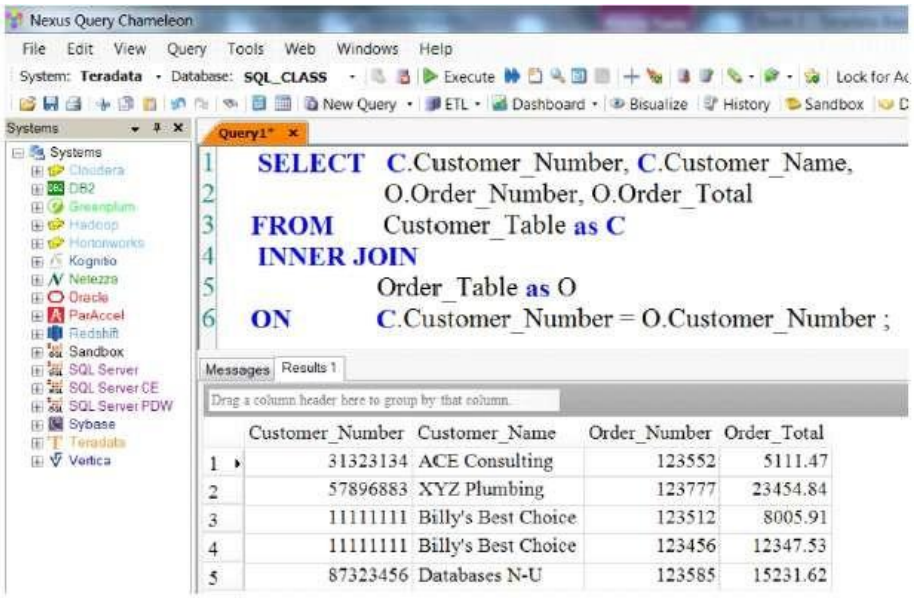
Using Aggregates



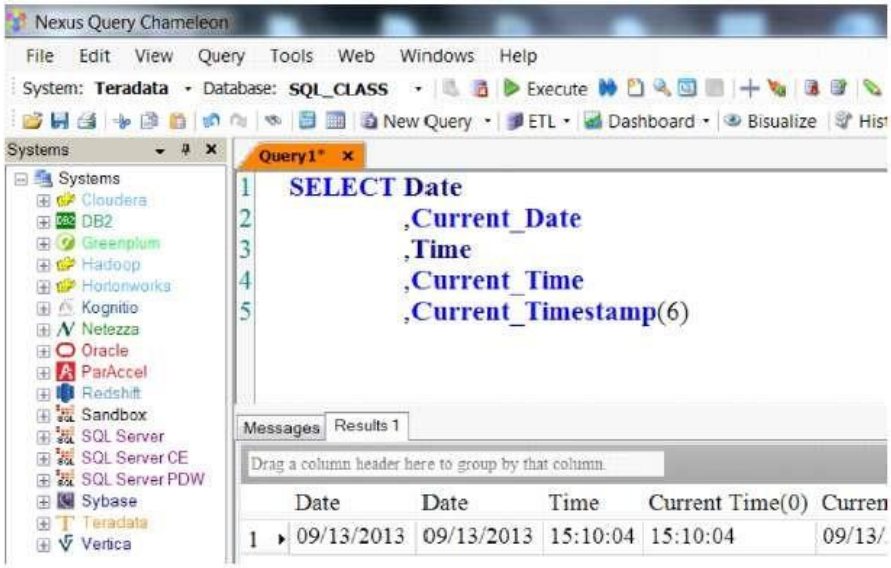
Performing a Join



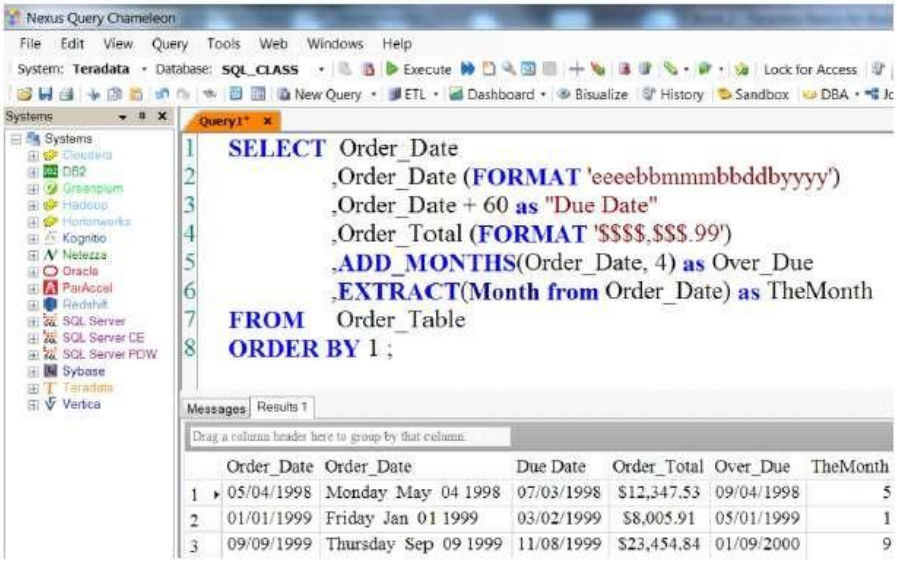
Performing a Join using ANSI Syntax



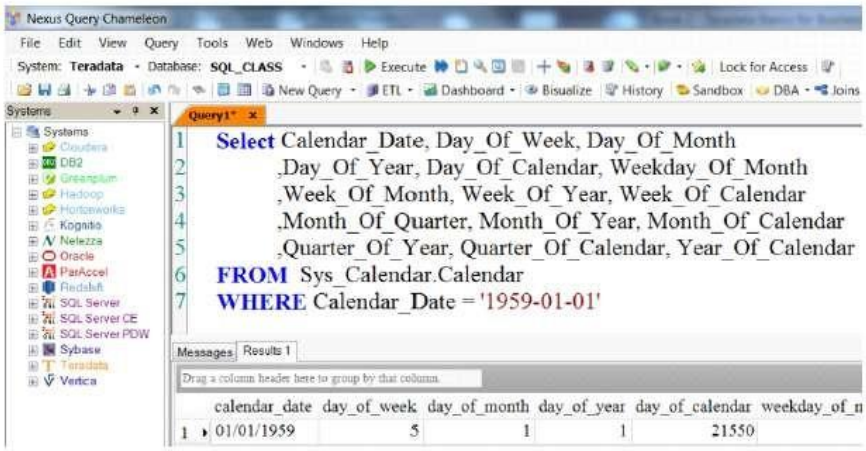
Using Date, Time and Timestamp



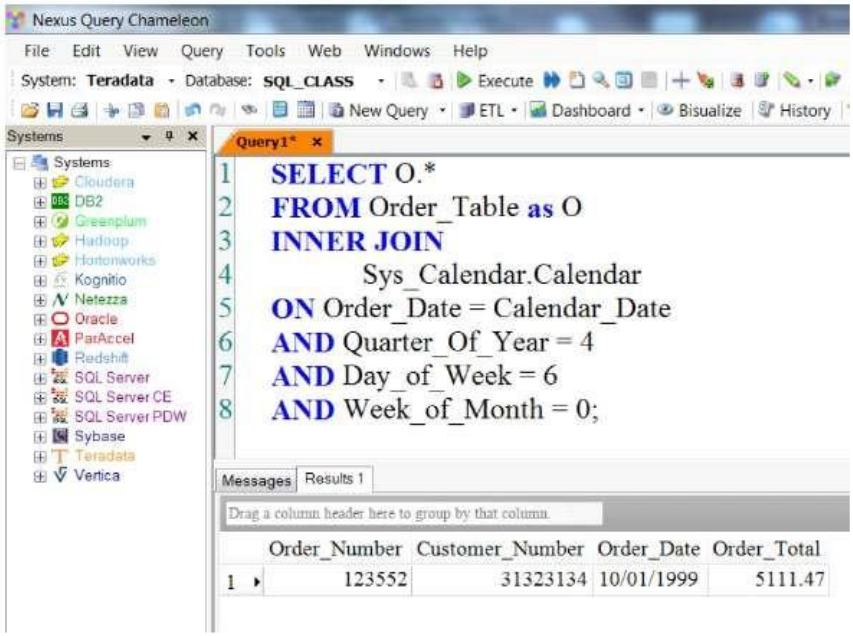
Using Date Functions



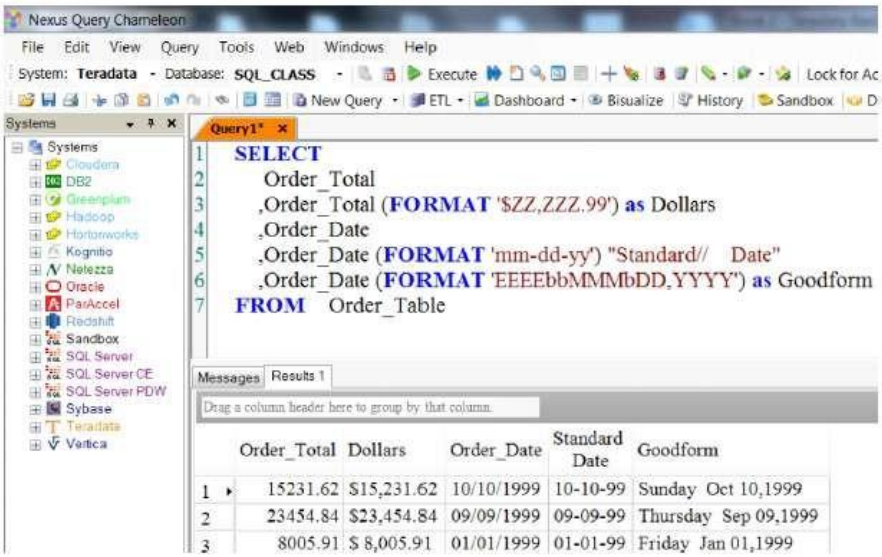
Using the System Calendar



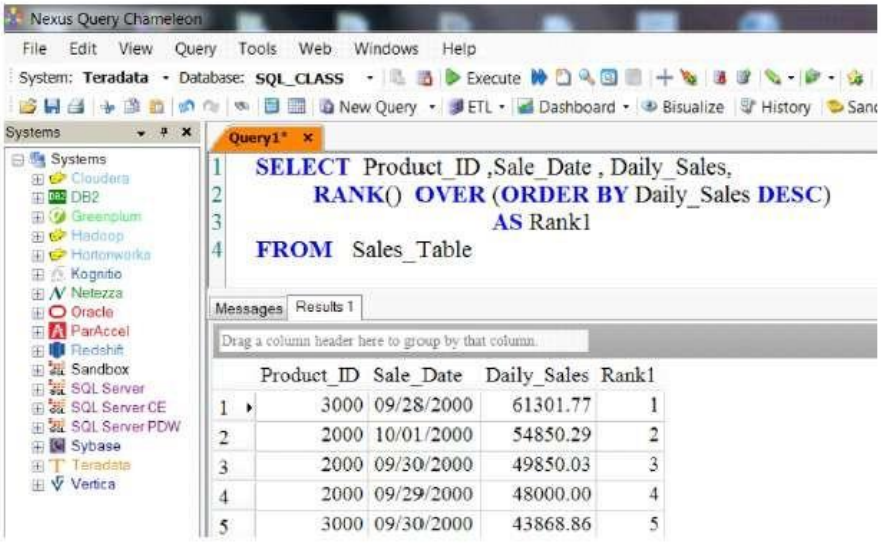
Using the System Calendar in a Query



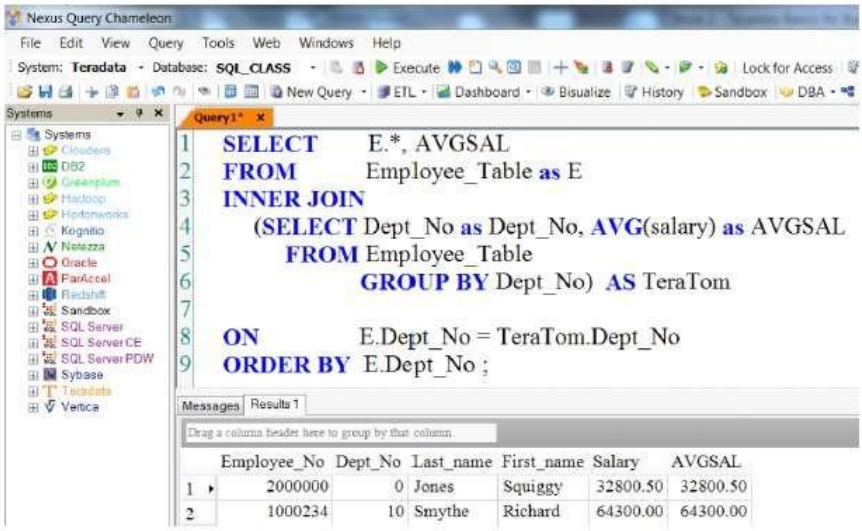
Formatting Data



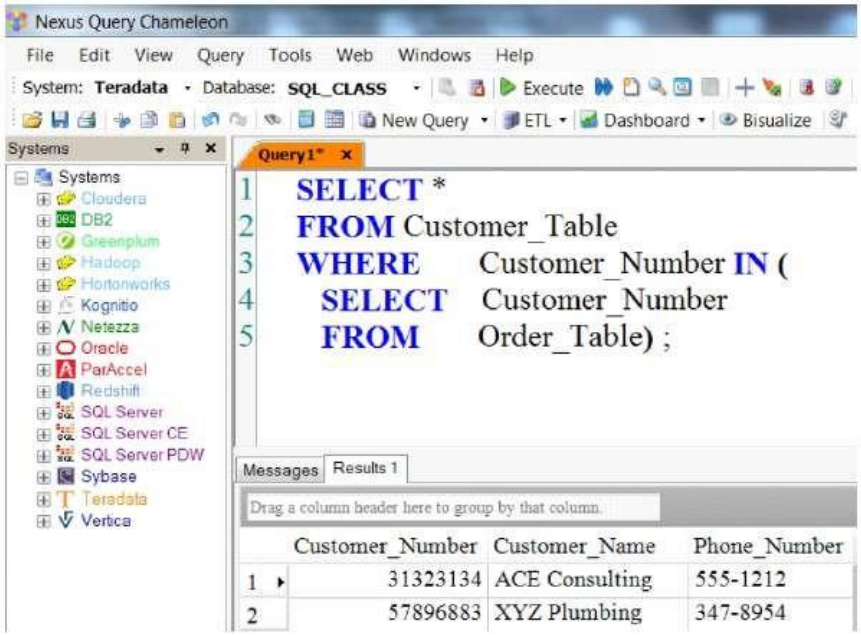
Using Rank



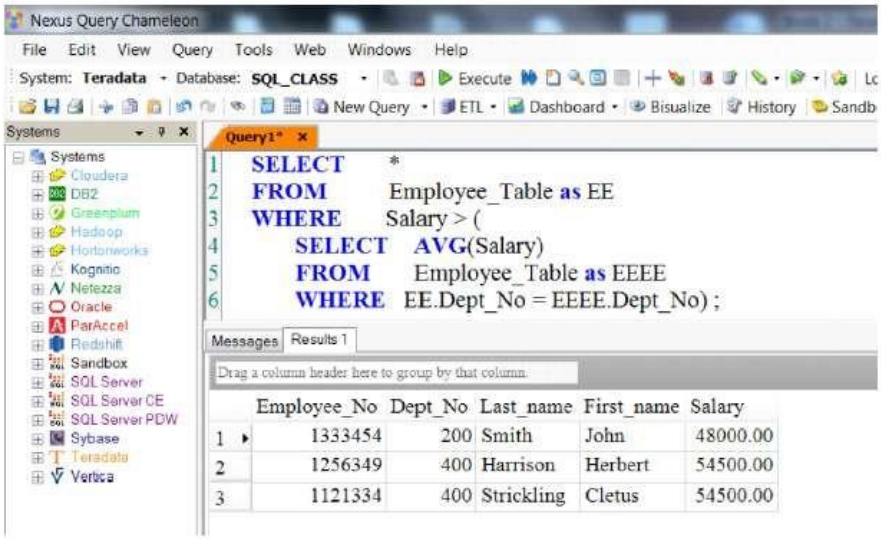
Using a Derived Table



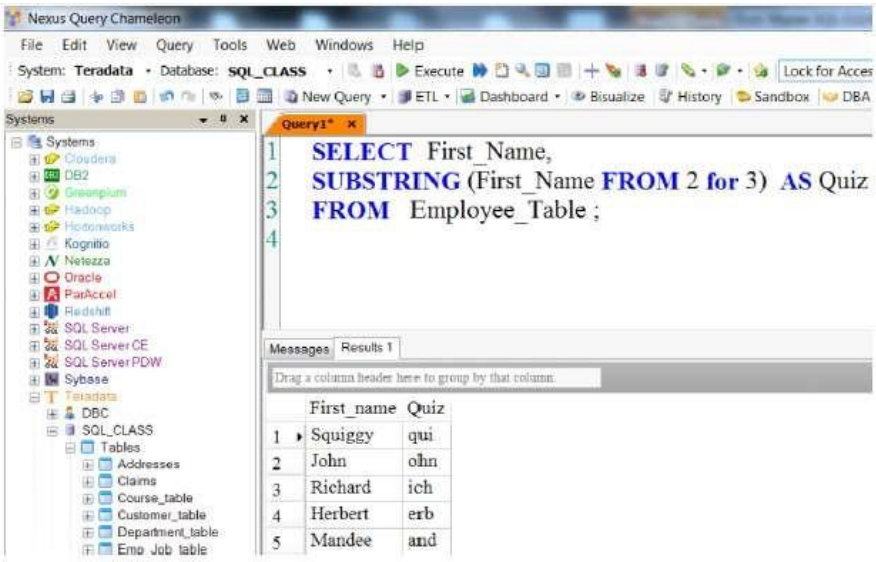
Using a Subquery



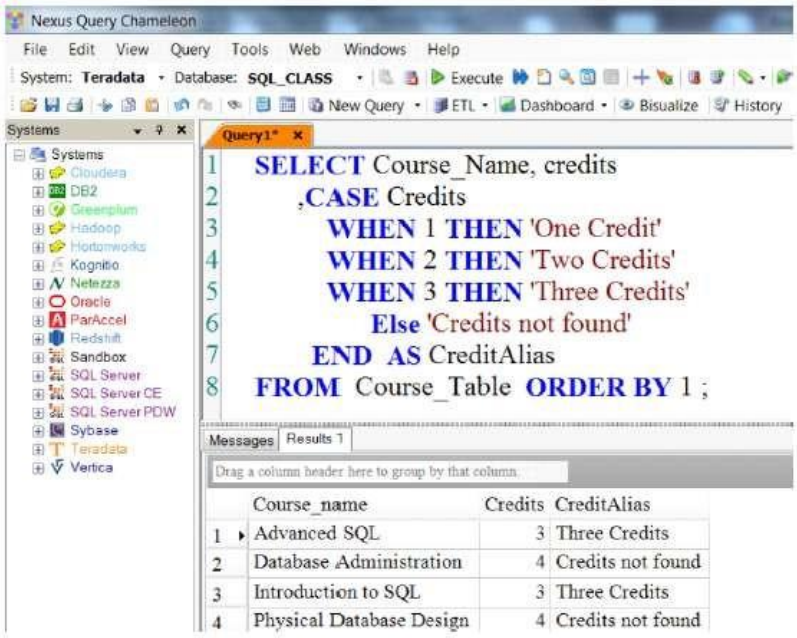
Correlated Subquery



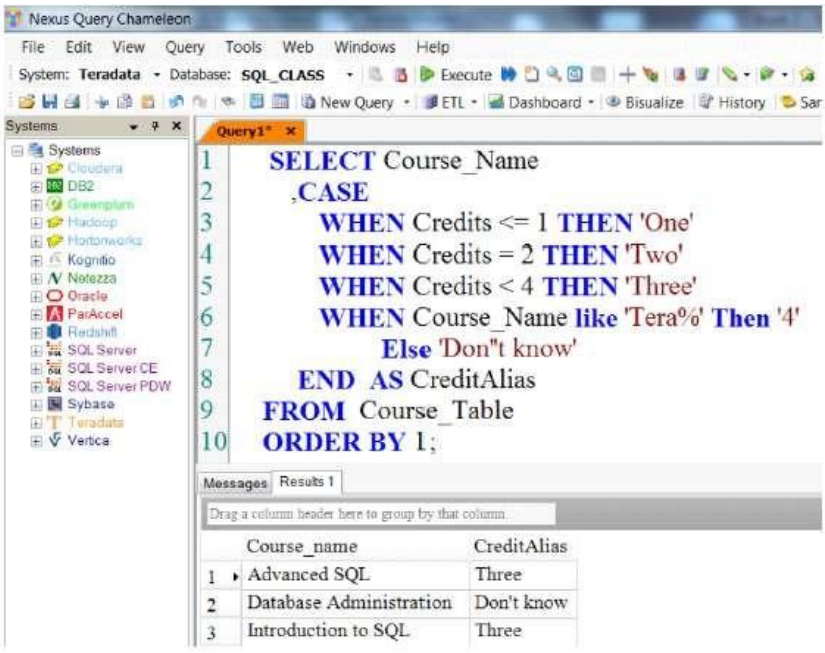
Using Substring



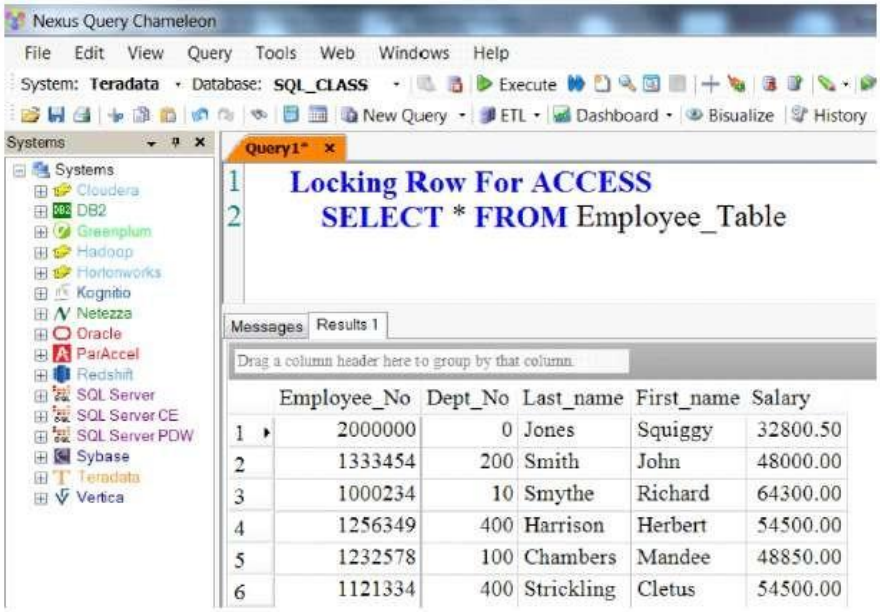
Basic CASE Statement



Advanced CASE Statement

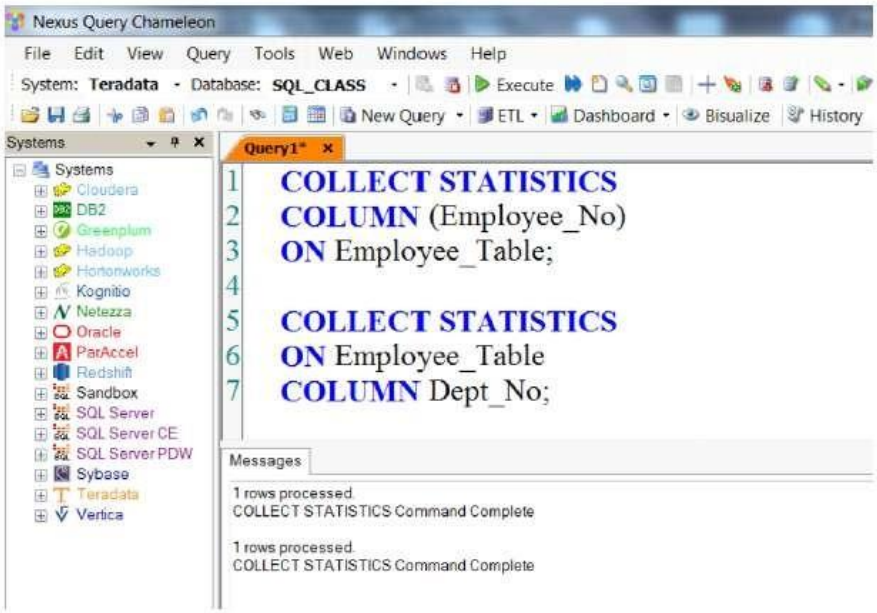


Using an Access Lock in your SQL

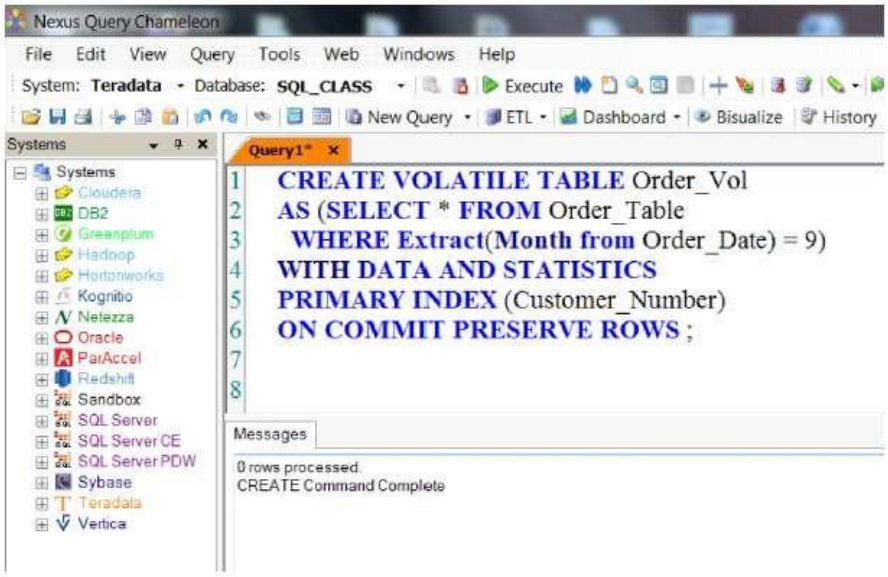


	Employee_No	Dept_No	Last_name	First_name	Salary
1	2000000	0	Jones	Squiggy	32800.50
2	1333454	200	Smith	John	48000.00
3	1000234	10	Smythe	Richard	64300.00
4	1256349	400	Harrison	Herbert	54500.00
5	1232578	100	Chambers	Mandee	48850.00
6	1121334	400	Strickling	Cletus	54500.00

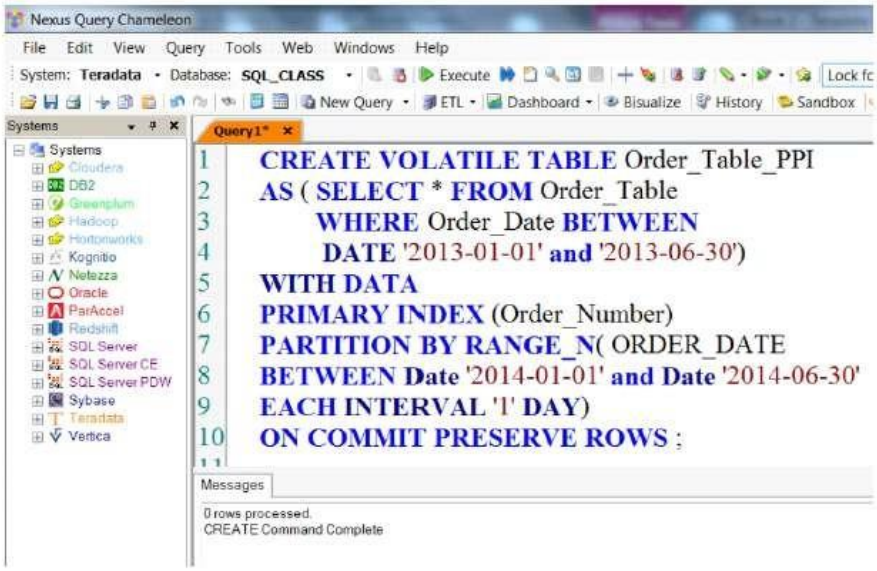
Collect Statistics



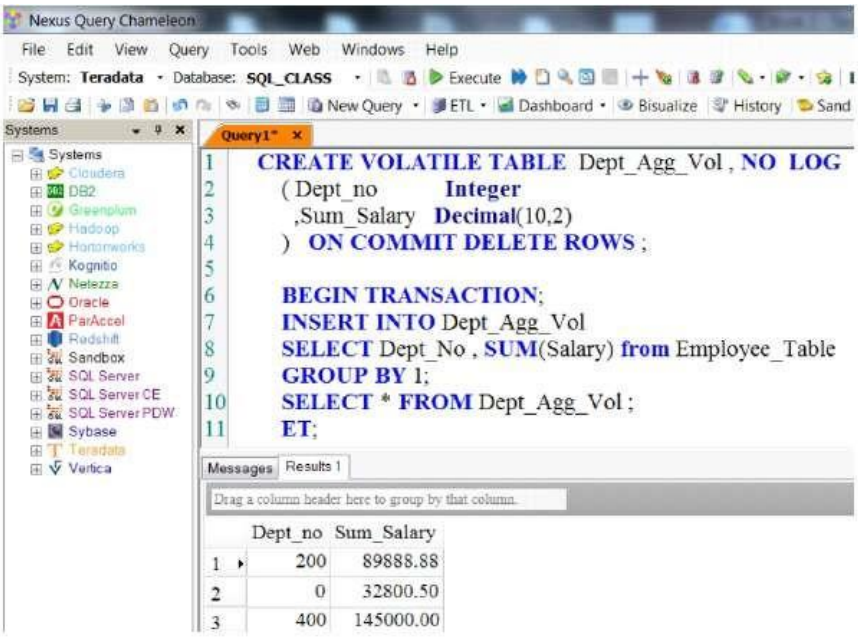
CREATING a Volatile Table with a Primary Index



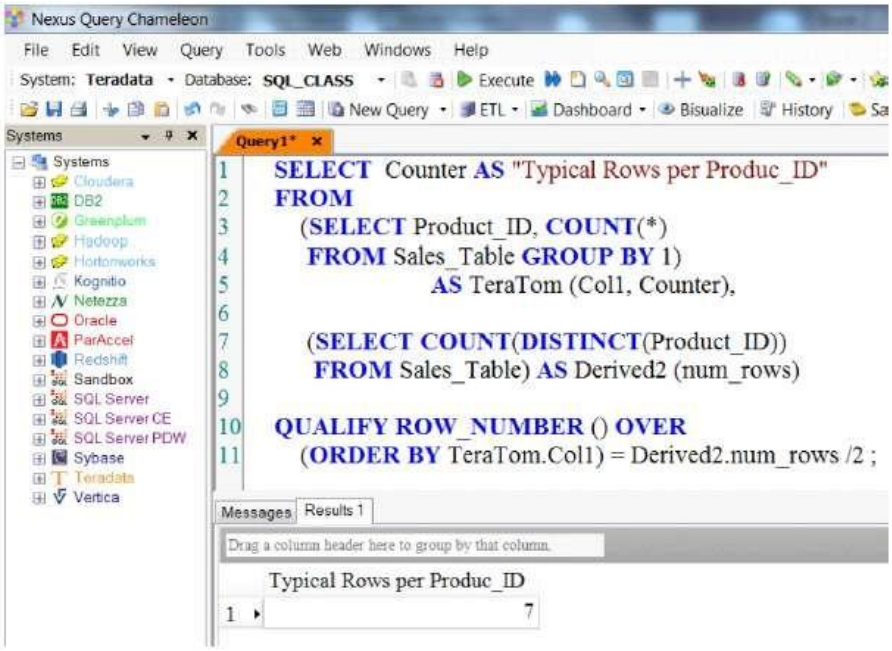
CREATING a Volatile Table that is Partitioned (PPI)



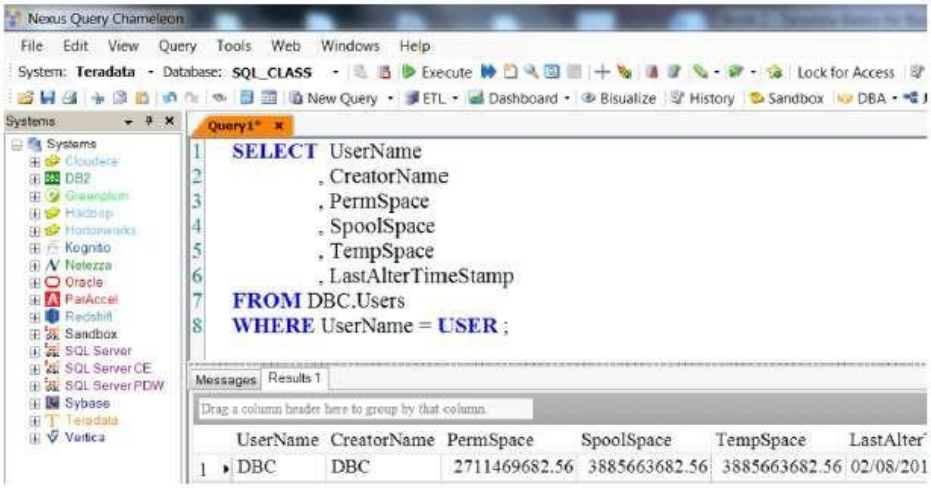
CREATING a Volatile Table that is deleted after the Query



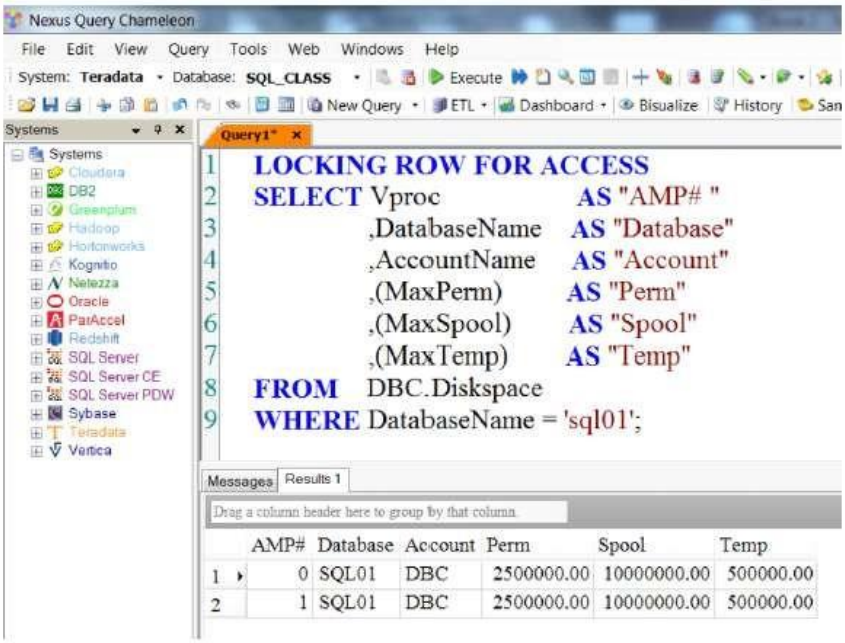
Finding the Typical Rows per Value for specific column



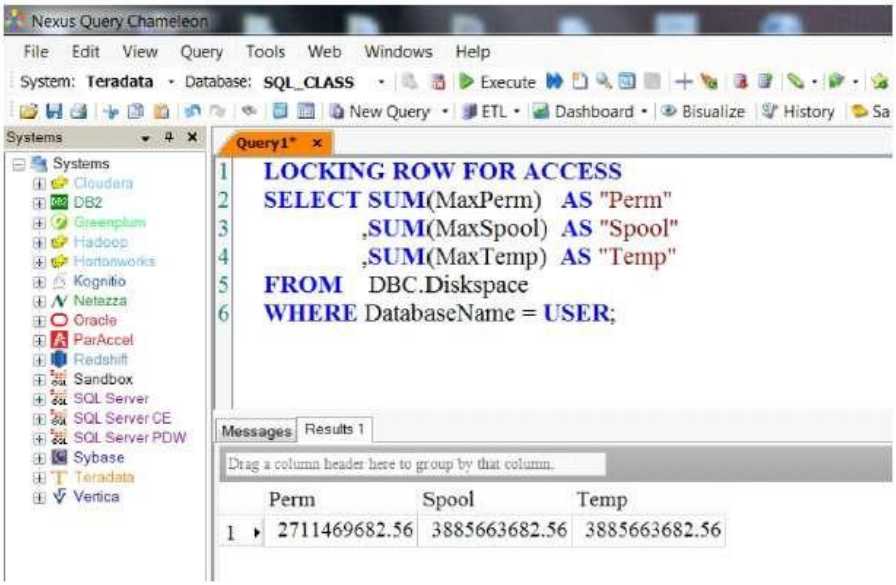
Finding out how much Space you have



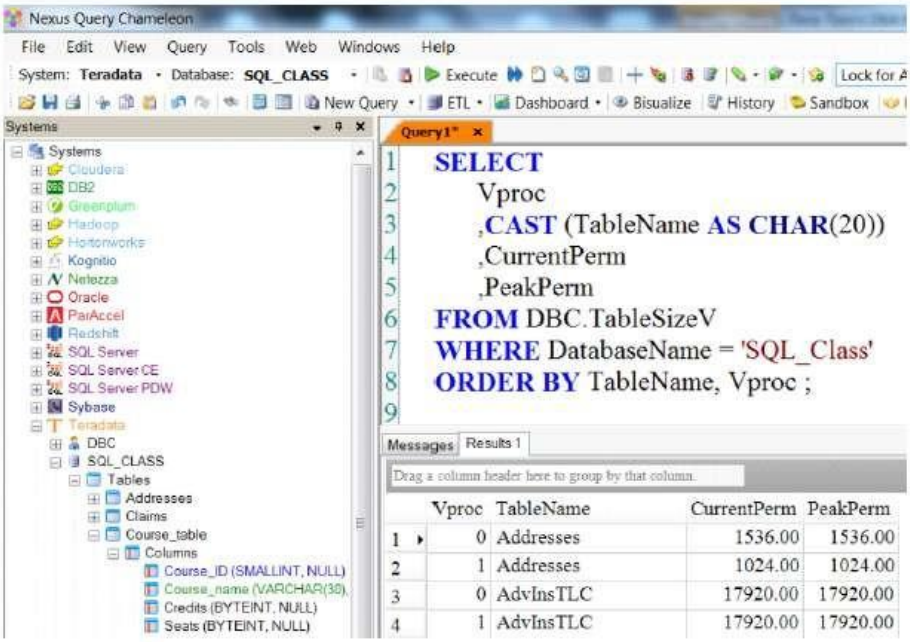
How much Space you have Per AMP



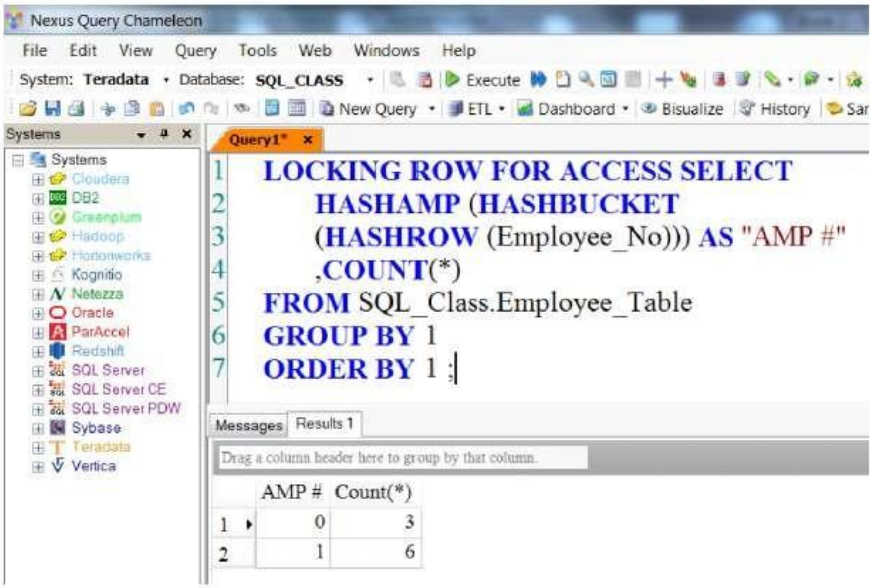
Finding your Space



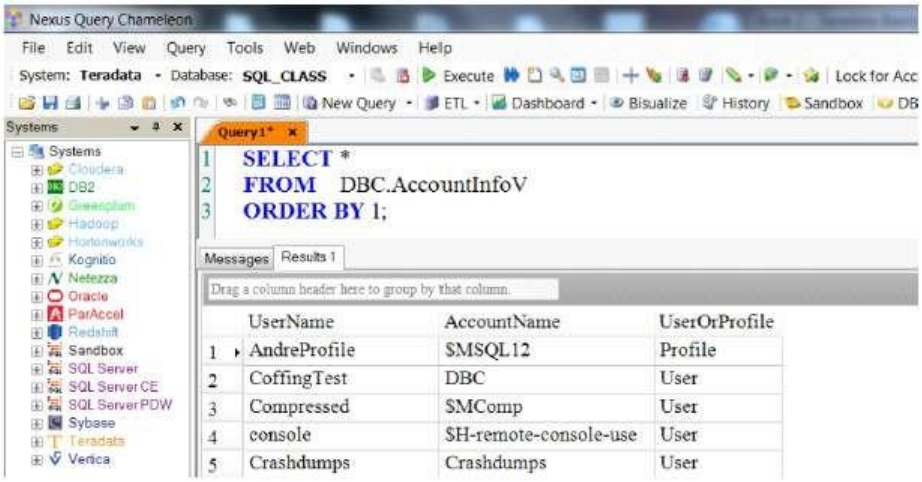
Finding Space Skew in Tables in a Database



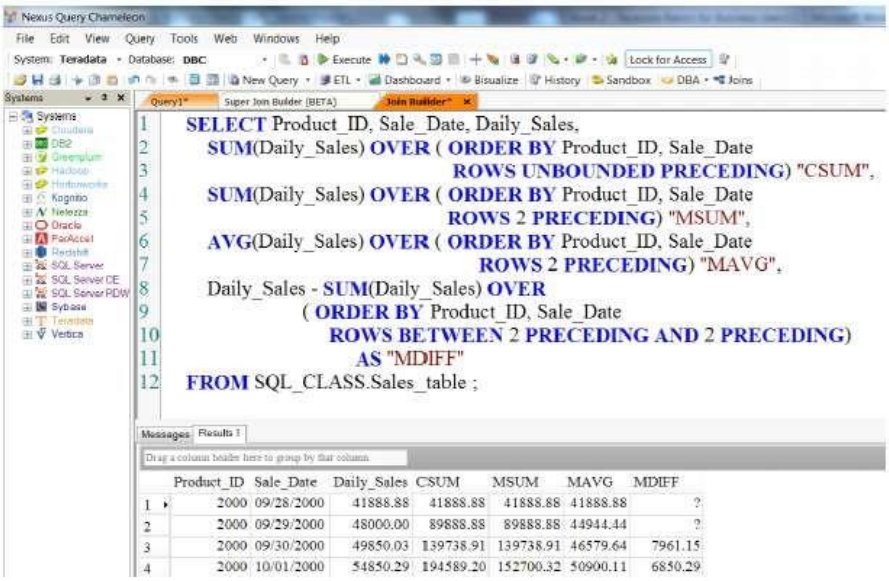
Finding the Number of rows per AMP for a Column



Finding Account Information



Ordered Analytics


Nexus Query Chameleon
File Edit View Query Tools Web Windows Help
System: Teradata Database: DBC
Execute
Lock for Access
New Query
ETL
Dashboard
Bisualize
History
Sandbox
DBA
Joins
Systems
Query1*
Super Join Builder (BETA)
Join Builder*
SELECT Product_ID, Sale_Date, Daily_Sales,
SUM(Daily_Sales) OVER (ORDER BY Product_ID, Sale_Date
ROWS UNBOUNDED PRECEDING) "CSUM",
SUM(Daily_Sales) OVER (ORDER BY Product_ID, Sale_Date
ROWS 2 PRECEDING) "MSUM",
AVG(Daily_Sales) OVER (ORDER BY Product_ID, Sale_Date
ROWS 2 PRECEDING) "MAVG",
Daily_Sales - SUM(Daily_Sales) OVER
(ORDER BY Product_ID, Sale_Date
ROWS BETWEEN 2 PRECEDING AND 2 PRECEDING)
AS "MDIFF"
FROM SQL_CLASS.Sales_table ;
Messages
Results 1
Drag a column header here to group by that column
Product_ID Sale_Date Daily_Sales CSUM MSUM MAVG MDIFF
1 2000 09/28/2000 41888.88 41888.88 41888.88 41888.88 ?
2 2000 09/29/2000 48000.00 89888.88 89888.88 44944.44 ?
3 2000 09/30/2000 49850.03 139738.91 139738.91 46579.64 7961.15
4 2000 10/01/2000 54850.29 194589.20 152700.32 50900.11 6850.29

View Functions

View Functions

Overview

"It is easier to go down a hill than up it, but the view is much better at the top."
- Arnold Bennett

Creating a Simple View

```
                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
```

```
CREATE View Employee_V AS
SELECT      Employee_No
           ,First_Name
           ,Last_Name
           ,Dept_No
FROM Employee_Table ;
```

The purposes of views are to restrict access to certain columns, derive columns or Join Tables, and to restrict access to certain rows (if a WHERE clause is used).

Basic Rules for Views

1. No ORDER BY inside the View CREATE (some exceptions exist)
2. All Aggregation needs to have an ALIAS
3. Any Derived columns (such as Math) needs an ALIAS

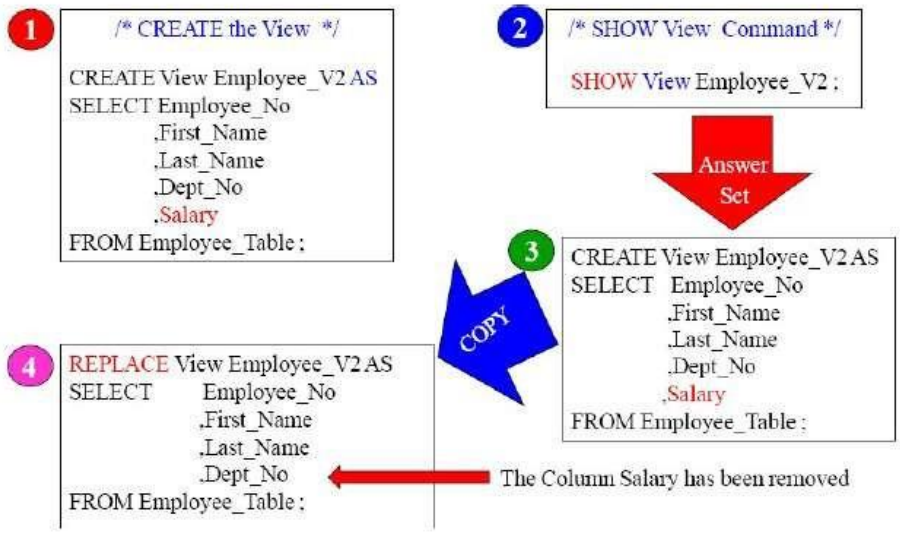
```
CREATE View Department_Salaries AS
SELECT     Dept_No
          ,SUM(Salary) as SumSal
          ,SUM(Salary) / 12 as MonthSal
FROM Employee_Table
GROUP BY 1;
```

```
SEL *
FROM Department_Salaries
Order By 1 ;
```

```
Dept_No   SumSal     MonthSal
-------  ---------  --------
      ?   32800.50   2733.38
     10   64300.00   5358.33
    100   48850.00   4070.83
    200   89888.88   7490.74
    300   40200.00   3350.00
    400  145000.00  12083.33
```

Above, are the basic rules of Views with excellent examples.

How to Modify a View



The REPLACE Keyword will allow a user to change a view.

Exceptions to the ORDER BY Rule inside a View

There are EXCEPTIONS to the ORDER BY rule. The TOP command allows a view to work with an ORDER BY inside. ANSI OLAP statements also work inside a View.

How to Get HELP with a View

1

```
HELP View  Command

HELP View Emp_View ;
```

Column Name	Type	Comment	Nullable	Format	Title
Employee_No	?		?	?	?
Dept_No	?		?	?	?
Last_Name	?		?	?	?
First_Name	?		?	?	?
Salary	?		?	?	?

Max Length	Decimal Total Digits	Decimal Fractional Digits	Range Low	Range High
?	?	?	?	?
?	?	?	?	?
?	?	?	?	?
?	?	?	?	?

UpperCase	Table/View?		Default value	Char Type	IdCol Type
?	?	?	?	?	
?	?	?	?	?	
?	?	?	?	?	
?	?	?	?	?	

The Help View command does little but show you the columns.

Views sometimes CREATED for Formatting or Row Security

```
CREATE VIEW  empl_200_v   AS
  SELECT  Employee_No AS Emp_No
         ,Last_Name    AS Last
         ,salary/12  (format '$$$$,$$9.99')  AS Monthly_Salary
FROM    Employee_Table
WHERE  Dept_No = 200 ;
```

```
  SELECTING from  A View

SELECT *
FROM Empl_200_v
ORDER BY Monthly_Salary ;
```

```
Emp_No  Last_Name  Monthly_Salary
_______ _________  ______________
1324657 Coffing        $3,490.74
1333454 Smith          $4,000.00
```

Views are designed to do many things. In the example above, this view formats and derives data, limits columns, and also limits the rows coming back with a WHERE.

Another Way to Alias Columns in a View CREATE

```
CREATE VIEW empl_200_v (Emp_Nbr, Last, Monthly_Salary)
        AS      SELECT Employee_No
                       ,Last_Name
                       ,Salary/12  (format '$$$$,$$9.99')
                FROM   Employee_Table
                WHERE Dept_No = 200 ;
```

Aliases are here

```
SELECT *
FROM Empl_200_v
ORDER BY Monthly_Salary ;
```

Emp_No	Last_Name	Monthly_Salary
1324657	Coffing	$3,490.74
1333454	Smith	$4,000.00

Will this View CREATE Error? No! It won't error because it's Aliased above!

Resolving Aliasing Problems in a View CREATE

```
CREATE VIEW empl_200_v (Emp_Nbr, Last, Monthly_Salary)
        AS SELECT Employee_No
                 ,Last_Name
                 ,Salary/12 (format '$$$$,$$9.99') as Sal_Monthly
           FROM Employee_Table
           WHERE Dept_No = 200 ;
```

```
SELECT *
FROM Empl_200_v
ORDER BY 3 ;
```

Emp_No	Last_Name	Monthly_Salary
1324657	Coffing	$3,490.74
1333454	Smith	$4,000.00

First Alias!

The ALIAS for Salary / 12 that'll be used in this example is MONTHLY_SALARY. It came first at the top, even though it is aliased in the SELECT list also.

Resolving Aliasing Problems in a View CREATE

```
CREATE VIEW  empl_200_v  (Emp_Nbr, Last, Monthly_Salary)
       AS  SELECT  Employee_No
                  ,Last_Name
                  ,Salary/12  (format '$$$$,$$9.99') as Sal_Monthly
           FROM Employee_Table
           WHERE  Dept_No = 200 ;
```

```
SELECT *
FROM Empl_200_v
ORDER BY Sal_Monthly ;
```

What will happen in the above query?

Resolving Aliasing Problems in a View CREATE

```
CREATE VIEW  empl_200_v  (Emp_Nbr, Last, Monthly_Salary)
       AS  SELECT  Employee_No
                  ,Last_Name
                  ,Salary/12  (format '$$$$,$$9.99') as Sal_Monthly
           FROM Employee_Table
```

```
WHERE  Dept_No = 200 ;
```

```
SELECT *
FROM Empl_200_v
ORDER BY Sal_Monthly ;
```

Doesn't Recognize

ERROR

If you ALIAS at the top, then that is the only ALIAS that the query can recognize. So, it is a good idea to alias at the top or the bottom, but not do both.

CREATING Views for Complex SQL such as Joins

```
CREATE VIEW Customer_Order_v  AS
SELECT   Customer_Name   AS Customer
        ,Order_Number
        ,Order_Total  (FORMAT  '$$$,$$9.99' ) AS Total_Amount
FROM     Customer_Table   AS  Cust
        ,Order_Table      AS  Ord
WHERE   Cust.Customer_Number = Ord.Customer_Number ;
```

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

Customer	Order_Number	Total_Amount
Ace Consulting	123552	$5,111.47
Billy's Best Choice	123456	$12,346.53
Billy's Best Choice	123512	$8,005.91
Databases N-U	123585	$15,231.62
XYZ Plumbing	123777	$23,454.84

A huge reason for Views, other than security, is to make Complex SQL easy for users. This view already has the Inner Join built into it, but users just SELECT.

WHY certain columns need Aliasing in a View

```
CREATE VIEW  Aggreg_Order_v  AS
SELECT Customer_Number
       ,Order_Date/100+190000  (format '9999-99')  AS  Yr_Mth_Orders
       ,COUNT(Order_Total)  AS Order_Cnt
       ,SUM(Order_Total)    AS  Order_Sum
       ,AVG(Order_Total)    AS  Order_Avg
FROM    Order_Table
GROUP BY  Customer_Number, Yr_Mth_Orders ;
```

```
SELECT Customer_Number
      ,Order_Sum
FROM Aggreg_Order_v ;
```

Customer_Number	Order_Sum
31323134	5111.47
87323456	15231.62
11111111	8005.91
11111111	12347.53
57896883	23454.84

When you CREATE a view, you have to ALIAS any aggregation or derived data (such as math). Why? So you can SELECT it later without having to do a SELECT *. Here, we only chose two columns and used their ALIAS to retrieve them.

Aggregates on View Aggregates

```
CREATE VIEW Aggreg_Order_v AS
SELECT Customer_Number
      ,Order_Date/100+190000 (format '9999-99') AS Yr_Mth_Orders
      ,COUNT(Order_Total) AS Order_Cnt
      ,SUM(Order_Total)   AS Order_Sum
      ,AVG(Order_Total)   AS Order_Avg
FROM Order_Table
GROUP BY Customer_Number, Yr_Mth_Orders ;
```

```
SELECT Customer_Number
      ,Order_Sum
FROM Aggreg_Order_v ;
```

Customer_Number	Order_Sum
31323134	5111.47
87323456	15231.62
11111111	8005.91
11111111	12347.53
57896883	23454.84

```
SELECT SUM (Order_Sum)
FROM Aggreg_Order_v ;
```

SUM(Order_Sum)
64151.37

The examples above show how we put a SUM on the aggregate Order_Sum.

Locking Row for Access

Lock Placed 

```
CREATE VIEW Emp_HR_v AS
Locking Row for ACCESS
SELECT Employee_No
      ,Dept_No
      ,Last_Name
      ,First_Name
FROM   Employee_Table ;
```

```
SELECT * FROM Emp_HR_v;
```

The Employee_Table used above will automatically use an ACCESS Lock, which allows ACCESS during UPDATES or table loads.

Most views utilize the Locking row for ACCESS command. This is because they want to be able to read while a table is being updated and loaded into. If the user knows a dirty read won't have a huge effect on their job, why not make a view lock with an ACCESS Lock, thus preventing unnecessary waiting?

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,
```

```
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;
```

```
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

Altering a Table

```
CREATE VIEW Emp_HR_v AS
SELECT  Employee_No
       ,Dept_No
       ,Last_Name
       ,First_Name
FROM   Employee_Table ;
```

```
-- Altering the actual Table
ALTER TABLE Employee_Table
ADD Mgr_No INTEGER ;
```

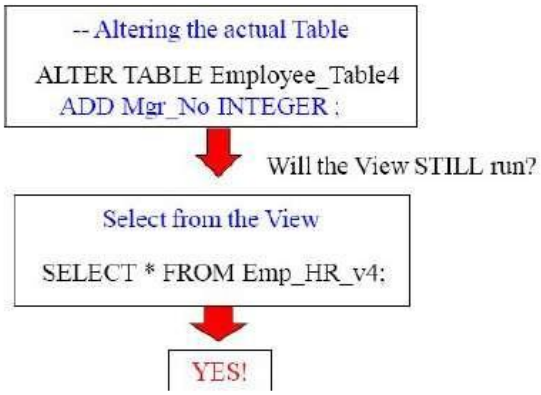
Will the View STILL run?

```
-- Select from the View
SELECT * FROM Emp_HR_v;
```

This view will run after the table has added an additional column!

Altering a Table after a View has been created

```
CREATE VIEW Emp_HR_v4 AS
SELECT   *
FROM    Employee_Table4 ;
```



This view runs after the table has added an additional column, but it won't include Mgr_No in the view results even though there is a SELECT * in the view. The View includes only the columns present when the view was CREATED.

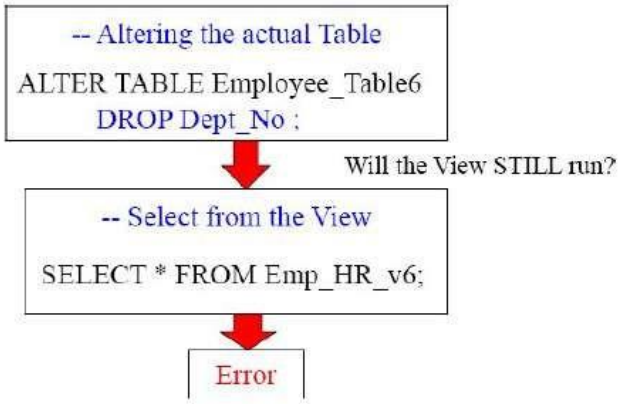
A View that errors After an ALTER

```
CREATE VIEW Emp_HR_v5 AS
SELECT  Employee_No
       ,Dept_No
       ,Last_Name
       ,First_Name
FROM   Employee_Table5 ;
```

This view will NOT run after the table has dropped a column referenced in the view.

Troubleshooting a View

```
CREATE VIEW Emp_HR_v6 AS
SELECT *
FROM   Employee_Table6 ;
```



This view will NOT run after the table has dropped a column referenced in the view, even though the View was CREATED with a SELECT *. At View CREATE Time, the columns present were the only ones the view considered responsible for. Dept_No was one of those columns. Once Dept_No was dropped, the view no longer works.

Updating Data in a Table through a View

```
CREATE VIEW Emp_HR_v8 AS
SELECT *
FROM   Employee_Table8;
```

```
--Updating the table through the View

   UPDATE Emp_HR_V8
      SET Salary = 88888.88
 WHERE Employee_No = 2000000;
```

```
  --SELECT from the actual Table

SELECT *
FROM   Employee_Table8
WHERE  Employee_No = 2000000;
```

Employee_No	Dept_No	Last_Name	First_Name	Salary
2000000	?	Jones	Squiggy	88888.88

You can UPDATE a table through a View if you have the RIGHTS to do so.

Maintenance Restrictions on a Table through a View

There are a few restrictions that disallow maintenance activity on a view with an INSERT, UPDATE or DELETE request. A view cannot be used for maintenance if it:

1. Performs a join operation - more than one table
2. Selects the same column twice - wouldn't know which one to use
3. Derives data - because it does not undo the math or calculation
4. Performs aggregation - because this eliminates detail data
5. Uses OLAP functions - because OLAP data is calculated
6. Uses a DISTINCT or GROUP BY - eliminates duplicate rows

Perform maintenance on a table through a view, but see the restrictions above first.