CONTENTS :

Set 1 :

1) Aggregation Function
2) Basic SQL Function
3) Collect Statistics
4) Data Manipulation Language (DML)
5) Date Functions

Aggregation Function

Aggregation Function

Overview

"Teradata climbed Aggregate Mountain and delivered a better way to Sum It."
- Tera-Tom Coffing

Quiz – You calculate the Answer Set in your own Mind

Aggregation_Table

Employee_No	Salary
423400	100000.00
423401	100000.00
423402	NULL

```
SELECT AVG(Salary) as "AVG"
      ,Count(Salary) as SalCnt
      ,Count(*)      as RowCnt
FROM Aggregation_Table ;
```

AVG	SalCnt	RowCnt

Please fill in the values you think will be in the Answer.

What would the result set be from the above query? The next slide shows answers!

Answer – You calculate the Answer Set in your own Mind

Aggregation_Table

Employee_No	Salary
423400	100000.00
423401	100000.00
423402	NULL

```
SELECT AVG(Salary) as "AVG"
      ,Count(Salary) as SalCnt
      ,Count(*)      as RowCnt
FROM Aggregation_Table ;
```

AVG	SalCnt	RowCnt
100000.00	2	3

Here are all the Correct answers

Here are your answers!

The 3 Rules of Aggregation

```
    Aggregation_Table

Employee_No      Salary
      423400   100000.00
      423401   100000.00
      423402        NULL
```

```
SELECT AVG(Salary)
      ,Count(Salary)
      ,Count(*)
FROM Aggregation_Table ;
```

1) Aggregates Ignore Null Values.

2) Aggregates WANT to come back in one row.

3) You CAN'T mix Aggregates with normal columns unless you use a GROUP BY.

AVG(Salary) =	Count(Salary) =	Count(*) =
$100000.00	2	3

There are Five Aggregates

Aggregation_Table

Employee_No	Salary
423400	100000.00
423401	100000.00
423402	NULL

There are FIVE AGGREGATES which are the following:

MIN – The Minimum Value.

MAX – The Maximum Value.

AVG – The Average of the Column Values.

SUM – The Sum Total of the Column Values.

COUNT – The Count of the Column Values.

Quiz – How many rows come back?

Employee_Table

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

Query

```
SELECT  MIN (Salary)
        ,MAX (Salary)
        ,SUM (Salary)
        ,AVG  (Salary)
        ,Count(*)
FROM    Employee_Table ;
```

How many rows will the above query produce in the result set? The answer is one.

Troubleshooting Aggregates

Employee_Table

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

```
SELECT  Dept_No
       ,MIN (Salary)
       ,MAX (Salary)
       ,SUM (Salary)
       ,AVG  (Salary)
       ,Count(*)
FROM   Employee_Table ;
```

NON-Aggregate

How many rows will the above query produce in the result set? None, because this query errors!

GROUP BY when Aggregates and Normal Columns Mix

Employee_Table

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

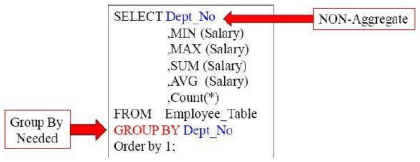
```
SELECT  Dept_No
       ,MIN (Salary)
       ,MAX (Salary)
       ,SUM (Salary)
       ,AVG  (Salary)
       ,COUNT(*)
FROM   Employee_Table
GROUP BY Dept_No;
```

NON-Aggregate

Group By Needed

A GROUP BY statement is needed when mixing aggregates and non-aggregates.

GROUP BY Delivers one row per Group



Dept_No	Min(Salary)	Max(Salary)	Sum(Salary)	AVG(Salary)	Count(*)
?	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	41888.88	48000.00	89888.88	44944.44	2
300	40200.00	40200.00	40200.00	40200.00	1
400	36000.00	54500.00	145000.00	48333.33	3

The Group BY Dept_No allows for the Aggregates to be calculated per Dept_No.

GROUP BY Dept_No or GROUP BY 1 the same thing

```
SELECT Dept_No
      ,MIN (Salary)
      ,MAX (Salary)
      ,SUM (Salary)
      ,AVG (Salary)
      ,Count(*)
FROM Employee_Table
GROUP BY Dept_No
ORDER BY Dept_No;
```

Both Queriesare the same to the system.

```
SELECT Dept_No
      ,MIN (Salary)
      ,MAX (Salary)
      ,SUM (Salary)
      ,AVG (Salary)
      ,Count(*)
FROM Employee_Table
GROUP BY 1
ORDER BY 1;
```

Dept_No	Min(Salary)	Max(Salary)	Sum(Salary)	AVG(Salary)	Count(*)
?	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	41888.88	48000.00	89888.88	44944.44	2
300	40200.00	40200.00	40200.00	40200.00	1
400	36000.00	54500.00	145000.00	48333.33	3

Both queries above produce the same result. The GROUP BY allows you to either name the column or use the number in

the SELECT list, just like the ORDER BY.

Limiting Rows and Improving Performance with WHERE

Employee_Table

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

```
SELECT Dept_No, MIN (Salary), MAX (Salary), SUM (Salary)
       , AVG  (Salary) , COUNT(*)
FROM   Employee_Table
WHERE Dept_No IN (200, 400)
GROUP BY Dept_No
Order by 1 ;
```

WHERE Clause acts as a filter before any Calculations are done

Will Dept_No 300 be calculated? Of course you know it will…NOT!

WHERE Clause in Aggregation limits unneeded Calculations

```
SELECT Dept_No, MIN (Salary), MAX (Salary), SUM (Salary)
       , AVG  (Salary) , COUNT(*)
FROM   Employee_Table
WHERE Dept_No IN (200, 400)
GROUP BY Dept_No
Order by 1 ;
```

WHERE Clause acts as a filter before any Calculations are done

Dept_No	Min(Salary)	Max(Salary)	Sum(Salary)	AVG(Salary)	Count(*)
200	41888.88	48000.00	89888.88	44944.44	2
400	36000.00	54500.00	145000.00	48333.33	3

The system eliminates reading any other Dept_No other than 200 and 400. This means that only the Dept_No of 200 and 400 will come off the disk to be calculated.

Keyword HAVING tests Aggregates after they are Totaled

```
SELECT Dept_No, MIN (Salary), MAX (Salary), SUM (Salary)
       , AVG  (Salary) , COUNT(*)
FROM   Employee_Table  WHERE Dept_No in (200, 400)
GROUP BY Dept_No
HAVING Count(*) > 2 ;
```

HAVING Clause acts as a filter on all Aggregates after they are totaled.

Previous Answer Set

Dept_No	Min(Salary)	Max(Salary)	Sum(Salary)	AVG(Salary)	Count(*)
200	41888.88	48000.00	89888.88	44944.44	2
400	36000.00	54500.00	145000.00	48333.33	3

NEW Answer Set
???????????????

Can you calculate what the new Answer Set will be after the HAVING Clause is implemented?

The HAVING Clause only works on Aggregate Totals. The WHERE filters rows to be excluded from calculation, but the HAVING filters the Aggregate totals after the calculations, thus eliminating certain Aggregate totals.

Keyword HAVING is like an Extra WHERE Clause for Totals

```
SELECT Dept_No, MIN (Salary), MAX (Salary), SUM (Salary)
       , AVG  (Salary) , COUNT(*)
FROM   Employee_Table  WHERE Dept_No in (200, 400)
GROUP BY Dept_No
HAVING Count(*) > 2 ;
```

HAVING Clause acts as a filter on all Aggregates after they are totaled.

Previous Answer Set without the HAVING Statement

Dept_No	Min(Salary)	Max(Salary)	Sum(Salary)	AVG(Salary)	Count(*)
200	41888.88	48000.00	89888.88	44944.44	2
400	36000.00	54500.00	145000.00	48333.33	3

New Answer Set using the HAVING Statement

Dept_No	Min(Salary)	Max(Salary)	Sum(Salary)	AVG(Salary)	Count(*)
400	34000.00	54500.00	145000.00	48333.33	3

The HAVING Clause only works on Aggregate Totals after they are totaled. It is a final check after aggregation is complete. Now only the totals with Count(*) > 2 can return.

Getting the Average Values per Column

```
SELECT 'Product_ID' AS "Column Name"
,COUNT(*) / COUNT(DISTINCT(Product_ID)) AS "Average Rows"
FROM Sales_Table ;
```

Column_Name	Average_Rows
Product_ID	7

```
SELECT 'Sale_Date' AS "Column Name"
,COUNT(*) / COUNT(DISTINCT(Sale_Date)) AS "Average Rows"
FROM Sales_Table ;
```

Column_Name	Average_Rows
Sale_Date	3

The first query retrieved the average rows per value for the column Product_ID. The example below did the same for the column Sale_Date.

Average Values per Column for All Columns in a Table

```
SELECT 'Product_ID' AS "Column Name"
       ,COUNT(*) / COUNT(DISTINCT(Product_ID)) AS "Average Rows"
       ,'Sale_Date' AS "Column Name2"
       ,COUNT(*) / COUNT(DISTINCT(Sale_Date)) AS "Average Rows2"
FROM Sales_Table ;
```

Column_Name	Average_Rows	Column_Name2	Average_Rows2
Product_ID	7	Sale_Date	3

The query above retrieved the average rows per value for both columns in the table.

Three types of Advanced Grouping

Sales_Table

Product_ID	Sale_Date	Daily_Sales
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	49950.03
3000	2000-09-30	43968.86
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 in this table are displayed

There are three advanced grouping options:

GROUP BY Grouping Sets
GROUP BY Rollup
GROUP BY Cube

Be prepared to be amazed. There are three advanced options listed above for grouping data. Each is more powerful than the one before. The following pages will give great examples.

GROUP BY Grouping Sets

```
SELECT Product_ID
      ,EXTRACT (MONTH FROM Sale_Date) AS MTH
      ,EXTRACT (YEAR FROM Sale_Date) AS YR
      ,SUM(Daily_Sales) AS SUM_Daily_Sales
FROM Sales_Table
GROUP BY GROUPING SETS (Product_ID, MTH, YR)
ORDER BY Product_ID Desc, MTH Desc, YR Desc;
```

Product_ID	MTH	YR	SUM_Daily_Sales	
3000	?	?	224587.82	Product 3000 sales
2000	?	?	306611.81	Product 2000 sales
1000	?	?	331204.72	Product 1000 sales
?	10	?	443634.99	October sales all years
?	09	?	418769.36	Sept. sales all years
?	?	2000	862404.35	Grand Total

GROUP BY GROUPING Sets above will show you what your Daily_Sales were for each Product_ID, for each month, and for each year.

GROUP BY Rollup

```
SELECT Product_ID
      ,EXTRACT (MONTH FROM Sale_Date) AS MTH
      ,EXTRACT (YEAR FROM Sale_Date) AS YR
      ,SUM(Daily_Sales) AS SUM_Daily_Sales
FROM Sales_Table
GROUP BY ROLLUP (Product_ID, MTH, YR)
ORDER BY Product_ID Desc, MTH Desc, YR Desc;
```

Not all 16 rows in this answer set could be displayed

Product_ID	MTH	YR	SUM_Daily_Sales	
3000	10	2000	84908.06	October, 2000 sales
3000	10	?	84908.06	October sales all years
3000	09	2000	139679.76	September, 2000 sales
3000	09	?	139679.76	
3000	?	?	224587.82	All Product 3000 sales
2000	10	2000	166872.90	
2000	10	?	166872.90	October sales all years
2000	09	2000	139738.91	

GROUP BY Rollup Result Set

Product_ID	MTH	YR	SUM_Daily_Sales
3000	10	2000	84908.06
3000	10	?	84908.06
3000	9	2000	139679.76
3000	9	?	139679.76
3000	?	?	224587.82
2000	10	2000	166872.90
2000	10	?	166872.90
2000	9	2000	139738.91
2000	9	?	139738.91
2000	?	?	306611.81
1000	10	2000	191854.03
1000	10	?	191854.03
1000	9	2000	139350.69
1000	9	?	139350.69
1000	?	?	331204.72
?	?	?	862404.35

This is the full result set from the previous GROUP BY **ROLLUP** query.

GROUP BY Cube

```
SELECT Product_ID
      ,EXTRACT (MONTH FROM Sale_Date) AS MTH
      ,EXTRACT (YEAR FROM Sale_Date) AS YR
      ,SUM(Daily_Sales) AS SUM_Daily_Sales
FROM Sales_Table
GROUP BY CUBE (Product_ID, MTH, YR)
ORDER BY Product_ID Desc, MTH Desc, YR Desc;
```

Product_ID	MTH	YR	SUM_Daily_Sales	
3000	10	2000	84908.06	October 2000 sales
3000	10	?	84908.06	October sales all years
3000	9	2000	139679.76	September 2000 sales
3000	9	?	139679.76	October sales all years
3000	?	2000	224587.82	Prod 3000 sales per year
3000	?	?	224587.82	Prod 3000 sales all years
2000	10	2000	166872.90	
2000	10	?	166872.90	

Not all 24 rows in this answer set could be displayed

GROUP BY ROLLUP displays Daily_Sales for each Product_ID, for each distinct month, for each month per year, for each year, plus a grand total.

GROUP BY CUBE Result Set

Product_ID	MTH	YR	SUM_Daily_Sales	
3000	10	2000	84908.06	Prod 3000 October, 2000 sales
3000	10	?	84908.06	Prod 3000 October sales all years
3000	9	2000	139679.76	Prod 3000 September, 2000 sales
3000	9	?	139679.76	Prod 3000 September sales all years
3000	?	2000	224587.82	Prod 3000 sales for the year 2000
3000	?	?	224587.82	Prod 3000 sales for all years
2000	10	2000	166872.90	Prod 2000 October, 2000 sales
2000	10	?	166872.90	Prod 2000 October sales all years
2000	9	2000	139738.91	Prod 2000 September, 2000 sales
2000	9	?	139738.91	Prod 2000 September sales all years
2000	?	2000	306611.81	Prod 2000 sales for the year 2000
2000	?	?	306611.81	Prod 2000 sales for all years
1000	10	2000	191854.03	Prod 1000 October, 2000 sales
1000	10	?	191854.03	Prod 1000 October sales all years
1000	9	2000	139350.69	Prod 1000 September, 2000 sales
1000	9	?	139350.69	Prod 1000 September sales all years
1000	?	2000	331204.72	Prod 1000 sales for the year 2000
1000	?	?	331204.72	Prod 1000 sales for all years
?	10	2000	443634.99	Total Sales for October, 2000
?	10	?	443634.99	Total Sales for all October dates
?	9	2000	418769.36	Total Sales for September, 2000
?	9	?	418769.36	Total Sales for all September dates
?	?	2000	862404.35	Total Sales for the year 2000
?	?	?	862404.35	Total Sales for all years totaled

Use the Nexus for all Groupings

In Nexus, just right click on the Sales_Table and choose Super Join Builder. Then, select all the columns. Then, choose the Analytics tab on the top right. Choose Grouping Sets in the Analytics Tab. Then, drag the Product_ID column to the Product. Drag the Sale_Date to the Date Column. Then, drag the Daily_Sales column to the Sum. Then, Check Box all the Group BY Functions on the right of the screen and then hit Execute or Send SQL to Nexus, and you are Done!

Testing Your Knowledge – Basic Aggregation

Employee_Table

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

First, SELECT the AVERAGE Salary from the Employee_Table.

Testing Your Knowledge – Multiple Aggregates

Employee_Table

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

Now, SELECT the AVERAGE Salary and the SUM of the Salary from the Employee_Table.

Testing Your Knowledge- Group By

Employee_Table

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

Now, SELECT the AVERAGE Salary and the SUM of the Salary from the Employee_Table PER DEPARTMENT (Dept_No).

Testing Your Knowledge – Using a Where Clause

Employee_Table

Employee_No	Dept_No	Last_Name	First_Name	Salary
1232578	100	Chambers	Mandee	48850.00
1256349	400	Harrison	Herbert	54500.00
2341214	800	Reilly	William	36000.00
2312225	300	Larkins	Loraine	40200.00
2000000	?	Jones	Squiggy	32800.50
1000234	10	Smythe	Richard	32800.00
1121334	400	Stickling	Cletus	54500.00
1324657	200	Coffing	Billy	41888.88
1333454	200	Smith	John	48000.00

After that, SELECT the AVERAGE Salary and the SUM of the Salary from the Employee_Table PER DEPARTMENT (Dept_No). However, I only want to see the people from Department 200, 300, 400.

Testing Your Knowledge- Using Having

Employee_Table

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

After that, SELECT the AVERAGE Salary and the SUM of the Salary from the Employee_Table PER DEPARTMENT (Dept_No). However, I only want to see Department 200, 300, 400 which has an AVERAGE Salary of over 43,000.

Final Answer to Test Your Knowledge on Aggregates

Answer Set

Dept_No	AVG(Salary)	Sum(Salary)
200	44944.44	89888.88
400	43316.67	144950.00

```
Select Dept_No, AVG(Salary), SUM(Salary)
From Employee_Table
Where Dept_No IN (200, 300, 400)
Group By Dept_No
Having AVG(Salary) > 43000
```

This should be your final answer set. The query under it should be approximately what you wrote to attain such an answer set. How'd you do?

Basic SQL Functions

Basic SQL Functions

Introduction

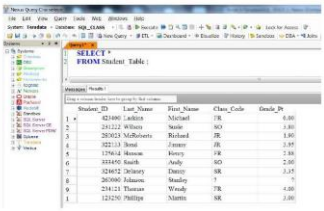
Student_Table

Student_ID	Last_Name	First_Name	Class_Code	Grade_Pt
423400	Larkins	Michael	FR	0.00
125634	Hanson	Henry	FR	2.88
280023	McRoberts	Richard	JR	1.90
260000	Johnson	Stanley	?	?
231222	Wilson	Susie	SO	3.80
234121	Thomas	Wendy	FR	4.00
324652	Delaney	Danny	SR	3.35
123250	Phillips	Martin	SR	3.00
322133	Bond	Jimmy	JR	3.95
333450	Smith	Andy	SO	2.00

The Student_Table above will be used in our early SQL Examples

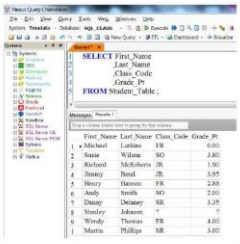
This is a pictorial of the Student_Table which we will use to present some basic examples of SQL and get some hands-on experience with querying this table. This book attempts to show you the table, show you the query, and show you the result set.

SELECT * (All Columns) in a Table



Mostly every SQL statement will consist of a SELECT and a FROM. You SELECT the columns you want to see on your report, and an Asterisk (*) means you want to see all columns in the table on the returning answer set!

SELECT Specific Columns in a Table



Column names must be separated by commas. Notice that only the columns requested come back on the report, not all columns. Also, notice that the order of the columns in the SQL is the same order on the report.

Using the Best Form for Writing SQL

Student_Table

Student_ID	Last_Name	First_Name	Class_Code	Grade_Pt
423400	Larkins	Michael	FR	0.00
125634	Hanson	Henry	FR	2.88
280023	McRoberts	Richard	JR	1.90
260000	Johnson	Stanley	?	?
231222	Wilson	Susie	SO	3.80
234121	Thomas	Wendy	FR	4.00
324652	Delaney	Danny	SR	3.35
123250	Phillips	Martin	SR	3.00
322133	Bond	Jimmy	JR	3.95
333450	Smith	Andy	SO	2.00

```
SELECT First_Name,
       Last_Name,
       Class_Code,
       Grade_Pt
FROM     Student_Table ;
```

```
SELECT First_Name
      ,Last_Name
      ,Class_Code
      ,Grade_Pt
FROM     Student_Table ;
```

Can you spot the difference between these two examples?

Why is the example on the right better even though they are functionally equivalent?

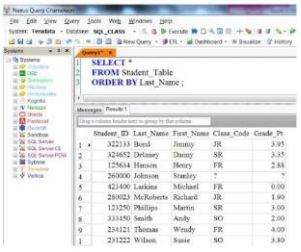
Commas in the Front or in the Back?

Commas in the front (example 1) is Tera-Tom's recommendation to writing, but the next page is an even better example for a company standard. Both queries will produce the same answer set and have the same performance

Place your Commas in front for better Debugging Capabilities

Having commas in front to separate column names makes it easier to debug.

Sort the Data with the ORDER BY Keyword



	Student_ID	Last_Name	First_Name	Class_Code	Grade_Pt
1	322133	Bond	Jimmy	JR	3.95
2	324652	Delaney	Danny	SR	3.35
3	125634	Hanson	Henry	FR	2.88
4	260000	Johnson	Stanley	?	?
5	423400	Larkins	Michael	FR	0.00
6	280023	McRoberts	Richard	JR	1.90
7	123250	Phillips	Martin	SR	3.00
8	333450	Smith	Andy	SO	2.00
9	234121	Thomas	Wendy	FR	4.00
1	231222	Wilson	Susie	SO	3.80

Rows typically come back to the report in random order. To order the result set, you must use an ORDER BY. When you order by a column, it will order in ASCENDING order. This is called the Major Sort!

ORDER BY Defaults to Ascending

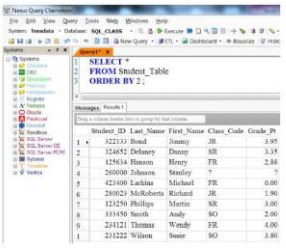
Sorts the Answer Set In Ascending Order By Last_Name

```
SELECT    *
FROM      Student_Table
ORDER BY Last_Name ;
```

Student_ID	Last_Name	First_Name	Class_Code	Grade_Pt
322133	Bond	Jimmy	JR	3.95
324652	Delaney	Danny	SR	3.35
125634	Hanson	Henry	FR	2.88
260000	Johnson	Stanley	?	?
423400	Larkins	Michael	FR	0.00
280023	McRoberts	Richard	JR	1.90
123250	Phillips	Martin	SR	3.00
333450	Smith	Andy	SO	2.00
234121	Thomas	Wendy	FR	4.00
231222	Wilson	Susie	SO	3.80

When you use the ORDER BY statement, it will default to ascending order. But you can change that if you like. I will show you how to do that in a few pages.

Use the Name or the Number in your ORDER BY Statement



```
SELECT *
FROM Student_Table
ORDER BY 2 ;
```

	Student_ID	Last_Name	First_Name	Class_Code	Grade_Pt
1	322133	Bond	Jimmy	JR	3.95
2	324652	Delaney	Danny	SR	3.35
3	125634	Hanson	Henry	FR	2.88
4	260000	Johnson	Stanley	?	?
5	423400	Larkins	Michael	FR	0.00
6	280023	McRoberts	Richard	JR	1.90
7	123250	Phillips	Martin	SR	3.00
8	333450	Smith	Andy	SO	2.00
9	234121	Thomas	Wendy	FR	4.00
1	231222	Wilson	Susie	SO	3.80

The ORDER BY can use a number to represent the sort column. The number 2 represents the second column on the report. This is also going to default to ascending.

The ORDER BY can use a number to represent the sort column. The number 2 represents the second column on the report. This is also going to default to ascending.

Two Examples of ORDER BY using Different Techniques

```
SELECT *
FROM Student_Table
ORDER BY 5 ;
```

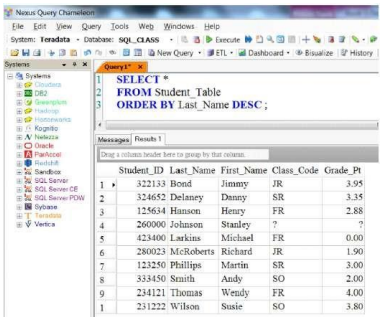
Same Query

```
SELECT *
FROM Student_Table
ORDER BY Grade_Pt ;
```

Student_ID	Last_Name	First_Name	Class_Code	Grade_Pt
260000	Johnson	Stanley	?	?
423400	Larkins	Michael	FR	0.00
280023	McRoberts	Richard	JR	1.90
333450	Smith	Andy	SO	2.00
125634	Hanson	Henry	FR	2.88
123250	Phillips	Martin	SR	3.00
324652	Delaney	Danny	SR	3.35
231222	Wilson	Susie	SO	3.80
322133	Bond	Jimmy	JR	3.95
234121	Thomas	Wendy	FR	4.00

You have the option of using a number instead of the column name. The columns number is represented by what position it is in the SELECT statement, not the table. If you use an * in your Select Statement, then the columns number is represented by the position it is in the table. The two above queries are the same.

Changing the ORDER BY to Descending Order



	Student_ID	Last_Name	First_Name	Class_Code	Grade_Pt
1	322133	Bond	Jimmy	JR	3.95
2	324652	Delaney	Danny	SR	3.35
3	125634	Hanson	Henry	FR	2.88
4	260000	Johnson	Stanley	?	?
5	423400	Larkins	Michael	FR	0.00
6	280023	McRoberts	Richard	JR	1.90
7	123250	Phillips	Martin	SR	3.00
8	333450	Smith	Andy	SO	2.00
9	234121	Thomas	Wendy	FR	4.00
1	231222	Wilson	Susie	SO	3.80

If you want the data sorted in descending order just place DESC at the end.

NULL Values sort First in Ascending Mode (Default)

```
SELECT *
FROM     Student_Table
ORDER BY 5;
```

```
SELECT *
FROM     Student_Table
ORDER BY Grade_Pt;
```

Student_ID	Last_Name	First_Name	Class_Code	Grade_Pt
260000	Johnson	Stanley	?	?
423400	Larkins	Michael	FR	0.00
280023	McRoberts	Richard	JR	1.90
333450	Smith	Andy	SO	2.00
125634	Hanson	Henry	FR	2.88
123250	Phillips	Martin	SR	3.00
324652	Delaney	Danny	SR	3.35
231222	Wilson	Susie	SO	3.80
322133	Bond	Jimmy	JR	3.95
234121	Thomas	Wendy	FR	4.00

Nulls

The default for ORDER BY is Ascending mode (ASC). Notice that this places the Null Values at the beginning of the Answer Set.

NULL Values sort Last in Descending Mode (DESC)

```
SELECT *
FROM     Student_Table
ORDER BY 5 DESC;
```

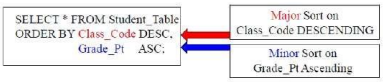
```
SELECT *
FROM     Student_Table
ORDER BY Grade_Pt DESC;
```

Student_ID	Last_Name	First_Name	Class_Code	Grade_Pt
234121	Thomas	Wendy	FR	4.00
322133	Bond	Jimmy	JR	3.95
231222	Wilson	Susie	SO	3.80
324652	Delaney	Danny	SR	3.35
123250	Phillips	Martin	SR	3.00
125634	Hanson	Henry	FR	2.88
333450	Smith	Andy	SO	2.00
280023	McRoberts	Richard	JR	1.90
423400	Larkins	Michael	FR	0.00
260000	Johnson	Stanley	?	?

Nulls are Last in DESC Order

You can ORDER BY in descending order by putting a DESC after the column name or its corresponding number. Null Values will sort Last in DESC order.

Major Sort vs. Minor Sorts



Student_ID	Last_Name	First_Name	Class_Code	Grade_Pt
123250	Phillips	Martin	SR	3.00
324652	Delaney	Danny	SR	3.35
333450	Smith	Andy	SO	2.00
231222	Wilson	Susie	SO	3.80
280023	McRoberts	Richard	JR	1.90
322133	Bond	Jimmy	JR	3.95
423400	Larkins	Michael	FR	0.00
125634	Hanson	Henry	FR	2.88
234121	Thomas	Wendy	FR	4.00
260000	Johnson	Stanley	?	?

Major Sorts First In DESC Order

Minor Sorts On Ties

Major sort is how things are sorted, but a minor sort kicks in if there are Major Sort ties.

Multiple Sort Keys using Names vs. Numbers

```
SELECT Employee_No
      ,Dept_No
      ,First_Name
      ,Last_Name
      ,Salary
FROM Employee_Table
ORDER BY Dept_No DESC
      ,Last_Name ASC
      ,Salary DESC;
```

```
SELECT Employee_No
      ,Dept_No
      ,First_Name
      ,Last_Name
      ,Salary
FROM Employee_Table
ORDER BY 2 DESC,
         4,
         5 DESC;
```

These queries sort identically

Queries can have a multiple columns in the ORDER BY. The first column in an ORDER BY is called the MAJOR SORT.

Those after it are MINOR SORTS.

Both of these Queries do the same thing. Once they sort Dept_No column in DESC order, they'll sort any ties by LAST_NAME. If any ties still occur, they'll sort by SALARY. Let me show you a real world example in the next slide!

Sorts are Alphabetical, NOT Logical

```
SELECT * FROM Student_Table
ORDER BY Class_Code ;
```

Student_ID	Last_Name	First_Name	Class_Code	Grade_Pt
260000	Johnson	Stanley	?	?
234121	Thomas	Wendy	FR	4.00
125634	Hanson	Henry	FR	2.88
423400	Larkins	Michael	FR	0.00
322133	Bond	Jimmy	JR	3.95
280023	McRoberts	Richard	JR	1.90
231222	Wilson	Susie	SO	3.80
333450	Smith	Andy	SO	2.00
324652	Delaney	Danny	SR	3.35
123250	Phillips	Martin	SR	3.00

This sorts alphabetically, but Sophomores (SO) logically come after Freshman

Change the query to Order BY Class_Code logically (FR, SO, JR, SR, ?)

Using A CASE Statement to Sort Logically

```
SELECT * FROM Student_Table
ORDER BY CASE Class_Code
           WHEN 'FR' THEN 1
           WHEN 'SO' THEN 2
           WHEN 'JR' THEN 3
           WHEN 'SR' THEN 4
                     ELSE 5
         END;
```

CASE in the ORDER BY Statement

Student_ID	Last_Name	First_Name	Class_Code	Grade_Pt
234121	Thomas	Wendy	FR	4.00
125634	Hanson	Henry	FR	2.88
423400	Larkins	Michael	FR	0.00
333450	Smith	Andy	SO	2.00
231222	Wilson	Susie	SO	3.80
322133	Bond	Jimmy	JR	3.95
280023	McRoberts	Richard	JR	1.90
324652	Delaney	Danny	SR	3.35
125250	Phillips	Martin	SR	3.00
260000	Johnson	Stanley	?	?

We are using a CASE Statement to Order BY Class_Code logically (FR, SO, JR, SR,)

How to ALIAS a Column Name

```
SELECT  First_Name  AS Fname
       ,Last_Name       Lname
       ,Class_Code      "Class Code"
       ,Grade_Pt   AS "AVG"
       ,Student_ID AS STU_ID
FROM    Student_Table ;
```

Different Techniques for Aliasing

ALIAS Rules!

1) AS is optional
2) Use Double Quotes when Spaces are in the Alias name
3) Use Double Quotes when the Alias is a reserved word

When you ALIAS a column, you give it a new name for the report header. You should always reference the column using the ALIAS everywhere else in the query. You never need Double Quotes in SQL unless you are Aliasing.

A Missing Comma can by Mistake become an Alias

```
SELECT First_Name, Last_Name, Class_Code  Grade_Pt
FROM    Student_Table ;
```

Missing a Comma

First_Name	Last_Name	Grade_Pt
Michael	Larkins	FR
Susie	Wilson	SO
Richard	McRoberts	JR
Jimmy	Bond	JR
Henry	Hinson	FR
Andy	Smith	SO
Danny	Delaney	SR
Stanley	Johnson	?
Wendy	Thomas	FR
Martin	Phillips	SR

Aliased as Grade_Pt

Column names must be separated by commas. Notice in this example, there is a comma missing between Class_Code and Grade_Pt. This will result in only three columns appearing on your report, and one of them will be titled wrong.

The Title Command and Literal Data

```
SELECT 'Character Data'
      ,'Character Data'  (TITLE 'Character// Data')
       ,123  (TITLE 'Nimeric Data')
       ,'Character Data' (TITLE '  My//Stacked//Example') ;
```

Stacks the Report Header

Character Data	Character Data	Numeric Data	MY Stacked Example
Character Data	Character Data	123	Character Data

A Literal Value brings back the Literal Value! Also, notice that the word 'Character' is stacked over the 'Data' portion of the heading for the second column using the Nexus Query Chameleon. So, as an alternative, a TITLE can be used instead of an alias which allows the user to include spaces in the output title.

The difference between an ALIAS and a TITLE is that the ALIAS can be used in the SQL again, such as in the ORDER BY or WHERE statements. But, a TITLE is only good for the report heading. Notice that Title uses Single Quotes not double quotes.

Comments using Double Dashes are Single Line Comments

Comment

```
-- Double Dashes provide a single line comment
SELECT    *
FROM      Student_Table
ORDER BY  Grade_Pt ;
```

Student_ID	Last_Name	First_Name	Class_Code	Grade_Pt
260000	Johnson	Stanley	?	?
423400	Larkins	Michael	FR	0.00
280023	McRoberts	Richard	JR	1.90
333450	Smith	Andy	SO	2.00
125634	Hanson	Henry	FR	2.88
123250	Phillips	Martin	SR	3.00
324652	Delaney	Danny	SR	3.35
231222	Wilson	Susie	SO	3.80
322133	Bond	Jimmy	JR	3.95
234121	Thomas	Wendy	FR	4.00

Double dashes make a single line comment that will be ignored by the system.

Comments for Multi-Lines

Comment

```
/* This is how you can make multi-line comments
to express what is going on in the code. */

SELECT    *
FROM      Student_Table
ORDER BY  Grade_Pt ;
```

Student_ID	Last_Name	First_Name	Class_Code	Grade_Pt
260000	Johnson	Stanley	?	?
423400	Larkins	Michael	FR	0.00
280023	McRoberts	Richard	JR	1.90
333450	Smith	Andy	SO	2.00
125634	Hanson	Henry	FR	2.88
123250	Phillips	Martin	SR	3.00
324652	Delaney	Danny	SR	3.35
231222	Wilson	Susie	SO	3.80
322133	Bond	Jimmy	JR	3.95
234121	Thomas	Wendy	FR	4.00

Slash Asterisk starts a multi-line comment, and Asterisk Slash ends the comment.

Comments for Multi-Lines as Double Dashes per Line

Comments

```
-- This is how you can make multi-line comments
-- also to express what is going on in the code.
SELECT    *
FROM      Student_Table
ORDER BY  Grade_Pt ;
```

Student_ID	Last_Name	First_Name	Class_Code	Grade_Pt
260000	Johnson	Stanley	?	?
423400	Larkins	Michael	FR	0.00
280023	McRoberts	Richard	JR	1.90
333450	Smith	Andy	SO	2.00
125634	Hanson	Henry	FR	2.98
123250	Phillips	Martin	SR	3.00
324652	Delaney	Danny	SR	3.35
231222	Wilson	Susie	SO	3.80
322133	Bond	Jimmy	JR	3.95
234121	Thomas	Wendy	FR	4.00

You can make multi-line comments with double dashes on each line.

A Great Technique for Comments to Look for SQL Errors

```
SELECT  Student_ID
       ,Last_Name
       ,First_Name
       ,Class_Code as Sum
       ,Grade_Pt
FROM    Student_Table
WHERE   Grade_Pt > 3.6
```

ERROR

Comment

```
SELECT Student_ID
       ,Last_Name
       ,First_Name
--     ,Class_Code as Sum
       ,Grade_Pt
FROM    Student_Table
WHERE   Grade_Pt > 3.6
```

Student_ID	Last_Name	First_Name	Grade_Pt
234121	Thomas	Wendy	4.00
231222	Wilson	Susie	3.80
322133	Bond	Jimmy	3.95

Sometimes you get an error in your SQL, and it is difficult to find. When our first query ran, it produced an error. We were not sure if our Class_Code was the error, so we commented that line out and ran our SQL again. Everything ran perfectly the next time, so we knew the Class_Code line must have been the

error. What was the error? The alias Sum is a reserved word. Comments can be used to test lines for errors.

Collect Statistics

Collect Statistics

The Teradata Parsing Engine (Optimizer) is Cost Based

The Parsing Engine (PE) is often referred to as the Teradata Optimizer, and it will actually generate several plans to choose from and ultimately choose the one with the lowest cost of resources. This is critical to performance in supporting mixed workloads ranging from OLTP to large joins and Decision Support (DS). All cost based optimizers require statistical information about the data and the machine resources (CPU, disk, memory, processors, etc.).

The other type of optimizer is a rules based optimizer which is designed for transactional On-Line Transaction Processing (OLTP) workloads where queries are well known and the data has been logically and physically structured to support OLTP workloads.

A cost based optimizer is much better than an optimizer that is rule based for data warehouses.

The Purpose of Collect Statistics

The Teradata Parsing Engine (PE) is in charge of creating the PLAN for the AMPs to follow. The PE works best when Statistics have been collected on a table. Then it knows:

1. The number of rows in the table
2. The average row size
3. Information on all Indexes in which statistics were collected
4. The range of values for the column(s) in which statistics were collected
5. The number of rows per value for the column(s) in which statistics were collected
6. The number of NULLs for the column(s) in which statistics were collected

The purpose of the COLLECT STATISTICS command is to gather and store demographic data for one or more columns or indices of a table or join index. This process computes a statistical profile of the collected data, and stores the synopsis in the Data Dictionary (DD) inside USER DBC for use during the PE's optimizing phase of SQL statement parsing. The optimizer uses this synopsis data to generate efficient table access and join plans. Do NOT COLLECT Statistics on all columns in the table.

When Teradata Collects Statistics, it creates a Histogram

Histogram of Employee_Table Column Last_Name

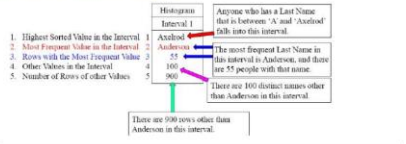
	Interval 1	Interval 2	Interval 3	Interval 4
1	Axelrod	Brubaker	Custer	Dzoba
2	Anderson	Bell	Cane	Davis
3	55	150	50	160
4	100	200	300	50
5	900	800	900	800

When statistics are collected, Teradata does a full table scan, sorts the column or index, places them into a default of 250 intervals, and then provides the above Histogram.

1. Highest Sorted Value in the Interval
2. Most Frequent Value in the Interval
3. Rows with the Most Frequent Value
4. Other Values in the Interval
5. Number of Rows of other Values

This is what is stored in statistics. This is tricky to understand at first, but recognize first that there are 55 people with a Last_Name of Anderson, 150 Bells, 50 Canes, and 160 with the name Davis. Each interval shows the most popular value and row count.

The Interval of the Collect Statistics Histogram



The PE now knows there are 55 Andersons in the table, and it assumes, for any other name falling between 'A' and 'Axelrod', that there are 9 values for each (900 / 100 = 9)

When statistics are collected, Teradata does a full table scan, sorts the column or index, places them into a default of 250 intervals, and then provides the above Histogram. This is what the PE uses to build a plan. Above, you see only interval one of 250.

Histogram Quiz

	Histogram of Employee_Table Column Last_Name			
	Interval 1	Interval 2	Interval 3	Interval 4
1	Axelrod	Brubaker	Custer	Dzoba
2	Anderson	Bell	Cane	Davis
3	55	150	50	160
4	100	200	300	50
5	900	800	900	800

1. Highest Sorted Value in the Interval
2. Most Frequent Value in the Interval
3. Rows with the Most Frequent Value
4. Other Values in the Interval
5. Number of Rows of other Values

1. Which Interval would the PE look to find the Last_Name of 'Apple'?_____
2. How many people are in the Employee_Table with a Last_Name of 'Davis'?_____
3. In Interval 2 how many other names are there other than 'Bell'?_____
4. How many people named 'Baker' would Teradata estimate?_____
5. How many people name 'Donaldson' would Teradata estimate?_____
6. How many people named 'Cooper' would Teradata estimate?_____

Answers to Histogram Quiz

	Histogram of Employee_Table Column Last_Name			
	Interval 1	Interval 2	Interval 3	Interval 4
1	Axelrod	Brubaker	Custer	Dzoba
2	Anderson	Bell	Cane	Davis
3	55	150	50	160
4	100	200	300	50
5	900	800	900	800

1. Highest Sorted Value in the Interval
2. Most Frequent Value in the Interval
3. Rows with the Most Frequent Value
4. Other Values in the Interval
5. Number of Rows of other Values

1. Which Interval would the PE look to find the Last_Name of 'Apple'? Interval 1
2. How many people are in the Employee_Table with a Last_Name of 'Davis'? 160
3. In Interval 2 how many other names are there other than 'Bell'? 200 other names
4. How many people named 'Baker' would Teradata estimate? 4 (800 / 200)
5. How many people name 'Donaldson' would Teradata estimate? 16 (800 / 50)
6. How many people named 'Cooper' would Teradata estimate? 6 (900 / 300)

What to COLLECT STATISTICS On?

You don't COLLECT STATISTICS on all columns and indexes because it takes up too much space for unnecessary reasons, but you do collect on:

- All Non-Unique Primary Indexes and All Non-Unique Secondary Indexes
- Non-indexed columns used in joins
- The Unique Primary Index of small tables (less than 1,000 rows per AMP)
- Columns that frequently appear in WHERE search conditions or in the WHERE clause of joins
- Primary Index of a Join Index
- Secondary Indexes defined on any join index
- Join index columns that frequently appear on any additional join index columns that frequently appear in WHERE search conditions

The first time you collect statistics, you collect them at the index or column level. After that, you just collect statistics at the table level and all previous columns collected previously are collected again. It is a mistake to collect statistics only once and then never do it again. COLLECT STATISTICS each time a table's data changes by 10%.

Why Collect Statistics?

What does collect statistics do to help the PE come up with a better plan?

- Access Path – The PE will easily choose and use any Primary Index access (UPI or NUPI), and it will also easily choose a Unique Secondary Index (USI), but statistics really help the PE decide whether or not to do a Full Table Scan or use a Non-Unique Secondary Index (NUSI) or if it can use multiple NUSI's ANDed together to perform a NUSI bitmap.
- Join Method – When you collect statistics, it gives Teradata a better idea whether or not to do a merge join, product join, hash join, or nested join.
- Join Geography – When two rows are joined together, they must physically be located on the same AMP. The only way that this happens naturally is if the join column (PK/FK) is the Primary Index of both tables. Most of the time, this is not the case and Teradata must decide the Join Geography of how it will relocate the rows to co-locate them on the same AMP. Will it redistribute (rehash by the join column) one or both of the tables, or will it duplicate the smaller table across all AMPs? A redistribution or duplication are the paths to co-location.
- Join Order – All joins are performed two tables at a time. What will be the best order to join the tables together? When two or more tables are involved, this becomes very important.

It is the access path, the join method, the join geography, and the order that makes statistics collection so vital to all Teradata systems.

How do you know if Statistics were collected on a Table?

Syntax: HELP Statistics <Table Name>

```
Help Statistics Employee_Table ;

Date      Time      Unique Values  Column Names
12/06/19  20:50:29              9  Employee_No
12/06/19  20:50:30              6  Dept_No
12/06/19  20:50:30              9  No,Last_Name, First_Name
```

2

```
Help Statistics Hierarchy_Table;

ERROR [HY000] [Teradata][ODBC Teradata Driver][Teradata Database]
There are no statistics defined for the table.
HELP STATISTICS Command Failed.
```

Careful: This looks like an error, but it is merely stating that No Statistics were Collected!

The HELP Statistics command will show you what statistics have been collected, or specifically tell you that no statistics were collected on the table.

A Huge Hint that No Statistics Have Been Collected

```
EXPLAIN SELECT * FROM New_Employee_Table ;
```

3) We do an all-AMPs RETRIEVE step from SQL_CLASS.New_Employee_Table by way of an all-rows scan with no residual conditions into Spool1 (group_amps), which is built locally on the AMPs. The size of Spool 1 is estimated with low confidence to be 12 rows (684 bytes). The estimated time for this step is 0.03 seconds.

```
COLLECT STATISTICS ON New_Employee_Table
Column Employee_No ;

EXPLAIN SELECT * FROM New_Employee_Table ;
```

3) We do an all-AMPs RETRIEVE step from SQL_CLASS.New_Employee_Table by way of an all-rows scan with no residual conditions into Spool1 (group_amps), which is built locally on the AMPs. The size of Spool 1 is estimated with high confidence to be 9 rows (513 bytes). The estimated time for this step is 0.03 seconds.

If you run an Explain on a query and the row estimate has No Confidence or Low Confidence, then that is a sign that no statistics were collected. Notice how the Explain above changed to High Confidence after we collected statistics on the table.

The Basic Syntax for COLLECT STATISTICS

1. `COLLECT STATISTICS on <Tablename> COLUMN <Column Name > ;`
2. `COLLECT STATISTICS on <Tablename> INDEX (<Column Name(s)>);`

Here are three actual examples

A. COLLECT STATISTICS on Employee_Table COLUMN Employee_No ; — Employee_No is the Primary Index. Always collect at the column level for single indexes.

B. COLLECT STATISTICS on Employee_Table COLUMN Dept_No ;

C. COLLECT STATISTICS on Employee_Table INDEX (First_Name, Last_Name); — Only collect at the Index level for multi-column indexes.

The example commands above provide good fundamentals and concepts to follow.

COLLECT STATISTICS Examples for a better Understanding

- COLLECT STATISTICS on Employee_Table COLUMN Dept_No ; — Here is how you COLLECT STATISTICS on a single column.
- COLLECT STATISTICS on Employee_Table COLUMN Employee_No ; — Here is how you COLLECT STATISTICS on a single Index.
- COLLECT STATISTICS on Employee_Table INDEX(First_Name, Last_Name); — Here is how to COLLECT STATISTICS on a Multi-column Index.
- COLLECT STATISTICS on Employee_Table Column (Employee_No, Dept_No) — Here is how to COLLECT STATISTICS on multiple columns that are often used together in the SQL WHERE Clause.
- COLLECT STATISTICS on Employee_Table — Here is how to Refresh STATISTICS on columns and indexes previously collected on when the table has changed by at least 10%.

The New Teradata V14 Way to Collect Statistics

In previous versions, Teradata required that you had to Collect Statistics for each column separately, thus always performing a full table scan each time. Those days are over!

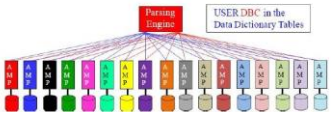
Old Way New Teradata V14 Way

```
COLLECT STATISTICS COLUMN
(First_Name, Last_Name)
ON Employee_Table ;

COLLECT STATISTICS COLUMN
(First_Name)
ON Employee_Table;

COLLECT STATISTICS COLUMN
(Dept_No)
ON Employee_Table;
```

```
COLLECT STATISTICS
  COLUMN(First_Name, Last_Name)
 ,COLUMN(First_Name)
 ,COLUMN(Dept_No)
ON Employee_Table;
```

The new way to collect statistics in Teradata V14 is to do it all at the same time. This is a much better strategy. Only a single table scan is required, instead of 3 table scans using the old approach. This is an incredible improvement.

Where Does Teradata Keep the Collected Statistics?



The Collect Statistics information is kept in user DBC in the Data Dictionary. The rows are spread evenly across all AMPs in three tables:

1. DBC.Indexes(for multi-column indexes only)
2. DBC.TVFields(for all columns and single column indexes)
3. DBC.StatsTbl(Teradata V14 and beyond)

DBC is the most powerful user, and DBC owns the Data Dictionary (DD) so it makes sense that DBC will house the statistics in DBC tables. In V14, the statistics are housed in the DBC.StatsTbl relieving the contention for the DBC.Indexes and TVFields tables.

The Official Syntaxes for COLLECT STATISTICS

1 Syntax 1

```
COLLECT STATISTICS [ USING SAMPLE ]
    ON [ TEMPORARY ] [ <table-name> | <join-index-name> | <hash-index-name> ]
  [ COLUMN [ <column-name> | (<column-list>) ]
    | [ UNIQUE ] INDEX ( <index-name> [ ALL ] | (<column-list>) )
                        [ ORDER BY [ HASH | VALUES ] [ <column-name> ] ] ] ;
```

2 Syntax 2

```
COLLECT STATISTICS [ USING SAMPLE ]
  [ COLUMN [ <column-name> | (<column-list>) ]
    | [ UNIQUE ] INDEX ( <index-name> [ ALL ] | (<column-list>) )
                        [ ORDER BY [ HASH | VALUES ] [ <column-name> ] ] ]
    ON [ TEMPORARY ] [ <table-name> | <join-index-name> | <hash-index-name> ] ;
```

How to Recollect STATISTICS on a Table

Here is the syntax for re-collecting statistics on a table.

```
COLLECT STATISTICS ON <Tablename> ;
```

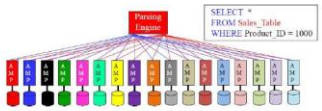
Below is an actual example

```
COLLECT STATISTICS ON Employee_Table;
```

This will NOT Collect on all columns in the table, but only refresh the columns and indexes currently collected.

The first time you collect statistics, you do it for each individual column or index that you want to collect on. When a table changes its data by 10% due to Inserts, Updates, or Deletes, you merely use the command above, and it re-collects on the same columns and indexes previously collected on.

Teradata Always Does a Random AMP Sample



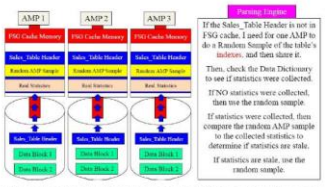
The Parsing Engine will hash the Table ID for a table being queried, and then use the Hash Map to determine which AMP will be assigned to do a Random AMP Sample for this table.

Remember that a Random AMP sample only applies to indexed columns and table row counts.

In the old days, Teradata never did a Random AMP Sample unless statistics were not collected. But, these days Teradata always does a Random AMP Sample before placing the Table Header inside each AMP's FSG Cache. This allows Teradata to compare these statistics with collected statistics to determine if statistics are old and stale. If the statistics are determined to be out of date, then the Random AMP Sample is used.

Random Sample is kept in the Table Header in FSG Cache

Random Sample is kept in the Table Header in FSG Cache



Teradata compares the collected statistics to a Random AMP Sample (obtained by sampling a single AMP before placing the Table Header in FSG Cache). This compare determines if the statistics will be used or if they should be replaced by the sample.

Multiple Random AMP Samplings

The PE does Random AMP Sampling based on the Table ID. The Table ID is hashed and that AMP is always selected as the sampled AMP for that table. This assures that no single AMP will be tasked for too many tables, but if the table is badly skewed this can confuse the PE.

So now more than one AMP can be sampled when generating row counts for a query plan for much better estimations on row count, row size, and rows per value estimates per table.

In the DBS Control area field 65 can now set the standard for how AMPs are sampled.

65. RandomAmpSampling – this field determines the number of AMPs to be sampled for getting the row

estimates of a table. The valid values are D, L, M, N or A.

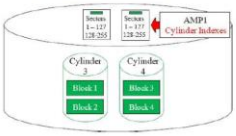
- D - The default is one AMP sampling (D is the default unless changed.)
- L - Maximum of two AMPs sampling
- M - Maximum of five AMPs sampling
- N - Node Level sampling (all the AMPs in a node are sampled).
- A - System Level sampling (all the AMPs in the system are sampled).

Multiple AMPs can now be used for the random AMP sample, so a higher number of AMPs sampled will provide better estimates to counter skewed results. But, it can cause short running queries to run slower just so long running queries can run faster.

How a Random AMP gets a Table Row count

For row counts read 1 or 2 cylinders from 1 (or more) AMPs. Calculate the number of rows in the table by taking the:

1. Average Number of Rows per Block in the sampled Cylinder(s)
2. Multiple this by the Number of Data Blocks in the sampled Cylinder(s)
3. Multiple this by the Number of Cylinders for this table on this AMP(s)
4. Multiple this by the Number of AMPs in the system



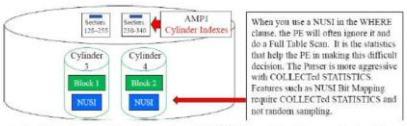
If a table (or index subtable) spans more than 1 cylinder, it will sample the first and the last cylinder. If it fits into 1 cylinder, it will only sample that one cylinder. This way, the AMP can do some simple math to estimate the total row count for a table.

Random AMP Estimates for NUSI Secondary Indexes

For Non-Unique Secondary Indexes (NUSI) estimates, read 1 or 2 cylinders from the NUSI subtable, and then count the number of NUSI values in the cylinder(s).

The table row count is divided by the NUSI row counts to get a Rows Per NUSI Value.

It also assumes the number of distinct values on one AMP = total distinct values.



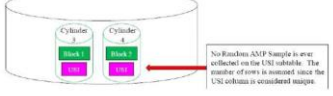
When you use a NUSI in the WHERE clause, the PE will often ignore it and do a Full Table Scan. It is the statistics that help the PE in making this difficult decision. The Parser is more aggressive with COLLECTed STATISTICS. Features such as NUSI Bit Mapping require COLLECTed STATISTICS and not random sampling.

The Random AMP reads a couple of cylinders and some data NUSI blocks, and then does some simple math to estimate the Rows per NUSI Value. The PE then knows how strong or weak the WHERE clause is using the NUSI, and if it should even use the NUSI. This is the most important decision for the Parsing Engine. Should it just do a Full Table Scan, or use the NUSI? That is the biggest reason the PE needs statistics. That is why you should always collect statistics on all NUSI indexes.

USI Random AMP Samples are Not Considered

Random AMP sampling assumes that the number of distinct values in a USI equals its cardinality, so it does not read the index subtable for USI equality conditions. Because equality conditions on a unique index return only one row, the PE automatically uses the USI without considering statistics. However, if a USI will frequently be specified in non-equality statements, such as range constraints, then you should collect statistics on the Unique Secondary Index.

```
SELECT *
FROM Sales_Data_Mart
WHERE Sale_Date BETWEEN
'2013-01-01' AND '2013-01-31';
```



No Random AMP Sample is ever collected on the USI subtable. The number of rows is assumed since the USI column is considered unique.

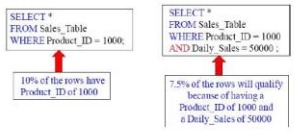
You really only need to collect statistics on a Unique Secondary Index column if there are a lot of SQL statements on non-equality conditions such as range queries.

There's No Random AMP Estimate for Non-Indexed Columns

For non-indexed columns without statistics, the optimizer uses a fixed formula to estimate the number of rows. This is sometimes referred to as heuristics.

Teradata assumes 10% for one column in an equality condition in the WHERE clause.

Assumes 7.5% for two columns, each in an equality condition, and ANDed together.



Teradata does not do a Random AMP Sample for non-indexed columns that are used in the WHERE clause of the SQL. It uses the above for a quick and dirty estimate.

Summary of the PE Plan if No Statistics Were Collected

Today's Teradata systems always perform a random AMP sample even if tables have statistics. Then, they compare the random AMP sample with the statistics to determine if the statistics are stale.

A random AMP is selected for a random sample. Two things happen:
1) Indexes are sampled on the random AMP, and the PE estimates based on the total number of AMPs in the system.
2) If a column in the WHERE clause of the SQL is not an index, the PE assumes that 10% of the rows will come back. If two columns are in the WHERE clause then it assumes 7.5% of the rows will come back. If three columns are in the WHERE Clause, it assumes 5%.

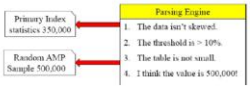
A random AMP sample is selected by the PE, so if it finds there are no statistics on the table or if the statistics are old and stale it has options.

Stale Statistics Detection and Extrapolation

The PE estimates the table row count based on the Primary Index collection of a table (called the histogram), but it also does a Random AMP Sample for comparison. If these two metrics differ by a threshold of 10% or for small tables (10,000 new rows), then statistics are considered stale.

Any cardinality estimations will now use extrapolation. What is extrapolation? This means that the PE will estimate based on past statistics and go with new estimations. In other words, the PE will derive its own estimate based on past history and the random sample.

Small tables with less than 25 rows per AMP, or for skewed tables, no extrapolation will done.



When statistics are determined to be stale, the PE will use the Random AMP sample and also extrapolate, which means to estimate new statistics based on historical data.

Extrapolation for Future Dates

AMP 1
Order_Table
Jan 2014
Feb 2014
Mar 2014
Apr 2014
May 2014
Jun 2014
Jul 2014
Aug 2014
Sep 2014
Oct 2014
Nov 2014
Dec 2014

AMP 2
Order_Table
Jan 2014
Feb 2014
Mar 2014
Apr 2014
May 2014
Jun 2014
Jul 2014
Aug 2014
Sep 2014
Oct 2014
Nov 2014
Dec 2014

```
EXPLAIN
SELECT *
FROM Order_Table
WHERE Order_Date BETWEEN
'2015-01-01' AND '2015-03-31';
```

Parsing Engine

1. I only have statistics for 2014.
2. During the first quarter of 2014 I had 1,000,000 orders.
3. Orders have been increasing by 10%.
4. I think I can extrapolate and estimate that there will be 1,100,000 orders in the first quarter of 2015.

Above, you can see the PE only has collect statistics for the year 2014, but the Explain is asking about 2015 data. Teradata will extrapolate new estimates for future dates based on history and growth estimates. This allows for better estimates and less concern about statistics collection.

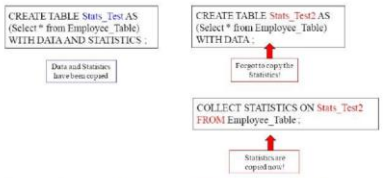
How to Copy a Table with Data and the Statistics?

This next example is pretty amazing. Assume that the original Employee_Table had COLLECT STATISTICS on the columns Employee_No and Dept_No. The new table we have called Employee_Table_New will have DDL exactly like the Employee_Table plus data plus the statistics. Yes, the exact same statistics will be copied to the new table. Below is the actual example!

```
CREATE TABLE Employee_Table_New AS Employee_Table
WITH DATA
AND Statistics;
```

The example above will CREATE a new table called Employee_Table_New, and it will have the exact same DDL as the Employee_Table, the exact same data, and the exact same statistics.

COLLECT STATISTICS Directly From another Table



In Teradata V13 and above, you can Collect Statistics directly from another table.

How to Copy a Table with NO Data and the Statistics?

This next example is clever. Assume that the original Employee_Table had COLLECT STATISTICS on the columns Employee_No and Dept_No. The new table we have called Employee_Table_99 will have DDL exactly like the Employee_Table but NO data. It will have the Statistics, but they will be Zeroed Statistics.

```
CREATE TABLE Employee_Table_99 AS Employee_Table
with NO DATA
AND Statistics;
```

Table DDL copied with NO Data and Zeroed Statistics on the columns Employee_No and Dept_No.

Once you have loaded Employee_Table_99 with the data you want by using BTEQ, FastLoad, an INSERT/SELECT, Nexus, or the ETL tool of your choice so you can then Recollect Statistics!

```
COLLECT STATISTICS ON Employee_Table_99 ;
```

You have just Re-Collected Statistics on Employee_Table_99 for the columns Employee_No and Dept_No. The easy re-collection on the columns previously collected on (after the data is loaded) was the entire purpose of getting the Zeroed Statistics in the first place. Make sure you recollect after your data is loaded though!

When to COLLECT STATISTICS Using only a SAMPLE

You might consider Collecting Statistics with SAMPLE if:

1. You are collecting statistics on a very large table.
2. When collecting statistics becomes a problem with system performance or cost because the system is so busy.

Don't consider Collecting Statistics with SAMPLE if:

1. The tables are small.
2. To replace all existing full scan Collect Statistics.
3. If the column's data is skewed badly.

COLLECT STATISTICS can be very time consuming because it performs a full table scan and then performs a lot of statistical calculations. Because Collect Statistics runs infrequently and benefits query optimization, it is considered a necessary task. Without statistics, query performance will suffer. The bad news about sampled statistics is that they may not be as accurate, which could negatively affect the PE's plans. In most cases, sampled statistics are better than no statistics. Don't use Sample unless necessary!

Examples of COLLECT STATISTICS Using only a SAMPLE

COLLECT STATISTICS USING SAMPLE on Employee_Table COLUMN Dept_No ;	Here is how you COLLECT STATISTICS on a single column.
COLLECT STATISTICS USING SAMPLE on Employee_Table COLUMN Employee_No ;	Here is how you COLLECT STATISTICS on a single Index.
COLLECT STATISTICS USING SAMPLE on Employee_Table INDEX (First_Name, Last_Name);	Here is how to COLLECT STATISTICS on a Multi-column Index.
COLLECT STATISTICS on Employee_Table ;	Here is how to Refresh STATISTICS on a Table.

If you recollect statistics on a sample, it recollects with the same sample amount!

Sampled statistics are generally more accurate for data that is not skewed. For example, columns or indexes that are unique or nearly unique are not skewed. Because the PE needs to be aware of skewed data, you should not collect with sample on skewed data. That is why sampling is generally more appropriate for indexes than non-indexed column(s). If you recollect statistics on a Sample, it Recollects with the same Sample!

Examples of COLLECT STATISTICS for V14

To collect sample statistics using the system default sample:

```
COLLECT STATISTICS USING SYSTEM SAMPLE COLUMN (Product_ID) ON Sales_Table;
```

To collect sample statistics by scanning 15 percent of the rC1Ws and use 100 intervals:

```
COLLECT STATISTICS USING SAMPLE 15 PERCENT AND MAXINTERVALS 10
COLUMN (Product_ID) AS Product_Stats ON Sales_Table;
```

To change sample statistics to 20 percent (for Product_ID) and use 250 intervals:

```
COLLECT STATISTICS USING SAMPLE 20 PERCENT AND MAXINTERVALS 25
COLUMN (Product_ID) AS Prod_stats ON Sales_Table;
```

To display the COLLECT STATISTICS statements for a table

```
SHOW STATISTICS ON Sales_Table;
```

To display statistics details - summary section, high bias values, and intervals

```
SHOW STATISTICS VALUES COLUMN Product_ID ON Sales_Table;
```

How to Collect Statistics on a PPI Table on the Partition

Here is the syntax for collecting statistics on a PPI table.

```
COLLECT STATISTics on <Tablename> COLUMN p_qTrTiON;
```

Here is an actual example

```
COLLECT STATISTICS on Order_Table_PPI COLUMN PARTITION;
```

Three reasons to Collect on the Partition:

- The Parsing Engine will have a better plan for PPI Tables.
- This helps the most with Partition Elimination on Range Queries.
- This is especially helpful when a table has a lot of empty partitions.

The Parsing Engine can use this information to better estimate the query cost when there are a significant number of empty partitions. If PARTITION statistics are not collected, empty partitions may cause the Parsing Engine to underestimate the number of rows in a partition. You shouldn't use WITH SAMPLE to collect on Partitions.

Teradata V12 and V13 Statistics Enhancements

In V12 Extrapolate Statistics is designed to more accurately provide for a statistical estimate for date range-based queries that specify a "future" date that is outside the bounds of the current statistics. This results in less re-collections.

In V12 Stale Statistics Detection compares the Random AMP Sample with the statistics collected and determines if they are stale, and should not be used.

In V13 Statistics can now be collected on Volatile Tables.

In V13 PARTITION statistic capabilities have been added to Global Temporary Tables. In V13 Multi-Column statistics are now available on Hash and Join Indexes.

In V13 Sample Statistics are available on Tables, Volatile Tables, Global Temporary Tables, Hash Indexes and Join Indexes, including the Partition Columns.

Teradata V14 Statistics Enhancements

- There is now a SUMMARY option to collect table-level statistics.
- SYSTEM SAMPLE option allows the system to determine the sampled system percentage.
- Sampling options have been enhanced (e.g., SAMPLE n PERCENT).
- Statistics are stored in DBC.StatsTbl to reduce access contention and improve performance.
- New views DBC.StatsV, DBC.ColumnStatsV, DBC.MultiColumnStatsV, and IndexStatsV.
- SHOW STATISTICS statement reports detailed statistics in plain text or XML formatting.
- Internal PE enhancements for histogram structure and use, including:
  - Storing statistics data in their native Teradata data types without losing precision
  - Enhanced extrapolation methods for stale statistics
  - Maintaining statistics history

Teradata V14 now allows you to determine a sampling percentage for sampled statistics. You can even collect/recollect either summary statistics or both full and summary statistics combined. You can now collect statistics on global temporary tables, and you can provide a name for statistics collected while also being able to specify the column ordering for multicolumn statistics. There is also a dedicated statistics cache that is designed to improve query optimization time.

Teradata V14 Summary Statistics

New in Teradata 14.0, table-level statistics known as "summary statistics" are collected whenever column or index statistics are collected. Summary statistics do not cause their own histogram to be built, but rather they create important facts about the table undergoing collection that are held in the new DBC.StatsTbl. Here are some of the items in "summary statistics":

- Row count
- Average block size
- block level compression metrics
- Temperature

One critical advantage is that the optimizer now uses summary stats to get the most up-to-date row count from the table in order to provide more accurate extrapolations.

Here is how you can see the Summary Statistics.

```
SHOW SUMMARY STATISTICS VALUES ON Employee_Table;
```

You can specifically request summary statistics for a table, but you never need to do that because each individual statistics collection statement causes summary stats to be gathered (it is a quick process). It is best in V14 to now write your collection scripts to do it all in one scan Vs. multiple statements.

Teradata V14 MaxValueLength

```
COLLECT STATISTICS
   USING MAXVALUELENGTH 50
   COLUMN ( Product_Name)
ON Product_Table ;
```

Before V14, whenever you collected statistics, Teradata only placed the first 16 bytes in the statistics so long names were cut off. Now, the default is 25 bytes, but you can use the MaxValueLength keyword (example above) to specify the length you want.

MAXVALUELENGTH lets you expand the length of the values contained in the histogram for that statistic. The new default length is 25 bytes, when previously it was 16. If needed, you can specify well over 1000 bytes for a maximum value length. The 16-byte limit on value sizes in earlier releases was always padded to 16 bytes. Now, the length can be longer, but no padding is done.

Teradata V14 MaxIntervals

```
COLLECT STATISTICS
  USING MaxIntervals 500
  COLUMN ( Last_Name)
ON Employee_Table ;
```

Before V14, whenever you collected statistics, Teradata did a full table scan on the values, sorted them, and then placed them into 200 intervals. Now, the default is 250 intervals, but you can specify (example above) the number of intervals you desire.

Each statistics interval highlights its single most popular value, and the number of rows that carry that value are recorded. The rest of the values in the interval are estimated. By increasing the number of intervals, the optimizer can accurately get a better row count for a greater number of the most popular values. A larger number of intervals can be useful if you have widespread skew on a column or index you are collecting statistics on and you want more individual high-row-count values to be represented in the histogram. The range is 0 - 500 for MaxIntervals.

Teradata V14 Sample N Percent

```
COLLECT STATISTICS
  USING Sample 20 Percent
  COLUMN ( Last_Name)
ON Employee_Table ;
```

Using Sample before defaulted each time to the system parameter, but now you can specifically state the percent you want for each column or index.

SAMPLE n PERCENT allows you to specify sampling at the individual statistics collection level, rather than at the system level. Now, different levels of statistics sampling to different columns and indexes can be performed. The better you get at knowing your data and the queries upon them, the more you can specifically use the sampling to better help the PE.

Teradata Statistics Wizard

The Teradata Statistics Wizard is a graphical tool that automates the collection and re-collection of statistics, helping the DBA to better manage the statistics process.

Now the DBA can specify a workload to be analyzed and receive statistics recommendations.

Any database or collection of tables, indexes, or columns can be selected for the collection, or re-collection of statistics.

Recommendations can be based on table demographics and general heuristics.

Defer execution of and schedule the arbitrary collection/re-collections for later.

Display and modify the interval statistics for a column or index.

Provides numerous reports on statistics recommendations, update cost analysis and table usage.

The Statistics Wizard can be used to help the DBA with statistics.

Data Manipulation Language

Data Manipulation Language (DML)

INSERT Syntax # 1

The following syntax of the INSERT does not use the column names as part of the command. Therefore, it requires that the VALUES portion of the INSERT match each and every column in the table with a data value or a NULL.

```
INS[ERT] [INTO] <table-name> VALUE
(<literal-data-value>[,,<literal-data-value> );
```

Note: Using INS instead of INSERT is not ANSI compliant.

The INSERT statement is used to put a new row into a table. A status is the only returned value from the database; no rows are returned to the user. It must account for all the columns in a table using either a data value or a NULL. When executed, the INSERT places a single new row into a table. Although it can run as a single row insert, primarily it is used in utilities like BTEQ, FastLoad, MultiLoad, TPump or some other application that reads a data record and uses the data to build a new row in a table.

INSERT Example with Syntax 1

```
INSERT INTO  My_Table  VALUES
( 'My character data', 124.56, 102587, , NULL, '2000-12-31' ) ;
```

My character data	124.56	102587	NULL	NULL	2000-12-31

After the execution of the above INSERT, there is a new row with the first character data value of 'My character data' going into Column1, the decimal value of 124.56 into Column2, the integer 102587 into Column3, NULL values into Column4 and Column5, and a date into Column6.

The NULL expressed in the VALUES list is the literal representation for no data. However, the two commas (,,) that follow the positional value for Column3 also represent missing data. The commas are placeholders or delimiters for the data values. When no data value is coded, the end result is a NULL.

INSERT Syntax # 2

The syntax of the second type of INSERT follows:

```
INS[ERT] [INTO] <table-name>
(<column-name>[,,<column-name> ]
VALUES
(<literal-data-value>[,,<literal-data-value> );
```

Note: Using INS instead of INSERT is not ANSI compliant.

This is another form of the INSERT statement that can be used when some of the data is not available. It allows for the missing values (NULL) to be eliminated from the list in the VALUES clause. It is also the best format when the data is arranged in a different sequence than the CREATE TABLE, or when there are more nulls (unknown values) than available

data values.

INSERT Example with Syntax 2

```
INSERT INTO  My_Table  ( Column2, Column1, Column3, Column6 )
VALUES  ( 124.56, 'My character data', 12587, '2000-12-31') ;
```

My character data	124.56	102587	NULL	NULL	2000-12-31

The above statement incorporates both of the reasons to use this syntax. First, notice that the column names Column2 and Column1 have been switched, to match the data values. Also, notice that Column4 and Column5 do not appear in the column list, therefore they are assumed to be NULL. This is a good format to use when the data is coming from a file and does not match the order of the table columns.

INSERT Example with Syntax 3

```
INSERT INTO  My_Table
(Column2=124.56, Column1='My character data', Column3=12587,
Column6='2000-12-31') ;
```

My character data	124.56	102587	NULL	NULL	2000-12-31

The third form of the INSERT statement can be used to insert the same row as the previous INSERT. It might look like this.

Using NULL for Default Values

Either of the next two INSERT statements may be used to build a row with no data values in My_Table:

```
INSERT INTO  My_Table  VALUES (,,,,,) ;
```

```
INSERT INTO My_Table VALUES
(NULL,NULL,NULL,NULL,NULL,NULL) ;
```

NULL	NULL	NULL	NULL	NULL	NULL

Teradata now has the ANSI DEFAULT VALUES functionality. Although an INSERT statement could easily put a null value into a table column, it requires it to use the NULL reserved word or by omitting a value for that column(s) between commas.

INSERT/SELECT Command

The syntax of the INSERT / SELECT is:

```
INS[ERT] [INTO] <table-name>
```

```
  SELECT <column-name> [,<column-name>]
FROM <table-name>)
```

Although the INSERT is great for adding a single row not currently present in the system, an INSERT/SELECT is even better when the data already exists within Teradata. In this case, the INSERT is combined with a SELECT. However, no rows are returned to the user. Instead, they go into the table as new rows.

The SELECT reads the data values from the one or more columns in one or more tables and uses them as the values to INSERT into another table. Simply put, the SELECT takes the place of the VALUES portion of the INSERT.

This is a common technique for building data marts, interim tables and temporary tables. It is normally a better and much faster alternative than extracting the rows to a data file, then reading the data file and inserting the rows using a utility.

INSERT/SELECT Example using All Columns (*)

When all columns are desired to make an exact copy of the second table, and both tables have the exact same number of columns in the exact same order with the exact same data types. An * may be used in the SELECT to read all columns without a WHERE clause, as in the example below:

```
INSERT INTO My_Table
SELECT * FROM My_Original_Table;
```

Like all SELECT operations without a WHERE clause, a full table scan occurs and all the rows of the second table are inserted into My_Table, using only the data values from the columns listed.

INSERT/SELECT Example with Less Columns

When fewer than all the columns are desired, either of the following INSERT / SELECT statements will do the job:

```
INSERT INTO My_Table
   SELECT (Column1, Column2, Column3 , , , '2010-01-01')
      FROM My_Original_Table;
```

```
INSERT INTO My_Table (Column2, Column1, Column3, Column6)
   SELECT Column2, Column1, Column3 , CURRENT_DATE
     FROM My_Original_Table ;
```

In both of the above examples, only the first three and the last columns are receiving data. In the first INSERT, the data is a literal date. The second INSERT uses the CURRENT_DATE. Both are acceptable, depending on what is needed.

Working with the same concept of a normal INSERT, when using the column names, the only data values needed are for these columns. They must be in the same sequence as the column list, not the CREATE TABLE. Therefore, omitted data values or column names become a NULL data value.

INSERT/SELECT to Build a Data Mart

As an example of a data mart, it might be desirable to build a summary table using something like the following:

```
INSERT INTO My_Table
  SELECT ( Column1, SUM(Column2), AVG(Column3),
           COUNT(Column4), AVG(CHAR(Column5)),
           AVG(CHAR(Column6)) )
  FROM My_Original_Table
  GROUP BY 1 ;
```

However used, the INSERT / SELECT can be a powerful tool for creating rows from the rows already contained in one or more other tables.

Fast Path INSERT/SELECT

```
INSERT INTO My_Table SELECT * FROM My_Original_Table ;
```

When the table being loaded is empty, the INSERT / SELECT is very fast. This is especially true when all columns and all rows are being copied. Remember, the table being loaded must be empty to attain the speed. If there is even one row already in the table, it negates the ability to take the Fast Path.

There are two reasons behind this speed. First, there is no need to Transient Journal an identifier for each inserted row. Recovery, if needed, is to empty the table. No other type of recovery can be easier or faster.

When all columns and all rows are requested from the existing table, and they exactly match the columns in the new table, there is no need to use spool. The rows go straight into the table being loaded. Additionally, when all rows are being selected, Teradata does not bother to read the individual rows. Instead, each AMP literally copies the blocks of the original table to blocks for the new table.

These reasons are why it is called the Fast Path. To use this technique, the order of the columns in both tables must match exactly, and so must the data types. Otherwise, spool must be used to rearrange the data values or translate from one data type to the other.

NOT quite the Fast Path INSERT/SELECT

What if it is necessary to retrieve the rows from multiple tables for the INSERT?

Multiple INSERT / SELECT operations could be performed as follows:

```
INSERT INTO My_Table    SELECT * FROM My_Original_Table_1;
INSERT INTO My_Table    SELECT * FROM My_Original_Table_2;
INSERT INTO My_Table    SELECT * FROM My_Original_Table_3;
```

The first INSERT/SELECT into My_Table loads the empty table extremely fast, even with millions of rows. However, the table is no longer empty, and the subsequent INSERT is much slower because it cannot use the fast path. All inserted rows must be identified in the Transient Journal. It can more than double the processing time.

The real question is: How does one make the entire individual SELECT operations act as one, so that the table stays empty until all rows are available for the INSERT?

UNION for the Fast Path INSERT/SELECT

One way get the Fast Path is to use the UNION command to perform all SELECT operations in parallel before the first row is inserted into the new table. Therefore, all rows are read from the various tables combined into a single answer set in spool, and then loaded into the empty table. All of this is done at high speed.

For instance, if all the rows from three different tables are needed to populate the new table, the applicable statement might look like the following:

```
INSERT INTO My_Table
 SELECT * FROM My_Original_Table_1
  UNION
 SELECT * FROM My_Original_Table_2
  UNION
 SELECT * FROM My_Original_Table_3 ;
```

Again, the above statement assumes that all four tables have exactly the same columns. Whether or not that would ever be the case in real life, this is used as an example. However, at this point we know the columns in the SELECT must match the columns in the table to be loaded, no matter how that is accomplished.

BTEQ for the Fast Path INSERT/SELECT

A second alternative method is available using BTEQ. The key here is that BTEQ can do multiple SQL statements as a single transaction for the SELECT and the INSERT operations. The only way to do that is to delay the actual INSERT, until all of the rows from all the select operations have completed. Then, the INSERT is performed as a part of the same transaction into the empty table.

```
.logon localtd/dbc
Password: **********
Logon successfully completed
BTEQ - Enter your DBC/SQL request or BTEQ commands:

INSERT INTO My_Table
  SELECT * FROM My_Original_Table_1
; INSERT INTO My_Table
  SELECT * FROM My_Original_Table_2
; INSERT INTO My_Table
  SELECT * FROM My_Original_Table_3;
```

By having another SQL command on the same line as the semi-colon (;), in BTEQ, they all become part of the same multi-statement transaction. Therefore, all are inserting into an empty table, and it is much faster than doing each INSERT individually.

The UPDATE Command Basic Syntax

```
UPD[ATE] <table-name>
[FROM <table-name> [AS<alias-name>]]
  SET <column-name> = { <expression-or-data-value> | <data-value> }
    [,, <column-name> = <expression-or-data-value> | <data-value> ]
    [WHERE <condition-test> ]
        [AND <condition-test> ... ] [ OR <condition-test> ... ] [ALL] ;
```

The UPDATE statement is used to modify data values in one or more columns of one or more existing rows. A status is the only returned value from the database; no rows are returned to the user.

When business requirements call for a change to be made in the existing data, the UPDATE is the SQL statement to use. In order for the UPDATE to work, it must know a few things about the data row(s) involved. Like all SQL, it must know which table to use for making the change, which column or columns to change, and the change to make within the data.

Two UPDATE Examples

```
UPDATE My_Table
  SET Column2 = 256
      ,Column4 = 'Mine'
      ,Column5 = 'Yours'
 WHERE Column1 = 'My character data';
```

```
UPDATE My_Table
   SET Column2 = Column2 + 256
WHERE Column1 = 'My character data'
      AND Column4 = 'Mine'
      AND Column5 = 'Yours';
```

The first UPDATE command modifies all rows that contain 'My character data' including the one that was inserted earlier in this chapter. It changes the values in three columns with new data values provided after the equal sign (=):

The next UPDATE uses the same table as the above statement. However, this time it modifies the value in a column based on its current value and adds 256 to it. The UPDATE determines which row(s) to modify with compound conditions written in the WHERE clause based on values stored in other columns.

Subquery UPDATE Command Syntax

```
UPD[ATE] <table-name>
[ FROM <table-name> [AS <alias-name> ]]
  SET <column-name> = ( <expression-or-data-value> | <data-value> )
    [,, <column-name> = <expression-or-data-value> | <data-value> ]
WHERE <column-name> [,, <column-name> ]
  IN ( SELECT <column-name> [,,<column-name> ]
         FROM <table-name> [ AS <alias-name> ]
        [WHERE <condition-test> ,, ] ) [ ALL ] ;
```

Sometimes it is necessary to update rows in a table when they match rows in another table. To accomplish this, the tables must have one or more columns in the same domain. The matching process then involves either a subquery or join processing.

Notice in the above syntax example, that creating an alias for the table being updated is not compatible with a FROM clause. This change was made in V2R4.

Example of Subquery UPDATE Command

To change rows in My_Table using another table called Ctl_Tbl, the following UPDATE uses a subquery operation to accomplish the operation:

```
UPDATE My_Table
 SET Column3 = 20000000
WHERE Column2 IN ( SELECT Column2 FROM Ctl_Tbl
                                 WHERE Column3 > 5000
      AND ctl_tbl.Column4 IS NOT NULL ) ;
```

Sometimes it is necessary to update rows in a table when they match rows in another table. To accomplish this, the tables must have one or more columns in the same domain. The matching process then involves either a subquery or join processing.

Notice above, that creating an alias for the table being updated is not compatible with a FROM clause, and remember that the change was made in V2R4.

Join UPDATE Command Syntax

```
UPD[ATE] <table-name>
  [ FROM  <table-name> [ AS <alias-name> ] ]
  SET <column-name> = ( <expression-or-data-value> | <data-value> )
    [,, <column-name> = <expression-or-data-value> | <data-value> ]
WHERE [<table-name>.]<column-name> = [<table-name>.]<column-name>
    [ AND <condition-test> ] [ OR <condition-test> ] [ ALL ] ;
```

When adding an alias to the UPDATE, the alias becomes the table name and MUST be used in the WHERE clause when qualifying columns.

Example of an UPDATE Join Command

To change rows in My_Table using another table called Ctl_Tbl, the following UPDATE uses a join to accomplish the operation. Both examples are equivalent.

```
UPDATE My_Table
  FROM Ctl_Tbl AS Ctbl
   SET Column3 = 20000000
      ,Column5 = 'A'
WHERE My_Table.Column2 = Ctbl.Column2
AND My_Table.Column3 > 5000   AND Ctl_Tbl.Column4 IS NOT NULL ) ;
```

```
 UPDATE My_Table
  SET Column3 = 20000000
    , Column5 = 'A'
WHERE mytable.Column2 = Ctl_tbl.Column2
AND mytable.Column3 > 5000 AND Ctl_tbl.Column4 IS NOT NULL ) ;
```

In reality, the FROM is optional. This is because Teradata can dynamically include a table by qualifying the join column with the table name. The FROM is only needed to make an alias for the join tables. The second UPDATE is the same as the above without the FROM for Ctl_Tbl.

Fast Path UPDATE

The following INSERT/SELECT "updates" the values in Column3 and Column5 in every row of My_Table, using My_Table_Copy:

```
INSERT INTO My_Table_Copy
       SELECT Column1
             ,Column2
             ,Column3*1.05
             ,Column4
             ,'A'
             ,Column6
       FROM My_Table ;
```

When the above command finishes, My_Table_Copy contains every row from My_Table with the needed update.

The DELETE Command Basic Syntax

```
DEL[ETE] [ FROM ] <table-name> [ AS <alias-name> ]
[WHERE condition] [ALL] ;
```

The DELETE statement has one function, and that is to remove rows from a table. A status is the only returned value from the database, no rows are returned to the user. One of the fastest things that Teradata does is to remove ALL rows from a table.

The reason for its speed is that it simply moves all of the sectors allocated to the table onto the free sector list in the AMP's Cylinder Index. It is the fast path, and there is no OOPS command unless the explicit transaction has not yet completed. In that case, a ROLLBACK statement can be issued to undo the delete operation before a COMMIT. Otherwise, the rows are gone and it will take either a backup tape or a BEFORE image in the Permanent Journal to perform a manual rollback. Be Very CAREFUL with DELETE. It can come back to bite you if you're not careful.

Two DELETE Examples to DELETE ALL Rows in a Table

```
DELETE FROM My_Table ;
DEL My_Table ;
```

Both examples will delete all the rows in the table.

Since the FROM and the ALL are optional, and the DELETE can be abbreviated, the second example still removes all rows from a table and executes exactly the same as the above statement.

A DELETE Example: Deleting only Some of the Rows

```
DELETE FROM My_Table
WHERE Column6 < 1001131;
```

The DELETE example above only removes the rows that contained a date value less than 1001231 (December 31, 2000) in Column6 (DATE, data type) and leaves all rows newer than or equal to the date.

Subquery and Join DELETE Command Syntax

The subquery syntax for the DELETE statement raises:

```
DELETE <table-name>
WHERE (<column-name> [,...<column-name>] )
  IN (SELECT <column-name> [,...<column-name>]
      FROM <table-name> [ AS <alias-name> ]
      [WHERE condition ...] [ ALL ] ) ;
```

The join syntax for DELETE statement follows:

```
DELETE <table-name>
[ FROM <table-name> [ AS <alias> ]
WHERE (<table-name>.<column-name>=<table-name>.<column-name
  [ AND <condition> ] [ ALL ] ) ;
```

Sometimes it is desirable to delete rows from one table based on their existence in or by matching a value stored in another table. For example, you may be asked to give a raise to all people in the Awards Table. To access these rows from another table for comparison, a subquery or a join operation can be used.

Example of Subquery DELETE Command:

To remove rows from My_Table using another table called Control_Del_Tbl, the next DELETE uses a subquery operation to accomplish the operation.

```
DELETE FROM My_Table
WHERE Column1,Column2 IN ( SELECT Column2 FROM Control_Del_Tbl
                           WHERE Column3 > 5000 );
```

The above uses a Subquery and the DELETE command.

Example of Join DELETE Command

To remove the same rows from My_Table using a join with the table called Control_Del_Tbl, the following is another technique to accomplish the same DELETE operation as the subquery example on the previous page.

```
DELETE My_Table
  FROM Control_Del_Tbl AS Ctl_Tbl
 WHERE My_Table.Column1 = Ctl_Tbl.Column1
   AND My_Table.Column2 = Ctl_Tbl.Column2
   AND Ctl_Tbl.Column3 > 5000;
```

The previous example could also be written using the format below. However, an alias cannot be used with this format.

```
DELETE My_Table
WHERE My_Table.Column1,Column2 IN ( SELECT Column1,Column2 FROM Control_Del_Tbl
                                    WHERE Column3 > 5000 );
```

```
DELETE My_Table
  WHERE My_Table.Column2 = Control_Del_Tbl.Column2
          AND My_Table.Column1 = Control_Del_Tbl.Column1
          AND Control_Del_Tbl.Column4 IS NULL ;
```

The above uses a Join and the DELETE, and is equivalent to the previous subquery example we saw on the previous page.

Fast Path DELETE

Fast Path DELETE always occur when the WHERE clause is omitted.

However, most of the time, it is not desirable to delete all of the rows. Instead, it is more practical to remove older rows to make room for newer rows, or periodically purge data rows beyond the scope of business requirements.

For instance, the table is supposed to contain twelve months' worth of data, and it is now month number thirteen. It is now time to get rid of rows that are older than twelve months.

As soon as a WHERE clause is used in a DELETE, it must take the slow path to delete the rows. This simply means that it must log or journal a copy of each deleted row. This is to allow for the potential that the command might fail. If that should happen, Teradata can automatically put the deleted rows back into the table using a ROLLBACK. As slow as this additional processing makes the command, it is necessary to insure data integrity.

To use the Fast Path, a technique is needed that eliminates the journal logging. The trick is, again, to use a Fast Path INSERT / SELECT. This means, we insert the rows that need to be kept into an empty table.

Fast Path DELETE Example # 1

Normal Path Processing for the DELETE (uses the Transient Journal):

```
DELETE FROM My_Table
WHERE Column6  <  1001231 ;
```

There are three different methods for using Fast Path Processing in BTEQ for a DELETE. The first method uses an INSERT/SELECT. It will be fast, but it does require privileges for using the appropriate DDL. It also requires that additional PERM space be available for temporarily holding both the rows to be kept and all of the original rows at the same time.

```
INSERT INTO My_Table_Copy
   SELECT * FROM My_Table WHERE Column6  >  1001230
; DROP TABLE My_Table
; RENAME My_Table_Copy to My_Table ;
```

The first example does NOT use the Fast Path Delete, but the second example does.

Fast Path DELETE Example # 2

Normal Path Processing for the DELETE (uses the Transient Journal):

```
DELETE FROM My_Table
WHERE Column4  <  1001231 ;
```

This next method also uses an INSERT/SELECT and will be fast. It does not require privileges for using any DDL. It probably will not be faster than the first method, since the rows must all be put back into the original table. However, the table is empty and the Fast Path will be used:

```
INSERT INTO My_Table_Copy
    SELECT * FROM My_Table WHERE Column4 >= 1001230
; DELETE My_Table
; INSERT INTO My_Table
    SELECT * FROM My_Table_Copy ;
```

The first example does NOT use the Fast Path Delete, but the second example does.

Fast Path DELETE Example # 3

This INSERT/SELECT uses a Global temporary table to prepare for the single transaction to copy My_Table in BTEQ:

```
INSERT INTO My_Global_Table_Copy
    SELECT * FROM My_Table WHERE Column4 >= 1001230
; DELETE My_Table
; INSERT INTO My_Table
    SELECT * FROM My_Global_Table_Copy ;
```

A Global Temporary Tables requires that TEMPORARY space be available for temporarily holding the rows to be kept and all of the original rows at the same time. A Volatile Temporary table could also be used. Its space comes from spool. However, it requires a CREATE statement to build the table, unlike Global Temporary tables. More information on Temporary tables is available in this book.

If you are not using BTEQ, these statements can be used in a macro. This works because macros always execute as a transaction.

MERGE INTO

Here are the V2R5 MERGE Rules:

1. The Source relation must be a single row.
2. The Primary Index of the target relation must use an equality condition to a numeric constant or a string literal in the ON Clause.
3. You cannot request an error log.

Here are the V12 and above MERGE Rules:

1. The Source relation can be a either a single row or multiple rows.
2. The Primary Index of the target relation must be bound by use of an equality condition to an explicit term or to a column set of the source relation.
3. The Primary Index of the target table cannot be updated.
4. The INSERT if the WHEN NOT MATCHED Clause is specified, must have the value specified in the ON clause with the target table primary index of the target table, which causes the INSERT to be on the local AMP.

MERGE merges a source row set into a target table based on whether there is a MATCH or whether there is a NOT MATCH condition. If there is a MATCH, then Teradata will UPDATE the row, but if there is a NOT MATCH condition, it will INSERT the row. This is a Teradata Extension pre V12 and an ANSI Version on Teradata V12.

MERGE INTO Example that Matches

The first query shows Squiggy Jones in the Employee_Table with a salary of 32500.00.

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

    Employee_No  Dept_No  Last_Name  First_Name  Salary
    -----------  -------  ---------  ----------  --------
        2000000        ?  Jones      Squiggy     32500.00
```

I will now perform a MERGE that will have a MATCH.

```
MERGE INTO Employee_Table
 USING VALUES (2000000, NULL, 'Jones', 'Squiggy', 40000.00)
  AS E (Emp, Dept, Las, Fir, Sal)
ON Employee_No = Emp
 WHEN MATCHED THEN
  UPDATE set salary = Sal
 WHEN NOT MATCHED THEN
  INSERT VALUES (Emp, Dep, Las, Fir, Sal);
```

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

    Employee_No  Dept_No  Last_Name  First_Name  Salary
    -----------  -------  ---------  ----------  --------
        2000000        ?  Jones      Squiggy     40000.00
```

MERGE INTO Example that does NOT Match

I will now perform a MERGE that will have a NOT MATCH situation because Employee_No 3000000 does not exist in the Employee_Table.

```
MERGE INTO Employee_Table
 USING VALUES (3000000, 400, 'Coffing', 'TeraTom', 300000.00)
  AS E (Emp, Dep, Las, Fir, Sal)
ON Employee_No = Emp
 WHEN MATCHED THEN
  UPDATE set salary = Sal
 WHEN NOT MATCHED THEN
  INSERT VALUES (Emp, Dep, Las, Fir, Sal);
```

```
SELECT   * FROM   Employee_Table   WHERE Employee_No = 3000000;

    Employee_No  Dept_No  Last_Name  First_Name  Salary
    -----------  -------  ---------  ----------  --------
```

```
3000000      400  coffing    TeraTom    300000.00
```

There is no employee 3000000 that exists before the MERGE INTO statement, but once the MERGE INTO statement runs (and doesn't find a Match), it INSERTS employee 3000000 into the table.

OReplace

OReplace will replace characters or eliminate them.

Just give the replace string as a zero length string and it removes the character.

So if col1 contains '123*45*67*89'
and you:

OReplace (col1,"*","")

the result is

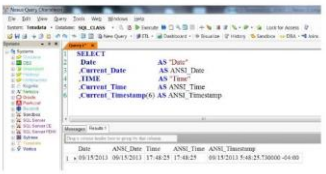
123456789

The OReplace function is a UDF that will replace certain characters with others.

Date Functions

Date Functions

Date, Time, and Current_Timestamp Keywords



Above are the keywords you can utilize to get the date, time, or timestamp. These are reserved words that the system will deliver to you when requested. The Keyword TIMESTAMP will fail. Notice the -04:00 at the end of the ANSI_Timestamp. That is the time zone offset. This system is 4 hours behind Greenwich Mean Time (GMT).

Dates are stored internally as INTEGERS from a Formula

```
INTEGERDATE = ((Year -1900) * 10000) + (Month * 100) + Day

    /* Example -Tom's Birthday January 10, 1959*/

INTEGERDATE = ((1959 - 1900) * 59 <-- Year Portion
            * 10000) = 590000
          + (Month * 100) = 590100 <-- Month Portion
              + Day = 590110 <-- Day Portion

    /* Example -Tom's Birthday January 10, 1999 */
                  990110

    /* Example -Tom's Birthday January 10, 2000 */
                 1000110

    /* Send Tom a birthday present on January 10, 2014 */
                 1140110
```

The reason the Smart Calendar works so well is that it stores EVERY date in Teradata as something known as an INTEGERDATE.

Displaying Dates for INTEGERDATE and ANSIDATE

```
SELECT Date              AS "Date"
      ,Current_Date      AS Display_Date
```

INTEGERDATE (YYMMDD) ANSIDATE(YYYY-MM-DD)

June 30, 2012 June 30, 2012

Date	Display_Date	Date	Display_Date
12/06/30	12/06/30	2012-06-30	2012-06-30

NEXUS Query Chameleon MM/DD/YYYY

Date	Display_Date
06/30/2012	06/30/2012

Teradata in release V2R3 defaulted to a display of YY/MM/DD. This is called the INTEGERDATE. This can be changed to ANSIDATE, which is YYYY-MM-DD for a specific session or by Default if the DBA changes the DATEFORM in DBS Control. This has nothing to do with how the date is stored internally. It has to do with the display of dates when using any ODBC tool or load utility. Above are some examples.

DATEFORM

DATEFORM Controls the default display of dates.

DATEFORM display choices are either INTEGERDATE or ANSIDATE.

INTEGERDATE is (YY/MM/DD) and ANSIDATE is (YYYY-MM-DD).

DATEFORM is the expected format for import and export of dates in Load Utilities.

Can be over-ridden by USER or within a Session at any time.

The Default can be changed by the DBA by changing the DATEFORM in DBSControl.

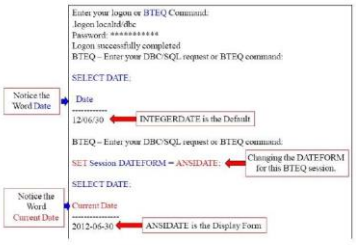
INTEGERDATE (YY/MM/DD) June 30, 2012

ANSIDATE(YYYY-MM-DD) June 30, 2012

Date	INTEGERDATE	Date	ANSIDATE
12/06/30	12/06/30	2012-06-30	2012-06-30

Teradata in release V2R3 defaulted to a display of YY/MM/DD. This is called the INTEGERDATE. This can be changed to ANSIDATE, which is YYYY-MM-DD for a specific session or by Default if the DBA changes the DATEFORM in DBS Control. This has nothing to do with how the date is stored internally. It has to do with the display of dates when using any ODBC tool or load utility.

Changing the DATEFORM in Client Utilities such as BTEQ



Date, Time, and Timestamp Recap

```
SELECT Date          AS "Date"
      ,Current_Date  AS ANSI_Date
```

INTEGERDATE (YY/MM/DD) — June 30, 2012

ANSIDATE (YYYY-MM-DD) — June 30, 2012

Date	ANSI_Date
12/06/30	12/06/30

Date	ANSI_Date
2012-06-30	2012-06-30

Dates are converted to an integer through a formula before being stored.

Dates are displayed by default as INTEGERDATE YY/MM/DD.

The DBA can set up the system to display as ANSIDATE YYYY-MM-DD.

Keywords Date or Current_Date will return the date automatically.

Time,Current_Time and Current_Timestamp are keywords to return time.

The Nexus Query Chameleon displays dates as MM/DD/YYYY.

Timestamp Differences

```
SELECT Current_Timestamp(0) AS Col1
      ,Current_Timestamp(6) AS Col2
```

Answer Set

Col1	Col2
2011/03/22 10:34:44	2011/03/22 10:34:44.123456

Date Space Time — Milliseconds

A timestamp has the date separated by a space and the time. In our second example, we have asked for 6 milliseconds.

Finding the Number of Hours between Timestamps

Create the Table

```
Create Table Car_Maintenance
(Car_ID           Integer
,Start_Timestamp  Timestamp
,End_Timestamp    Timestamp
) Primary Index (Car_ID) ;
```

Load Test Data

```
INSERT INTO Car_Maintenance
( 1, Current_Timestamp - Interval '7' day, Current_Timestamp);
INSERT INTO Car_Maintenance
( 2, Current_Timestamp - Interval '1' day, Current_Timestamp);
INSERT INTO Car_Maintenance
( 3, Current_Timestamp - Interval '2' day, Current_Timestamp);
INSERT INTO Car_Maintenance
( 4, Current_Timestamp - Interval '3' day, Current_Timestamp);
```

```
SELECT Car_ID , Start_Timestamp, End_Timestamp
 , (CAST(End_Timestamp AS DATE) -CAST(Start_Timestamp AS DATE)) * 24.0
 + (EXTRACT(HOUR FROM End_Timestamp)   -EXTRACT(HOUR FROM Start_Timestamp)) * 1.0
 + (EXTRACT(MINUTE FROM End_Timestamp) -EXTRACT(MINUTE FROM Start_Timestamp)) / 60.0
 + (EXTRACT(SECOND FROM End_Timestamp) -EXTRACT(SECOND FROM Start_Timestamp)) / 3600.0
                                                                    AS Hours
FROM  Car_Maintenance ORDER BY Car_ID ;
```

Car_ID	Start_Timestamp	End_Timestamp	Hours
1	08/07/2013 3:14:19.000000	08/14/2013 3:14:19.000000	168.000000
2	08/13/2013 3:13:12.640000	08/14/2013 3:13:12.640000	24.000000
3	08/12/2013 3:13:27.370000	08/14/2013 3:13:27.370000	48.000000
4	08/11/2013 3:13:39.040000	08/14/2013 3:13:39.040000	72.000000

The above example is how you find the number of hours between Timestamps.

Troubleshooting Timestamp

```
SELECT Timestamp(0) AS Col1
     , Timestamp(6) AS Col2
```

Error

There is Date and Current_Date (both work).

There is Time and Current_Time (both work).

There is NO Timestamp, but only Current_Timestamp!

There is NO Timestamp KEYWORD, but only ANSI's Current_Timestamp!

Add or Subtract Days from a date

```
SELECT Order_Date
      ,Order_Date + 40 as "Due Date"
      ,Order_Total
```

```
      ,"Due date" -10 as Discount
      ,Order_Total *.98 (FORMAT '$$$$,$$$.99')
FROM  Order_Table
ORDER BY 1 ;
```

Order_Date	Due_Date	Order_Total	Discount	Discounted
05/04/1999	07/03/1999	12347.53	06/23/1999	12100.57
01/01/1999	03/02/1999	8005.91	02/20/1999	7845.79
09/08/1999	11/08/1999	23454.84	10/28/1999	22985.74
10/01/1999	11/30/1999	5111.47	11/20/1999	5009.24
10/10/1999	12/09/1999	15231.62	11/29/1999	14926.98

When you add or subtract from a Date you are adding/subtracting Days

Because Dates are stored internally on disk as integers, it makes it easy to add days to the calendar. In the query above, we are adding 60 days to the Order_Date.

A Summary of Math Operations on Dates

1. DATE - DATE = Interval (days between dates)
2. DATE + or - Integer = Date

Let's find the number of days Tera-Tom has been alive since his 2014 birthday.

```
SELECT (1140110(date)) - (590110 (date)) (Title 'Tera-Tom's Age In Days') ;
```

Tera-Tom's Age In Days
20089

Below is the same exact query, but with a clearer example of the dates.

```
SELECT ('2014-01-10' (date)) -('1959-01-10' (date)) (Title 'Tera-Tom's Age In Days');
```

Tera-Tom's Age In Days
20089

A DATE – DATE is an interval of days between dates. A DATE + or - Integer = Date. Both queries above perform the same function; but the top query uses the internal date functions, and the query on the bottom does dates the traditional way.

Using a Math Operation to find your Age in Years

1. DATE - DATE = Interval (days between dates)
2. DATE + or - Integer = Date

Let's find the number of days Tera-Tom has been alive since his 2014 birthday.

```
SELECT (1140110(date)) - (590110 (date)) (Title 'Tera-Tom's Age In Days') ;
```

Tera-Tom's Age In Days
20089

Let's find the number of years Tera-Tom has been alive since his 2014 birthday.

```
SELECT ((1140110(date)) - (590110 (date))) / 365 (Title 'Tera-Tom's Age In Years') ;
```

Tera-Tom's Age In Days
55

A DATE – DATE is an interval of days between dates. A DATE + or - Integer = Date. Both queries above perform a Date function; but the top query brings back Tom's age in days, and the bottom query brings back Tom's age in years.

Find What Day of the week you were Born

Let's find the actual day of the week Tern-Tom was born

```
SEL 'Tern-Tom was born on day ' (((990110(date)) - (101(date))) MOD 7  (TITLE '');
```

Tern-Tom was born on day 5

This will produce No Title

Result	Day of the Week
0	Monday
1	Tuesday
2	Wednesday
3	Thursday
4	Friday
5	Saturday
6	Sunday

This chart can be used in conjunction with the above SQL

The above subtraction results in the number of days between the two dates. Then the MOD 7 divides by 7 to get rid of the number of weeks and results in the remainder. A MOD 7 can only result in values 0 thru 6 (always 1 less than the MOD operator). Since January 1, 1900 (101(date)) is a Monday, Tom was born on a Saturday.

The ADD_MONTHS Command

Order_Table

Order_Number	Customer_Number	Order_Date	Order_Total
123456	11111111	1998/05/04	12347.53
123512	11111111	1999/01/01	8005.91
123552	31323134	1999/10/01	5111.47
123585	87323456	1999/10/10	15231.62
123777	57896893	1999/09/09	23454.54

```
SELECT Order_Date
      ,Add_Months (Order_Date,2) as "Due Date2"
      ,Order_Total
FROM   Order_Table ORDER BY 1 ;
```

Order_Date	Due Date2	Order_Total
05/04/1998	07/04/1998	12347.53
01/01/1999	03/01/1999	8005.91
09/09/1999	11/09/1999	23454.54
10/01/1999	12/01/1999	5111.47
10/10/1999	12/10/1999	15231.62

This is the Add_Months Command. What you can do with it is add a month or many months to your columns date. Can you convert this to one year?

Using the ADD_MONTHS Command to Add 1 Year

Order_Table

Order_Number	Customer_Number	Order_Date	Order_Total
123456	11111111	1998/05/04	12347.53
123512	11111111	1999/01/01	8005.91

123552	31323134	1999/10/01	5111.47
123585	87323456	1999/10/10	15231.62
123777	57896883	1999/09/09	23454.54

```
SELECT Order_Date
      ,Add_Months (Order_Date,12) as "Due Date"
      ,Order_Total
FROM   Order_Table
ORDER BY 1 ;
```

There is no Add_Year command, so put in 12 months for 1-year

The Add_Months command adds months to any date. Above we used a great technique that would give us 1 year. Can you give me 5 years?

Using the ADD_MONTHS Command to Add 5 Years

Order_Table

Order_Number	Customer_Number	Order_Date	Order_Total
123456	11111111	1998/05/04	12347.53
123512	11111111	1999/01/01	9005.92
123552	31323134	1999/10/01	5111.47
123585	87323456	1999/10/10	15231.62
123777	57896883	1999/09/09	23454.54

```
SELECT Order_Date
      ,Add_Months (Order_Date,12 * 5) as "Due Date"
      ,Order_Total
FROM   Order_Table
ORDER BY 1 ;
```

In this example we multiplied 12 months times 5 for a total of 5 years!

Above you see a great technique for adding multiple years to a date. Can you now SELECT only the orders in September?

The EXTRACT Command

Order_Table

Order_Number	Customer_Number	Order_Date	Order_Total
123456	11111111	1998/05/04	12347.53
123512	11111111	1999/01/01	9005.92
123552	31323134	1999/10/01	5111.47
123585	87323456	1999/10/10	15231.62
123777	57896883	1999/09/09	23454.54

```
SELECT Order_Date
      ,Add_Months (Order_Date,12 * 5) as "Due Date"
      ,Order_Total
FROM   Order_Table
WHERE  EXTRACT(Month from Order_Date) = 9
ORDER BY 1 ;
```

The EXTRACT command extracts portions of Date, Time, and Timestamp.

This is the Extract command. It extracts a portion of the date. It can be used in the SELECT list, the WHERE Clause, or the ORDER BY Clause!

EXTRACT from DATES and TIME

```
SELECT Current_Date
      ,EXTRACT(Year from Current_Date) as Yr
      ,EXTRACT(Month from Current_Date) as Mo
      ,EXTRACT(Day from Current_Date) as Da
      ,Current_Time
      ,EXTRACT(Hour from Current_Time) as Hr
      ,EXTRACT(Minute from Current_Time) as Mn
      ,EXTRACT(Second from Current_Time) as Sc
      ,EXTRACT(TIMEZONE_HOUR from Current_Time) as Th
      ,EXTRACT(TimeZONE_MINUTE from Current_Time) as Tm ;
```

Answer Set

Current_Date	Yr	Mo	Da	Current_Time (CT)	Hr	Mn	Sc	Th	Tm
2013/03/22	2013	03	22	20:01:14 20	1	14	0	0	0

Just like the Add_Months, the EXTRACT Command is a Temporal Function or a Time-Based Function.

CURRENT_DATE and EXTRACT or Current_Date and Math

```
SELECT Current_Date
      ,EXTRACT(Year from Current_Date) as Yr
      ,EXTRACT(Month from Current_Date) as Mo
      ,EXTRACT(Day from Current_Date) as Da
      ,Current_Date/ 10000 +1900 as YrMath
      ,(Current_Date / 100) Mod 100 as MoMath
      ,Current_Date Mod 100 as DayMath ;
```

Math can be used to extract portions of a Date!

Answer Set

Current_Date	Yr	Mo	Day	YrMath	MoMath	DayMath
2013/03/22	2013	03	22	2013	03	22

The Extract Temporal Function can be used to extract a portion of a date. As you can see, Basic Arithmetic accomplishes the same thing.

CAST the Date of January 1, 2011 and the Year 1800

```
SELECT
 cast('2011-01-01' as date)  as ANSI_Literal
,cast(1110101 as date)       as INTEGER_Literal
,cast('11-01-01' as date)    as YY_Literal
,cast(Date '2011-01-01' as Integer) as Dates_Stored
,cast(Date '1800-01-01' as Integer) as Dates_1800s
```

Answer Set

ANSI_Literal	INTEGER_Literal	YY_Literal	Dates_Stored	Dates_1800s
01/01/2011	01/01/2011	01/01/1911	111010	-999999

The Convert and Store (CAST) command is used to give columns a different data type temporarily for the life of the query.

Notice our dates and how they're stored.

The System Calendar

Teradata systems have a table called Caldates.

Caldates has only one column in it called Cdates.

Cdates is a date column that contains a row for each date starting from January 1, 1900 to December 31, 2100.

No user can access the table Caldates directly.

Views in the Sys_Calendar database accesses Caldates.

A view called Calendar is how USERS work with the calendar.

Users use Sys_Calendar.Calendar for advanced dates.

In every Teradata system, there is something known as a System Calendar (or as Teradata calls it "Sys_Calendar.Calendar"). Get ready for AWESOME!

Using the System Calendar in Its Simplest Form

```
SELECT * FROM Sys_Calendar.Calendar
WHERE Calendar_Date = '1959-01-10';
```

Birthday of Tera-Tom

```
Calendar_Date = 01/10/1959'
day_of_week = 7 (Sunday = 1)
day_of_month = 10
day_of_year = 10
day_of_calendar = 21559 (since Jan 1, 1900)
weekday_of_month = 2
week_of_month = 1 (0 for partial week for any month not starting with Sunday)
week_of_year = 1
week_of_calendar = 3079 (since Jan 1, 1900)
month_of_quarter = 1
month_of_year = 1
month_of_calendar = 709 (since Jan 1, 1900)
quarter_of_year = 1
quarter_of_calendar = 237 (since Jan 1, 1900)
year_of_calendar = 1959
```

Tera-Tom was born on a Saturday! It was the first full week of the month, the first full week of the year, and it was the first quarter of the year!

How to really use the Sys_Calendar.Calendar

```
DATE  '1999-01-10' is stored as 990110
DATE '2000-01-10' is stored as 1000110
```

4 bytes store Date_col internally because dates are considered a 4-byte integer.

Storing Time Internally

```
CREATE SET TABLE TIMEZONE_table ,FALLBACK ,
  NO BEFORE JOURNAL,
  NO AFTER JOURNAL,
  CHECKSUM = DEFAULT
   (Date_col             Date,
    TIME_col             TIME(6),
    TIMETIMEZONE_col1    TIME(6) WITH TIME ZONE,
    TIMESTAMP_col        TIMESTAMP(6),
    TIMEZONE_col         TIMESTAMP(6) WITH TIME ZONE)
UNIQUE PRIMARY INDEX (TIMEZONE_col) ;
```

```
Time(n) stored as HHMMSS.nnnnnn
```

It takes 6 bytes to store Time_col internally.

Storing TIME with TIME ZONE Internally

```
CREATE SET TABLE TIMEZONE_table ,FALLBACK ,
  NO BEFORE JOURNAL,
  NO AFTER JOURNAL ,
  CHECKSUM = DEFAULT
   (Date_col            Date,
    TIME_col            TIME(6),
    TIMETIMEZONE_col    TIME(6) WITH TIME ZONE,
    TIMESTAMP_col       TIMESTAMP(6),
    TIMEZONE_col        TIMESTAMP(6) WITH TIME ZONE)
UNIQUE PRIMARY INDEX (TIMEZONE_col) ;
```

```
Time(n) WITH ZONE stored as HHMMSS.nnnnnn+HHMM
```

It takes 8 bytes to store TimeTimezone_col internally.

Storing Timestamp Internally

```
CREATE SET TABLE TIMEZONE_table ,FALLBACK ,
  NO BEFORE JOURNAL,
  NO AFTER JOURNAL ,
  CHECKSUM = DEFAULT
  (Date_col            Date,
   TIME_col            TIME(6),
   TIMETIMEZONE_col    TIME(6) WITH TIME ZONE,
   TIMESTAMP_col       TIMESTAMP(6),
   TIMEZONE_col        TIMESTAMP(6) WITH TIME ZONE)
UNIQUE PRIMARY INDEX (TIMEZONE_col) ;
```

```
TimeStamp(n) stored as YYMMDDHHMMSS.nnnnnn
```

It takes **10 bytes** to store TimeStamp_col internally.

Storing Timestamp with TIME ZONE Internally

```
CREATE SET TABLE TIMEZONE_table ,FALLBACK ,
  NO BEFORE JOURNAL,
  NO AFTER JOURNAL ,
  CHECKSUM = DEFAULT
   (Date_col           Date,
    TIME_col           TIME(6),
    TIMETIMEZONE_col   TIME(6) WITH TIME ZONE,
    TIMESTAMP_col      TIMESTAMP(6),
    TIMEZONE_col       TIMESTAMP(6) WITH TIME ZONE)
UNIQUE PRIMARY INDEX (TIMEZONE_col);
```

```
TimeStamp(n) With Zone stored as YYMMDDHHMMSS.nnnnnn+HHMM
```

It will take 12 bytes to store Timezone_col internally.

Storing Date, Time, and Timestamp with Zone Internally

```
CREATE SET TABLE TIMEZONE_table ,FALLBACK ,
  NO BEFORE JOURNAL,
  NO AFTER JOURNAL ,
  CHECKSUM = DEFAULT
  (Date_col           Date,
   TIME_col           TIME(6),
   TIMETIMEZONE_col   TIME(6) WITH TIME ZONE,
   TIMESTAMP_col      TIMESTAMP(6),
   TIMEZONE_col       TIMESTAMP(6) WITH TIME ZONE)
UNIQUE PRIMARY INDEX (TIMEZONE_col) ;
```

Date	Stored Internally	4 Bytes
Time(n)	Stored Internally	6 Bytes
Time(n) With Zone	Stored Internally	8 Bytes
Timestamp(n	Stored Internally	10 Bytes
Timestamp(n) with zone	Stored Internally	12 Bytes

Each data type increases its internal storage by 2 bytes.

Time Zones

A time zone relative to London(UTC) might be:

```
LA---------Miami----------Frankfurt-----------Hong Kong
 +8:00       +05:00          00:00             -08:00
```

A time zone relative to New York (EST) might be:

```
LA---------Miami----------Frankfurt-----------Hong Kong
 +3:00       00:00          -05:00             -13:00
```

Time zones are set either at the system level (DBS Control), the user level (when user is created or modified), or at the session level as an override.

Teradata has the ability to access and store both the hours and the minutes reflecting the difference between the user's time zone and the system time zone. From a World perspective, this difference is normally the number of hours between a specific location on Earth and the United Kingdom location that was historically called Greenwich Mean Time (GMT). Since the Greenwich observatory has been decommissioned, the new reference to this same time zone is called Universal Time Coordinate (UTC).

Setting Time Zones

A Time Zone should be established for the system and every user in each different time zone.

Setting the system default time zone is done by the DBA in the DBSControl record:

```
MODIFY GENERAL 16 = x   /* Hours,   n= -12 to 13   */
MODIFY GENERAL 17 = x   /* Minutes, n = -59 to 59  */
```

Setting a User's time zone requires choosing either LOCAL, NULL, or an explicit value:

```
CREATE USER  Tera-Zone
TIME ZONE    = LOCAL    /* use system level   */
             = NULL     /* no default, set to system or session level at logon */
             = '16:00'  /* explicit setting   */
             = '-06:30' /* explicit setting   */
```

Setting a Session's time zone:

```
SET TIME ZONE LOCAL ; /* use system level */
SET TIME ZONE USER ;  /* use user level   */
SET TIME ZONE INTERVAL '08:00' HOUR TO MINUTE ; /* explicit setting */
```

A Teradata session can modify the time zone without requiring a logoff and logon.

Seeing your Time Zone

Help Session ;

User Name	Account Name	Logon Date	Logon Time	Current Database	Collation	Char Set	Transaction Semantics	Current Dateform	Session Time Zone
DBC	DBC	12/06/17	15:55:39	SQL_CLASS	ASCII	ASCII	Teradata	IntegerDate	00:00

Not all output is displayed above from the HELP Session

A user's time zone is now part of the information maintained by Teradata. The settings can be seen in the extended information available in the HELP SESSION request. Teradata converts all TIME and TIMESTAMP values to Universal Time Coordinate (UTC) prior to storing them. All operations, including hashing, collation, and comparisons that act on TIME and TIMESTAMP values, are performed using their UTC forms. This will allow users to CAST the information to their local times.

Creating a Sample Table for Time Zone Examples

```
CREATE TABLE Tstamp_Test
 (
  TS_Zone CHAR(3)
 ,TS_With_Zone TIMESTAMP(6) WITH TIME ZONE
 ,TS_Without_Zone TIMESTAMP(6)
 )
UNIQUE PRIMARY INDEX ( TS_Zone );
```

A user's time zone is now part of the information maintained by Teradata. The settings can be seen in the extended information available in the HELP SESSION request.

Inserting Rows in the Sample Table for Time Zone Examples

```
Enter your logon or BTEQ Command:
.logon localtd/dbc
Password: 3536373839*
Logon successfully completed
BTEQ - Enter your DBC/SQL request or BTEQ command:

        INSERT INTO Tstamp_Test ('EST', timestamp '2000-10-01 08:12:00',
                                        timestamp '2000-10-01 08:12:00');

SET TIME ZONE INTERVAL '05:00' HOUR TO MINUTE ;
        INSERT INTO Tstamp_Test ('UTC', timestamp '2000-10-01 08:12:00',
                                        timestamp '2000-10-01 08:12:00');

SET TIME ZONE INTERVAL -'03:00' HOUR TO MINUTE ;
        INSERT INTO Tstamp_Test ('PST', timestamp '2000-10-01 08:12:00',
                                        timestamp '2000-10-01 08:12:00');

SET TIME ZONE INTERVAL -'11:00' HOUR TO MINUTE ;
        INSERT INTO Tstamp_Test ('NZT', timestamp '2000-10-01 08:12:00',
                                        timestamp '2000-10-01 08:12:00');
```

Selecting the Data from our Time Zone Table

```
SELECT * FROM Tstamp_Test ;
```

TS_Zone	TS_with_Zone	TS_Without_Zone
UTC	2000-10-01 08:12:00.000000+05:00	2000-10-01 08:12:00.000000
EST	2000-10-01 08:12:00.000000+00:00	2000-10-01 08:12:00.000000
PST	2000-10-01 08:12:00.000000-03:00	2000-10-01 08:12:00.000000
HKT	2000-10-01 08:12:00.000000-11:00	2000-10-01 08:12:00.000000

Notice the Accompanying Time Zone Offsets

Our Insert statements were done at 08:12:00 exactly. Notice the Time Zone offsets in the column TS_with_Zone and how they're not there for the column TS_Without_Zone. Teradata converts all TIME and TIMESTAMP values to Universal Time Coordinate (UTC) prior to storing them. All operations, including hashing, collation, and comparisons that act on TIME and TIMESTAMP values, are performed using their UTC forms. This will allow users to CAST the information to their local times.

Normalizing our Time Zone Table with a CAST

```
SELECT TS_Zone, TS_with_Zone
    ,CAST(TS_with_Zone AS TIMESTAMP(6)) AS T_Normal
FROM Tstamp_Test ORDER BY 3 ;
```

TS_Zone	TS_with_Zone	T_Normal
UTC	2000-10-01 08:12:00.000000-05:00	2000-10-01 03:12:00.000000
EST	2000-10-01 08:12:00.000000+00:00	2000-10-01 08:12:00.000000
PST	2000-10-01 08:12:00.000000-03:00	2000-10-01 11:12:00.000000
HKT	2000-10-01 08:12:00.000000-11:00	2000-10-01 19:12:00.000000

The System is on EST Time. The New Times are Normalized to the time zone of the System!

Notice that the Time Zone value was added to, or subtracted from, the time portion of the time stamp to adjust it to a perspective of the same time zone. As a result, it has normalized the different Times Zones in respect to the system time.

As an illustration, when the transaction occurred at 8:12 AM locally in the PST Time Zone, it was already 11:12 AM in EST, the location of the system. The times in the columns have been normalized in respect to the time zone of the system.

Intervals for Date, Time and Timestamp

Interval Chart

Simple Intervals	More involved Intervals
YEAR	DAY TO HOUR
MONTH	DAY TO MINUTE
DAY	DAY TO SECOND
HOUR	HOUR TO MINUTE
MINUTE	HOUR TO SECOND
SECOND	MINUTE TO SECOND

To make Teradata SQL more ANSI compliant and compatible with other RDBMS SQL, Teradata has added INTERVAL processing. Intervals are used to perform DATE, TIME and TIMESTAMP arithmetic and conversion.

Although Teradata allowed arithmetic on DATE and TIME, it was not performed in accordance to ANSI standards and

therefore, an extension instead of a standard. With INTERVAL being a standard instead of an extension, more SQL can be ported directly from an ANSI compliant database to Teradata without conversion.

Interval Data Types and the Bytes to Store Them

Interval Chart

Bytes	Data Type	Comments
2	INTERVAL YEAR	
4	INTERVAL YEAR TO MONTH	
2	INTERVAL MONTH	
2	INTERVAL MONTH TO DAY	
2	INTERVAL DAY	
8	INTERVAL DAY TO MINUTE	
10/12	INTERVAL DAY TO SECOND	10 for 3 2-bit systems; 12 for 64-bit
2	INTERVAL HOUR 2	
4	INTERVAL HOUR TO MINUTE 4	
8	INTERVAL HOUR TO SECOND 8	
2	INTERVAL MINUTE 2	
6/8	INTERVAL MINUTE TO SECOND	6 for 32-bit systems; 8 for 64-bit
6/8	INTERVAL SECOND	6 for 32-bit systems; 8 for 64-bit

The Basics of a Simple Interval

```
SELECT Current_Date as Our_Date
      ,Current_Date + Interval '1' Day   as Plus_1_Day
      ,Current_Date + Interval '3' Month as Plus_3_Months
      ,Current_Date + Interval '5' Year  as Plus_5_Years ;
```

Our_Date	Plus_1_Day	Plus_3_Months	Plus_5_Years
06/18/2013	06/19/2013	09/18/2013	06/18/2018

In the example SQL above, we take a simple date and add 1 day, 3 months, and 5 years. Notice that our current_date is 06/18/2012 and that our intervals come out perfectly.

Troubleshooting the Basics of a Simple Interval

```
SELECT Date '2012-01-29' as Our_Date
      ,Date '2012-01-29' + INTERVAL '1' Month as Leap_Year ;
```

Our_Date	Leap_Year
01/29/2012	02/29/2012

```
SELECT Date '2011-01-29' as Our_Date
      ,Date '2011-01-29' + INTERVAL '1' Month as Leap_Year ;
```

Error–Invalid Date

The first example works because we added 1 month to the date '2012-01-29' and we got '2012-02-29'. Because this was a leap year, there actually is a date of February 29, 2012. The next example is the real point. We have a date of '2011-01-29' and we add 1-month to that. But there is no February 29th in 2011, so the query fails.

Interval Arithmetic Results

DATE and TIME arithmetic Results using intervals

DATE	- DATE	= Interval
TIME	- TIME	= Interval
TIMESTAMP	- TIMESTAMP	= Interval
DATE	- or + Interval	= DATE
TIME	- or + Interval	= TIME
TIMESTAMP	- or + Interval	= TIMESTAMP
Interval	- or + Interval	= Interval

To use DATE and TIME arithmetic, it is important to keep in mind the results of various operations. The above chart is your Interval guide.

A Date Interval Example

```
SELECT (DATE '1999-10-01' - DATE '1988-10-01') DAY  AS Actual_Days ;
```

ERROR – Interval Field Overflow

The Error occured because the default for all intervals is 2 digits.

```
SELECT (DATE '1999-10-01' - DATE '1988-10-01') DAY(4)  AS Actual_Days ;
```

Makes the output 4 digits

Actual_Days
4017

The default for all intervals is 2 digits. We received an overflow error because the Actual_Days is 4017. The second example works because we declared the output to be 4 digits (the maximum for intervals).

A Time Interval Example

Makes the output 3 digits

```
SELECT  (TIME '12:45:01' - TIME '10:10:01') HOUR        AS Actual_Hours
       ,(TIME '12:45:01' - TIME '10:10:01') MINUTE(3)   AS Actual_Minutes
       ,(TIME '12:45:01' - TIME '10:10:01') SECOND(4)   AS Actual_Seconds
       ,(TIME '12:45:01' - TIME '10:10:01') SECOND(4,4) AS Actual_Seconds4
```

Actual_Hours	Actual_Minutes	Actual_Seconds	Actual_Seconds4
2	155	9300.000000	9300.0000

```
SELECT  (TIME '12:45:01' - TIME '10:10:01') HOUR        AS Actual_Hours
       ,(TIME '12:45:01' - TIME '10:10:01') MINUTE      AS Actual_Minutes
       ,(TIME '12:45:01' - TIME '10:10:01') SECOND(4)   AS Actual_Seconds
       ,(TIME '12:45:01' - TIME '10:10:01') SECOND(4,4) AS Actual_Seconds4
```

ERROR – Interval Field Overflow

The default for all intervals is 2 digits, but notice in the top example we put in 3 digits for Minute, 4 digits for Second, and 4,4 digits for the Actual_Seconds4. If we had not, we would have received an overflow error as in the bottom example.

A - DATE Interval Example

```
SELECT Current_Date,
       INTERVAL '-2' YEAR + CURRENT_DATE as Two_years_Ago;
```

Date	Two_Year_Ago
06/18/2012	06/18/2010

The above Interval example uses a - '2' to go back in time.

A Complex Time Interval Example using CAST

Below is the syntax for using the CAST with a date:

```
SELECT CAST (<interval> AS INTERVAL <interval> )
FROM <table-name> ;
```

The following converts an INTERVAL of 6 years and 2 months to an INTERVAL number of months:

```
SELECT CAST( (INTERVAL '6-02' YEAR TO MONTH) AS INTERVAL MONTH );
```

6-02
74

The CAST function (Convert and Store) is the ANSI method for converting data from one type to another. It can also be used to convert one INTERVAL to another INTERVAL representation. Although the CAST is normally used in the SELECT list, it works in the WHERE clause for comparison purposes.

A Complex Time Interval Example using CAST

This request attempts to convert 1300 months to show the number of years and months. Why does the first example fail, but the second find success?

```
SELECT CAST(INTERVAL '1300' MONTH AS INTERVAL YEAR TO MONTH) as
YrMo;
```

ERROR

```
SELECT CAST(INTERVAL '1300' MONTH as interval YEAR(3) TO MONTH) as YrMo;
```

YrMo
108-04

The top query failed because the INTERVAL result defaults to 2-digits and we have a 3-digit answer for the year portion (108). The bottom query fixes that specifying 3-digits. The biggest advantage in using the INTERVAL processing is that SQL written on another system is now compatible with Teradata.

The OVERLAPS Command

Compatibility: Teradata Extension

The syntax of the OVERLAPS command is:

```
SELECT <literal>
       WHERE (<start-date-time>, <end-date-time>) OVERLAPS
             (<start-date-time>, <end-date-time>) ;
```

```
SELECT 'The Dates Overlap' (TITLE '')
WHERE   (DATE '2001-01-01', DATE '2001-11-30')
OVERLAPS
        (DATE '2001-10-15', DATE '2001-12-31');
```

Answer → The Dates Overlap

When working with dates and times, sometimes it is necessary to determine whether two different ranges have common points in time. Teradata provides a Boolean function to make this test for you. It is called OVERLAPS; it evaluates true if multiple points are in common, otherwise it returns a false. The literal is returned because both date ranges have from October 15 through November 30 in common.

An OVERLAPS Example that Returns No Rows

```
SELECT 'The dates overlap' (TITLE '')
WHERE  (DATE '2001-01-01', DATE '2001-11-30')
OVERLAPS
       (DATE '2001-11-30', DATE '2001-12-31') ;
```

Answer → No rows found

The above SELECT example tests two literal dates and uses the OVERLAPS to determine whether or not to display the character literal.

The literal was not selected because the ranges do not overlap. This means the common single date of November 30 does not constitute an overlap. When dates are used, 2 days must be involved. When time is used, 2 seconds must be contained in both ranges.

The OVERLAPS Command using TIME

```
SELECT 'The Times Overlap' (TITLE '')
WHERE  (TIME '08:00:00', TIME '02:00:00')
OVERLAPS
       (TIME '02:01:00', TIME '04:15:00') ;
```

Answer → The Times Overlap

The above SELECT example tests two literal times and uses the OVERLAPS to determine whether or not to display the character literal.

This is a tricky example, and it is shown to prove a point. At first glance, it appears as if this answer is incorrect because 02:01:00 looks like it starts 1 second after the first range ends. However, the system works on a 24-hour clock when a date and time (timestamp) are not used together. Therefore, the system considers the earlier time of 2AM time as the start, and the later time of 8 AM as the end of the range. Therefore, not only do they overlap, the second range is entirely contained in the first range.

The OVERLAPS Command using a NULL Value

```
SELECT 'The Times Overlap' (TITLE '')
WHERE  (TIME '10:00:00', NULL)
```

```
OVERLAPS
      (TIME '01:01:00', TIME '04:15:00') ;
```

Answer → No Rows Found

The above SELECT example tests two literal dates and uses the OVERLAPS to determine whether or not to display the character literal:

When using the OVERLAPS function, there are a couple of situations to keep in mind:

1. A single point in time, i.e. the same date, does not constitute an overlap. There must be at least one second of time in common for TIME, or one day when using DATE.
2. Using a NULL as one of the parameters, the other DATE or TIME constitutes a single point in time instead of a range.