teradata.

Teradata Vantage SQL Intermediate

Version 17.10.4

93908
Student Guide

Trademarks

Teradata Vantage SQL Intermediate

Version 17.10.4

Module 0 – Course Overview

Module 1 – OLAP Functions – Group Aggregates

Module 2 – OLAP Functions – Window Frames

Module 3 – OLAP Functions – Ranking

Module 4 – OLAP Functions – RESET WHEN

Module 5 – Scalar Subqueries

Module 6 – CREATE TABLE AS

Module 7 – Temporary Tables

Module 8 – Data Manipulation – INSERT, UPDATE, DELETE

Module 9 – Data Manipulation – MERGE INTO

Module 10 – Views

Module 11 – Macros

Module 12 – Type Cast Functions

Module 13 – Advanced String Functions

Module 14 – Data Types and Functions – Binary

Appendix A: Review Questions

Module 0: Course Overview

Teradata Vantage SQL Intermediate

Course Objectives

After successfully completing this course, you will be able to:

- Write Group Aggregates using PARTITION BY in a Window Function
- Filter the result of a Window Function using QUALIFY
- Add a frame to a Window Function to calculate ordered aggregates such as a Cumulative Sum or a Moving Average
- Access columns in a different row using Analytic Functions such as LAG and FIRST_VALUE
- Work with Ranking functions such as DENSE_RANK, CUME_DIST and PERCENTILE_CONT
- Dynamically add subpartitions to the window definition using RESET WHEN
- Handle NULLs in Window Functions
- Understand how and when to use scalar subqueries and correlated subqueries
- Clone an existing table
- Materialize the result of a Select in a new table
- Use various forms of volatile and global temporary tables
- Write SQL to modify data in a table, using the UPDATE, INSERT, DELETE and MERGE statements
- Create and use views and macros
- Convert data types using functions such as TO_CHAR, TO_DATE and TO_BYTES
- Apply advanced string functions such as CSV, STRTOK and NVP
- Apply bit manipulation functions such as BITOR and GETBIT

Course Description

Who should attend

- Power Users
- Application Developers
- Data Analysts
- Database Administrators
- Architects/Designers

Prerequisites

- To get the most out of this training, you should have the following knowledge or experience.
  - Introduction to Teradata Database
  - Teradata SQL Basics
  - Teradata SQL Intermediate

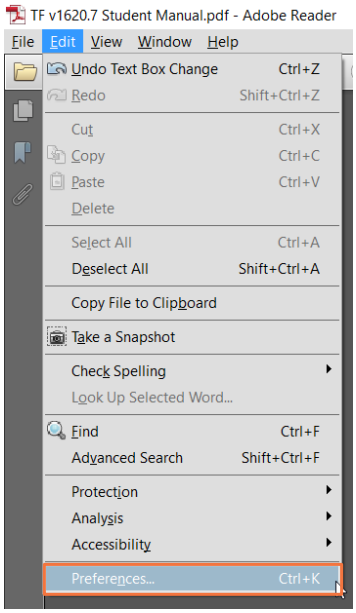
Class Format

- 6 * 4 hours Virtual Instructor Lead Training (VILT)
- Extensive hands-on labs

Course Modules

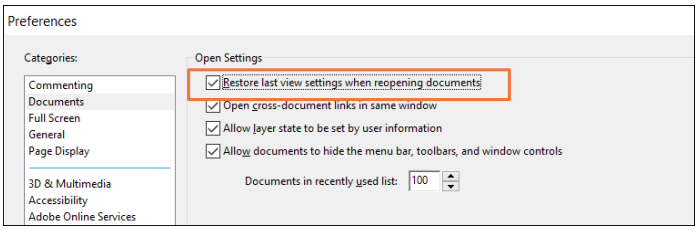
1. OLAP Functions: Group Aggregates
2. OLAP Functions: Window Frames
3. OLAP Functions: Ranking
4. OLAP Functions: RESET WHEN
5. Scalar Subqueries
6. Creating Tables from Existing Tables
7. Volatile and Global Temporary Tables
8. Data Manipulation: Insert, Update, Delete
9. Data Manipulation: Merge
10. Data Definition: Views
11. Data Definition: Macros
12. Type Cast Functions
13. Advanced String Functions
14. Bit/Byte Functions

Open PDF File at Last Page Viewed

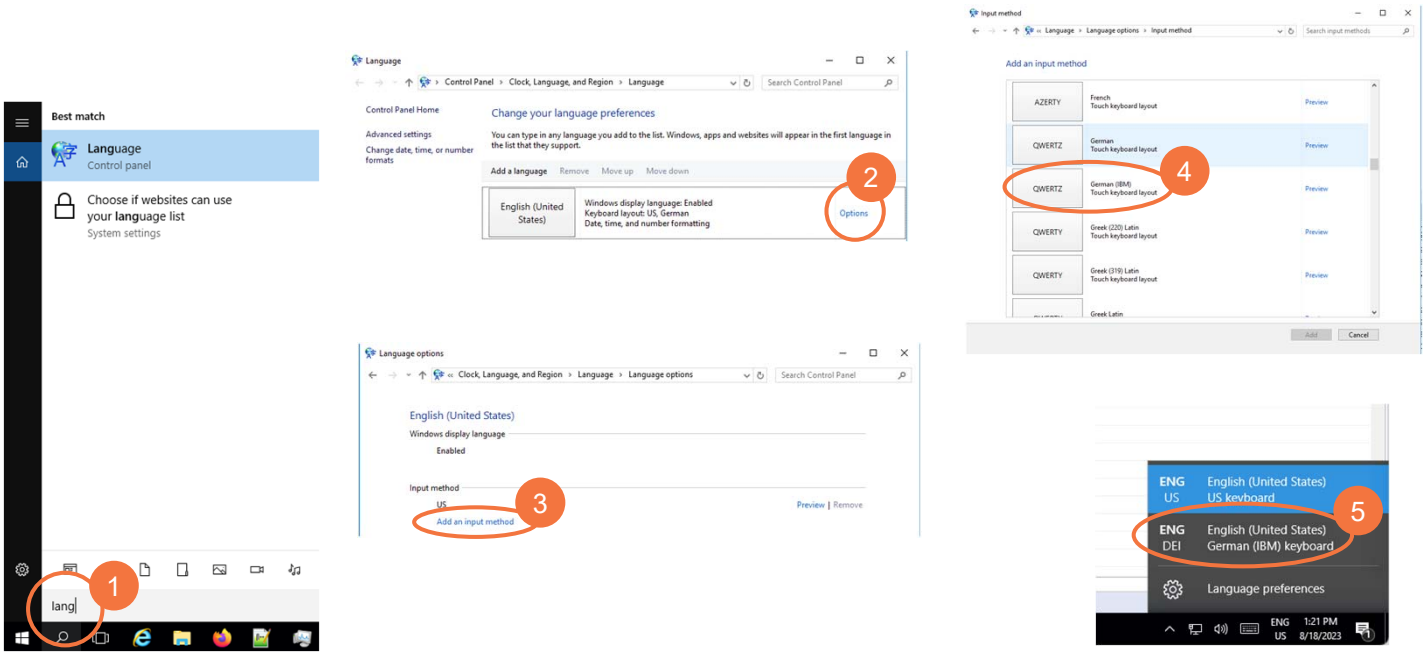


You may want to set your preferences so you can open your PDF documents at the last page that was viewed before closing the document.

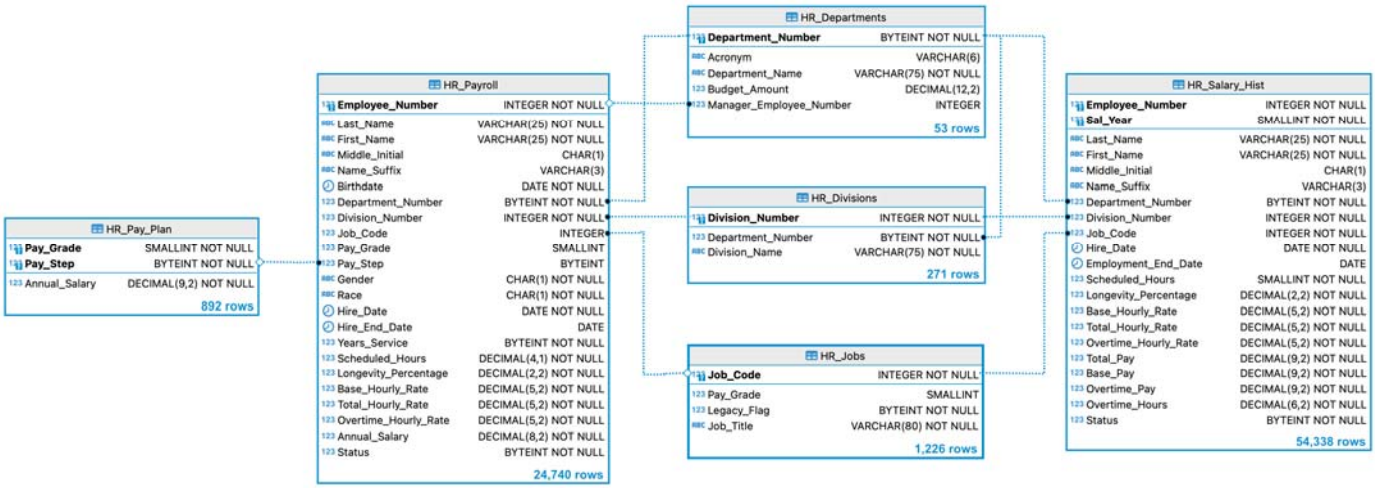
Edit > Preferences > Documents



Switching the Keyboard on AWS WorkSpaces

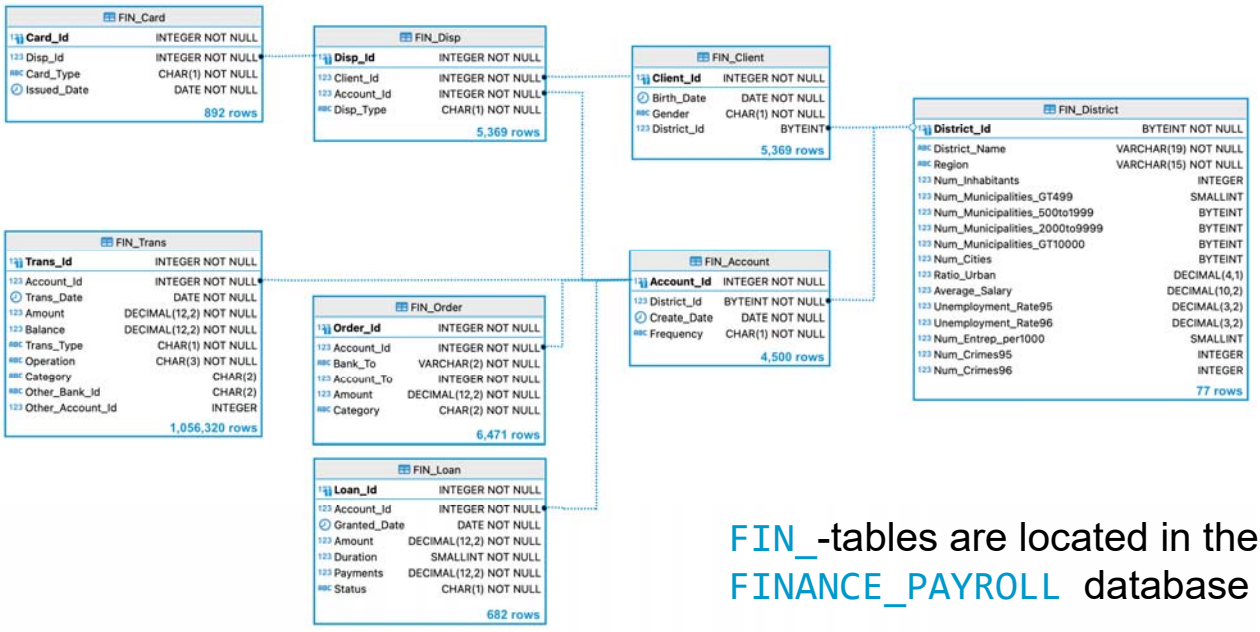


Lab Environment: Payroll Tables



HR_-tables are located in the FINANCE_PAYROLL database

Lab Environment: Financial Tables



`FIN_`-tables are located in the `FINANCE_PAYROLL` database

Module 1: OLAP Functions – Group Aggregates

Teradata Vantage SQL Intermediate

Objectives

After completing this module, you will be able to:

- Use Window Functions to perform Group Aggregates.
- Explain how the term “OVER” is used.
- Use PARTITION BY to group results into windows by one or more columns.
- Use QUALIFY to filter on Window Aggregate result sets.
- Explain the logical order of steps when processing queries.

Combining Detail and Aggregated Values

```
SELECT
   department_number  AS dept#
  ,MAX(salary_amount) AS MaxSal
  ,AVG(salary_amount) AS AvgSal
FROM employee
WHERE dept# in (301, 501)
GROUP BY department_number;
```

```
dept#   MaxSal   AvgSal
----- -------- --------
  301 29450.00 29350.00
  501 66000.00 50031.25
```

Aggregate functions do not provide row-level detail.

```
SELECT
   department_number AS dept#
  ,last_name
  ,salary_amount     AS salary
FROM employee
WHERE dept# in (301, 501);
```

```
dept# last_name   salary
----- --------- --------
  301 Kubic          NULL
  301 Kanieski   29250.00
  301 Stein      29450.00
  501 Rabbit     26500.00
  501 Wilson     53625.00
  501 Ratzlaff   54000.00
  501 Runyon     66000.00
```

No detail rows after aggregation

```
SELECT
   department_number    AS dept#
  ,last_name
  ,salary_amount        AS salary
  ,salary - AVG(salary) AS diff2avg
  ,salary - MAX(salary) AS diff2max
FROM employee
WHERE dept# in (301, 501);
```

Combining both detail and aggregated value fails.

3504 Selected non-aggregate values must be part of the associated group.

Combining Detail and Aggregated Values (cont.)

- Join to a Derived Table doing the aggregation
  - Detail and aggregate values can be used
- Disadvantage
  - Same table(s) used twice, join needed for final result
  - Joins and WHERE-conditions must be repeated

```
dept# job_code emp#  last_name    salary   diff2avg   diff2max
----- -------- ----- --------- --------- ---------- ----------
  301  312,102 1,008 Kanieski  29,250.00    -100.00    -200.00
  501  512,101 1,018 Ratzlaff  54,000.00   3,968.75 -12,000.00
  301  312,101 1,006 Stein     29,450.00     100.00       0.00
  301  311,100 1,019 Kubic           NULL       NULL       NULL
  501  512,101 1,023 Rabbit    26,500.00 -23,531.25 -39,500.00
  501  511,100 1,017 Runyon    66,000.00  15,968.75       0.00
  501  512,101 1,015 Wilson    53,625.00   3,593.75 -12,375.00
```

```
SELECT
   e.department_number AS dept#
  ,e.job_code
  ,e.employee_number   AS emp#
  ,e.last_name
  ,e.salary_amount     AS salary
  ,salary - eg.AvgSal  AS diff2avg
  ,salary - eg.MaxSal  AS diff2max
FROM employee AS e
JOIN
 ( SELECT
     department_number
    ,AVG(salary_amount) AS AvgSal
    ,MAX(salary_amount) AS MaxSal
   FROM employee
   GROUP BY department_number
   WHERE department_number IN (301, 501)
 ) AS eg
ON dept# = eg.department_number
WHERE e.department_number IN (301, 501);
```

3) We do an all-AMPs SUM step in TD_MAP1 to aggregate from TD01.employee by way of an all-rows scan with a condition of ("TD01.employee.department_number IN (301 ,501)"), grouping by field1 (TD01.employee.department_number). Aggregate Intermediate Results are computed globally, then placed in Spool 4 in TD_Map1. The size of Spool 4 is estimated with no confidence to be 2 rows (90 bytes). The estimated time for this step is 0.01 seconds.
4) We do an all-AMPs RETRIEVE step in TD_Map1 from Spool 4 (Last Use) by way of an all-rows scan into Spool 1 (used to materialize view, derived table, table function or table operator eg) (all_amps), which is built locally on the AMPs. The size of Spool 1 is estimated with no confidence to be 2 rows (82 bytes). The estimated time for this step is 0.00 seconds.
5) We do an all-AMPs RETRIEVE step in TD_Map1 from Spool 1 (Last Use) by way of an all-rows scan with a condition of ("eg.DEPARTMENT_NUMBER IN (301 ,501)") into Spool 6 (all_amps), which is duplicated on all AMPs in TD_Map1. The size of Spool 6 is estimated with no confidence to be 4 rows (132 bytes). The estimated time for this step is 0.00 seconds.
6) We do an all-AMPs JOIN step in TD_Map1 from Spool 6 (Last Use) by way of an all-rows scan, which is joined to TD01.e by way of an all-rows scan with a condition of ("TD01.e.department_number IN (301 ,501)"). Spool 6 and TD01.e are joined using a product join, with a join condition of ("TD01.e.department_number = DEPARTMENT_NUMBER"). The result goes into Spool 2 (group_amps), which is built locally on the AMPs. The size of Spool 2 is estimated with no confidence to be 6 rows (582 bytes). The estimated time for this step is 0.01 seconds.
7) Finally, we send out an END TRANSACTION step to all AMPs involved in processing the request.

```
-> The contents of Spool 2 are sent back to the user as the result of
   statement 1.  The total estimated time is 0.02 seconds.
```

Combining Detail and Aggregated Values (cont.)

- Easier using a CTE to get the base data.
  - No repeated joins or WHERE-conditions needed
- Disadvantage
  - Still same table(s) used twice, join needed for final result

dept#	job_code	emp#	last_name	salary	diff2avg	diff2max
301	312,102	1,008	Kanieski	29,250.00	-100.00	-200.00
501	512,101	1,018	Ratzlaff	54,000.00	3,968.75	-12,000.00
301	312,101	1,006	Stein	29,450.00	100.00	0.00
301	311,100	1,019	Kubic	NULL	NULL	NULL
501	512,101	1,023	Rabbit	26,500.00	-23,531.25	-39,500.00
501	511,100	1,017	Runyon	66,000.00	15,968.75	0.00
501	512,101	1,015	Wilson	53,625.00	3,593.75	-12,375.00

```
WITH cte AS
 (
   SELECT
      e.department_number AS dept#
     ,e.job_code
     ,e.employee_number   AS emp#
     ,e.last_name
     ,e.salary_amount     AS salary
   FROM employee AS e
   WHERE e.department_number IN (301, 501)
 )
SELECT
   e.*
  ,salary - eg.AvgSal AS diff2avg
  ,salary - eg.MaxSal AS diff2max
FROM cte AS e
JOIN
 (
   SELECT
      dept#
     ,Avg(salary) AS AvgSal
     ,Max(salary) AS MaxSal
   FROM cte
   GROUP BY dept#
 ) AS eg
  ON e.dept# = eg.dept#;
```

3) We do an all-AMPs SUM step in TD_MAP1 to aggregate from TD01.e by way of an all-rows scan with a condition of ("TD01.e.department_number IN (301 ,501)"), grouping by field1 (TD01.e.department_number). Aggregate Intermediate Results are computed globally, then placed in Spool 4 in TD_Map1. The size of Spool 4 is estimated with no confidence to be 2 rows (90 bytes). The estimated time for this step is 0.01 seconds.
4) We do an all-AMPs RETRIEVE step in TD_Map1 from Spool 4 (Last Use) by way of an all-rows scan into Spool 1 (used to materialize view, derived table, table function or table operator eg) (all_amps), which is built locally on the AMPs. The size of Spool 1 is estimated with no confidence to be 2 rows (82 bytes). The estimated time for this step is 0.00 seconds.
5) We do an all-AMPs RETRIEVE step in TD_Map1 from Spool 1 (Last Use) by way of an all-rows scan with a condition of ("eg.DEPT# IN (301 ,501)") into Spool 6 (all_amps), which is duplicated on all AMPs in TD_Map1. The size of Spool 6 is estimated with no confidence to be 4 rows (132 bytes). The estimated time for this step is 0.00 seconds.
6) We do an all-AMPs JOIN step in TD_Map1 from Spool 6 (Last Use) by way of an all-rows scan, which is joined to TD01.e in view cte.e by way of an all-rows scan with a condition of ("TD01.e in view cte.e.department_number IN (301 ,501)"). Spool 6 and TD01.e are joined using a product join, with a join condition of ("TD01.e.department_number = DEPT#"). The result goes into Spool 2 (group_amps), which is built locally on the AMPs. The size of Spool 2 is estimated with no confidence to be 6 rows (582 bytes). The estimated time for this step is 0.01 seconds.
7) Finally, we send out an END TRANSACTION step to all AMPs involved in processing the request.

```
-> The contents of Spool 2 are sent back to the user as the result of
   statement 1.  The total estimated time is 0.02 seconds.
```

Group Aggregates to the Rescue

- **Window functions** calculate aggregates without loosing detail rows.
- Aggregate functions can be used against a *window frame* rather than a *grouping clause.*
  - A *window* is specified by the `OVER()` phrase.
  - `PARTITION BY` creates groups of rows.
    - Equivalent to GROUP BY.
- Both detail and aggregated value available.
  - Result probably sorted by partitioning columns.
  - No guaranteed order without adding ORDER BY.

```
SELECT
   department_number                          AS dept#
  ,AVG(salary_amount) OVER (PARTITION BY dept#) AS AvgSal
  ,MAX(salary_amount) OVER (PARTITION BY dept#) AS MaxSal
  ,COUNT(*)           OVER (PARTITION BY dept#) AS CntStar
FROM employee
WHERE dept# in (301, 501);
```

```
SELECT
   department_number  AS dept#
  ,AVG(salary_amount) AS AvgSal
  ,MAX(salary_amount) AS MaxSal
  ,COUNT(*)           AS CntStar
FROM employee
WHERE dept# in (301, 501)
GROUP BY dept#;
```

dept#	AvgSal	MaxSal	CntStar
301	29350.00	29450.00	3
501	50031.25	66000.00	4

dept#	AvgSal	MaxSal	CntStar
301	29350.00	29450.00	3
301	29350.00	29450.00	3
301	29350.00	29450.00	3
501	50031.25	66000.00	4
501	50031.25	66000.00	4
501	50031.25	66000.00	4
501	50031.25	66000.00	4

The Window Feature

The ANSI SQL:2016 window feature provides a way to dynamically define a subset of data, or window, in an ordered relational database table. A window is specified by the **OVER()** phrase, which can include the following clauses inside the parentheses:

- `PARTITION BY`
- `ORDER BY`
- `RESET WHEN`
- `ROWS`

This module only covers the first clause, PARTITION BY.

Group Window Function

- Defines a group for which an aggregation is to be produced.
- Uses PARTITION BY clause to define a group that is a subset of the answer set.

Group Aggregates

- Window aggregate results can be independently mixed with detail row content.
  - Less code
  - Better performance, single access to tables

```
SELECT
   e.department_number AS dept#
  ,e.job_code
  ,e.employee_number   AS emp#
  ,e.last_name
  ,e.salary_amount     AS salary
  ,salary - eg.AvgSal  AS diff2avg
  ,salary - eg.MaxSal AS diff2max
FROM employee AS e
JOIN
 ( SELECT
     department_number
    ,AVG(salary_amount) AS AvgSal
    ,MAX(salary_amount) AS MaxSal
   FROM employee
   GROUP BY department_number
   WHERE department_number IN (301, 501)
 ) AS eg
ON dept# = eg.department_number
WHERE e.department_number IN (301, 501);
```

4 steps in Explain

```
SELECT
   department_number AS dept#
  ,job_code
  ,employee_number   AS emp#
  ,last_name
  ,salary_amount     AS salary
  ,salary
   - AVG(salary_amount)
     OVER (PARTITION BY dept#) AS diff2avg
  ,salary
   - MAX(salary_amount)
     OVER (PARTITION BY dept#) AS diff2max
FROM employee
WHERE dept# in (301, 501);
```

2 steps in Explain

dept#	job_code	emp#	last_name	salary	diff2avg	diff2max
301	312,102	1,008	Kanieski	29,250.00	-100.00	-200.00
301	311,100	1,019	Kubic	NULL	NULL	NULL
301	312,101	1,006	Stein	29,450.00	100.00	0.00
501	512,101	1,018	Ratzlaff	54,000.00	3,968.75	-12,000.00
501	512,101	1,023	Rabbit	26,500.00	-23,531.25	-39,500.00
501	511,100	1,017	Runyon	66,000.00	15,968.75	0.00
501	512,101	1,015	Wilson	53,625.00	3,593.75	-12,375.00

Window Functions are different than standard aggregate functions. Their aggregate results can be independently mixed with detail row content. The "old way" must join the same table together to perform the average, which is not as efficient, and is more difficult to write and understand.

Explain Window Aggregate

```
3) We do an all-AMPs RETRIEVE step in TD_MAP1 from TD01.employee by
   way of an all-rows scan with a condition of (
   "TD01.employee.department_number IN (301 ,501)") into Spool 2
   (all_amps), which is built locally on the AMPs.  The size of Spool
   2 is estimated with no confidence to be 7 rows (385 bytes).  The
   estimated time for this step is 0.00 seconds.
4) We do an all-AMPs STAT FUNCTION step in TD_Map1 from Spool 2 (Last
   Use) by way of an all-rows scan into Spool 5 (Last Use), which is
   redistributed by hash code to all AMPs in TD_Map1.  The result
   rows are put into Spool 1 (group_amps), which is built locally on
   the AMPs.  The size is estimated with no confidence to be 7 rows (
   504 bytes).
```

Different Partitions

- Each OVER can apply a different PARTITION BY.
  - ( ) indicates all rows, i.e., the *grand total*
  - Columns can be referenced by name or alias, but not by column position.
  - Each unique PARTITION BY results in a STAT FUNCTION step in Explain.
    - Multiple aggregates referencing the same partition are executed in the same step.

```
SELECT
   department_number AS dept#
  ,job_code
  ,employee_number   AS emp#
  ,last_name
  ,salary_amount     AS salary
  ,AVG(salary)
   OVER ()                        AS by_all
  ,AVG(salary)
   OVER (PARTITION BY dept#)      AS by_dept
  ,AVG(salary)
   OVER (PARTITION BY job_code) AS by_job
FROM employee_sales.employee AS e
WHERE dept# in (301, 501);
```

dept#	job_code	emp#	last_name	salary	by_all	by_dept	by_job
301	311100	1019	Kubic	NULL	43137.50	29350.00	NULL
301	312101	1006	Stein	29450.00	43137.50	29350.00	29450.00
301	312102	1008	Kanieski	29250.00	43137.50	29350.00	29250.00
501	511100	1017	Runyon	66000.00	43137.50	50031.25	66000.00
501	512101	1023	Rabbit	26500.00	43137.50	50031.25	44708.33
501	512101	1015	Wilson	53625.00	43137.50	50031.25	44708.33
501	512101	1018	Ratzlaff	54000.00	43137.50	50031.25	44708.33

Window Aggregates are much simpler and easier to read and understand. If you had a query with multiple calculations you would need to build many derived tables, but with Window Aggregates you would just add more function calls.

```
3) We do an all-AMPs RETRIEVE step in TD_MAP1 from employee_sales.e
   by way of an all-rows scan with a condition of (
   "employee_sales.e.department_number IN (301 ,501)") into Spool 2
   (all_amps), which is built locally on the AMPs.  The size of Spool
   2 is estimated with no confidence to be 6 rows (330 bytes).  The
   estimated time for this step is 0.00 seconds.
4) We do an all-AMPs STAT FUNCTION step in TD_Map1 from Spool 2 (Last
   Use) by way of an all-rows scan into Spool 5 (Last Use), which is
   assumed to be redistributed by value to all AMPs in TD_Map1.  The
   result rows are put into Spool 3 (all_amps), which is built
   locally on the AMPs.  The size is estimated with no confidence to
   be 6 rows (642 bytes).
5) We do an all-AMPs STAT FUNCTION step in TD_Map1 from Spool 3 (Last
   Use) by way of an all-rows scan into Spool 8 (Last Use), which is
   redistributed by hash code to all AMPs in TD_Map1.  The result
   rows are put into Spool 7 (all_amps), which is built locally on
   the AMPs.  The size is estimated with no confidence to be 6 rows (
   438 bytes).
6) We do an all-AMPs STAT FUNCTION step in TD_Map1 from Spool 7 (Last
   Use) by way of an all-rows scan into Spool 11 (Last Use), which is
   assumed to be redistributed by value to all AMPs in TD_Map1.  The
   result rows are put into Spool 1 (group_amps), which is built
   locally on the AMPs.  The size is estimated with no confidence to
   be 6 rows (480 bytes).
```

The QUALIFY Clause

- Applies conditions on the result of a Window Aggregate.
  - Just like HAVING for aggregates.
- QUALIFY is a Teradata extension to Standard SQL.
  - Without QUALIFY the Select must be wrapped in a Derived Table/CTE and then filtered using WHERE.

```
SELECT
   department_number           AS dept#
  ,job_code
  ,employee_number             AS emp#
  ,last_name
  ,salary_amount               AS salary
  ,salary
   - AVG(salary_amount)
     OVER (PARTITION BY dept#) AS diff2avg
FROM employee
WHERE dept# in (301, 501)
QUALIFY diff2avg > 0
AND COUNT(*)
    OVER (PARTITION BY dept#) > 3
ORDER BY diff2avg DESC;
```

```
dept# job_code emp# last_name   salary  diff2avg
----- -------- ---- --------- -------- ---------
  301   311100 1019 Kubic     57700.00  18900.00 X
  301   312101 1006 Stein     29450.00  -9350.00 X
  301   312102 1008 Kanieski  29250.00  -9550.00 X
  501   511100 1017 Runyon    66000.00  15968.75 ✓
  501   512101 1018 Ratzlaff  54000.00   3968.75 ✓
  501   512101 1015 Wilson    53625.00   3593.75 ✓
  501   512101 1023 Rabbit    26500.00 -23531.25 X
```

```
dept# job_code emp# last_name   salary  diff2avg
----- -------- ---- --------- -------- ---------
  501   511100 1017 Runyon    66000.00  15968.75
  501   512101 1018 Ratzlaff  54000.00   3968.75
  501   512101 1015 Wilson    53625.00   3593.75
```

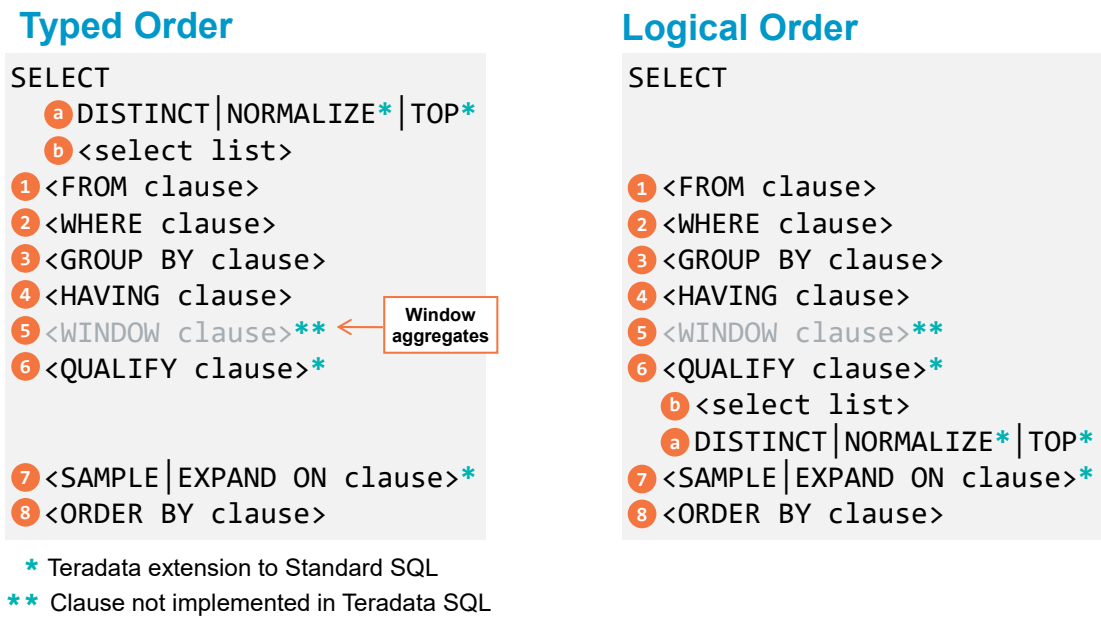
The QUALIFY clause
A conditional clause in the SELECT statement that filters results of a previously computed ordered analytical function according to user-specified search conditions.

The major difference between QUALIFY and HAVING is that with QUALIFY the filtering is based on the result of performing various ordered analytical functions on the data.

Same result using a Derived Table

```
SELECT -- column list needed because "cnt" is not to be projected
   dept#, job_code, emp#, last_name, salary, diff2avg
FROM
 ( SELECT
      department_number           AS dept#
     ,job_code
     ,employee_number             AS emp#
     ,last_name
     ,salary_amount               AS salary
     ,salary
      - AVG(salary_amount)
        OVER (PARTITION BY dept#) AS diff2avg
     ,COUNT(*)
        OVER (PARTITION BY dept#) AS cnt
   FROM employee
   WHERE dept# in (301, 501)
 ) AS dt
WHERE diff2avg > 0
  AND cnt > 3
ORDER BY diff2avg DESC;
```

Logical Query Processing Order



Logical query processing
Each select *clause* defines what is available to the clauses in subsequent steps and transforms the data step-by-step from the input tables into the final result set. And because Standard SQL works on tabular structures, rows and columns, both input and output of each clause are "tables".

But the order in which you type the query clauses is different from the order in which they get logically interpreted. Each of the following clauses take a *virtual* table as input and produce another *virtual* table as output:

① The `FROM` clause builds a *virtual* table based on the tables, views and joins defined.
② `WHERE` filters the result of the FROM.
③ `GROUP BY` partitions the data into groups, aggregates all rows of a group into a single row and returns a grouped table.
④ `HAVING` fillters the grouped table.
⑤ The `<WINDOW clause>` is part of the Window Function definition (`OVER`), but the WINDOW syntax is not implemented in Teradata. It also partitions the data into groups and applies aggregates/rankings/etc., but (unlike GROUP BY) without removing detail rows.
⑥ `QUALIFY` filters the result of Window Functions.
ⓑ The projected columns in the `SELECT` list are created.
ⓐ `DISTINCT`, `NORMALIZE` and `TOP` are mutually exclusive.
⑦ `SAMPLE` return one or more samples of rows specified.
`EXPAND ON` creates a regular time series of rows per input row.
⑧ `ORDER BY` finally sorts rows for output.

Because Teradata was implemented before there was Standard SQL Teradata's parser doesn't enforce the correct order of query clauses, in fact you can use any order as long as a query starts with SELECT. Teradata will happily execute

```
SELECT … ORDER BY … QUALIFY … FROM … HAVING … GROUP BY
```

Please don’t do this, stay with the “official” order, any fellow user trying to understand your queries or trying to port them to another DBMS will be grateful.

Nested Aggregation

- Aggregates are calculated *before* window aggregates
  - ⇒ Aggregates can be nested in window aggregates

    `MAX(AVG(salary_amount)) OVER ()` ✓
  - ⇒ Window aggregates cannot be nested in aggregates

    `AVG(MAX(salary_amount) OVER ())` ✗

```
SELECT
   department_number   AS dept#
  ,AVG(salary_amount)  AS avgsal
  ,COUNT(*)            AS cnt
  ,MAX(avgsal) OVER () AS maxavg
  ,avgsal - maxavg     AS diff2max
FROM employee
WHERE dept# BETWEEN 300 AND 499
  AND salary_amount IS NOT NULL
GROUP BY dept#
1 HAVING cnt > 1
2 QUALIFY diff2max < -5000;
```

```
dept#   salary
-----  --------
  301 29250.00
  301 29450.00
  401 24500.00
  401 25525.00
  401 36300.00
  401 37850.00
  401 43100.00
  401 46000.00
  402 52500.00
  403 31000.00
  403 31200.00
  403 37900.00
  403 43700.00
  403 49700.00
```

```
dept#   avgsal cnt  1
-----  -------- ---
  301 29350.00   2 ✓
  401 35545.83   6 ✓
  402 52500.00   1 ✗
  403 38700.00   5 ✓
```

```
dept# avgsal   cnt   maxavg diff2max  2
----- -------- --- -------- --------
  301 29350.00   2 38700.00 -9350.00 ✓
  401 35545.83   6 38700.00 -3154.17 ✗
  403 38700.00   5 38700.00     0.00 ✗
```

```
dept# avgsal   cnt   maxavg diff2max
----- -------- --- -------- --------
  301 29350.00   2 38700.00 -9350.00
```

Window aggregate functions can include aggregate functions because they are evaluated *after* the HAVING clause.

To aggregate the result of window aggregate functions a Derived Table or CTE is required.

Usage Notes

- PARTITION BY cannot reference columns by position (unlike GROUP BY or ORDER BY)
  - Not a syntax error
  - Interpreted as a *numeric literal* ⇒ All rows in a single partition
  - Same result as no PARTITION BY, but not automatically rewritten by the optimizer
  - Probably skewed to a single AMP
    - Bad runtime
    - Possible *2646 No more spool space* error

```
SELECT
  department_number AS dept#
  ,last_name
  ,salary_amount     AS salary
  ,MAX(salary_amount) OVER (PARTITION BY 1) AS MaxSal
  ,COUNT(*)           OVER (PARTITION BY 1) AS Cnt
FROM employee
WHERE dept# IN (402, 501, 999)
ORDER BY 1;
```

Invalid

Valid

dept#	last_name	salary	MaxSal	Cnt
402	Daly	52500.00	100000.00	7
402	Crane	NULL	100000.00	7
501	Runyon	66000.00	100000.00	7
501	Rabbit	26500.00	100000.00	7
501	Wilson	53625.00	100000.00	7
501	Ratzlaff	54000.00	100000.00	7
999	Trainer	100000.00	100000.00	7

Summary

Group window aggregates:

- Like aggregation, use SUM, COUNT, MIN, MAX, and AVG.
- Unlike aggregation, retain the detail data of each row.
- Can be partitioned into groups.
- Can be used with QUALIFY.
- Occur after aggregation in the order of operations.

Module 1: Review Questions

1. Which two are true about PARTITION BY?
   a. It cannot reference an aggregate function.
   b. It can reference a column by name or alias.
   c. It must reference a projected column.
   d. PARTITION BY and GROUP BY may both be present within the same projection.
2. There is an EMPLOYEE table with columns name, dept and salary.
   Which SQL statement produces a report that shows only employees having a salary greater than their departmental average salary?

```
SELECT name, dept, salary
FROM EMPLOYEE AS e
```

a.
```
GROUP BY dept
HAVING  salary > AVG(salary);
```

b.
```
WHERE   salary > AVG(salary);
```

c.
```
QUALIFY salary > AVG(salary) OVER (PARTITION BY dept);
```

d.
```
HAVING  salary > AVG(salary) OVER (PARTITION BY dept);
```

Check your understanding of the concepts discussed in this module by completing the review questions as directed by your instructor.

Module 1: Review Questions (cont.)

3. What is the correct order of query clauses?
   a. SELECT-FROM-WHERE-QUALIFY-GROUP BY-HAVING-SAMPLE-ORDER BY
   b. SELECT-FROM-WHERE-GROUP BY-HAVING-SAMPLE-QUALIFY-ORDER BY
   c. SELECT-FROM-WHERE-GROUP BY-HAVING-QUALIFY-SAMPLE-ORDER BY
   d. SELECT-FROM-WHERE-SAMPLE-GROUP BY-HAVING-QUALIFY-ORDER BY
4. What is the logical query processing order?
   a. SELECT-FROM-WHERE-GROUP BY-HAVING-QUALIFY-SAMPLE-ORDER BY
   b. FROM-WHERE-GROUP BY-HAVING-QUALIFY-SELECT-SAMPLE-ORDER BY
   c. FROM-WHERE-GROUP BY-HAVING-SAMPLE-QUALIFY-SELECT-ORDER BY
   d. SELECT-FROM-WHERE-SAMPLE-GROUP BY-HAVING-QUALIFY-ORDER BY
5. Which of the following is invalid syntax?
   a. `SUM(col) OVER (PARTITION BY 1))`
   b. `SUM(col) OVER (PARTITION BY col2 / 10))`
   c. `SUM(SUM(col)) OVER ()`
   d. `SUM(SUM(col) OVER ())`

Window Functions: PARTITION BY

Labs

Lab 1
Window Functions: PARTITION BY

teradata.

Write a query to return details of the employee earning the highest `annual_salary` for each `department_number` between 40 and 50. Order by descending salary.

```
HELP TABLE hr_payroll;
```

```
*** Query completed. 12 rows found. 5 columns returned.

   emp#  first_name last_name dept# annual_salary
-------  ---------- --------- ----- -------------
276,758  Sharon     Broome       40    175,000.28
 56,464  Carl       Dabadie      50    145,510.04
537,250  Murphy     Paul         50    145,510.04
375,721  Brian      Bernard      46    123,242.08
349,437  Eric       Romero       44    123,242.08
 38,687  Sharon     Campbell     41    106,870.14
395,030  Greta      Meche        41    106,870.14
487,384  Patti      Wallace      45    106,366.00
248,711  Russel     Smith        48    101,924.16
154,881  Barbara    Leblanc      48    101,924.16
407,720  Mertis     Edwards      47     61,224.28
378,089  Sidney     Allison      43     36,405.20
```

Lab 2
Window Functions: PARTITION BY

teradata.

Modify the previous query to add a column indicating the percentage this salary represents of all salaries for the department.
Return only rows where the percentage is over 1%.
Order by descending percentage.

```
HELP TABLE hr_payroll;
```

```
*** Query completed. 8 rows found. 7 columns returned.

   emp# first_name last_name dept# annual_salary % dept sal    dept_sal
------- ---------- --------- ----- ------------- ---------- -------------
407,720 Mertis     Edwards      47     61,224.28      13.27     461,392.10
378,089 Sidney     Allison      43     36,405.20      10.13     359,366.28
248,711 Russel     Smith        48    101,924.16       5.62   1,812,808.66
154,881 Barbara    Leblanc      48    101,924.16       5.62   1,812,808.66
487,384 Patti      Wallace      45    106,366.00       3.16   3,364,405.46
276,758 Sharon     Broome       40    175,000.28       1.95   8,986,370.62
375,721 Brian      Bernard      46    123,242.08       1.91   6,443,657.58
349,437 Eric       Romero       44    123,242.08       1.89   6,536,300.16
```

Lab 3
Window Functions: PARTITION BY

teradata.

```
HELP TABLE hr_salary_hist;
```

Write a query based on employees with a salary (total_pay) > 10000 calculating the average salary for each job_code <> 999999 and each sal_year between 2014 to 2017.
Add a column showing the difference of the salary to this average salary and return only rows where the salary is more than twice the average.
Order by descending salary within year.

```
*** Query completed. 32 rows found. 10 columns returned.
```

sal_year	emp#	first_name	last_name	dept#	job_code	avg_pay	total_pay	above_avg	overtime_hours
2014	220,779	Cory	Reech	50	5,015	82,162.76	195,736.96	113,574.20	2,626.10
2014	416,916	Arthur	Munoz	50	5,005	48,168.77	140,062.40	91,893.63	2,196.60
2014	371,521	J	Fontenot	50	5,010	73,184.04	157,886.83	84,702.79	2,120.17
2014	346,055	Mickey	Duncan	50	5,005	48,168.77	125,307.33	77,138.56	2,003.76
2014	349,461	Jason	Martin	50	5,005	48,168.77	123,804.83	75,636.06	2,004.35
2014	306,789	Samuel	White	50	5,005	48,168.77	97,912.92	49,744.15	1,161.50
2014	138,657	Stacy	Denicola	52	120,211	42,493.35	89,422.14	46,928.79	209.00
2015	371,521	J	Fontenot	50	5,010	77,540.23	180,814.81	103,274.58	2,423.25
2015	416,916	Arthur	Munoz	50	5,010	77,540.23	169,838.28	92,298.05	2,585.92
2015	346,055	Mickey	Duncan	50	5,005	52,211.69	139,672.35	87,460.66	2,203.67
2015	349,461	Jason	Martin	50	5,005	52,211.69	124,577.09	72,365.40	1,884.75
2015	306,789	Samuel	White	50	5,005	52,211.69	112,032.02	59,820.33	1,362.16
2015	322,970	Alex	Wall	20	550,921	34,676.88	87,382.76	52,705.88	0.00
2015	328,049	Roderick	Taylor	70	151,370	23,573.81	52,892.44	29,318.63	1,352.75
2015	438,391	Carol	Hornsby-Mccoy	12	400,280	20,496.65	43,640.00	23,143.35	0.00
2015	356,026	Glendora	Tate	12	400,280	20,496.65	41,317.50	20,820.85	0.00
2016	416,916	Arthur	Munoz	50	5,010	78,992.82	172,956.15	93,963.33	2,425.08
2016	349,461	Jason	Martin	50	5,005	52,188.33	124,905.60	72,717.27	1,788.41
2016	400,475	Debbie	Dean	52	123,125	50,721.98	119,468.51	68,746.53	1,266.25
2016	488,445	Calvin	Bailey	70	151,380	33,200.91	69,710.57	36,509.66	2,008.00
2016	326,054	Willie	Sheppard	70	151,370	23,576.19	54,466.08	30,889.89	969.75
2016	371,076	Terri	Parnell	70	300,011	28,437.38	58,375.16	29,937.78	0.00
2016	438,391	Carol	Hornsby-Mccoy	12	400,280	18,419.53	38,602.50	20,182.97	0.00

Lab 4
Window Functions: PARTITION BY

teradata.

Modify the previous, to show only employees who earned more than twice the average in *at least 3 of the 4 years between 2014 and 2017*.

```
HELP TABLE hr_salary_hist;
```

```
*** Query completed. 11 rows found. 10 columns returned.
```

sal_year	emp#	first_name	last_name	dept#	job_code	avg_pay	total_pay	above_avg	overtime_hours
2014	416,916	Arthur	Munoz	50	5,005	48,168.77	140,062.40	91,893.63	2,196.60
2014	349,461	Jason	Martin	50	5,005	48,168.77	123,804.83	75,636.06	2,004.35
2014	306,789	Samuel	White	50	5,005	48,168.77	97,912.92	49,744.15	1,161.50
2015	416,916	Arthur	Munoz	50	5,010	77,540.23	169,838.28	92,298.05	2,585.92
2015	349,461	Jason	Martin	50	5,005	52,211.69	124,577.09	72,365.40	1,884.75
2015	306,789	Samuel	White	50	5,005	52,211.69	112,032.02	59,820.33	1,362.16
2016	416,916	Arthur	Munoz	50	5,010	78,992.82	172,956.15	93,963.33	2,425.08
2016	349,461	Jason	Martin	50	5,005	52,188.33	124,905.60	72,717.27	1.788.41
2017	416,916	Arthur	Munoz	50	5,010	78,322.62	169,237.58	90,914.96	2,253.25
2017	349,461	Jason	Martin	50	5,005	52,643.33	130,875.53	78,232.20	1,857.25
2017	306,789	Samuel	White	50	5,005	52,643.33	107,659.12	55,015.79	1,158.75

Lab 5
Window Functions: PARTITION BY

teradata.

Write a report showing the sum of salaries (`total_pay`) per department and the percentage these sums represent of the sum of *all* salaries.

Base the calculation on data from `sal_year` 2017 only.

Return only rows where the percentage is over 2%.

Order by descending sums.

```
HELP TABLE hr_salary_hist;
HELP TABLE hr_departments;
```

```
*** Query completed. 11 rows found. 4 columns returned.

department_name                dept#          sum_pay  % all sal
------------------------------ ----- ------------- ----------
Police Department                 50 48,918,175.00      24.77
Fire Department                   51 34,842,032.22      17.64
Library Board of Control          12 15,473,693.88       7.84
Emergency Medical Services        52 13,332,944.67       6.75
Public Works                      70 11,016,505.33       5.58
Maintenance                       77  8,723,611.88       4.42
Human Development and Services    60  8,407,801.38       4.26
Finance Department                41  6,727,032.66       3.41
City Court                        20  5,643,296.53       2.86
Parish Attorney                    5  4,747,061.85       2.40
Development                       73  4,280,227.82       2.17
```

Window Functions: PARTITION BY

Lab Solutions

Lab 1 Solution
Window Functions: PARTITION BY

teradata.

Write a query to return details of the employee earning the highest `annual_salary` for each `department_number` between 40 and 50.
Order by descending salary.

```
SELECT
   employee_number AS emp#
  ,first_name
  ,last_name
  ,department_number AS dept#
  ,annual_salary
FROM hr_payroll
WHERE dept# BETWEEN 40 AND 50
QUALIFY
   annual_salary
   = Max(annual_salary)
     Over (PARTITION BY dept#)
ORDER BY annual_salary DESC
;
```

Lab 2 Solution
Window Functions: PARTITION BY

teradata.

Modify the previous query to add a column indicating the percentage this salary represents of all salaries for the department.
Return only rows where the percentage is over 1%.
Order by descending percentage.

```
SELECT
   employee_number AS emp#
  ,first_name
  ,last_name
  ,department_number AS dept#
  ,annual_salary
  ,100 * annual_salary / dept_sal AS "% dept sal"
  ,Sum(annual_salary)
   Over (PARTITION BY dept#) AS dept_sal
FROM hr_payroll
WHERE dept# BETWEEN 40 AND 50
QUALIFY
   annual_salary
   = Max(annual_salary)
     Over (PARTITION BY dept#)
  AND "% dept sal" > 1
ORDER BY "% dept sal" DESC;
```

Lab 3 Solution
Window Functions: PARTITION BY

teradata.

Write a query based on employees with a salary (total_pay) > 10000 calculating the average salary for each job_code <> 999999 and each sal_year between 2014 to 2017.
Add a column showing the difference of the salary to this average salary and return only rows where the salary is more than twice the average.
Order by descending salary within year.

```
SELECT
   sal_year
  ,employee_number AS emp#
  ,first_name
  ,last_name
  ,department_number AS dept#
  ,job_code
  ,Avg(total_pay) Over (PARTITION BY job_code, sal_year) AS avg_pay
  ,total_pay
  ,total_pay - avg_pay AS above_avg
  ,overtime_hours
FROM hr_salary_hist AS t
WHERE total_pay > 10000
  AND job_code <> 999999
  AND sal_year BETWEEN 2014 AND 2017
QUALIFY total_pay > 2 * avg_pay
ORDER BY sal_year, total_pay DESC;
```

Lab 4 Solution
Window Functions: PARTITION BY

Modify the previous to show only employees who earned more than twice the average in *at least 3 of the 4 years between 2014 and 2017*.

```
SELECT *
FROM
 (
   SELECT
      sal_year
     ,employee_number AS emp#
     ,first_name
     ,last_name
     ,department_number AS dept#
     ,job_code
     ,Avg(total_pay) Over (PARTITION BY job_code, sal_year) AS avg_pay
     ,total_pay
     ,total_pay - avg_pay AS above_avg
     ,overtime_hours
   FROM hr_salary_hist AS t
   WHERE total_pay > 10000
     AND job_code <> 999999
     AND sal_year BETWEEN 2014 AND 2017
   QUALIFY total_pay > 2 * avg_pay
 ) AS dt
QUALIFY Count(*) Over (PARTITION BY emp#) >= 3
ORDER BY sal_year, total_pay DESC;
```

Lab 5 Solution
Window Functions: PARTITION BY

- Write a report showing the sum of salaries (`total_pay`) per department and the percentage these sums represent of the sum of *all* salaries.
- Base the calculation on data from `sal_year` 2017 only.
- Return only rows where the percentage is over 2%.
- Order by descending sums.

```
SELECT
   d.department_name
  ,d.department_number AS dept#
  ,Sum(s.total_pay) AS sum_pay
  ,100 * sum_pay
   / Sum(sum_pay)
     Over () AS "% all sal"
FROM hr_salary_hist AS s
JOIN hr_departments AS d
  ON s.department_number = d.department_number
WHERE s.sal_year = 2017
GROUP BY 1,2
QUALIFY "% all sal" > 2
ORDER BY sum_pay DESC
;
```

Module 2: OLAP Functions – Window Frames

Teradata Vantage SQL Intermediate

Objectives

After completing this module, you will be able to:

- Explain how to use ROWS and ORDER BY to define a window
- Write Cumulative Window Aggregate queries
- Write Moving Window Aggregate queries
- Write Remaining Window Aggregate queries
- Use the LAG & LEAD Functions
- Use the FIRST_VALUE & LAST_VALUE Functions
- Emulate DISTINCT window aggregates

Adding a Frame to the Window

- A *Windowed Aggregate* is newly calculated for each row within the PARTITION based on all rows between a *starting* row and an *ending* row.
- This is called a *frame*, introduced by the keyword **ROWS**.
- *Starting* and *ending* row might be fixed or relative to the current row based on the following keywords:
  - **UNBOUNDED PRECEDING** ⇒ all rows before the current row ⇒ fixed start
  - **UNBOUNDED FOLLOWING** ⇒ all rows after the current row ⇒ fixed end
  - ***x* PRECEDING** ⇒ *x* rows before the current row (offset) ⇒ relative start
  - ***y* FOLLOWING** ⇒ *y* rows after the current row (offset) ⇒ relative end
  - **CURRENT ROW** ⇒ the current row
- When ROWS is ommitted there's an implicit default frame, the *Group Window:*

Default: Group Window

```
AVG(salary) OVER ( )
AVG(salary) OVER (ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING)
```

The Four Window Frame Types

- **Group** Window: Both starting and ending row are fixed, calculation is based on all rows of a partition (Syntax hardly used because it's the default anyway).
  `ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING`
- **Cumulative** Window: The starting row is fixed, calculation is based on an increasing number of rows.
  `ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW`
- **Remaining** Window: The ending row is fixed, calculation is based on a decreasing number of rows.
  `ROWS BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING`
- **Moving** Window: Both starting and ending rows are relative, calculation is based on a fixed number of rows.
  `ROWS BETWEEN x PRECEDING AND y FOLLOWING`
- The last three window frames work on a logically *ordered* dataset.
  - An `ORDER BY` clause must be added to each frame to ensure that rows are processed in a specific sequence.
  - These logical orders are independent of the final ORDER BY.

OLAP has its own aggregate functions and provides 4 basic modes of operation. With OLAP you can aggregate based on a group of rows, a cumulative set of rows, a moving set of rows and a remaining set of rows.

1. The first case is a group window – this includes every row in the window.
2. The cumulative window is all rows up to, and including, the current row.
3. The next case is the Remaining window and includes the current row and subsequent rows.
4. The last case is the Moving window. It can evaluate rows before and after the current row, as well as the current row.

It is important to remember, the last three window cases include the concept of position within the dataset. So you will need to ensure the rows are ordered. The GROUP window does not require ordering since all rows are included.

Cumulative Window

- Calculation is logically based on a steadily *increasing* number of rows.
- For each added row the aggregation is repeated based on the increased number of rows in the window.
- For better performance this logic is simplified to
  Add the current value to the result of the previous row.

```
Row 1:  1  1  1  1  1  1
Row 2:  2  2  2  2  2  2
Row 3:  3  3  3  3  3  3
Row 4:  4  4  4  4  4  4
Row 5:  5  5  5  5  5  5
Row 6:  6  6  6  6  6  6
...
```

```
SELECT
   department_number AS dept#
  ,employee_number   AS emp#
  ,last_name
  ,SUM(salary_amount)
   OVER (ORDER BY emp#
         ROWS BETWEEN UNBOUNDED PRECEDING
                  AND CURRENT ROW) AS cum_sum
  ,salary_amount AS salary
FROM employee_sales.employee AS e
WHERE dept# in (301, 501);
```

dept#	emp#	last_name	cum_sum	salary
301	1006	Stein	29,450.00	29,450.00
301	1008	Kanieski	58,700.00	+ 29,250.00
501	1015	Wilson	112,325.00	+ 53,625.00
501	1017	Runyon	178,325.00	+ 66,000.00
501	1018	Ratzlaff	232,325.00	+ 54,000.00
301	1019	Kubic	232,325.00	+ NULL
501	1023	Rabbit	258,825.00	+ 26,500.00

```
3) We do an all-AMPs RETRIEVE step in TD_MAP1 from employee_sales.e
   by way of an all-rows scan with a condition of (
   "employee_sales.e.department_number IN (301 ,501)") into Spool 2
   (all_amps), which is built locally on the AMPs.  The size of Spool
   2 is estimated with no confidence to be 6 rows (306 bytes).  The
   estimated time for this step is 0.00 seconds.
4) We do an all-AMPs STAT FUNCTION step in TD_Map1 from Spool 2 (Last
   Use) by way of an all-rows scan into Spool 5 (Last Use), which is
   assumed to be redistributed by value to all AMPs in TD_Map1.  The
   result rows are put into Spool 1 (group_amps), which is built
   locally on the AMPs.  The size is estimated with no confidence to
   be 6 rows (510 bytes).
5) Finally, we send out an END TRANSACTION step to all AMPs involved
   in processing the request.
-> The contents of Spool 1 are sent back to the user as the result of
   statement 1.
```

Cumulative Window (cont.)

- ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
  can be shortened to ROWS UNBOUNDED PRECEDING.
- Each frame can be ordered and partitioned individually.
  - Each unique combination of PARTITION/ORDER BY adds another *STAT FUNCTION step* in Explain.

```
SELECT
   department_number AS dept#
  ,employee_number   AS emp#
  ,last_name
  ,salary_amount     AS salary
  ,SUM(salary_amount) OVER (PARTITION BY dept#)                                        AS sum_by_dept
  ,SUM(salary_amount) OVER (                ORDER BY emp# DESC ROWS UNBOUNDED PRECEDING) AS cum_sum_desc
  ,SUM(salary_amount) OVER (                ORDER BY emp#      ROWS UNBOUNDED PRECEDING) AS cum_sum
FROM employee AS e
WHERE dept# in (301, 501);
```

dept#	emp#	last_name	salary	sum_by_dept	cum_sum_desc	cum_sum
301	1006	Stein	29,450.00	58,700.00	258,825.00	29,450.00
301	1008	Kanieski	29,250.00	58,700.00	229,375.00	58,700.00
501	1015	Wilson	53,625.00	200,125.00	200,125.00	112,325.00
501	1017	Runyon	66,000.00	200,125.00	146,500.00	178,325.00
501	1018	Ratzlaff	54,000.00	200,125.00	80,500.00	232,325.00
301	1019	Kubic	NULL	58,700.00	26,500.00	232,325.00
501	1023	Rabbit	26,500.00	200,125.00	26,500.00	258,825.00

```
1) First, we lock EMPLOYEE_SALES.e in TD_MAP1 for read on a reserved
   RowHash to prevent global deadlock.
2) Next, we lock EMPLOYEE_SALES.e in TD_MAP1 for read.
3) We do an all-AMPs RETRIEVE step in TD_MAP1 from EMPLOYEE_SALES.e
   by way of an all-rows scan with a condition of (
   "EMPLOYEE_SALES.e.department_number IN (301 ,501)") into Spool 2
   (all_amps), which is built locally on the AMPs.  The size of Spool
   2 is estimated with no confidence to be 6 rows (306 bytes).  The
   estimated time for this step is 0.00 seconds.
4) We do an all-AMPs STAT FUNCTION step in TD_Map1 from Spool 2 (Last
   Use) by way of an all-rows scan into Spool 5 (Last Use), which is
   redistributed by hash code to all AMPs in TD_Map1.  The result
   rows are put into Spool 3 (all_amps), which is built locally on
   the AMPs.  The size is estimated with no confidence to be 6 rows (
   390 bytes).
5) We do an all-AMPs STAT FUNCTION step in TD_Map1 from Spool 3 (Last
   Use) by way of an all-rows scan into Spool 8 (Last Use), which is
   assumed to be redistributed by value to all AMPs in TD_Map1.  The
   result rows are put into Spool 7 (all_amps), which is built
   locally on the AMPs.  The size is estimated with no confidence to
   be 6 rows (438 bytes).
6) We do an all-AMPs STAT FUNCTION step in TD_Map1 from Spool 7 (Last
   Use) by way of an all-rows scan into Spool 11 (Last Use), which is
   assumed to be redistributed by value to all AMPs in TD_Map1.  The
   result rows are put into Spool 1 (group_amps), which is built
   locally on the AMPs.  The size is estimated with no confidence to
   be 6 rows (606 bytes).
7) Finally, we send out an END TRANSACTION step to all AMPs involved
   in processing the request.
-> The contents of Spool 1 are sent back to the user as the result of
```

statement 1.

Frames and ORDER BY

- Without ORDER BY the optimizer simply sorts the frame by *all* columns.
- To get a repeatable result the ORDER BY column(s) must be unique.

```
SELECT
   department_number AS dept#
  ,employee_number   AS emp#
  ,last_name
  ,salary_amount     AS salary
  ,SUM(salary_amount)
   OVER (ORDER BY dept#
         ROWS BETWEEN UNBOUNDED PRECEDING
                  AND CURRENT ROW) AS cum_sum
FROM employee AS e
WHERE dept# in (301, 501);
```

1st run

```
dept# emp# last_name    salary    cum_sum
----- ---- --------- --------- ----------
  301 1019 Kubic           NULL       NULL
  301 1006 Stein     29,450.00  29,450.00
  301 1008 Kanieski  29,250.00  58,700.00
  501 1015 Wilson    53,625.00 112,325.00
  501 1017 Runyon    66,000.00 178,325.00
  501 1023 Rabbit    26,500.00 204,825.00
  501 1018 Ratzlaff  54,000.00 258,825.00
```

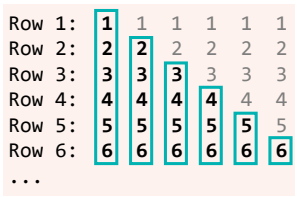
2nd run

```
dept# emp# last_name    salary    cum_sum
----- ---- --------- --------- ----------
  301 1008 Kanieski  29,250.00  29,250.00
  301 1006 Stein     29,450.00  58,700.00
  301 1019 Kubic           NULL  58,700.00
  501 1017 Runyon    66,000.00 124,700.00
  501 1023 Rabbit    26,500.00 151,200.00
  501 1015 Wilson    53,625.00 204,825.00
  501 1018 Ratzlaff  54,000.00 258,825.00
```

- No guaranteed order of the result set without adding the final ORDER BY.

Remaining Window

- Calculation is logically based on a steadily *decreasing* number of rows.
- For each added row the aggregation is repeated based on the decreased number of rows in the window.
- For better performance this logic is simply changed to a *Cumulative Window* with reversed ORDER BY.




```
(ORDER BY emp#  ASC ROWS BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING)
```
remaining

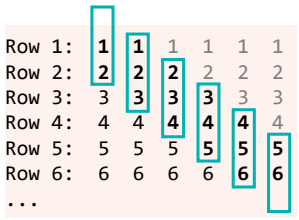
Logically Equivalent

```
(ORDER BY emp# DESC ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)
```
cumulative

- *Without the final ORDER BY the result will be sorted reverse.*

Moving Window

- Calculation is logically based on a window of a fixed number of rows *sliding* over the dataset.
- For each row the aggregation is repeated based on an offset from the current row.
- This offset might include non-existing rows.
  - Then the window size of first and last row(s) will be smaller.




```
SELECT
   department_number AS dept#
  ,employee_number   AS emp#
  ,last_name
  ,AVG(salary_amount)
   OVER (ORDER BY emp#
         ROWS BETWEEN 1 PRECEDING
                  AND 1 FOLLOWING) AS mov_avg
  ,salary_amount AS salary
FROM employee AS e
WHERE dept# in (301, 501);
```

dept#	emp#	last_name	salary	mov_avg
301	1006	Stein	29,450.00	29,350.00
301	1008	Kanieski	29,250.00	37,441.67
501	1015	Wilson	53,625.00	49,625.00
501	1017	Runyon	66,000.00	57,875.00
501	1018	Ratzlaff	54,000.00	60,000.00
301	1019	Kubic	NULL	40,250.00
501	1023	Rabbit	26,500.00	26,500.00

Rules

- Fixed order of the optional clauses: **[PARTITION BY] [ORDER BY] [ROWS]**
- The row specified by *group end* cannot precede the row specified by *group start*.
- A shorter syntax exists for a frame ending on CURRENT ROW:
  - `ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW` ⇒ `ROWS UNBOUNDED PRECEDING`
  - `ROWS BETWEEN n PRECEDING AND CURRENT ROW` ⇒ `ROWS n PRECEDING`
- CURRENT ROW can be replaced by an offset:
  - `ROWS BETWEEN UNBOUNDED PRECEDING AND 1 FOLLOWING`
  - `ROWS BETWEEN 3 PRECEDING AND UNBOUNDED FOLLOWING`
- The frame need not include the current row.
  - `ROWS BETWEEN UNBOUNDED PRECEDING AND 2 PRECEDING`
  - `ROWS BETWEEN 3 PRECEDING AND 2 PRECEDING`

Differences to Standard SQL

- Maximum value for *offset* 4096

  `ROWS BETWEEN 4096 PRECEDING AND 4096 FOLLOWING` ✓

  `ROWS BETWEEN 4096 PRECEDING AND 4097 FOLLOWING` ✗ *5486: Window size value is not acceptable*
- Performance related limit, all values in a frame must be kept in memory to calculate efficiently: *Add the new value to the list, remove the oldest one*.

- ORDER BY without ROWS defaults to a Group Window.
  - Default according to Standard SQL is **`RANGE UNBOUNDED PRECEDING`**.
  - `RANGE` syntax is not implemented in Teradata.
- Performance related limit.
  RANGE defines an *unknown* number of rows, possibly more than 4096.

Moving Difference

- Minimum size of a frame is a single row.
- A *MAX Group Window* with a single row window enables access to data in other rows.
  - Allows comparison based on data in the current and the previous row.
  - Allows calculation of a moving difference, etc.

```
SELECT
   item
  ,dt
  ,price
  ,Max(price)
   Over (ORDER BY dt
         ROWS BETWEEN 1 Preceding
                  AND 1 Preceding) AS prev_price
  ,price - prev_price               AS diff2prev
FROM vt_prices
WHERE item = 1;
```

item	dt	price	prev_price	diff2prev
1	2021-03-12	9.90	NULL	NULL
1	2021-05-02	8.95	9.90	-0.95
1	2021-05-09	9.90	8.95	0.95
1	2021-08-20	NULL	9.90	NULL
1	2021-09-04	10.95	NULL	NULL
1	2021-12-03	12.90	10.95	1.95
1	2022-01-15	10.90	12.90	-2.00
1	2022-02-12	8.95	10.90	-1.95

- No preceding row for the first row ⇒ NULL returned.
  - When NULLs are unwanted ⇒ apply COALESCE.

LAG & LEAD

```
MAX(salary_amount) OVER (ORDER BY emp# ROWS BETWEEN n PRECEDING AND n PRECEDING)
```

Logically Equivalent

```
LAG(salary_amount[, n]) OVER (ORDER BY emp#)
```

- Simpler and more flexible syntax to access data in different rows:
  - LAG returns data from a *preceding* row at a specified *offset* value
  - LEAD returns data from a *following* row at a specified *offset* value
  - *Offset* is specified as a number, defaults to 1
  - No ROWS clause

```
SELECT item, dt, price
  ,Lag(price)
   Over (ORDER BY dt) AS prev_price
  ,price - prev_price AS diff2prev
  ,Lead (price)
   Over (ORDER BY dt) AS next_price
  ,price - next_price AS diff2next
FROM vt_prices
WHERE item = 1
ORDER BY item, dt;
```

item	dt	price	prev_price	diff2prev	next_price	diff2next
1	2021-03-12	9.90	NULL	NULL	8.95	0.95
1	2021-05-02	8.95	9.90	-0.95	9.90	-0.95
1	2021-05-09	9.90	8.95	0.95	NULL	NULL
1	2021-08-20	NULL	9.90	NULL	10.95	NULL
1	2021-09-04	10.95	NULL	NULL	12.90	-1.95
1	2021-12-03	12.90	10.95	1.95	10.90	2.00
1	2022-01-15	10.90	12.90	-2.00	8.95	1.95
1	2022-02-12	8.95	10.90	-1.95	NULL	NULL

LAG/LEAD

The LAG function accesses data from the row preceding the current row at a specified offset value in a window group, while the LEAD function returns data from the row following the current row. If the offset value is outside the scope of the window, the user-specified default value is returned.

Syntax

```
{ LAG | LEAD } ( value_expression [, offset_spec ] ] )
  [ { RESPECT | IGNORE } NULLS ]
  OVER ( [ PARTITION BY expression ] [ order_by_clause ] )

offset_spec
    [ offset_value ] [, [ default_value_expression ] ]

order_by_clause
    ORDER BY expression [ ASC | DESC ] [ NULLS { FIRST | LAST } ]
      [ RESET WHEN expression ]
```

Teradata Extension

```
{ LAG | LEAD }
  ( value_expression [ { IGNORE | RESPECT } NULLS ] [, offset_spec ] ] )
  OVER ( [ PARTITION BY expression ] [ order_by_clause ] )
```

offset_value

A literal unsigned integer value between 0 and 4096. If not specified, the default value is 1. *offset_value* specifies the physical row position relative to the current row in a given window of rows. The row position is the row following the current row for the LEAD function, and the preceding row for the LAG function. An *offset_value* of 0 specifies the current row.

LAG & LEAD Null Handling

- A user-specified **default value** can be returned instead of NULL if the offset value is outside the scope of the window, i.e., before the first (LAG) or after the last row (LEAD).
- The `IGNORE NULLS` option replaces a NULL value at the *offset row* with the preceding (LAG) or following (LEAD) non-NULL value.

```
SELECT item, dt, price
  ,Lag(price)
   Over (ORDER BY dt) AS prev_price
  ,Lag(price, 1, 0)
   Over (ORDER BY dt) AS prev_default
  ,Lag(price) IGNORE NULLS
   Over (ORDER BY dt) AS prev_ignore
  ,Lag(price, 1, 0) IGNORE NULLS
   Over (ORDER BY dt) AS prev_def_ign
FROM vt_prices
WHERE item = 1
ORDER BY item, dt;
```

item	dt	price	prev_price	prev_default	prev_ignore	prev_def_ign
1	2021-03-12	9.90	NULL	0.00	NULL	0.00
1	2021-05-02	8.95	9.90	9.90	9.90	9.90
1	2021-05-09	9.90	8.95	8.95	8.95	8.95
1	2021-08-20	NULL	9.90	9.90	9.90	9.90
1	2021-09-04	10.95	NULL	NULL	9.90	9.90
1	2021-12-03	12.90	10.95	10.95	10.95	10.95
1	2022-01-15	10.90	12.90	12.90	12.90	12.90
1	2022-02-12	8.95	10.90	10.90	10.90	10.90

IGNORE NULLS

If *value_expression* returns a NULL value where the preceding or following row, as determined by the specified offset_value, is within the scope of the window group, LAG or LEAD ignores the NULL value.

LAG or LEAD then continues searching for the non-NULL *value_expression* in the preceding or following row, which may be far from the current row but within the scope of the window group. The search terminates at the window boundaries:

- For LAG, the search terminates at the first row of the window group.
- For LEAD, the search terminates at the last row of the window group.

At the end of the search, LAG or LEAD returns default_value_expression if no non-NULL *value_expression* is found.

If the preceding or following row is outside the scope of the window group, LAG or LEAD returns *default_value_expression*.
If the optional NULL clause is not specified, the default option is RESPECT NULLS.

RESPECT NULLS

If the preceding or following row determined by *offset_value* is within the scope of the window group, and if the *value_expression* evaluation returns a NULL, LAG or LEAD returns NULL. This setting indicates that the NULL value is not ignored.

If the preceding or following row is outside the scope of the window group, LAG or LEAD returns *default_value_expression*.

Caution: In presence of ties in the sort key of the Window Aggregate Function syntax, LAG and LEAD are non-deterministic. They return *value_expression* from any one of the rows with tied order by value.

Emulating DISTINCT

- The DISTINCT clause is not permitted in window aggregate functions.
- A nested LAG can be used to convert duplicate values to NULLs.

```
SELECT
   last_name
  ,dept#
  ,sal
  ,Count(sal_uniq) Over () AS cnt_distinct
  ,Avg  (sal_uniq) Over () AS avg_distinct
FROM
 (
   SELECT
      last_name
     ,department_number AS dept#
     ,Round(salary_amount, -3) AS sal
     ,CASE WHEN Lag(sal)
                Over (ORDER BY sal) = sal
           THEN NULL
           ELSE sal
      END AS sal_uniq
   FROM employee
   WHERE dept# IN (301, 501, 402)
 ) AS dt
ORDER BY sal;
```

last_name	dept#	sal	sal_uniq	cnt_distinct	avg_distinct
Crane	402	NULL	NULL	5	45,800.00
Kubic	301	NULL	NULL	5	45,800.00
Rabbit	501	27,000.00	27,000.00	5	45,800.00
Kanieski	301	29,000.00	29,000.00	5	45,800.00
Stein	301	29,000.00	NULL	5	45,800.00
Daly	402	53,000.00	53,000.00	5	45,800.00
Wilson	501	54,000.00	54,000.00	5	45,800.00
Ratzlaff	501	54,000.00	NULL	5	45,800.00
Runyon	501	66,000.00	66,000.00	5	45,800.00

The DISTINCT clause is not permitted in window aggregate functions:

`Count(DISTINCT salary) Over ()` fails with
3706 Syntax error: Distinct option is invalid with OVER phrase.

FIRST_VALUE

- **FIRST_VALUE** returns a value from the first row in a window frame.
  - Rows clause supported
  - Default frame is **ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW**
  - **IGNORE NULLS** option is supported to return the first non-NULL value in the set

Display the price difference to the initial price of an item.

```
SELECT item, dt, price
  ,First_Value(price)
   Over (ORDER BY dt)  AS first_price
  ,price - first_price AS diff2initial
FROM vt_prices
WHERE item = 1
ORDER BY item, dt;
```

item	dt	price	first_price	diff2initial
1	2021-03-12	9.90	9.90	0.00
1	2021-05-02	8.95	9.90	-0.95
1	2021-05-09	9.90	9.90	0.00
1	2021-08-20	NULL	9.90	NULL
1	2021-09-04	10.95	9.90	1.05
1	2021-12-03	12.90	9.90	3.00
1	2022-01-15	10.90	9.90	1.00
1	2022-02-12	8.95	9.90	-0.95

Syntax

```
{ FIRST_VALUE | LAST_VALUE } ( value_expression [ { IGNORE | RESPECT } NULLS ] ) OVER window
```

This returns the first value or last value in an ordered set of values.

Standard SQL window functions.

FIRST_VALUE and LAST_VALUE are especially valuable because they are often used as the baselines in calculations. For instance, with a partition holding sales data ordered by day, you may want to know how much the sales for each day were compared to the first sales day (FIRST_VALUE) for the period, or you may want to know, for a set of rows in increasing sales order, what the percentage size of each sale in the region was compared to the largest sale (LAST_VALUE) in the region.

IGNORE NULLS is particularly useful in populating an inventory table properly.

Caution:
In presence of ties in the sort key of the Window Aggregate Function syntax, FIRST_VALUE and LAST_VALUE are non-deterministic. They return *value_expression* from any one of the rows with tied order by value.

LAST_VALUE

- `LAST_VALUE` returns a value from the last row in a window frame.
  - Rows clause supported
  - Default frame is **`ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW`**
    - Usually changed to **`ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING`**
  - **`IGNORE NULLS`** option is supported to return the first non-NULL value in the set
  - Interchangeable with `FIRST_VALUE` and reversed order.

Display the price difference to the current price of an item.

```
SELECT item, dt, price
  ,Last_Value(price)
   Over (ORDER BY dt
         ROWS BETWEEN Unbounded Preceding
                  AND Unbounded Following) AS last_price
  ,price - last_price AS diff2current
FROM vt_prices
WHERE item = 1
ORDER BY item, dt;
```

item	dt	price	last_price	diff2current
1	2021-03-12	9.90	8.95	0.95
1	2021-05-02	8.95	8.95	0.00
1	2021-05-09	9.90	8.95	0.95
1	2021-08-20	NULL	8.95	NULL
1	2021-09-04	10.95	8.95	2.00
1	2021-12-03	12.90	8.95	3.95
1	2022-01-15	10.90	8.95	1.95
1	2022-02-12	8.95	8.95	0.00

Caution:
In presence of ties in the sort key of the Window Aggregate Function syntax, FIRST_VALUE and LAST_VALUE are non-deterministic. They return *value_expression* from any one of the rows with tied order by value.

Replacing NULLs with the Last Known Value

- `LAST_VALUE` can be used to return the last non-NULL value by adding the `IGNORE NULLS` option to the default window frame.
- A `LAG` with *offset* zero returns the same result.
- Similar to a COALESCE over *rows* instead of *columns*.

```
SELECT item, dt, price
  ,Last_Value(price IGNORE NULLS)
   Over (ORDER BY dt) AS last_price
  ,Lag(price, 0) IGNORE NULLS
   Over (ORDER BY dt) AS last_price2
FROM vt_prices
WHERE item = 1
ORDER BY item, dt;
```

item	dt	price	last_price	last_price2
1	2021-03-12	9.90	9.90	9.90
1	2021-05-02	8.95	8.95	8.95
1	2021-05-09	9.90	9.90	9.90
1	2021-08-20	NULL	9.90	9.90
1	2021-09-04	10.95	10.95	10.95
1	2021-12-03	12.90	12.90	12.90
1	2022-01-15	10.90	10.90	10.90
1	2022-02-12	8.95	8.95	8.95

Replacing NULLs with the Next Known Value

- `FIRST_VALUE` and `LEAD` can be applied accordingly to return the subsequent non-NULL value.

```
SELECT item, dt, price
  ,First_Value(price IGNORE NULLS)
   Over (ORDER BY dt
         ROWS BETWEEN CURRENT ROW
                  AND Unbounded Following) AS next_price
  ,Lead(price, 0) IGNORE NULLS
   Over (ORDER BY dt) AS next_price2
FROM vt_prices
WHERE item = 1
ORDER BY item, dt;
```

item	dt	price	next_price	next_price2
1	2021-03-12	9.90	9.90	9.90
1	2021-05-02	8.95	8.95	8.95
1	2021-05-09	9.90	9.90	9.90
1	2021-08-20	NULL	10.95	10.95
1	2021-09-04	10.95	10.95	10.95
1	2021-12-03	12.90	12.90	12.90
1	2022-01-15	10.90	10.90	10.90
1	2022-02-12	8.95	8.95	8.95

Summary

- A group window is defined by the clause – `UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING`.
- A cumulative window is defined by the clause – `UNBOUNDED PRECEDING`.
- A remaining window is defined by the clause – `UNBOUNDED FOLLOWING`.
- A moving window is defined by discrete values – `UNBOUNDED` is not referenced.
- `LAG/LEAD` allow access to data in other rows within a window.
- `FIRST_VALUE/LAST_VALUE` return value from the first or last row in a window frame.
- A moving difference can be computed using `LAG/LEAD`.

OLAP Functions: Window Frames

Labs

Lab 1
OLAP Functions: Window Frames

teradata.

```
HELP TABLE fin_trans;
```

Calculate the **average** and the **sum** of the transaction amounts
for each month of the years 2014 to 2018 (trans_date) and trans_type 'C'.
Add a **running total** of the transaction sums per year.
Order by year/month.

```
*** Query completed. 60 rows found. 5 columns returned.

   yy mm avgamt       sumamt        cumsum
----- -- ------ ------------ -------------
 2014  1 726.98 1,861,786.93  1,861,786.93
 2014  2 712.10 1,838,646.41  3,700,433.34
 2014  3 746.52 1,992,459.64  5,692,892.98
 2014  4 726.94 1,978,718.43  7,671,611.41
 2014  5 749.46 2,160,697.31  9,832,308.72
 2014  6 978.76 2,951,941.60 12,784,250.32
 2014  7 729.95 2,229,276.03 15,013,526.35
 2014  8 734.77 2,282,941.84 17,296,468.19
 2014  9 765.28 2,441,995.79 19,738,463.98
 2014 10 731.23 2,379,419.04 22,117,883.02
 2014 11 754.63 2,503,110.07 24,620,993.09
 2014 12 954.40 3,232,542.35 27,853,535.44
 2015  1 726.15 2,688,945.15  2,688,945.15
 2015  2 710.13 2,710,585.03  5,399,530.18
 2015  3 759.87 2,930,821.58  8,330,351.76
 2015  4 744.39 2,891,966.16 11,222,317.92
 2015  5 778.01 3,133,034.23 14,355,352.15
 2015  6 956.98 3,981,986.08 18,337,338.23
 2015  7 744.81 3,192,256.43 21,529,594.66
```

Lab 2
OLAP Functions: Window Frames

teradata.

```
HELP TABLE fin_trans;
```

Modify the previous lab and replace the running total with the difference of the current month and the same month previous year shown as a **factor current/previous**.
Only return months where the sums increased at least 45% over the previous year.

```
 *** Query completed. 17 rows found. 5 columns returned.

   yy mm avgamt       sumamt factor
----- -- ------ ------------ ------
 2015  2 710.13 2,710,585.03   1.47
 2015  3 759.87 2,930,821.58   1.47
 2015  4 744.39 2,891,966.16   1.46
 2016  2 724.89 4,011,560.97   1.48
 2016  4 739.62 4,281,686.21   1.48
 2016  7 748.25 4,946,663.09   1.55
 2016  8 750.85 5,080,222.29   1.56
 2016  9 761.43 5,351,363.07   1.62
 2016 10 769.65 5,563,825.65   1.61
 2016 11 763.12 5,640,250.92   1.63
 2016 12 977.86 7,476,733.19   1.63
 2017  1 757.19 6,224,104.20   1.61
 2017  2 736.29 6,129,629.17   1.53
 2017  3 770.56 6,476,555.89   1.55
 2017  4 769.42 6,519,329.03   1.52
 2017  5 773.01 6,669,500.88   1.49
 2017  6 971.67 8,712,945.20   1.51
```

Lab 3
OLAP Functions: Window Frames

teradata.

Write a query to return transaction details of account 4150 in January 2018.
Calculate a new account balance based on the balance of the first transaction and a running total of the amounts.
Order by trans_date.

```
HELP TABLE fin_trans;
```

```
*** Query completed. 15 rows found. 4 columns returned.

account_id trans_date    amount new_balance
---------- ---------- --------- -----------
      4150 2018-01-01 -1,970.00    2,715.43
      4150 2018-01-05   -166.90    2,548.53
      4150 2018-01-05   -300.60    2,247.93
      4150 2018-01-06   -200.00    1,787.93
      4150 2018-01-06   -260.00    1,987.93
      4150 2018-01-07     -7.50    1,780.43
      4150 2018-01-08   -200.00    1,580.43
      4150 2018-01-10  1,735.10    3,315.53
      4150 2018-01-12   -200.00    3,115.53
      4150 2018-01-24   -150.00    2,965.53
      4150 2018-01-25    -70.00    2,895.53
      4150 2018-01-29   -170.00    2,725.53
      4150 2018-01-31 -1,070.00    1,674.86
      4150 2018-01-31     -1.46    2,744.86
      4150 2018-01-31     20.79    2,746.32
```

Lab 4
OLAP Functions: Window Frames

teradata.

For each account_id in the year 2018 calculate the **average number of days** between dates with cash withdrawals (**trans_type** 'P').
Return only accounts with more than 12 transactions.
Order by account_id.

```
HELP TABLE fin_trans;
```

```
*** Query completed. 31 rows found. 3 columns returned.

account_id order_count avg_days
---------- ----------- --------
       180          14       27
       347          13       20
       412          14       25
       521          13       30
       863          13       25
       915          13       23
      1132          14       22
      1160          13       23
      1274          14       24
      1407          13       27
      2219          14       26
      2624          13       25
      2934          13       24
      3089          14       25
      3364          14       25
      3424          13       25
      3556          13       28
      3558          13       27
```

Example account 347:

```
 account_id trans_date diff
 ---------- ---------- ----
        347 2018-04-01
        347 2018-04-09    8
        347 2018-04-27   18
        347 2018-04-28    1
        347 2018-05-01    3
        347 2018-05-26   25
        347 2018-07-24   59
        347 2018-08-11   18
        347 2018-08-16    5
        347 2018-09-29   44
        347 2018-09-29   same day
        347 2018-10-26   27
        347 2018-11-12   17
                         ---
                         20 avg
```

⇒ 13 cash withdrawals with an average of 20 days between each date

OLAP Functions: Window Frames

Lab Solutions

Lab 1 Solution
OLAP Functions: Window Frame

teradata.

Calculate the **average** and the **sum** of the transaction amounts
for each month of the years 2014 to 2018 (trans_date) and trans_type 'C'.
Add a **running total** of the transaction sums per year.
Order by year/month.

```
SELECT
   Extract(YEAR From trans_date) AS yy
  ,Extract(MONTH From trans_date) AS mm
  ,Avg(amount) AS avgamt
  ,Sum(amount) AS sumamt
  ,Sum(sumamt)
   Over (PARTITION BY yy
         ORDER BY mm
         ROWS Unbounded Preceding) AS cumsum
FROM fin_trans
WHERE trans_type = 'C'
  AND Extract(YEAR From trans_date) BETWEEN 2014 AND 2018
GROUP BY 1,2
ORDER BY 1,2;
```

Lab 2 Solution
OLAP Functions: Window Frame

teradata.

Modify the previous lab and replace the running total with the difference of the current month and the same month previous year shown as a **factor current/previous**.
Only return months where the sums increased at least 45% over the previous year.

```
SELECT
   Extract(YEAR From trans_date) AS yy
  ,Extract(MONTH From trans_date) AS mm
  ,Avg(amount) AS avgamt
  ,Sum(amount) AS sumamt
  ,sumamt
    / Lag(sumamt, 12)
      Over (ORDER BY yy, mm) AS factor
FROM fin_trans
WHERE trans_type = 'C'
  AND Extract(YEAR From trans_date) BETWEEN 2014 AND 2017
GROUP BY 1,2
QUALIFY factor >= 1.45
ORDER BY 1,2;
```

A different way to get the same month previous year:

```
,sumamt
   / Lag(sumamt)
     Over (PARTITION BY mm
           ORDER BY yy) AS factor
```

Lab 3 Solution
OLAP Functions: Window Frame

Write a query to return transaction details of account 4150 in January 2018.
Calculate a new account balance based on the `balance` of the first transaction and a running total of the amounts.
Order by trans_date.

```
SELECT account_id
  ,trans_date
  ,amount
  ,First_Value(balance) Over (PARTITION BY account_id ORDER BY trans_date)
          + Sum(amount) Over (PARTITION BY account_id ORDER BY trans_date
                              ROWS Unbounded Preceding) AS new_balance
FROM fin_trans
WHERE account_id = 4150
AND trans_date BETWEEN DATE '2018-01-01' AND  DATE '2018-01-31'
ORDER BY trans_date
;
```

Lab 4
OLAP Functions: Window Frame

For each account_id in the year 2018 calculate the **average number of days** between dates with cash withdrawals (**trans_type** 'P').
Return only accounts with more than 12 transactions.
Order by account_id.

```
SELECT
   account_id
  ,Count(*) AS order_count
  ,Cast(Avg(diff) AS INT) AS avg_days
FROM
 (
   SELECT
     account_id
     ,trans_date
     ,NullIf(trans_date
             - Lag(trans_date)
               Over (PARTITION BY account_id
                     ORDER BY trans_date), 0) AS diff
   FROM fin_trans
   WHERE Extract(YEAR From trans_date) = 2018
   AND trans_type = 'P'
   QUALIFY
     Count(*)
      Over (PARTITION BY account_id) > 12
 ) dt
GROUP BY 1
ORDER BY 1;
```

Lab 4 (Simplified)
OLAP Functions: Window Frame

No OLAP-functions needed here, a simple aggregation works, too:

Divide the number of days between the first and the last transaction by the number of days (-1) with withdrawals.

```
SELECT
   account_id
  ,Count(*) AS order_count
  ,(Max(trans_date) - Min(trans_date))
    / (Count(DISTINCT trans_date) -1) AS avg_days
FROM fin_trans
WHERE Extract(YEAR From trans_date) = 2018
  AND trans_type = 'P'
GROUP BY 1
HAVING order_count > 12
ORDER BY 1
;
```

Module 3: OLAP Functions – Ranking

Teradata Vantage SQL Intermediate

Objectives

After completing this module, you will be able to:

- Contrast the differences between ROW_NUMBER and variations of RANK.
- Assign sequence values within a partition.
- Use relative ranking functions PERCENT_RANK and CUME_DIST.
- Divide a partition into equally sized buckets.
- Use the inverse distribution functions PERCENTILE_DISC and PERCENTILE_CONT.
- Calculate median values.
- Ignore NULLs in ranking functions.

ROW_NUMBER

- A Cumulative Count Star assigns a sequential number to any sortable data type.
- Standard SQL provides a shortcut: ROW_NUMBER

```
SELECT
   last_name
  ,department_number AS dept#
  ,Round(salary_amount, -3) AS sal
  ,Count(*)
   Over (ORDER BY sal DESC
         ROWS Unbounded Preceding) AS cum_cnt
  ,Row_Number()
   Over (ORDER BY sal DESC) AS row_num
FROM employee
WHERE dept# IN (301, 501, 402)
  AND sal IS NOT NULL;
```

last_name	dept#	sal	cum_cnt	row_num
Runyon	501	66,000.00	1	1
Ratzlaff	501	54,000.00	2	2
Wilson	501	54,000.00	3	3
Daly	402	53,000.00	4	4
Stein	301	29,000.00	5	5
Kanieski	301	29,000.00	6	6
Rabbit	501	27,000.00	7	7

- When the order by columns are not unique:
  - Rows with the same order get different row numbers
  - Multiple runs can produce different results

ROW_NUMBER Function Syntax

```
ROW_NUMBER ()
OVER ( [ PARTITION BY column_reference [,...] ]
         ORDER BY value_spec [,...]
       [ RESET WHEN condition ]
     )
```

Ranking Functions: RANK & DENSE_RANK

teradata.

- Standard SQL ranking algorithms
  - ROW_NUMBER
    - Sequential, no gaps
    - Non-deterministic
  - RANK
    - Duplicate values, gaps
  - DENSE_RANK
    - Duplicate values, no gaps
- Teradata extensions
  - RANK WITH TIES HIGH
    - Duplicate values, gaps
  - RANK WITH TIES AVG
    - Duplicate values, gaps

```
SELECT
   last_name
  ,department_number AS dept#
  ,Row_Number() Over (ORDER BY dept#)           AS rownum
  ,Rank() Over (ORDER BY dept#)                 AS rnk
  ,Rank() Over (ORDER BY dept# WITH TIES High)  AS r_high
  ,Rank() Over (ORDER BY dept# WITH TIES Avg)   AS r_avg
  ,Dense_Rank() Over (ORDER BY dept#)           AS r_dense
FROM employee
WHERE dept# IN (301,402,501,999);
```

last_name	dept#	rownum	rnk	r_high	r_avg	r_dense
Kubic	301	1	1	3	2.0	1
Kanieski	301	2	1	3	2.0	1
Stein	301	3	1	3	2.0	1
Daly	402	4	4	5	4.5	2
Crane	402	5	4	5	4.5	2
Ratzlaff	501	6	6	9	7.5	3
Wilson	501	7	6	9	7.5	3
Rabbit	501	8	6	9	7.5	3
Runyon	501	9	6	9	7.5	3
Trainer	999	10	10	10	10.0	4

Five Ranking algorithms exist, which will return the same output for unique data. But when the ORDER BY is non-unique they will produce different results with possibly duplicate numbers and gaps in the ranking. Standard SQL proposes syntax for three out of those five variations, while Teradata syntax extensions provide all of them.

Ordinal ranking "1234"
A sequential number. No duplicate values & no gaps, but non-deterministic, i.e. same row might get different ranking number when query runs a second time. Standard SQL ROW_NUMBER.

Standard competition ranking "1224"
One plus the number of rows with a value less than the current value. Duplicate values & gaps possible. Standard SQL RANK (or Teradata extension RANK WITH TIES LOW).

Modified competition ranking "1334"
The number of rows with a value less than or equal to the current value. Duplicate values & gaps possible. Teradata extension RANK WITH TIES HIGH.

Fractional ranking "1 2.5 2.5 4"
One plus the number of rows with a value less than the current value plus half the number of items equal to it. Duplicate values & gaps possible. Teradata extension RANK WITH TIES AVG.

Dense ranking "1223"
The number of distinct values less than or equal to the current value. Duplicate values, but no gaps. Standard SQL DENSE_RANK (or Teradata extension RANK WITH TIES DENSE)

Relative Ranking Functions: PERCENT_RANK & CUME_DIST

teradata.

PERCENT_RANK

- Calculates the relative position of the current value within a partition as the fraction of rows with a *value less than* the current value.
- Returns a DECIMAL(7,6)
  `0.000000 <= n <= 1.000000`
- Calculation based on (RANK-1)/(COUNT-1)

CUME_DIST (Cumulative Distribution)

- Calculates the relative position as the fraction of rows with a *value less than or equal to* the current value.
- Returns a DECIMAL(7,6)
  `0.000000 < n <= 1.000000`
- Based on (RANK WITH TIES HIGH)/(COUNT)

```
SELECT
   last_name
  ,department_number AS dept#
  ,salary_amount AS sal
  ,Rank()
   Over (ORDER BY sal DESC) AS rnk
  ,Percent_Rank()
   Over (ORDER BY sal DESC) AS pct_rnk
  ,Cume_Dist()
   Over (ORDER BY sal DESC) AS cum_dist
FROM employee
WHERE dept# IN (301, 401, 403)
  AND sal IS NOT NULL;
```

last_name	dept#	sal	rnk	pct_rnk	cum_dist
Villegas	403	49,700.00	1	0.000000	0.076923
Rogers	401	46,000.00	2	0.083333	0.153846
Brown	403	43,700.00	3	0.166667	0.230769
Brown	401	43,100.00	4	0.250000	0.307692
Hopkins	403	37,900.00	5	0.333333	0.384615
Trader	401	37,850.00	6	0.416667	0.461538
Johnson	401	36,300.00	7	0.500000	0.538462
Ryan	403	31,200.00	8	0.583333	0.615385
Lombardo	403	31,000.00	9	0.666667	0.692308
Stein	301	29,450.00	10	0.750000	0.769231
Kanieski	301	29,250.00	11	0.833333	0.846154
Hoover	401	25,525.00	12	0.916667	0.923077
Phillips	401	24,500.00	13	1.000000	1.000000

CUME_DIST is similar to PERCENT_RANK. Unlike PERCENT_RANK, which considers the RANK value in the presence of ties, CUME_DIST uses the highest tied rank, that is, the position of the last tied value when there are peers. CUME_DIST is the ratio of that position in the partition (RANK HIGH/NUM ROWS).

The range of values returned by CUME_DIST is > 0 to <= 1.
The range of values returned by ROW_NUMBER is >= 0 to <= 1.

NULL Handling

To project rows with NULLs, but exclude them from processing

- ROW_NUMBER & RANK: Order NULLs last and apply a CASE
- Relative ranking functions: Create two partitions NULL/NOT NULL and apply a CASE

```
SELECT
   last_name
  ,department_number AS dept#
  ,salary_amount AS sal
  ,CASE
     WHEN sal IS NOT NULL
     THEN Row_Number()
          Over (ORDER BY sal NULLS LAST)
   END AS rn_null
  ,CASE WHEN sal IS NOT NULL
        THEN Percent_Rank()
             Over (PARTITION BY CASE WHEN sal IS NOT NULL
                                     THEN 0 ELSE 1
                                END
                   ORDER BY sal)
   END AS pct_rnk
FROM employee
WHERE dept# IN (301, 501, 402);
```

last_name	dept#	sal	rn_null	pct_rnk
Rabbit	501	26,500.00	1	0.00
Kanieski	301	29,250.00	2	0.17
Stein	301	29,450.00	3	0.33
Daly	402	52,500.00	4	0.50
Wilson	501	53,625.00	5	0.67
Ratzlaff	501	54,000.00	6	0.83
Runyon	501	66,000.00	7	1.00
Crane	402	NULL	NULL	NULL
Kubic	301	NULL	NULL	NULL

QUANTILE

teradata.

QUANTILE(*n, sort_spec*)

- Distributes the rows in a partition ordered by ***sort_spec*** into ***n*** tiles with roughly the same number of rows.
- Assigns an Integer between **0** and **n-1**
- Rows with same order always placed *in the same bucket*
- Deprecated syntax, strongly discouraged
  - No PARTITION BY, must use GROUP BY instead
  - should be replaced by

```
n * (Rank() Over (ORDER BY sort_spec)-1)
 / Count(*) Over ()
```

QUANTILE ≠ NTILE

- Standard SQL supports similarly named function NTILE
- Assigns an Integer between **1** and **n**
- Rows with same order might be placed *in adjacent buckets*
  - Almost, but not exactly the same result:

```
n * (Row_Number() Over (ORDER BY sort_spec)-1)
 / Count(*) Over ()
```

```
SELECT
   last_name
  ,department_number AS dept#
  ,salary_amount AS sal
  ,Quantile(4, sal) AS qnt
  ,4 * (Rank()  Over (ORDER BY sal) - 1)
     / Count(*) Over () AS "QUANTILE"
FROM employee
WHERE dept# IN (301, 401, 403)
  AND sal IS NOT NULL;
```

last_name	dept#	sal	qnt	QUANTILE
Phillips	401	24,500.00	0	0
Hoover	401	25,525.00	0	0
Kanieski	301	29,250.00	0	0
Stein	301	29,450.00	0	0
Lombardo	403	31,000.00	1	1
Ryan	403	31,200.00	1	1
Johnson	401	36,300.00	1	1
Trader	401	37,850.00	2	2
Hopkins	403	37,900.00	2	2
Brown	401	43,100.00	2	2
Brown	403	43,700.00	3	3
Rogers	401	46,000.00	3	3
Villegas	403	49,700.00	3	3

Teradata's QUANTILE is a deprecated feature based on old syntax, but it's easy to rewrite

```
QUANTILE(n, data_col ASC)
...
GROUP BY part_col
```

to

```
n * (RANK() OVER(PARTITION BY part_col ORDER BY data_col) - 1)
 / COUNT(*) OVER(PARTITION BY part_col) AS "QUANTILE"
```

When the RANK in this formula is replaced by a ROW_NUMBER the number of rows per bucket can differ by at most one. If the number of rows *N* isn't divisible without reminder by the number of buckets *B* then the *N* **MOD** *B* extra rows are kind of *randomly* assigned to buckets. But in NTILE the *first* *N* **MOD** *B* buckets get an extra row.

To get an exactly matching output

```
 NTILE(B) OVER (PARTITION BY part_col ORDER BY data_col)
```

must be rewritten to

```
 ,COUNT(*)     OVER (PARTITION BY part_col) AS N
 ,ROW_NUMBER() OVER (PARTITION BY part_col ORDER BY data_col) AS rowno
 ,CASE
    WHEN  rowno   <= ((N/B)+1) * (N MOD B)
    THEN (rowno-1) / ((N/B)+1)
    ELSE (rowno-1 - (N MOD B)) / (N/B)
  END + 1 AS "NTILE"
```

This complex rewrite is still using a single *STAT FUNCTION step* in Explain, but reconsider if matching the exact output of NTILE is actually needed.

Inverse Distribution Functions: PERCENTILE_DISC

teradata.

`Percentile_Disc( n ) Within Group ( ORDER BY sort_spec )`

- Order-based *aggregate function*: ignores NULLs, no `PARTITION BY` ⇒ `GROUP BY`
- `0 <= n <= 1`
- Computes the *discrete percentile*
- Treats the group as a window partition of the `CUME_DIST` window function and returns the first value whose cumulative distribution value is *greater than or equal* to ***n***.

salary	cum_dist
49,700.00	0.076923
46,000.00	0.153846
43,700.00	0.230769
43,100.00	0.307692
37,900.00	0.384615
37,850.00	0.461538
36,300.00	0.538462
31,200.00	0.615385
31,000.00	0.692308
29,450.00	0.769231
29,250.00	0.846154
25,525.00	0.923077
24,500.00	1.000000

```
SELECT
   Percentile_Disc(0.2)
   Within Group (ORDER BY salary_amount DESC)
  ,Percentile_Disc(0.8)
   Within Group (ORDER BY salary_amount DESC)
FROM employee
WHERE department_number IN (301, 401, 403);
```

```
43,700.00
29,250.00
```

PERCENTILE_DISC

The `CUME_DIST` function computes the cumulative distribution of a set of values. The *inverse operation* to find the value for a given percentile is done using `PERCENTILE_DISC`.

According to Standard SQL `PERCENTILE_DISC(x)` is the first value with a `CUME_DIST` greater than or equal to ***x***. This logic will return a NULL for `x = 0`, but in this case `MIN` is a better option than `PERCENTILE_DISC(0)`.

Inverse Distribution Functions: PERCENTILE_CONT

teradata.

`Percentile_Cont( n ) Within Group ( ORDER BY sort_spec )`

- Order-based *aggregate function*: ignores NULLs, no `PARTITION BY` ⇒ `GROUP BY`
- `0 <= n <= 1`
- Computes the *continuous percentile*
- Treats the group as a window partition of the `PERCENT_RANK` window function and returns a value corresponding to *n* within the ordered set of values. Will interpolate between adjacent rows if needed.

salary	pct_rnk
49,700.00	0.000000
46,000.00	0.083333
43,700.00	0.166667
43,100.00	0.250000
37,900.00	0.333333
37,850.00	0.416667
36,300.00	0.500000
31,200.00	0.583333
31,000.00	0.666667
29,450.00	0.750000
29,250.00	0.833333
25,525.00	0.916667
24,500.00	1.000000

`Percentile_Cont(0.5) = MEDIAN`

- The `MEDIAN` is just the special case of the 50th percentile.

```
SELECT
   Percentile_Cont(0.2)
   Within Group (ORDER BY salary_amount DESC)
  ,Percentile_Cont(0.8)
   Within Group (ORDER BY salary_amount DESC)
FROM employee
WHERE department_number IN (301, 401, 403);
```

```
43,460.00
29,330.00
```

PERCENTILE_CONT

Whereas `PERCENTILE_DISC` returns one of the existing values, `PERCENTILE_CONT` is based on a linear interpolation between two consecutive values. First you calculate a hypothetical row number ***rn*** `= (`*x*` * (`*n*` - 1)) + 1`, where *x* is the percentile and *n* is the number of rows per group. If ***rn*** has no fractional part then you already found the result row, else you have to consider the `FLOOR(`***rn***`)` and `CEILING(`***rn***`)` rows and do a linear interpolation.

Summary

- Variations of the RANK function are available for assigning ranking values based on an ordering.
- RANK is a window aggregate and can reference PARTITION.
- ROW_NUMBER can be used to assign a sequence number to a result set.
- PERCENT_RANK and CUME_DIST can be used to assign a relative ranking to values in a window.
- Dense Ranks and finding Median values can also be derived from these functions.
- The QUANTILE function can be used to return a percentile, decile, quartile, etc.

OLAP Functions: Ranking

Labs

Lab 1
OLAP Functions: Ranking

teradata.

Write a query to return details of the employees earning the two highest `annual_salary` for each `department_number` between 40 and 50.
Order by descending salary.

```
HELP TABLE hr_payroll;
```

```
*** Query completed. 22 rows found. 6 columns returned.

emp#    first_name last_name dept# rnk annual_salary
------- ---------- --------- ----- --- -------------
276,758 Sharon     Broome       40   1    175,000.28
536,326 Darryl     Gissel       40   2    149,155.24
156,698 James      Llorens      40   2    149,155.24
 56,464 Carl       Dabadie      50   1    145,510.04
537,250 Murphy     Paul         50   1    145,510.04
349,437 Eric       Romero       44   1    123,242.08
375,721 Brian      Bernard      46   1    123,242.08
154,512 Claude     Leduff       50   2    118,758.12
152,269 Robert     Laughlin     44   2    112,063.12
 38,687 Sharon     Campbell     41   1    106,870.14
395,030 Greta      Meche        41   1    106,870.14
487,384 Patti      Wallace      45   1    106,366.00
357,936 Tiffani    Delapasse    41   2    105,422.20
154,881 Barbara    Leblanc      48   1    101,924.16
248,711 Russel     Smith        48   1    101,924.16
352,241 Annette    Bookter      46   2    100,202.18
370,827 Michael    Drago        48   2     92,726.14
304,638 Lynn       Maloy        45   2     92,726.14
407,720 Mertis     Edwards      47   1     61,224.28
 76,252 Marion     Niles        47   2     51,352.08
```

Lab 2
OLAP Functions: Ranking

teradata.

Write a query to return the top 30 `overtime_pay` of the `sal_year` 2017.
Consider only employees with at least 100 `overtime_hours`.
Add another column indicating the rank of the overtime pay within department.
Order by descending overtime pay.

```
HELP TABLE hr_salary_hist;
```

```
*** Query completed. 30 rows found. 9 columns returned.
```

emp#	first_name	last_name	dept#	total_pay	overtime_pay	overtime_hours	overtime_rank	dept_rank
416,916	Arthur	Munoz	50	169,237.58	108,117.48	2,253.25	1	1
403,636	Todd	Bourgoyne	50	167,481.71	101,548.69	1,952.33	2	2
346,055	Mickey	Duncan	50	141,234.89	84,256.68	1,875.25	3	3
349,461	Jason	Martin	50	130,875.53	78,120.85	1,857.25	4	4
370,088	Craig	Russell	50	142,034.22	76,011.76	1,459.25	5	5
330,159	Duke	Staples	50	137,405.58	74,169.86	1,521.50	6	6
347,191	Nicholas	Mcdonner	52	130,243.41	69,945.25	1,620.75	7	1
371,432	Leroy	Perkins	70	109,918.06	63,830.34	2,010.50	8	1
281,352	Randall	Wiedeman	50	129,610.45	60,596.21	1,139.42	9	7
313,700	Brandon	Johnson	50	117,202.14	56,826.42	1,187.33	10	8
312,576	Leon	Morris	70	104,723.98	55,708.32	1,561.50	11	2
16,349	Danny	Beck	70	107,615.54	54,230.64	1,439.75	12	3
32,573	Timothy	Browning	50	126,846.06	53,930.70	964.75	13	9
373,087	Kenneth	Bowman	50	114,672.88	53,030.66	1,084.33	14	10
371,521	J	Fontenot	50	121,772.09	52,694.35	973.50	15	11
306,789	Samuel	White	50	107,659.12	51,089.34	1,158.75	16	12
380,300	David	Wallace	50	120,250.91	50,724.79	927.99	17	13

Lab 3
OLAP Functions: Ranking

teradata.

```
HELP TABLE hr_salary_hist;
```

Modify the previous report and replace the department rank with a rank of the total_pay of all employees regardless of overtime.
Return the top 30 total_pay amounts of employees with at least 100 hours overtime.
Order by descending total pay.

```
*** Query completed. 11 rows found. 9 columns returned.
```

emp#	first_name	last_name	dept#	total_pay	overtime_pay	overtime_hours	overtime_rank	total_rank
416,916	Arthur	Munoz	50	169,237.58	108,117.48	2,253.25	1	5
403,636	Todd	Bourgoyne	50	167,481.71	101,548.69	1,952.33	2	6
370,088	Craig	Russell	50	142,034.22	76,011.76	1,459.25	5	8
346,055	Mickey	Duncan	50	141,234.89	84,256.68	1,875.25	3	9
330,159	Duke	Staples	50	137,405.58	74,169.86	1,521.50	6	11
349,461	Jason	Martin	50	130,875.53	78,120.85	1,857.25	4	14
347,191	Nicholas	Mcdonner	52	130,243.41	69,945.25	1,620.75	7	15
281,352	Randall	Wiedeman	50	129,610.45	60,596.21	1,139.42	9	16
32,573	Timothy	Browning	50	126,846.06	53,930.70	964.75	13	21
371,521	J	Fontenot	50	121,772.09	52,694.35	973.50	15	28
380,300	David	Wallace	50	120,250.91	50,724.79	927.99	17	30

Lab 4
OLAP Functions: Ranking

teradata.

Write a query to return the 3 accounts with the highest volume of cash withdrawals (`trans_type = 'P'`) per `district_id`.
Consider only accounts with a `sum(amount) < -50000`.
Order by volume within district.

```
HELP TABLE fin_account;
HELP TABLE fin_trans;
```

```
*** Query completed. 36 rows found. 4 columns returned.

district_id account_id rnk    volume
----------- ---------- --- ---------
          1       2486   1 -75697.20
          1       1475   2 -71663.80
          4        730   1 -70701.50
         64       7445   1 -64806.10
          4       3558   2 -63161.10
         14       3713   1 -63106.30
         12       6473   1 -60944.50
         60       2219   1 -60157.40
         74       7774   1 -59984.80
         52       8212   1 -58604.10
         55      10105   1 -57750.70
         74       3424   2 -57433.40
         27        212   1 -57025.90
         65       1012   1 -56929.00
          1       1132   3 -56914.60
         47       4321   1 -56403.40
         70       9707   1 -56378.80
         77       5228   1 -54980.00
         71        863   1 -54941.70
```

Lab 5
OLAP Functions: Quantile

teradata.

Report the transactions amounts of trans_type 'C' in 2017 (trans_date)
in 10 equal *height* buckets.
Add count, minimum, average, median, maximum and sum columns.

```
HELP TABLE fin_trans;
```

```
*** Query completed. 10 rows found. 7 columns returned.

 q    cnt  min_amt  avg_amt  med_amt  max_amt       sum_amt
-- ------ -------- -------- -------- -------- -------------
 0 10,787     0.01     6.15     6.68     8.42     66,303.03
 1 10,762     8.43    10.28    10.20    12.29    110,657.42
 2 10,769    12.30    14.66    14.61    17.22    157,866.97
 3 10,759    17.23    20.58    20.27    24.96    221,397.62
 4 10,825    24.97    59.33    35.13   170.00    642,204.73
 5 10,709   180.00   353.36   342.70   484.80  3,784,119.10
 6 10,764   485.40   624.43   617.80   840.40  6,721,416.05
 7 10,770   840.90 1,238.87 1,229.70 1,642.20 13,342,664.35
 8 10,771 1,642.30 2,043.66 2,032.80 2,451.00 22,012,276.85
 9 10,762 2,451.10 3,661.99 3,555.70 7,481.20 39,410,326.55
```

Lab 5 (optional)
OLAP Functions: Equal Sum Buckets

teradata.

Report the transactions amounts of trans_type 'C' in 2017 (trans_date)
in 10 equal *sum* buckets.
Add count, minimum, average, median, maximum and sum columns.

```
HELP TABLE fin_trans;
```

```
*** Query completed. 10 rows found. 7 columns returned.

 q    cnt  min_amt  avg_amt  med_amt  max_amt       sum_amt
-- ------ -------- -------- -------- -------- -------------
 0 71,098     0.01   121.61    19.01   641.60  8,646,322.87
 1  9,648   641.60   896.26   879.25 1,228.70  8,647,102.30
 2  5,935 1,228.90 1,456.75 1,458.60 1,688.00  8,645,793.70
 3  4,663 1,688.00 1,854.49 1,852.30 2,023.60  8,647,504.20
 4  3,973 2,023.60 2,176.22 2,178.70 2,332.60  8,646,109.75
 5  3,467 2,333.00 2,494.03 2,460.80 2,768.30  8,646,795.95
 6  2,810 2,768.50 3,077.14 3,075.70 3,393.60  8,646,766.45
 7  2,360 3,393.60 3,664.28 3,657.60 3,965.10  8,647,696.85
 8  2,019 3,965.50 4,281.87 4,287.30 4,592.10  8,645,100.50
 9  1,705 4,592.20 5,073.34 4,853.80 7,481.20  8,650,040.10
```

Lab 5 (optional)
OLAP Functions: WIDTH_BUCKET

teradata.

Report the transactions amounts of trans_type 'C' in 2017 (trans_date)
in 10 equal *width* buckets of 500 $ each.
Add count, minimum, average, median, maximum and sum columns.

```
HELP TABLE fin_trans;
```

```
*** Query completed. 11 rows found. 7 columns returned.

wb    cnt  min_amt  avg_amt  med_amt  max_amt       sum_amt
-- ------ -------- -------- -------- -------- -------------
 1 65,171     0.01    80.68    17.37   498.90  5,258,290.27
 2 12,295   500.00   680.16   648.00   999.30  8,362,613.70
 3  6,793 1,000.00 1,246.45 1,237.40 1,499.90  8,467,154.95
 4  6,716 1,500.00 1,750.47 1,752.80 1,999.50 11,756,126.85
 5  6,516 2,000.10 2,246.21 2,245.50 2,499.90 14,636,282.95
 6  2,335 2,500.30 2,743.87 2,738.20 2,999.20  6,406,940.50
 7  2,232 3,000.10 3,243.23 3,238.95 3,499.50  7,238,899.75
 8  1,990 3,501.15 3,729.66 3,716.00 3,999.30  7,422,024.20
 9  1,674 4,000.50 4,258.82 4,261.80 4,499.80  7,129,258.00
10  1,638 4,500.20 4,759.57 4,768.30 4,999.90  7,796,181.10
11    318 5,002.80 6,275.03 6,298.20 7,481.20  1,995,460.40
```

OLAP Functions: Ranking

Lab Solutions

Lab 1 Solution
OLAP Functions: Ranking

Write a query to return details of the employees earning the two highest `annual_salary` for each `department_number` between 40 and 50.
Order by descending salary.

```
SELECT
   employee_number AS emp#
  ,first_name
  ,last_name
  ,department_number AS dept#
  ,Dense_Rank()
   Over (PARTITION BY dept#
         ORDER BY annual_salary DESC) AS rnk
  ,annual_salary
FROM hr_payroll
WHERE dept# BETWEEN 40 AND 50
QUALIFY
   rnk <= 2
ORDER BY annual_salary DESC
;
```

Lab 2 Solution
OLAP Functions: Ranking

teradata.

Write a query to return the top 30 overtime_pay of the sal_year 2017.
Consider only employees with at least 100 overtime_hours.
Add another column indicating the rank of the overtime pay within department.
Order by descending overtime pay.

```
SELECT
   employee_number AS emp#
  ,first_name
  ,last_name
  ,department_number AS dept#
  ,total_pay
  ,overtime_pay
  ,overtime_hours
  ,Rank()
   Over (ORDER BY overtime_pay DESC) AS overtime_rank
  ,Rank()
   Over (PARTITION BY department_number
         ORDER BY overtime_pay DESC) AS dept_rank
FROM hr_salary_hist
WHERE sal_year = 2017
  AND overtime_hours >= 100
QUALIFY overtime_rank <= 30
ORDER BY overtime_rank;
```

Lab 3 Solution
OLAP Functions: Ranking

teradata.

Modify the previous report and replace the department rank with a rank of the total_pay of all employees regardless of overtime.
Return the top 30 total_pay amounts of employees with at least 100 hours overtime.
Order by descending total pay.

```
SELECT
   employee_number AS emp#
  ,first_name
  ,last_name
  ,department_number AS dept#
  ,total_pay
  ,overtime_pay
  ,overtime_hours
  ,Rank()
   Over (ORDER BY overtime_pay DESC) AS overtime_rank
  ,Rank()
   Over (ORDER BY total_pay DESC) AS total_rank
FROM hr_salary_hist
WHERE sal_year = 2017
QUALIFY total_rank <= 30
    AND overtime_hours >= 100
ORDER BY total_rank;
```

Lab 4 Solution
OLAP Functions: Ranking

Write a query to return the 3 accounts with the highest volume of cash withdrawals (`trans_type = 'P'`) per `district_id`.
Consider only accounts with a `sum(amount) < -50000`.
Order by volume within district.

```
SELECT
   a.district_id
  ,a.account_id
  ,RANK()
   OVER (PARTITION BY district_id
         ORDER BY volume) AS rnk
  ,SUM(amount) AS volume
FROM fin_account AS a
JOIN fin_trans AS t
  ON a.account_id = t.account_id
WHERE trans_type = 'P' -- cash withdrawal
GROUP BY 1, 2
HAVING volume < -50000
QUALIFY rnk <= 3
ORDER BY volume
;
```

Lab 5 Solution
OLAP Functions: Quantile

teradata.

Report the transactions amounts of trans_type 'C' in 2017 (trans_date) in 10 equal *height* buckets.
Add count, minimum, average, median, maximum and sum columns.

```
WITH cte AS
 (
   SELECT -- QUANTILE(10,amount) AS q
      10 * (Rank() Over (ORDER BY amount) -1) / Count(*) Over () AS q
     ,amount
   FROM fin_trans
   WHERE trans_type = 'C'
     AND trans_date BETWEEN DATE '2017-01-01' AND DATE '2017-12-31'
 )
SELECT
   q
  ,Count(*) AS Cnt
  ,Min(amount) AS min_amt
  ,Avg(amount) AS avg_amt
  ,Median(amount) AS med_amt
  ,Max(amount) AS max_amt
  ,Sum(amount) AS sum_amt
FROM cte
GROUP BY q
ORDER BY q;
```

Lab 5 (optional) Solution
OLAP Functions: Equal Sum Buckets

teradata.

Report the transactions amounts of trans_type 'C' in 2017 (trans_date) in 10 equal *area* buckets.
Add count, minimum, average, median, maximum and sum columns.

```
WITH cte AS
 (
   SELECT -- similar to RANK/COUNT = Cumulative Sum / Group Sum
        Sum(amount) Over (ORDER BY amount ROWS Unbounded Preceding) AS cumsum
       ,Sum(amount) Over (ORDER BY amount) AS grpsum
       ,Cast(10 * (Cast(cumsum AS NUMBER) - 0.01) / grpsum AS INT) AS q
       ,amount
   FROM fin_trans
   WHERE trans_type = 'C'
     AND trans_date BETWEEN DATE '2017-01-01'
                        AND DATE '2017-12-31'
 )
SELECT
   q
  ,Count(*) AS Cnt
  ,Min(amount) AS min_amt
  ,Avg(amount) AS avg_amt
  ,Median(amount) AS med_amt
  ,Max(amount) AS max_amt
  ,Sum(amount) AS sum_amt
FROM cte
GROUP BY q
ORDER BY q;
```

Lab 5 (optional) Solution
OLAP Functions: WIDTH_BUCKET

Report the transactions amounts of trans_type 'C' in 2017 (trans_date) in 10 equal *width* buckets of 500 $ each.
Add count, minimum, average, median, maximum and sum columns.

```
SELECT
   Width_Bucket(amount, 0, 5000, 10) AS wb
  ,Count(*) AS Cnt
  ,Min(amount) AS min_amt
  ,Avg(amount) AS avg_amt
  ,Median(amount) AS med_amt
  ,Max(amount) AS max_amt
  ,Sum(amount) AS sum_amt
FROM fin_trans
WHERE trans_type = 'C'    -- credit
  AND trans_date BETWEEN DATE '2017-01-01' AND DATE '2017-12-31'
GROUP BY wb
ORDER BY wb;
```

Module 4: OLAP Functions – RESET WHEN

Teradata Vantage SQL Intermediate

Objectives

After completing this module, you will be able to:

- Use RESET WHEN to dynamically add subpartitions to the window definition.
- Describe how RESET WHEN is processed.
- Rewrite RESET WHEN using nested Selects.

RESET WHEN

- OLAP calculation always restart at the *beginning* of a partition
- RESET WHEN feature allows restarting *within* a partition
- Adds dynamic condition handling to the query
  - `RESET WHEN condition` is evaluated at run-time
  - New sub-partition is created if the *condition* evaluates TRUE
- Optional part of the `ORDER BY` *clause* in the window definition
- Teradata extension to Standard SQL
- Simplified syntax which can be re-written using nested OLAP-functions (see next slide)

```
SELECT
   birthdate AS dt
  ,salary_amount AS amount
  -- cumulative sum which resets on NULL
  ,Sum(amount)
   Over (ORDER BY dt
         RESET WHEN amount IS NULL
         ROWS Unbounded Preceding
        ) AS cumsum
FROM employee
```

dt	amount	cumsum	
1965-04-23	46,000.00	46,000.00	
1967-01-31	49,700.00	95,700.00	
1972-02-18	37,900.00	133,600.00	
1972-12-11	NULL	NULL	Reset
1973-04-29	38,750.00	38,750.00	
1974-08-09	43,100.00	81,850.00	
1975-08-11	100,000.00	181,850.00	
1975-11-15	31,000.00	212,850.00	
1976-04-23	36,300.00	249,150.00	
1977-06-19	37,850.00	287,000.00	
1977-07-07	34,700.00	321,700.00	
1979-06-21	NULL	NULL	Reset
1979-12-11	52,500.00	52,500.00	
1980-01-14	25,525.00	78,025.00	
1981-11-10	66,000.00	144,025.00	
1983-10-15	29,450.00	173,475.00	
1984-01-16	43,700.00	217,175.00	
1984-05-31	54,000.00	271,175.00	
1985-09-10	31,200.00	302,375.00	
1987-03-04	53,625.00	356,000.00	
1987-07-14	NULL	NULL	Reset
1988-05-17	29,250.00	29,250.00	
1989-09-04	56,500.00	85,750.00	
1990-07-04	NULL	NULL	Reset
1992-10-29	26,500.00	26,500.00	
1993-08-10	24,500.00	51,000.00	

RESET WHEN Phrase
RESET WHEN is a Teradata extension to the ANSI SQL standard.

Depending on the evaluation of the specified condition, RESET WHEN determines the group or partition, over which the ordered analytical function operates. If the condition evaluates to TRUE, a new dynamic partition is created inside the specified window partition. To define a partition based on a column reference list, use the PARTITION BY phrase.

If there is no RESET WHEN phrase or PARTITION BY phrase, then the entire result set, delivered by the FROM clause, constitutes a single partition, over which the ordered analytical function executes. You can have different RESET WHEN clauses in the same SELECT list.

Note: A window specification that specifies a RESET WHEN clause must also specify an ORDER BY clause.

RESET WHEN Condition Rules
The condition in the RESET WHEN clause is equivalent in scope to the condition in a QUALIFY clause with the additional constraint that nested ordered analytical functions cannot specify conditional partitioning.
The condition is applied to the rows in all designated window partitions to create sub-partitions within the particular window partitions.

A RESET WHEN condition can contain the following:
- Ordered analytical functions that do not include the RESET WHEN clause
- Scalar subqueries
- Aggregate operators

In this example we want to create a cumulative sum, which resets when a NULL is found.

RESET WHEN Rewrite

```
WITH cte_dynamic_partition AS
 (
   SELECT
      birthdate AS dt
     ,salary_amount AS amount
     ,Sum(CASE WHEN amount IS NULL THEN 1 ELSE 0 END)
      Over (ORDER BY dt
            ROWS Unbounded Preceding ) AS sub_part
   FROM employee
 )
SELECT
   t.*
  ,Sum(amount)
   Over (PARTITION BY sub_part
         ORDER BY dt
         ROWS Unbounded Preceding
        ) AS cumsum
FROM cte_dynamic_partition AS t;
```

dt	amount	sub_part	cumsum
1965-04-23	46,000.00	0	46,000.00
1967-01-31	49,700.00	0	95,700.00
1972-02-18	37,900.00	0	133,600.00
1972-12-11	NULL	1	NULL
1973-04-29	38,750.00	1	38,750.00
1974-08-09	43,100.00	1	81,850.00
1975-08-11	100,000.00	1	181,850.00
1975-11-15	31,000.00	1	212,850.00
1976-04-23	36,300.00	1	249,150.00
1977-06-19	37,850.00	1	287,000.00
1977-07-07	34,700.00	1	321,700.00
1979-06-21	NULL	2	NULL
1979-12-11	52,500.00	2	52,500.00
1980-01-14	25,525.00	2	78,025.00
1981-11-10	66,000.00	2	144,025.00
1983-10-15	29,450.00	2	173,475.00
1984-01-16	43,700.00	2	217,175.00
1984-05-31	54,000.00	2	271,175.00
1985-09-10	31,200.00	2	302,375.00
1987-03-04	53,625.00	2	356,000.00
1987-07-14	NULL	3	NULL
1988-05-17	29,250.00	3	29,250.00
1989-09-04	56,500.00	3	85,750.00
1990-07-04	NULL	4	NULL
1992-10-29	26,500.00	4	26,500.00
1993-08-10	24,500.00	4	51,000.00

- Reset conditon is translated into a `0`/`1` flag in a nested Select
  - `1` when condition evaluates TRUE
- The dynamic partition is calculated using a *cumulative sum* over this flag and then added to the outer PARTITION BY

RESET WHEN is internally rewritten by the optimizer to nested Selects:

- First the *condition* is evaluated in a CASE expression returning a **`0/1`** flag
  - TRUE ⇒ **`1`**
  - FALSE or UNKNOWN ⇒ **`0`**
- Then a *cumulative sum* is calculated over this flag returning a sequence number which only increases when the condition is TRUE, i.e. all rows after a NULL share the same sequence.
- The outer Select finally adds this sequence to the PARTITION BY and calculates the final *cumulative* sum.

Explaining RESET WHEN

- Same Explain for both queries:

```
1) First, we lock EMPLOYEE_SALES.Employee in TD_MAP1 for read on a
   reserved RowHash to prevent global deadlock.
2) Next, we lock EMPLOYEE_SALES.Employee in TD_MAP1 for read.
3) We do an all-AMPs STAT FUNCTION step in TD_MAP1 from
   EMPLOYEE_SALES.Employee by way of an all-rows scan with no
   residual conditions into Spool 7 (Last Use), which is assumed to
   be redistributed by value to all AMPs in TD_Map1.  The result rows
   are put into Spool 5 (all_amps), which is built locally on the
   AMPs.  The size is estimated with low confidence to be 28 rows (
   1,148 bytes).  The estimated time for this step is 0.00 seconds.
4) We do an all-AMPs RETRIEVE step in TD_Map1 from Spool 5 (Last Use)
   by way of an all-rows scan into Spool 1 (all_amps), which is built
   locally on the AMPs.  The size of Spool 1 is estimated with low
   confidence to be 28 rows (1,148 bytes).  The estimated time for
   this step is 0.00 seconds.
5) We do an all-AMPs STAT FUNCTION step in TD_Map1 from Spool 1 (Last
   Use) by way of an all-rows scan into Spool 12 (Last Use), which is
   redistributed by hash code to all AMPs in TD_Map1.  The result
   rows are put into Spool 2 (group_amps), which is built locally on
   the AMPs.  The size is estimated with low confidence to be 28 rows
   (1,372 bytes).  The estimated time for this step is 0.00 seconds.
6) Finally, we send out an END TRANSACTION step to all AMPs involved
   in processing the request.
-> The contents of Spool 2 are sent back to the user as the result of
   statement 1.  The total estimated time is 0.01 seconds.
```

RESET WHEN Nested OLAP Function

- The RESET WHEN condition can be based on a nested OLAP function

```
SELECT
   birthdate AS dt
  ,salary_Amount AS amount
  ,Row_Number()
   Over (ORDER BY dt
         RESET WHEN amount < Lag(amount)
                             Over (ORDER BY dt)
        )-1  AS increases
FROM employee
WHERE amount IS NOT NULL;
```

dt	amount	increases	
1965-04-23	46,000.00	0	
1967-01-31	49,700.00	1	
1972-02-18	37,900.00	0	Reset
1973-04-29	38,750.00	1	
1974-08-09	43,100.00	2	
1975-08-11	100,000.00	3	
1975-11-15	31,000.00	0	Reset
1976-04-23	36,300.00	1	
1977-06-19	37,850.00	2	
1977-07-07	34,700.00	0	Reset
1979-12-11	52,500.00	1	
1980-01-14	25,525.00	0	Reset
1981-11-10	66,000.00	1	
1983-10-15	29,450.00	0	Reset
1984-01-16	43,700.00	1	
1984-05-31	54,000.00	2	
1985-09-10	31,200.00	0	Reset
1987-03-04	53,625.00	1	
1988-05-17	29,250.00	0	Reset
1989-09-04	56,500.00	1	
1992-10-29	26,500.00	0	Reset
1993-08-10	24,500.00	0	Reset

- Explain shows a 3rd STAT FUNCTION step.

In this example we want to mark any row with a 0 if the balance for a given month is less than or equal the previous month.

For cases where the balance is greater than the previous row we want to increment a counter:
1 for the first time, 2 for the second time and so on.

The ROW_NUMBER function can be used to generate an incremental value.

Within the RESET WHEN we need to compare the current row to the previous row to determine if the balance increased. Sounds like a good use for LAG.

Notice that 1 is subtracted from the ROW_NUMBER to make the counter start at 0.

This RESET WHEN is internally rewritten (see next slide) by the optimizer to nested Selects, but it needs a 3rd Select as OLAP functions can't be nested:

- First the LAG-based *condition* is evaluated in a CASE expression returning a **`0/1`** flag.
  - TRUE ⇒ **`1`**
  - FALSE or UNKNOWN ⇒ **`0`**

Must be done in an extra step as OLAP functions can't be nested.

- Then a *cumulative sum* is calculated over this flag returning a sequence number which only increases when the condition is TRUE, i.e. all rows after a decreasing amount share the same sequence as long as their amount increases.
- The outer Select finally adds this sequence to the PARTITION BY and calculates the final ROW_NUMBER.

RESET WHEN Nested OLAP Function Rewrite

```
WITH cte_flag AS
 (
   SELECT
      birthdate AS dt
     ,salary_Amount AS amount
     ,CASE WHEN amount < Lag(amount)
                         Over (ORDER BY dt)
           THEN 1
           ELSE 0
      END AS flag
   FROM employee
   WHERE amount IS NOT NULL
 )
, cte_dynamic_partition AS
 ( SELECT t.*
     ,Sum(flag)
      Over (ORDER BY dt
            ROWS Unbounded Preceding) AS sub_part
   FROM cte_flag AS t
 )
SELECT t.*
  ,Row_Number()
   Over (PARTITION BY sub_part
         ORDER BY dt
        )-1  AS increases
FROM cte_dynamic_partition AS t;
```

dt	amount	flag	sub_part	increases
1965-04-23	46,000.00	0	0	0
1967-01-31	49,700.00	0	0	1
1972-02-18	37,900.00	1	1	0
1973-04-29	38,750.00	0	1	1
1974-08-09	43,100.00	0	1	2
1975-08-11	100,000.00	0	1	3
1975-11-15	31,000.00	1	2	0
1976-04-23	36,300.00	0	2	1
1977-06-19	37,850.00	0	2	2
1977-07-07	34,700.00	1	3	0
1979-12-11	52,500.00	0	3	1
1980-01-14	25,525.00	1	4	0
1981-11-10	66,000.00	0	4	1
1983-10-15	29,450.00	1	5	0
1984-01-16	43,700.00	0	5	1
1984-05-31	54,000.00	0	5	2
1985-09-10	31,200.00	1	6	0
1987-03-04	53,625.00	0	6	1
1988-05-17	29,250.00	1	7	0
1989-09-04	56,500.00	0	7	1
1992-10-29	26,500.00	1	8	0
1993-08-10	24,500.00	1	9	0

- Flag must be calculated in another nested Select

Same Explain again, adding a 3rd STAT FUNCTION step:

```
3) We do an all-AMPs RETRIEVE step in TD_MAP1 from
   EMPLOYEE_SALES.employee by way of an all-rows scan with a
   condition of ("NOT (EMPLOYEE_SALES.employee.salary_amount IS NULL)")
   into Spool 6 (all_amps), which is built locally on the AMPs.  The
   size of Spool 6 is estimated with no confidence to be 26 rows (
   702 bytes).  The estimated time for this step is 0.00 seconds.
4) We do an all-AMPs STAT FUNCTION step in TD_Map1 from Spool 6 (Last
   Use) by way of an all-rows scan into Spool 9 (Last Use), which is
   assumed to be redistributed by value to all AMPs in TD_Map1.  The
   result rows are put into Spool 7 (all_amps), which is built
   locally on the AMPs.  The size is estimated with no confidence to
   be 26 rows (1,170 bytes).
5) We do an all-AMPs RETRIEVE step in TD_Map1 from Spool 7 (Last Use)
   by way of an all-rows scan into Spool 2 (all_amps), which is built
   locally on the AMPs.  The size of Spool 2 is estimated with no
   confidence to be 26 rows (1,014 bytes).  The estimated time for
   this step is 0.00 seconds.
6) We do an all-AMPs STAT FUNCTION step in TD_Map1 from Spool 2 (Last
   Use) by way of an all-rows scan into Spool 14 (Last Use), which is
   assumed to be redistributed by value to all AMPs in TD_Map1.  The
   result rows are put into Spool 12 (all_amps), which is built
   locally on the AMPs.  The size is estimated with no confidence to
   be 26 rows (1,066 bytes).  The estimated time for this step is
   0.00 seconds.
7) We do an all-AMPs RETRIEVE step in TD_Map1 from Spool 12 (Last
   Use) by way of an all-rows scan into Spool 1 (all_amps), which is
   built locally on the AMPs.  The size of Spool 1 is estimated with
   no confidence to be 26 rows (1,066 bytes).  The estimated time for
   this step is 0.00 seconds.
```

8) We do an all-AMPs **STAT** FUNCTION step in TD_Map1 from **Spool 1** (Last Use) by way of an **all-rows scan** into **Spool 19** (Last Use), which is **redistributed** by hash code to all AMPs in TD_Map1. The result rows are put into **Spool 3** (group_amps), which is **built locally** on the AMPs. The size is estimated with no confidence to be 26 rows (1,170 bytes). The estimated time for this step is 0.00 seconds.

Rewrite Recommendations

- Two OLAP calculations based on the same RESET WHEN condition add another STAT step in Explain
- But a manual rewrite will not increase the number of STAT steps

3 STAT steps

```
SELECT
   birthdate AS dt
  ,salary_amount AS amount
  -- cumulative sum which resets on NULL
  ,Sum(amount)
   Over (ORDER BY dt
         RESET WHEN amount IS NULL
         ROWS Unbounded Preceding
        ) AS cumsum
  -- group count which resets on NULL
  ,Count(amount)
   Over (ORDER BY dt
         RESET WHEN amount IS NULL
        ) AS cumcnt
FROM employee;
```

2 STAT steps

```
WITH cte_dynamic_partition AS
 ( SELECT
      birthdate AS dt
     ,salary_amount AS amount
     ,Sum(CASE WHEN amount IS NULL THEN 1 ELSE 0 END)
      Over (ORDER BY dt
            ROWS Unbounded Preceding) AS sub_part
   FROM employee
 )
SELECT t.*
  ,Sum(amount)
   Over (PARTITION BY sub_part
         ORDER BY dt
         ROWS Unbounded Preceding
        ) AS cumsum
  ,Count(amount)
   Over (PARTITION BY sub_part
        ) AS cumcnt
FROM cte_dynamic_partition AS t;
```

3) We do an all-AMPs **STAT** FUNCTION step in TD_MAP1 from **EMPLOYEE_SALES.employee** by way of an **all-rows scan** with no residual conditions into **Spool 7** (Last Use), which is assumed to be **redistributed** by value to all AMPs in TD_Map1. The result rows are put into **Spool 5** (all_amps), which is **built locally** on the AMPs. The size is estimated with low confidence to be 28 rows (1,148 bytes). The estimated time for this step is 0.00 seconds.
4) We do an all-AMPs **RETRIEVE** step in TD_Map1 from **Spool 5** (Last Use) by way of an **all-rows scan** into **Spool 1** (all_amps), which is **built locally** on the AMPs. The size of **Spool 1** is estimated with low confidence to be 28 rows (1,372 bytes). The estimated time for this step is 0.00 seconds.
5) We do an all-AMPs **STAT** FUNCTION step in TD_Map1 from **Spool 1** (Last Use) by way of an **all-rows scan** into **Spool 12** (Last Use), which is **redistributed** by hash code to all AMPs in TD_Map1. The result rows are put into **Spool 10** (all_amps), which is **built locally** on the AMPs. The size is estimated with low confidence to be 28 rows (1,596 bytes). The estimated time for this step is 0.00 seconds.
6) We do an all-AMPs **STAT** FUNCTION step in TD_Map1 from **Spool 10** (Last Use) by way of an **all-rows scan** into **Spool 15** (Last Use), which is **redistributed** by hash code to all AMPs in TD_Map1. The result rows are put into **Spool 2** (group_amps), which is **built locally** on the AMPs. The size is estimated with low confidence to be 28 rows (1,484 bytes).

Step removed in rewrite

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

Deterministic ORDER BY

- Remember: If ORDER BY columns are not unique the result of an OLAP will not be deterministic when a *cumulative* or *moving* window or Row_Number is used
- When RESET WHEN is manually rewritten two or three steps with the same non-unique ORDER exist
  - Within each step the order might differ!
- To get a correct result the same order must be applied in each step
  - `Row_Number` calculated in the first step and then used instead of the ORDER BY columns
  - Potentially less spool space, e.g INT vs. VarChar(50)
  - Automatically added in internal optimizer rewrite

```
WITH cte_dynamic_partition AS
 (
   SELECT
      birthdate AS dt
     ,salary_amount AS amount
     ,Sum(CASE WHEN amount IS NULL THEN 1 ELSE 0 END)
      Over (ORDER BY dt
            ROWS Unbounded Preceding ) AS sub_part
     ,Row_Number()
      Over (ORDER BY dt) AS sortcol
   FROM employee
 )
SELECT
   t.*
  ,Sum(amount)
   Over (PARTITION BY sub_part
         ORDER BY sortcol
         ROWS Unbounded Preceding
        ) AS cumsum
FROM cte_dynamic_partition AS t;
```

Caution:
Result is still not deterministic

Summary

- RESET WHEN is a Teradata extension to the ANSI SQL standard.
- It dynamically creates sub-partitions within window partitions.
- It can be added within all OLAP functions.
- If RESET WHEN is specified, then the ORDER BY clause must be specified also.
- You can have different RESET WHEN clauses in the same SELECT list.

OLAP Functions: RESET WHEN

Labs

Lab 1
OLAP Functions: RESET WHEN

teradata.

```
*** Query completed. 24 rows found. 3 columns returned.

yyyymm       volume sum_increased
------ ------------ -------------
201501 2,688,945.15             0
201502 2,710,585.03             1
201503 2,930,821.58             2
201504 2,891,966.16             0
201505 3,133,034.23             1
201506 3,981,986.08             2
201507 3,192,256.43             0
201508 3,256,677.50             1
201509 3,307,110.83             2
201510 3,452,239.31             3
201511 3,469,834.97             4
201512 4,591,267.24             5
201601 3,868,759.77             0
201602 4,011,560.97             1
201603 4,191,760.03             2
201604 4,281,686.21             3
201605 4,476,360.61             4
201606 5,787,912.07             5
201607 4,946,663.09             0
201608 5,080,222.29             1
201609 5,351,363.07             2
201610 5,563,825.65             3
201611 5,640,250.92             4
201612 7,476,733.19             5
```

```
HELP TABLE fin_trans;
```

Write a query based on *credit* transactions (`trans_type = 'C'`) from 2015 and 2016 returning the transaction *volume* (`sum(amount)`) per month.
Add in indicator showing the number of months where the volume keeps increasing.

Lab 2
OLAP Functions: RESET WHEN

teradata.

```
*** Query completed. 26 rows found. 5 columns returned.
```

account_id	overdrawn_from	overdrawn_to	num_days	min_balance
2,305	2017-06-19	2018-12-08	537	-2,368.45
5,092	2017-08-02	2018-12-09	494	-3,452.07
5,429	2017-12-10	2018-12-31	386	-3,322.30
4,356	2018-01-03	2018-12-31	362	-3,545.66
3,374	2015-08-18	2016-08-12	360	-315.71
10,036	2018-02-19	2018-11-16	270	-1,501.92
2,260	2017-12-18	2018-09-13	269	-1,644.67
4,127	2017-05-21	2018-02-08	263	-292.99
2,959	2018-03-06	2018-10-20	228	-1,171.17
3,326	2018-04-19	2018-12-03	228	-4,112.57
2,346	2018-03-14	2018-10-21	221	-279.06
4,059	2018-05-15	2018-12-11	210	-1,543.19
3,374	2017-06-08	2017-12-24	199	-443.62
6,609	2016-06-30	2016-12-24	177	-121.29
3,734	2018-07-10	2018-12-31	174	-1,279.01
5,343	2018-01-12	2018-06-10	149	-1,762.33
3,711	2018-03-04	2018-07-02	120	-623.58
7,203	2018-07-28	2018-11-25	120	-1,015.88
3,374	2016-09-04	2016-12-28	115	-617.44
4,618	2017-10-12	2018-02-02	113	-46.97
1,505	2018-08-25	2018-12-13	110	-1,354.48
5,250	2018-03-12	2018-06-30	110	-678.88
1,505	2018-04-27	2018-08-13	108	-1,414.59
1,878	2018-07-24	2018-11-09	108	-894.98
635	2016-06-08	2016-09-22	106	-162.56

```
HELP TABLE fin_trans;
```

Write a query finding accounts which were overdrawn (`balance < 0`) for at least 100 days.

Show the first date where the balance started being negative and the end date when it became positive again.

Add the duration and the minimum balance within this period.

Order by descending duration.

Lab 3
OLAP Functions: RESET WHEN

teradata.

```
HELP TABLE fin_trans;
```

```
*** Query completed. 2 rows found. 5 columns returned.

account_id overdrawn_from overdrawn_to num_days min_balance
---------- -------------- ------------ -------- -----------
     1,505     2018-04-27   2018-08-13      108   -1,414.59
     1,505     2018-08-25   2018-12-13      110   -1,354.48
```

Modify the previous report to return only those accounts which had *more than one* period with a balance below `-1000`.

OLAP Functions: RESET WHEN

Lab Solutions

Lab 1 Solution
OLAP Functions: RESET WHEN

```
SELECT
   Extract(YEAR From trans_date) * 100
   + Extract(MONTH From trans_date) AS yyyymm
  ,Sum(amount) AS volume
  ,Row_Number()
   Over (ORDER BY yyyymm
         RESET WHEN Lag(volume)
                    Over (ORDER BY yyyymm) > volume
     ) -1 AS sum_increased
FROM fin_trans
WHERE trans_type = 'C'
AND trans_date BETWEEN DATE '2015-01-01'
                   AND DATE '2016-12-31'
GROUP BY yyyymm
ORDER BY yyyymm
;
```

Write a query based on *credit* transactions (`trans_type = 'C'`) from 2015 and 2016 returning the transaction *volume* (`sum(amount)`) per month.
Add in indicator showing the number of months where the volume keeps increasing.

Lab 2 Solution
OLAP Functions: RESET WHEN

```
WITH cte AS
 (
   SELECT
      account_id
     ,trans_date
     ,balance
     ,Min(trans_date)
      Over (PARTITION BY account_id
            ORDER BY trans_date DESC
            RESET WHEN balance >= 0) AS min_date
   FROM fin_trans
 )
SELECT account_id
  ,min_date AS overdrawn_from
  ,Max(trans_date) AS overdrawn_to
  ,overdrawn_to - overdrawn_from AS num_days
  ,Min(balance) AS min_balance
FROM cte
GROUP BY 1, 2
HAVING num_days >= 100
ORDER BY 1,2;
```

Write a query finding accounts which were overdrawn (`balance < 0`) for at least 100 days.

Show the first date where the balance started being negative and the end date when it became positive again.

Add the duration and the minimum balance within this period.

Order by descending duration.

Lab 3 Solution
OLAP Functions: RESET WHEN

```
WITH cte AS
 (
   SELECT
      account_id
     ,trans_date
     ,balance
     ,Min(trans_date)
      Over (PARTITION BY account_id
            ORDER BY trans_date DESC
            RESET WHEN balance >= 0) AS min_date
   FROM fin_trans
 )
SELECT account_id
  ,min_date AS overdrawn_from
  ,Max(trans_date) AS overdrawn_to
  ,overdrawn_to - overdrawn_from AS num_days
  ,Min(balance) AS min_balance
FROM cte
GROUP BY 1, 2
HAVING num_days >= 100
   AND Min(balance) < -1000
QUALIFY Count(*) Over(PARTITION BY account_id) >= 2
ORDER BY 1,2;
```

Modify the previous report to return only those accounts which had *more than one* period with a balance below `-1000`.

Module 5: Scalar Subqueries

Teradata Vantage SQL Intermediate

Objectives

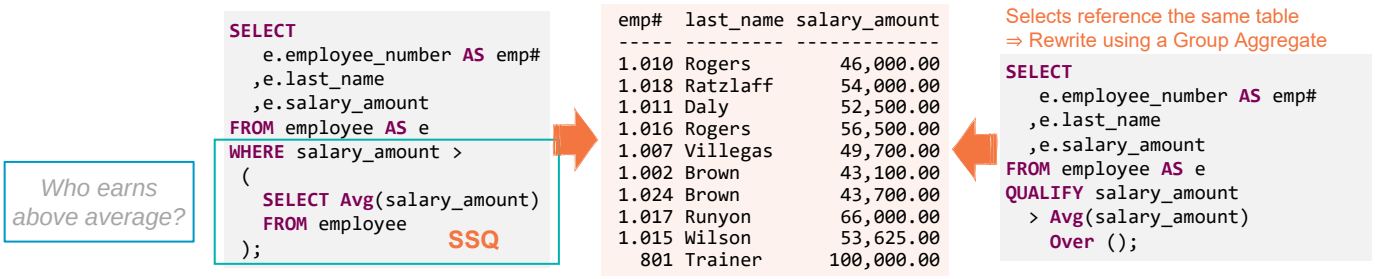
After completing this module, you will be able to:

- Differentiate between correlated and non-correlated scalar subqueries.
- Determine what happens when no rows are returned from a scalar subquery.
- Rewrite Scalar Subqueries for better performance

Types of Subqueries

- *Subqueries* return *one or more* columns and *zero or more* rows
  - Used in WHERE
- *Scalar Subqueries* return a *single* row and a *single* column ⇒ a single value
  - Can be used wherever an expression is allowed.
  - Returns NULL if no row found.

SubQuery	SQ
Correlated SubQuery	CSQ
Scalar SubQuery	SSQ
Correlated Scalar SubQuery	CSSQ



A *scalar subquery* is a subquery expression that can return a maximum of one value.

As with other types of subqueries, there are two types of scalar subqueries:
- *Correlated*
- *Noncorrelated*

A *correlated scalar subquery* returns a single value for each row of its correlated outer table set.

A *noncorrelated scalar subquery* returns a single value to its containing query.

You can think of a correlated scalar subquery as an extended column of the outer table set to which it is correlated. In the same way, you can think of a noncorrelated scalar subquery as a parameterized value.

Accordingly, you can specify a correlated scalar subquery in a statement in the same way as a column, while you can specify a noncorrelated scalar subquery in the same way as you would a parameterized value wherever a column or parameterized value is allowed. This includes specifying a scalar subquery as a standalone expression in various clauses of a DML statement, or specified within an expression.

ANSI Compliance
Scalar subqueries are compliant with the ANSI SQL:2016 standard.

Correlated Scalar Subqueries

```
SELECT
   e.employee_number AS emp#
  ,e.department_number AS dept#
  ,e.last_name
  ,salary_amount AS sal
  ,(
     SELECT budget_amount
     FROM department AS d
     WHERE e.department_number = d.department_number
   ) AS budget
  ,100 * sal / budget AS "% of budget"
FROM employee AS e
ORDER BY 1;
```

CSSQ

emp#	dept#	last_name	sal	budget	% of budget
801	999	Trainer	100,000.00	NULL	NULL
1,001	401	Hoover	25,525.00	982,300.00	2.60
1,002	401	Brown	43,100.00	982,300.00	4.39
1,003	401	Trader	37,850.00	982,300.00	3.85
1,004	401	Johnson	36,300.00	982,300.00	3.70
1,005	403	Ryan	31,200.00	932,000.00	3.35
1,006	301	Stein	29,450.00	465,600.00	6.33
1,007	403	Villegas	49,700.00	932,000.00	5.33
1,008	301	Kanieski	29,250.00	465,600.00	6.28
1,009	403	Lombardo	31,000.00	932,000.00	3.33
1,010	401	Rogers	46,000.00	982,300.00	4.68
1,011	402	Daly	52,500.00	308,000.00	17.05
1,012	403	Hopkins	37,900.00	932,000.00	4.07
1,013	401	Phillips	24,500.00	982,300.00	2.49
1,014	402	Crane	NULL	308,000.00	NULL

Selects reference different tables
⇒ Rewrite using an Outer Join

```
SELECT
   e.employee_number AS emp#
  ,e.department_number AS dept#
  ,e.last_name
  ,d.budget_amount AS budget
  ,100 * e.salary_amount / budget
FROM employee AS e
LEFT JOIN department AS d
ON e.department_number = d.department_number
;
```

Correlated Scalar Subqueries (cont.)

```
SELECT
   d.department_name AS dept_name
  ,d.budget_amount AS budget
  ,(
     SELECT Sum(salary_amount)
     FROM employee AS e
     WHERE e.department_number = d.department_number
   ) AS sumsal
  ,100 * sumsal / budget AS "% of budget"
FROM department AS d
ORDER BY budget
;
```

CSSQ

```
dept_name                budget     sumsal     % of budget
------------------------ ---------- ---------- -----------
NULL                           NULL       NULL        NULL
product planning          226,000.00       NULL        NULL
technical operations      293,800.00       NULL        NULL
marketing sales           308,000.00 200,125.00       64.98
NULL                      308,000.00  52,500.00       17.05
president                 400,000.00       NULL        NULL
research and development  465,600.00  58,700.00       12.61
education                 932,000.00 193,500.00       20.76
customer support          982,300.00 213,275.00       21.71
```

Selects reference different tables
⇒ Rewrite using an Outer Join

```
SELECT
   d.department_name AS dept_name
  ,d.budget_amount AS budget
  ,e.sumsal
  ,100 * e.sumsal / budget AS "% of budget"
FROM department AS d
LEFT JOIN
 (
   SELECT
      department_number
     ,Sum(salary_amount) AS sumsal
   FROM employee
   GROUP BY 1
 ) AS e
ON e.department_number =  d.department_number
ORDER BY budget;
```

Another possible rewrite:

```
SELECT
   d.department_name AS dept_name
  ,d.budget_amount AS budget
  ,Sum(e.salary_amount) AS sumsal
  ,100 * sumsal/budget AS "% of budget"
FROM department AS d
LEFT JOIN employee AS e
ON e.department_number =  d.department_number
GROUP BY 1,2
ORDER BY budget;
```

Rules and Restrictions

- The cardinality of a scalar subquery result cannot be greater than one.
  - *3669 More than one value was returned by a subquery.*
- If a scalar subquery returns 0 rows, then it evaluates to NULL.
- A scalar subquery can specify joins, both inner and outer, aggregation, etc.
- A scalar subquery can be either correlated or noncorrelated.
- A scalar subquery can be correlated to one or more outer tables.
- A scalar subquery can have nested levels of subqueries, which can be either scalar or nonscalar.
- A scalar subquery can be referenced in the WHERE, HAVING, GROUP BY, QUALIFY and ORDER BY clauses of a query.
- The following DML statements support scalar subqueries.
  - SELECT
  - DELETE
  - INSERT
  - UPDATE
  - ABORT / ROLLBACK

Performance Considerations

Rewriting a Scalar Subquery will result in better performance in many cases:

- One Scalar Subquery per table
  "Dispatcher Retrieve Step" in Explain ⇒ ✓ (No rewrite needed)
- Multiple Scalar Subqueries per table
  "Dispatcher Retrieve Step" per SSQ per table ⇒ ✗ (Rewrite improves performance)
- One Correlated Scalar Subquery per table
  Based on unique column (no aggregation) ⇒ Optimizer rewrites as Left Join ⇒ ✓
  Based on non-unique column (aggregation) ⇒ Complex plan ⇒ ✗
- Multiple Correlated Scalar Subqueries per table using the same conditions
  Based on unique column (no aggregation) ⇒ Optimizer rewrites as Left Join per CSSQ ⇒ ✗
  Based on non-unique column (aggregation) ⇒ Complex plan per CSSQ ⇒ ✗

```
SELECT
   d.department_name AS dept_name
  ,d.budget_amount AS budget
  ,(
     SELECT Sum(salary_amount)
     FROM employee AS e
     WHERE e.department_number =
           d.department_number
   ) AS sumsal
  ,100 * sumsal/budget AS "% of budget"
  ,(
     SELECT Count(salary_amount)
     FROM employee AS e
     WHERE e.department_number =
           d.department_number
   ) AS cntsal
FROM department AS d

ORDER BY budget;
```

```
SELECT
   d.department_name AS dept_name
  ,d.budget_amount AS budget
  ,e.sumsal
  ,100 * e.sumsal/budget AS "% of budget"
  ,e.cntsal
FROM department AS d
LEFT JOIN
 (
   SELECT
      department_number
     ,Sum(salary_amount) AS sumsal
     ,Count(*) AS cntsal
   FROM employee
   GROUP BY 1
 ) AS e
ON e.department_number =
   d.department_number
ORDER BY budget;
```

Summary

Scalar subqueries:

- Return only a single value.
- May be correlated or non-correlated.
- May be referenced wherever an expression is allowed.
- May perform bad and should be rewritten (if possible).

Scalar Subqueries

Labs

Lab 1
Scalar Subqueries

teradata.

Write a report showing all transactions of `account_id` 386 for the years 2017 and 2018.

Add a column calculating the sum of the transaction amounts of the *previous 4 weeks*.

Order by transaction date.

```
HELP TABLE fin_trans;
```

```
*** Query completed. 87 rows found. 4 columns returned.

account_id trans_date   amount sum_prev_28_days
---------- ---------- -------- ----------------
       386 2017-08-11   100.00          1875.70
       386 2017-08-11  1775.70          1875.70
       386 2017-08-31     5.01          1880.71
       386 2017-09-10   390.00           395.01
       386 2017-09-11  1775.70          2170.71
       386 2017-09-30    13.71          2179.41
       386 2017-10-10  -670.00          -656.29
       386 2017-10-11  1775.70          1119.41
       386 2017-10-31    20.02          1125.72
       386 2017-11-09 -1080.00         -1059.98
       386 2017-11-11  1775.70           715.72
       386 2017-11-30    23.51           719.21
       386 2017-12-09 -1250.00         -1226.49
       386 2017-12-11  2663.60          1437.11
       386 2017-12-31    28.57          1440.71
       386 2017-12-31    -1.46          1440.71
       386 2018-01-01  -264.00           912.71
       386 2018-01-01  -264.00           912.71
       386 2018-01-04   -96.00           816.71
       386 2018-01-08 -1520.00         -2587.99
       386 2018-01-08  -471.10         -2587.99
       386 2018-01-10  -216.00         -2803.99
       386 2018-01-11  1775.70         -1028.29
```

Lab 2
Scalar Subqueries

teradata.

Write a similar report to Lab 1 but show all transactions of *all* accounts and *all* years for accounts from `district_id` 1.

Order by account_id and transaction date.

```
HELP TABLE fin_trans;
HELP TABLE fin_account;
```

```
*** Query completed. 131812 rows found. 4 columns returned.

account_id trans_date   amount sum_prev_28_days
---------- ---------- -------- ----------------
         2 2013-02-26   110.00           110.00
         2 2013-03-12  2023.60          2133.60
         2 2013-03-28   370.00          2393.60
         2 2013-03-31     1.35          2394.95
         2 2013-04-12  2023.60          2394.95
         2 2013-04-27 -1100.00           924.95
         2 2013-04-30    10.95           934.55
         2 2013-05-12  2023.60           934.55
         2 2013-05-27 -1760.00           274.55
         2 2013-05-31    14.47           278.07
         2 2013-06-12  3035.40          1289.87
         2 2013-06-26 -2240.00           809.87
         2 2013-06-30    15.99           811.39
         2 2013-07-11  -320.00         -2544.01
         2 2013-07-12  2023.60          -520.41
         2 2013-07-19 -1314.50         -1834.91
         2 2013-07-26 -1030.00          -624.91
         2 2013-07-31    20.30          -622.06
         2 2013-07-31    -1.46          -622.06
         2 2013-08-05  -726.60         -1348.66
         2 2013-08-07  -290.00         -1638.66
```

Lab 3
Scalar Subqueries

teradata.

Modify the previous report and add a column showing the transaction count of the previous four weeks.

```
HELP TABLE fin_trans;
HELP TABLE fin_account;
```

```
*** Query completed. 131812 rows found. 5 columns returned.

account_id trans_date   amount sum_prev_28_days count_prev_28_days
---------- ---------- -------- ---------------- ------------------
         2 2013-02-26   110.00           110.00                  1
         2 2013-03-12  2023.60          2133.60                  2
         2 2013-03-28   370.00          2393.60                  2
         2 2013-03-31     1.35          2394.95                  3
         2 2013-04-12  2023.60          2394.95                  3
         2 2013-04-27 -1100.00           924.95                  3
         2 2013-04-30    10.95           934.55                  3
         2 2013-05-12  2023.60           934.55                  3
         2 2013-05-27 -1760.00           274.55                  3
         2 2013-05-31    14.47           278.07                  3
         2 2013-06-12  3035.40          1289.87                  3
         2 2013-06-26 -2240.00           809.87                  3
         2 2013-06-30    15.99           811.39                  3
         2 2013-07-11  -320.00         -2544.01                  3
         2 2013-07-12  2023.60          -520.41                  4
         2 2013-07-19 -1314.50         -1834.91                  5
         2 2013-07-26 -1030.00          -624.91                  5
         2 2013-07-31    20.30          -622.06                  6
         2 2013-07-31    -1.46          -622.06                  6
         2 2013-08-05  -726.60         -1348.66                  7
         2 2013-08-07  -290.00         -1638.66                  8
```

Lab 4
Scalar Subqueries

teradata.

Rewrite Lab 1 with a join instead of a Correlated Scalar Subquery.

```
HELP TABLE fin_trans;
```

```
*** Query completed. 87 rows found. 4 columns returned.

account_id trans_date   amount sum_prev_28_days
---------- ---------- -------- ----------------
       386 2017-08-11   100.00          1875.70
       386 2017-08-11  1775.70          1875.70
       386 2017-08-31     5.01          1880.71
       386 2017-09-10   390.00           395.01
       386 2017-09-11  1775.70          2170.71
       386 2017-09-30    13.71          2179.41
       386 2017-10-10  -670.00          -656.29
       386 2017-10-11  1775.70          1119.41
       386 2017-10-31    20.02          1125.72
       386 2017-11-09 -1080.00         -1059.98
       386 2017-11-11  1775.70           715.72
       386 2017-11-30    23.51           719.21
       386 2017-12-09 -1250.00         -1226.49
       386 2017-12-11  2663.60          1437.11
       386 2017-12-31    28.57          1440.71
       386 2017-12-31    -1.46          1440.71
       386 2018-01-01  -264.00           912.71
       386 2018-01-01  -264.00           912.71
       386 2018-01-04   -96.00           816.71
       386 2018-01-08 -1520.00         -2587.99
       386 2018-01-08  -471.10         -2587.99
       386 2018-01-10  -216.00         -2803.99
       386 2018-01-11  1775.70         -1028.29
```

Scalar Subqueries

Lab solutions

Lab 1 Solution
Scalar Subqueries

teradata.

Write a report showing all transactions of `account_id` 386 for the years 2017 and 2018.

Add a column calculating the sum of the transaction amounts of the *previous 4 weeks*.

Order by transaction date.

```
SELECT
   account_id
  ,trans_date
  ,amount
  ,(
     SELECT Sum(amount)
     FROM fin_trans AS t2
     WHERE t2.account_id = t.account_id
       AND t2.trans_date BETWEEN t.trans_date - 27
                             AND t.trans_date
   ) AS sum_prev_28_days
FROM fin_trans AS t
WHERE account_id = 386
  AND trans_date BETWEEN DATE '2017-01-01'
                     AND DATE '2018-12-31'
ORDER BY trans_date;
```

Lab 2 Solution
Scalar Subqueries

teradata.

Write a similar report to Lab 1 but show all transactions of *all* accounts and *all* years for accounts from `district_id` 1.

Order by account_id and transaction date.

```
SELECT
   t.account_id
  ,t.trans_date
  ,t.amount
  ,(
     SELECT Sum(amount)
     FROM fin_trans AS t2
     WHERE t2.account_id = t.account_id
       AND t2.trans_date BETWEEN t.trans_date - 27
                             AND t.trans_date
   ) AS sum_prev_28_days
FROM fin_trans AS t
JOIN fin_account AS a
  ON a.account_id = t.account_id
WHERE a.district_id = 1
ORDER BY t.account_id, t.trans_date;
```

Lab 3 Solution
Scalar Subqueries

teradata.

Modify the previous report and add a column showing the transaction count of the previous four weeks.

```
SELECT
   t.account_id
  ,t.trans_date
  ,t.amount
  ,(
     SELECT Sum(amount)
     FROM fin_trans AS t2
     WHERE t2.account_id = t.account_id
       AND t2.trans_date BETWEEN t.trans_date - 27
                             AND t.trans_date
   ) AS sum_prev_28_days
  ,(
     SELECT Count(*)
     FROM fin_trans AS t2
     WHERE t2.account_id = t.account_id
       AND t2.trans_date BETWEEN t.trans_date - 27
                             AND t.trans_date
   ) AS count_prev_28_days
FROM fin_account AS a
JOIN fin_trans AS t
  ON a.account_id = t.account_id
WHERE a.district_id = 1
ORDER BY t.account_id, t.trans_date;
```

Runtime doubled!

Lab 4 Solution
Scalar Subqueries

teradata.

Rewrite Lab 1 with a join instead of a Correlated Scalar Subquery.

```
SELECT t.account_id
  ,t.trans_date
  ,t.amount
  ,Sum(t2.amount)  AS sum_prev_28_days
FROM fin_trans AS t
JOIN fin_trans AS t2
  ON t2.account_id = t.account_id
 AND t2.trans_date BETWEEN t.trans_date - 27
                       AND t.trans_date
WHERE t.account_id = 386
  AND t.trans_date BETWEEN DATE '2017-01-01'
                       AND DATE '2018-12-31'
GROUP BY 1,2,3
ORDER BY t.trans_date;
```

Caution: Returns only 86 instead of 87 rows and not the same numbers.

On 2018-01-01 the same amount was transferred twice:

```
386 2018-01-01  -264.00          912.71
386 2018-01-01  -264.00          912.71
```

Aggregation combines these two rows into one:

```
386 2018-01-01  -264.00         1825.42
```

Lab 4 Solution (cont.)
Scalar Subqueries

Rewrite Lab 1 with a join instead of a Correlated Scalar Subquery.

```
SELECT t.account_id
  ,t.trans_date
  ,t.amount
  ,Sum(t2.amount)  AS sum_prev_28_days
FROM fin_trans AS t
JOIN fin_trans AS t2
  ON t2.account_id = t.account_id
 AND t2.trans_date BETWEEN t.trans_date - 27
                       AND t.trans_date
WHERE t.account_id = 386
  AND t.trans_date BETWEEN DATE '2017-01-01'
                       AND DATE '2018-12-31'
GROUP BY 1, 2, 3, trans_id
ORDER BY t.trans_date;
```

Adding the PK of the table to the GROUP BY fixes the result.

Lab 4 Solution (cont.)
Scalar Subqueries

teradata.

```
SELECT
   t.account_id
  ,t.trans_date
  ,t.amount
  ,Sum(t2.amount) AS sum_prev_28_days
  ,Count(*) AS count_prev_28_days
FROM fin_account AS a
JOIN fin_trans AS t
  ON a.account_id = t.account_id
JOIN fin_trans AS t2
  ON t2.account_id = t.account_id
 AND t2.trans_date BETWEEN t.trans_date - 27
                       AND t.trans_date
WHERE a.district_id = 1
GROUP BY 1,2,3, t.trans_id
ORDER BY
   t.account_id
  ,t.trans_date;
```

Added in Lab 3

Rewrite of Lab 2 & 3 with a join.

Adding COUNT doesn't change runtime.

```
-- 24 AMP system
RunTime IO_logical    CPU  SpoolUsage lab
------- ---------- ------ ----------- --------
   1.35     24,739  22.93 300,883,968 -- Lab 2-SSQ
   4.33     46,356  80.16 404,602,880 -- Lab 3-SSQ*2
   0.60      6,653   9.92  46,268,416 -- Lab 2-Join
   0.62      6,742   9.60  47,333,376 -- Lab 3-Join*2
```

Module 6: CREATE TABLE AS

Teradata Vantage SQL Intermediate

Objectives

After completing this module, you will be able to:

- Create a clone of an existing table.
- Create a new table using SELECT.
- Override attributes for the new table.
- Choose whether to ingest data or not.
- Choose to copy statistics or not.

CREATE TABLE AS *"Existing Table"*

This form of the syntax is used to create an exact copy of another table.

The source object MUST be a table, no view.

Steps:

- Verify the existence of the source.
- Clone the source table:
  - `WITH NO DATA` Definition only
  - `WITH DATA` Definition and data
- Verify what has been created.

Remember:
SHOW returns the *current* Data Dictionary definitions for the object.

Any slight variation may not create an exact copy.

```
CREATE TABLE employee
AS employee_sales.employee
WITH NO DATA
;
SHOW TABLE employee;
```

```
CREATE SET TABLE trainee_1.employee ,FALLBACK ,
     NO BEFORE JOURNAL,
     NO AFTER JOURNAL,
     CHECKSUM = DEFAULT,
     DEFAULT MERGEBLOCKRATIO,
     MAP = TD_MAP1
     (
      employee_number INTEGER NOT NULL,
      manager_employee_number INTEGER,
      department_number INTEGER,
      job_code INTEGER,
      last_name CHAR(20) CHARACTER SET LATIN
                         NOT CASESPECIFIC NOT NULL,
      first_name VARCHAR(30) CHARACTER SET LATIN
                         NOT CASESPECIFIC NOT NULL,
      hire_date DATE FORMAT 'YYYY-MM-DD' NOT NULL,
      birthdate DATE FORMAT 'YYYY-MM-DD' NOT NULL,
      salary_amount DECIMAL(10,2))
UNIQUE PRIMARY INDEX Employee_Index ( employee_number );
```

Required Privileges

The privileges required to execute CREATE TABLE with an AS clause are identical to those for a standard CREATE TABLE statement.

```
CREATE TABLE new_table
AS [ tablename | (select statement) ]
WITH [NO] DATA
[ AND [NO] [ STATISTICS | STAT[S] ] ]
;
```

Copying Table Definitions

You can create work or test tables that are identical to, or based on, base tables in a database. For example, you can use work tables with the same structure as base tables to speed up processing because it is faster to insert rows into a unpopulated work table and later merge the updates into a persistent base table than it is to update the base table directly one update at a time.

You can create test tables that are identical to some or all of the base tables used by live applications for prototyping new applications on a test system.

The AS clause option of CREATE TABLE provides a simple method for copying some or all definitions from an existing table to a new table.

The AS … WITH NO DATA clause copies only the source table or query expression definitions to the new table, while the AS … WITH DATA clause copies both the source table definitions and its data to the new table.

CREATE TABLE AS – Cloning Attributes

Most column level attributes are copied:

- Column names
- Data types
- Default values
- NOT NULL constraints
- CHECK constraints
- UNIQUE constraints
- PRIMARY KEY constraints

Most table level attributes are copied:

- SET / MULTISET
- All Indexes

Relations to other tables are not copied:

- Foreign Key References
- Triggers

This will make the new table multiset.

```
CREATE MULTISET TABLE dept1 AS department
WITH NO DATA;
```

Caution!
Adding an index drops all existing indexes. The USI causes the Primary Index to default to the first defined column as a NUPI.

```
CREATE TABLE dept1 AS department
WITH NO DATA
UNIQUE INDEX (department_name);
```

This will keep the UPI and add the USI.

```
CREATE TABLE dept1 AS department
WITH NO DATA
UNIQUE PRIMARY INDEX (Department_Number)
UNIQUE INDEX (department_name);
```

The following list describes attributes that are not copied to a target table from the source table or query expression using an AS clause with CREATE TABLE.

- The following source table attributes are not copied to the target table because they affect multiple tables.
  - REFERENCES constraints
  - Triggers
- When an existing source table (not a subquery) is specified in the AS clause and indexes are specified in the Index Definition clause, then source table indexes are not copied to the target table.

CREATE TABLE AS *SELECT*

This form of the syntax is used to materialize the result of a SELECT.

All column and table level attributes are reset to defaults or dropped:

- FORMAT reset
- Default values dropped
- NOT NULL constraints dropped
- CHECK constraints dropped
- UNIQUE constraints dropped
- PRIMARY KEY constraints dropped
- Indexes dropped ⇒ *NUPI on 1st column* or *NoPI*

Following attributes depend on session mode:

- Teradata mode: `SET NOT CASESPECIFIC`
- ANSI mode: `MULTISET CASESPECIFIC`

```
CREATE TABLE employee
AS ( SELECT * FROM employee_sales.employee )
WITH DATA
;
SHOW TABLE employee;
```

```
CREATE SET TABLE trainee_1.employee ,FALLBACK ,
     NO BEFORE JOURNAL,
     NO AFTER JOURNAL,
     CHECKSUM = DEFAULT,
     DEFAULT MERGEBLOCKRATIO,
     MAP = TD_MAP1
     (
      employee_number INTEGER,
      manager_employee_number INTEGER,
      department_number INTEGER,
      job_code INTEGER,
      last_name CHAR(20) CHARACTER SET LATIN
                         NOT CASESPECIFIC,
      first_name VARCHAR(30) CHARACTER SET LATIN
                         NOT CASESPECIFIC,
      hire_date DATE FORMAT 'YY/MM/DD',
      birthdate DATE FORMAT 'YY/MM/DD',
      salary_amount DECIMAL(10,2))
PRIMARY INDEX ( employee_number );
```

When you specify a subquery as your source definition:

- Column-level attributes are not copied, except for data type.
- A column that maps to an expression assumes the data type of the expression result.
- Table-level attributes are not copied.
- Indexes from tables specified in the subquery are not copied. Any indexes you want for the target table must be specified explicitly. If you do not specify an explicit primary index, the system chooses a default primary index.

Renaming Columns

Any valid Select can be materialized.

Rules and restrictions are similar to *Views*, *Derived Tables* and *Common Table Expressions:*

- No ORDER BY allowed
- Computed columns must be aliased

```
SHOW TABLE emp1;
```

```
CREATE SET TABLE trainee_1.emp1 ,FALLBACK,
     NO BEFORE JOURNAL,
     NO AFTER JOURNAL,
     CHECKSUM = DEFAULT,
     DEFAULT MERGEBLOCKRATIO,
     MAP = TD_MAP1
     (
      emp# INTEGER,
      department_number INTEGER,
      monthly_sal DECIMAL(10,2))
PRIMARY INDEX ( emp );
```

```
CREATE TABLE trainee_1.emp1 AS
 (
   SELECT
      employee_number AS emp#            Optional
     ,department_number
     ,salary_amount / 12 AS monthly_sal  Required
   FROM employee
 )
WITH DATA;
```

```
CREATE TABLE emp1
 (
   emp#                    Required
  ,department_number       Required
  ,monthly_sal             Required
 )
AS
 (
   SELECT
      employee_number AS nnn
     ,department_number
     ,salary_amount / 12
   FROM employee
 )
WITH DATA;
```

When using both forms for aliasing in the same request, the alias list overrides the AS alias.

Changing Column Attributes

Every column level attribute besides the data type can be set.

```
CREATE TABLE dept1
AS
 (
   SELECT
     department_number              AS dept
    ,CAST(budget_amount AS INTEGER) AS budget
   FROM department
 )
WITH DATA;
```

```
CREATE TABLE dept1
 (
   dept DEFAULT 0 UNIQUE NOT NULL
  ,budget CHECK (budget > 0)
 )
AS
 (
   SELECT
      department_number
     ,CAST(budget_amount AS INTEGER)
   FROM department
 )
WITH DATA;
```

Column attributes not available from the select statement must be included using this form for alias listing.

Note that CAST must be used in the projection.

Usefulness is questionable, may be finally similar to a full Create Table without listing the data types.

You cannot specify *data types* for source table columns, but you can specify *column attributes* for them. You can also specify table level constraints for the target table to be created.

Adding Unique and Primary Key Constraints

`PRIMARY KEY` or `UNIQUE` constraints may be simple methods to create UPIs and USIs.

Recall, however, that these both must be defined as `NOT NULL`.

Each alias must match an existing column in order of the `CREATE TABLE`.

`SHOW TABLE` doesn't show the index definitions.

You may have to use `HELP INDEX` or `dbc.IndicesV` to see what has been implemented.

```
CREATE TABLE dept1
 (
   deptno   UNIQUE      NOT NULL
  ,deptname PRIMARY KEY NOT NULL
  ,budget
  ,manager
 )
AS department
WITH DATA
;
SHOW TABLE dept1
;
```

```
CREATE SET TABLE TRAINEE_1.dept1 ,FALLBACK ,
     NO BEFORE JOURNAL,
     NO AFTER JOURNAL,
     CHECKSUM = DEFAULT,
     DEFAULT MERGEBLOCKRATIO,
     MAP = TD_MAP1
     (
      deptno INTEGER NOT NULL,
      deptname CHAR(30) CHARACTER SET LATIN NOT CASESPECIFIC NOT NULL,
      budget DECIMAL(10,2),
      manager INTEGER,
PRIMARY KEY ( deptname ),
UNIQUE ( deptno ))
;
```

You can also append an AND STATISTICS option to a WITH [NO] DATA clause to copy any collected source table statistics or empty histograms to a target table. If you specify WITH NO DATA AND STATISTICS, the system sets up the appropriate statistical histograms in the dictionary, but does not populate them with source table statistics. This is referred to as *zeroed statistics*.

```
HELP INDEX dept1;
```

```
 Unique? Primary//or//Secondary? Column Names Index Id
 ------- ---------------------- ------------ --------
 Y       P                      deptname            1
 Y       S                      deptno              4
```

```
SELECT
   IndexNumber
  ,IndexType
  ,UniqueFlag
  ,IndexName
  ,ColumnName
  ,ColumnPosition
FROM dbc.IndicesV
WHERE DatabaseName = DATABASE
  AND TableName = 'dept1'
ORDER BY IndexNumber, ColumnPosition
;
```

```
 IndexNumber IndexType UniqueFlag IndexName ColumnName ColumnPosition
 ----------- --------- ---------- --------- ---------- --------------
           1 K         Y          NULL      deptname                1
           4 U         Y          NULL      deptno                  1
```

Copying Statistics

Statistics may be propagated from a source table to a target table under certain conditions. This may save valuable resources.

```
CREATE TABLE trainee_1.dept1
AS department
WITH DATA
AND STATISTICS
;
```

Propagates:
- The specified column definitions and the data associated with those columns.
- Statistics are copied.

```
CREATE TABLE trainee_1.dept1
AS department
WITH NO DATA
AND STATISTICS
;
```

Propagates:
- The specified column definitions only.
- Statistics definition is copied but set to 0 rows.

```
COLLECT STATS ON trainee_1.dept1
FROM department
;
```

Statistics can also be copied from an existing table with matching column names and data types.

Summary

- You can use CREATE TABLE AS to easily create new tables from existing tables.
- You can specify explicit table attributes for the new table.
- You can specify explicit column attributes for the new table.
- You can create a clone or use a subquery to build a new table.
- You can change SET to MULTISET.
- You can propagate statistics to the new table from the sourcing table.

Module 7: Temporary Tables

Teradata Vantage SQL Intermediate

Objectives

After completing this module, you will be able to:

- use non-permanent tables holding intermediate results
- create, modify and populate non-permanent tables
- understand the characteristics of non-permanent tables
- identify volatile table limitations of non-permanent tables
- collects statistics on non-permanent tables
- create indexs on non-permanent tables
- distinguish between the two ON COMMIT options
- simplify processes using views and macros

Reasons for Interim Tables

Interim Tables may be used to:

- develop report processes
- hold common data used in subsequent SQL statements
- simplify subsequent SQL statements
- hold denormalized data for
  - aggregations
  - repeating groups
- save resources
- avoid unnecessary access right checks

Permanent Tables as Interim Tables

Using permant tables for intermediate storage of data requires:

- Creating the table
  - stores the definition and access rights in the data dictionary
  - may be used by many users with appropriate access rights
  - consumes permanently user disk space until dropped
- Data can be loaded when created (`CREATE TABLE AS`) or with `INSERT SELECT`
- Table defintion can be modified with ALTER TABLE
- Table must be dropped if not needed anymore

```
CREATE TABLE daily_net_trans
 (
   account_number INTEGER
  ,total_trans_amount DECIMAL(14,2)
 )
UNIQUE PRIMARY INDEX (Account_Number)
;

INSERT INTO daily_net_trans
SELECT
   account_number
  ,SUM(trans_amount)
FROM fin_trans
GROUP BY account_number
;

SELECT *
FROM  daily_net_trans
WHERE account_number IN
  (  20035223
     ,20024048
     ,20045853
  );
```

Pros and Cons for Interim Perm Tables

Pros about this strategy:

- Simpler SQL
- Doesn't have to continually reproduce aggregate results
- PI choice of table may facilitate subsequent SQL
- Open for use by multiple sessions if needed
- Space survives a restart
- Subsequent updates, inserts, and deletes can be performed on the rows of the table

Cons about this strategy:

- Separate steps to create and populate table (or `CREATE TABLE AS … WITH DATA`)
- Requires extra perm space for temp table
- Table may need to be dropped when no longer needed
  - If not, then extra maintenance may be required
- Data Dictionary access is necessary for creating, dropping and any DML

Handling Temporarily Needed Data

- To avoid unnecessary usage of system resources, it might be useful to save data temporarily.
- Using a permanent table will save CPU time but requires access rights and consumes disk space.

	Scope	Lifetime	Syntax	Access Rights required?	Space usage
Derived Table	Statement	Statement runtime	Defined in FROM	no	Spool Space
Common Table Expression (CTE)	Statement	Statement runtime	Defined in WITH clause	no	Spool Space
View	Permanent	DROP VIEW	CREATE VIEW	yes	Spool Space
Volatile Table (This module)	Session	End of session *or* DROP TABLE	CREATE VOLATILE TABLE (Session local)	no	Spool Space
Global Temporary Table (This module)	Session	**Base table**: DROP TABLE **Instance**: End of session *or* DROP TEMPORARY TABLE	CREATE GLOBAL TEMPORARY TABLE	yes	**Instance** materialized in Temp Space
Permanent Table	Permanent	DROP TABLE	CREATE TABLE	yes	Perm Space

Derived Tables & Common Table Expressions

- Nested Select statement similar to a view ("*named table expression*")
- Defined in **`FROM`** (*Derived Table*) or the **`WITH`** clause (*Common Table Expression*)
- Local to the query
- Discarded when query finishes
- Uses Spool space

Views

- **`CREATE VIEW`** stores the source code of a Select in the Data Dictionary
- Can be accessed like a table in **`FROM`** ("*named table expression*")
- Requires access rights
- Uses Spool space

Volatile Tables

- Uses **`CREATE VOLATILE TABLE`** syntax
- Local to the session (available to all queries during the session)
- Discarded automatically at session end
- No Data Dictionary involvement
- Uses Spool space

Global Temporary Tables

- Uses **`CREATE GLOBAL TEMPORARY TABLE`** syntax (DBA creates the *base table*)
- *Base table* definition stored in Data Dictionary
- Requires access rights
- INSERT materializes an *instance* local to the session (like volatile tables)
- *Instance* discarded at session end
- Uses Temp space

Volatile Table Syntax

```
CREATE VOLATILE TABLE vt_deptsal
 (
   deptno    SMALLINT
  ,avgsal    DECIMAL(9,2)
  ,maxsal    DECIMAL(9,2)
  ,minsal    DECIMAL(9,2)
  ,sumsal    DECIMAL(9,2)
  ,empcnt    SMALLINT
 )
;
INSERT INTO vt_deptsal
SELECT
   department_number
  ,AVG(salary_amount)
  ,MAX(salary_amount)
  ,MIN(salary_amount)
  ,SUM(salary_amount)
  ,COUNT(employee_number)
FROM  employee
GROUP BY 1
;                *** Insert completed. 7 rows added.
SELECT *
FROM vt_deptsal
;                *** Query completed. No rows found.
```

- `LOG` indicates that a transaction journal is maintained. This is the default.
- Use `NO LOG` to improve performance
- `ON COMMIT DELETE ROWS` indicates to delete all table rows after a commit (*end transaction* step). This is the default.
- Use `ON COMMIT PRESERVE ROWS` to keep the content after a commit.
- The primary index defaults to using the first column

Required Privileges
No privileges are required to create, access, modify, or drop volatile tables.

The definition is of a volatile table is retained in memory only for the duration of the session in which it is defined. Space usage is charged to the login user spool space. Because volatile tables are private to the session that creates them, the system does not check the creation, access, modification, and drop privileges. A single session can materialize up to 1,000 volatile tables.

The contents and the definition of a volatile table are dropped when a system reset occurs.

You cannot create a column-partitioned volatile table or normalized volatile table.

You cannot create secondary, hash, or join indexes on a volatile table.

If you frequently reuse particular volatile table definitions, consider writing a macro that contains the CREATE TABLE text for those volatile tables (or switch to a Global Temporary Table).

Volatile Table Traps

Teradata mode session

```
CREATE VOLATILE TABLE vt_deptsal
 (
   deptno    SMALLINT
  ,avgsal    DECIMAL(9,2)
  ,maxsal    DECIMAL(9,2)
  ,minsal    DECIMAL(9,2)
  ,sumsal    DECIMAL(9,2)
  ,empcnt    SMALLINT
 )
;
INSERT INTO vt_deptsal
SELECT
   department_number
  ,AVG(salary_amount)
  ,MAX(salary_amount)
  ,MIN(salary_amount)
  ,SUM(salary_amount)
  ,COUNT(employee_number)
FROM  employee
GROUP BY 1;
*** Insert completed. 7 rows added.

SELECT *
FROM vt_deptsal;
*** Query completed. No rows found.
```

In Teradata session mode each (single or multi-statement) request is a transaction.

- Multiple request can be wrapped in
  `BEGIN TRANSACTION;`
  …
  `END TRANSACTION;`

The `INSERT` statement has been executed successful.

But, because it is an implicit transaction and the default is `ON COMMIT DELETE ROWS`, the rows are deleted when reaching `END TRANSACTION`.

The following `SELECT` reflects this.

In ANSI session mode the `COMMIT;` statement finishes a transaction.

- But clients like Teradata Studio usually switch on an *Autocommit* option which automatically submits `COMMIT;` after every request.

ON COMMIT …
Action to perform when a transaction completes.

Delete or preserve the contents of an instance of a global temporary or volatile table when a transaction completes.
This option is valid for global temporary and volatile tables only.

ON COMMIT DELETE ROWS
Clears an instance of a global temporary or volatile table of all rows.
DELETE is the default.

ON COMMIT PRESERVE ROWS
Retains the rows in the global temporary or volatile table after the transaction commits.

LOG
Transient journaling is performed for global temporary and volatile tables.
Update, insert, or delete operations on the global temporary or volatile table are logged in the transient journal. This is the default.

NO LOG
Transient journal logging of rows is not performed, reducing the system overhead of logging.
If an error or restart occurs and the table is defined as NO LOG, then any update, insert, or delete operations on the global temporary or volatile table cannot be recovered.
If the table is defined as NO LOG, a transient journal is generated for the transaction and the content of any materialized global temporary table or volatile table is emptied when a transaction aborts.

Volatile Table Trap Avoided

```
-- you cannot use ALTER TABLE
DROP TABLE vt_deptsal
;
-- create a new table
CREATE VOLATILE TABLE vt_deptsal
 (
   deptno    SMALLINT
  ,avgsal    DECIMAL(9,2)
  ,maxsal    DECIMAL(9,2)
  ,minsal    DECIMAL(9,2)
  ,sumsal    DECIMAL(9,2)
  ,empcnt    SMALLINT
 )
ON COMMIT PRESERVE ROWS
;
INSERT INTO vt_deptsal
SELECT
   department_number
  ,AVG(salary_amount)
  ,MAX(salary_amount)
  ,MIN(salary_amount)
  ,SUM(salary_amount)
  ,COUNT(employee_number)
FROM  employee
GROUP BY 1
;
*** Insert completed. 7 rows added.
```

Use ON COMMIT PRESERVE ROWS to avoid unnecessary confusion.

```
SELECT *
FROM vt_deptsal
ORDER BY maxsal DESC;
```

deptno	avgsal	maxsal	minsal	sumsal	empcnt
402	52.500,00	52.500,00	52.500,00	52.500,00	2
NULL	43.316,67	56.500,00	34.700,00	129.950,00	3
403	38.700,00	49.700,00	31.000,00	193.500,00	6
999	100.000,00	100.000,00	100.000,00	100.000,00	1
401	35.545,83	46.000,00	24.500,00	213.275,00	7
301	29.350,00	29.450,00	29.250,00	58.700,00	3
501	50.031,25	66.000,00	26.500,00	200.125,00	4

```
-- returns a list of all volatile tables in the current session.
HELP VOLATILE TABLE
;
```

Pros and Cons for Volatile Tables

Pros about this strategy:

- Simpler to use.
- PI choice of table may facilitate subsequent SQL.
- Local (private) (i.e., cannot be viewed from other sessions).
- Subsequent updates, inserts, and deletes can be performed on the rows of the table.
- Table is automatically dropped at session end or manually by user.
- No dictionary access required.
- You can collect statistics on volatile table.

Cons about this strategy:

- Separate steps to create and populate table or use `CREATE TABLE AS … WITH DATA`
- Requires extra spool space for temp table.
- Secondary indexes allowed in `CREATE VOLATILE TABLE` but cannot be created with `CREATE INDEX`.
- Holds on to spool longer (data may survive transaction).

Volatile Table Restrictions

Up to 1000 volatile tables are allowed for a single session.

At the time you create a volatile table, the name must be unique among all global and permanent object names in the database that has the name of the login user.

A user can create a volatile table with the same name in different sessions, they don't see each other.

Options not permitted:

- Permanent journaling
- Referential integrity
- CHECK constraints
- Column default values
- Column titles
- Named indexes

```
-- database name if specified must be logon name
CREATE VOLATILE TABLE username.table1 ...;

-- database name always default to logon name
-- independent of the session default database
CREATE VOLATILE TABLE table1 ...;

-- cannot use foreign spool space -> error
CREATE VOLATILE TABLE databasename.table1 ...;
```

Volatile Tables
The primary index for a volatile table can be nonpartitioned or row-partitioned. The table can also be defined without a primary index (NoPI).

When you create an unqualified volatile temporary table, the login user space is used as the default database for the table, regardless of the default database that is currently specified.
When you create a qualified volatile table, you must specify the login user database. Otherwise, the system returns an error.

Global Temporary Tables

Global Temporary Tables:

- Require a base definition which is stored in the DD.
- An instance is materialized by first SQL INSERT statement to the table.

They are very similar to Volatile Tables:

- Each instance of global temp table is local to a user and session.
- Materialized tables are dropped automatically at session end.
- Have LOG and ON COMMIT PRESERVE/DELETE options.
- Materialized table contents aren't shareable with other sessions.

Differences from Volatile Tables:

- Base definition is permanent and kept in DD.
- Requires DML privileges necessary to materialize the table.
- Space is charged against an allocation of "temporary space".
- User can materialize up to 2000 global tables per session.
- They can survive a system restart.
- They are "tuneable" – i.e., object of CREATE INDEX and COLLECT STATS.
- DEFAULT values, named CHECK constraints and INDEXes are allowed.

Global Temporary Tables Syntax

Base table definition is stored in DD/D
Like Volatile tables, the default is: ON COMMIT DELETE ROWS
ALTER TABLE can be used to change defaults if no materialized instance exists.
Access rights needs to be granted to foreign users to be able to materialize a temp table.

```
CREATE GLOBAL TEMPORARY TABLE gt_deptsal
 (
   deptno    SMALLINT
  ,avgsal    DECIMAL(9,2)
  ,maxsal    DECIMAL(9,2)
  ,minsal    DECIMAL(9,2)
  ,sumsal    DECIMAL(9,2)
  ,empcnt    SMALLINT
 )
UNIQUE PRIMARY INDEX (deptno)
ON COMMIT PRESERVE ROWS
;
SHOW TABLE gt_deptsal
;
```

```
CREATE SET GLOBAL TEMPORARY TABLE trainee_1.gt_deptsal,
     FALLBACK ,
     CHECKSUM = DEFAULT,
     DEFAULT MERGEBLOCKRATIO,
     MAP = TD_MAP1,
     LOG
     (
      deptno SMALLINT,
      avgsal DECIMAL(9,2),
      maxsal DECIMAL(9,2),
      minsal DECIMAL(9,2),
      sumsal DECIMAL(9,2),
      empcnt SMALLINT)
UNIQUE PRIMARY INDEX ( deptno )
ON COMMIT PRESERVE ROWS;
```

Global Temporary Tables
Global temporary tables have a persistent *definition* but do not have persistent *contents* across sessions.

The following list describes characteristics of global temporary tables:

- Space usage is charged to the temporary space of the user. A minimum of 4KB times the number of AMPs on the system of permanent space is also required to contain the table header for each global temporary table.
- A single session can materialize up to 2,000 global temporary tables at one time. You materialize a global temporary table locally by referencing it in a data manipulation request. To materialize a global temporary table, you must have the appropriate privilege on the base global temporary table or on the containing database or user as required by the request that materializes the table.
- Any number of different sessions can materialize the same table definition, but the contents are different depending on the DML requests made against the individual materialized tables during the course of a session.
- You can log global temporary table updates by specifying the LOG option for the CREATE TABLE statement. LOG is the default.
- You can save the contents of a materialized global temporary table across transactions by specifying ON COMMIT PRESERVE ROWS as the last keywords in the CREATE TABLE statement. The default is not to preserve table contents after a transaction completes (DELETE).
- The primary index for a global temporary table can be nonpartitioned or row-partitioned. The table can be defined without a primary index.

Map for Global Temporary Tables and Volatile Tables
You can specify an existing contiguous or sparse map for global temporary tables and volatile tables.

Global Temporary Tables Syntax (cont.)

```
INSERT INTO gt_deptsal
SELECT
   department_number
  ,AVG(salary_amount)
  ,MAX(salary_amount)
  ,MIN(salary_amount)
  ,SUM(salary_amount)
  ,COUNT(employee_number)
FROM  employee
GROUP BY 1
;
SELECT *
FROM gt_deptsal
;
```

The developer grants the rights `INSERT` and `SELECT` on that table to users/roles.

The user materializes a session local instance of the table by inserting into the table.

- A row is inserted in `dbc.TempTables` to note the instance.

The table remains materialized until logoff, or until it is dropped.

You can perform other DML on the materialized instance.

deptno	avgsal	maxsal	minsal	sumsal	empcnt
402	52.500,00	52.500,00	52.500,00	52.500,00	2
NULL	43.316,67	56.500,00	34.700,00	129.950,00	3
403	38.700,00	49.700,00	31.000,00	193.500,00	6
999	100.000,00	100.000,00	100.000,00	100.000,00	1
401	35.545,83	46.000,00	24.500,00	213.275,00	7
301	29.350,00	29.450,00	29.250,00	58.700,00	3
501	50.031,25	66.000,00	26.500,00	200.125,00	4

The database does not check privileges for the materialized instances of global temporary tables because those tables exist only for the duration of the session in which they are materialized.

Global Temporary Tables – Space Allocation

Temporary space, like perm space, survives a system reset.

Like spool space, it is deallocated at the session end.

Check temp available to user via `dbc.UsersV`.

```
SELECT
   TempSpace
  ,SpoolSpace
FROM dbc.UsersV;
```

```
TempSpace      SpoolSpace
-------------- --------------
39.662.995.374 39.662.995.374
```

Check used space via `dbc.DiskSpaceV`.

- Reports space for all instances in all sessions of a user combined, no table level info available.

```
SELECT
   SUM(MaxTemp)     AS temp_available
  ,SUM(CurrentTemp) AS temp_used
FROM dbc.DiskSpaceV
WHERE DatabaseName = USER;
```

```
temp_available temp_used
-------------- ---------
 5,000,000,000   393,216
```

`dbc.AllTempTablesV` carries one row for each materialized instance of any temporary table.

```
SELECT *
FROM dbc.AllTempTablesV
WHERE UserName = USER;
```

```
HostNo SessionNo UserName  B_DatabaseName B_TableName E_TableId
------ --------- --------- -------------- ----------- -----------------
     1     1.229 trainee_1 trainee_1      GT_DEPTSAL  00-80-01-00-00-00
```

Global Temporary Tables
Stored Definition vs. Materialized Instance

teradata.

The materialized instance may be modified locally, it may be different to the stored definition.

Base Table	Materialized Instance
HELP TABLE	n/a
SHOW TABLE	SHOW TEMPORARY TABLE
DROP TABLE	DROP TEMPORARY TABLE
CREATE INDEX … ON	CREATE INDEX … ON TEMPORARY
DROP INDEX … ON	DROP INDEX … ON TEMPORARY
HELP INDEX	HELP TEMPORARY INDEX
COLLECT STATISTICS ON	COLLECT STATISTICS ON TEMPORARY
HELP STATISTICS	HELP TEMPORARY STATISTICS
DROP STATISTICS ON	DROP STATISTICS ON TEMPORARY

Renaming Volatile and Global Temporary Tables

You cannot rename a volatile table.

You can rename a global temporary table only when there are no materialized instances of that table anywhere in the system.

Summary

- Volatile tables use *spool space*.
- Global Temporary Tables use *temp space*.
- ON COMMIT DELETE ROWS is the default.
- Volatile Tables and instances of Global Temporary Tables last until the user's session logoff, then the database drops them automatically.
- Volatile tables are best suited for ad-hoc usage.
- Global temporary tables require a different syntax structure for manipulating base definitions than for materialized instances
- Global temporary tables can have materialized instances that may be "tuned" for performance by defining indexes and collecting statistics on them.

Temporary Tables

Labs

Lab 1
Volatile Tables

teradata.

Write a report showing all transactions of *all* accounts and *all* years for `district_id` 1.
Add a column calculating the sum of the transaction amounts of the previous 4 weeks.

Order by account_id and transaction date.

- To improve performance materialize all rows for district_id 1 in a Volatile Table first.
  - Do you need a SET or a MULTISET table?
  - Choose the Primary Index based on the following *Correlated Scalar Subquery*.
- Rewrite both the Scalar Subquery & the Join solution.

Compare runtime to the previous solutions and check how much it improved.

```
HELP TABLE fin_trans;
HELP TABLE fin_account;
```

Rewrite of Scalar Subqueries Lab 2

```
*** Query completed. 131812 rows found. 4 columns returned.

account_id trans_date   amount sum_prev_28_days
---------- ---------- -------- ----------------
         2 2013-02-26   110.00           110.00
         2 2013-03-12  2023.60          2133.60
         2 2013-03-28   370.00          2393.60
         2 2013-03-31     1.35          2394.95
         2 2013-04-12  2023.60          2394.95
         2 2013-04-27 -1100.00           924.95
         2 2013-04-30    10.95           934.55
         2 2013-05-12  2023.60           934.55
         2 2013-05-27 -1760.00           274.55
         2 2013-05-31    14.47           278.07
         2 2013-06-12  3035.40          1289.87
         2 2013-06-26 -2240.00           809.87
         2 2013-06-30    15.99           811.39
         2 2013-07-11  -320.00         -2544.01
         2 2013-07-12  2023.60          -520.41
         2 2013-07-19 -1314.50         -1834.91
         2 2013-07-26 -1030.00          -624.91
         2 2013-07-31    20.30          -622.06
         2 2013-07-31    -1.46          -622.06
         2 2013-08-05  -726.60         -1348.66
         2 2013-08-07  -290.00         -1638.66
```

Lab 2
Volatile Tables

teradata.

Modify the previous report and add a column showing the transaction count of the previous four weeks.

```
HELP TABLE fin_trans;
HELP TABLE fin_account;
```

Rewrite of Scalar Subqueries Lab 3

```
*** Query completed. 131812 rows found. 5 columns returned.

account_id trans_date   amount sum_prev_28_days count_prev_28_days
---------- ---------- -------- ---------------- ------------------
         2 2013-02-26   110.00           110.00                  1
         2 2013-03-12  2023.60          2133.60                  2
         2 2013-03-28   370.00          2393.60                  2
         2 2013-03-31     1.35          2394.95                  3
         2 2013-04-12  2023.60          2394.95                  3
         2 2013-04-27 -1100.00           924.95                  3
         2 2013-04-30    10.95           934.55                  3
         2 2013-05-12  2023.60           934.55                  3
         2 2013-05-27 -1760.00           274.55                  3
         2 2013-05-31    14.47           278.07                  3
         2 2013-06-12  3035.40          1289.87                  3
         2 2013-06-26 -2240.00           809.87                  3
         2 2013-06-30    15.99           811.39                  3
         2 2013-07-11  -320.00         -2544.01                  3
         2 2013-07-12  2023.60          -520.41                  4
         2 2013-07-19 -1314.50         -1834.91                  5
         2 2013-07-26 -1030.00          -624.91                  5
         2 2013-07-31    20.30          -622.06                  6
         2 2013-07-31    -1.46          -622.06                  6
         2 2013-08-05  -726.60         -1348.66                  7
         2 2013-08-07  -290.00         -1638.66                  8
```

Compare runtime to lab 1 solutions.

Temporary Tables

Lab solutions

Lab 1 Solution (SSQ) Volatile Tables

Write a report showing all transactions of *all* accounts and *all* years for `district_id` 1.
Add a column calculating the sum of the transaction amounts of the previous 4 weeks.
Order by account_id and transaction date.

```
CREATE MULTISET VOLATILE TABLE vt_amounts AS
 (
   SELECT
      t.account_id
     ,t.trans_date
     ,t.amount
   FROM fin_account AS a
   JOIN fin_trans AS t
     ON a.account_id = t.account_id
   WHERE a.district_id = 1
 ) WITH DATA
PRIMARY INDEX(account_id)
ON COMMIT PRESERVE ROWS
;
```

```
SELECT
   t.account_id
  ,t.trans_date
  ,t.amount
  ,(
     SELECT Sum(amount)
     FROM vt_amounts AS t2
     WHERE t2.account_id = t.account_id
       AND t2.trans_date BETWEEN t.trans_date - 27
                             AND t.trans_date
   ) AS sum_prev_28_days
FROM vt_amounts AS t
ORDER BY t.account_id, t.trans_date
;
```

Runtime improved

Lab 1 Solution (Join)
Volatile Tables

teradata.

Write a report showing all transactions of *all* accounts and *all* years for `district_id` 1.
Add a column calculating the sum of the transaction amounts of the previous 4 weeks.
Order by account_id and transaction date.

```
DROP  TABLE vt_amounts;

CREATE MULTISET VOLATILE TABLE vt_amounts AS
 (
   SELECT
      t.account_id
     ,t.trans_date
     ,t.amount
     ,t.trans_id
   FROM fin_account AS a
   JOIN fin_trans AS t
     ON a.account_id = t.account_id
   WHERE a.district_id = 1
 ) WITH DATA
PRIMARY INDEX(account_id)
ON COMMIT PRESERVE ROWS;
```

```
SELECT
   t.account_id
  ,t.trans_date
  ,t.amount
  ,Sum(t2.amount) AS sum_prev_28_days
FROM vt_amounts AS t
JOIN vt_amounts AS t2
  ON t2.account_id = t.account_id
 AND t2.trans_date BETWEEN t.trans_date - 27
                       AND t.trans_date
GROUP BY 1,2,3,t.trans_id
ORDER BY t.account_id, t.trans_date
;
```

Runtime improved

Lab 2 Solution (SSQ) Volatile Tables

teradata.

Modify the previous report and add a column showing the transaction count of the previous four weeks.

```
SELECT
   t.account_id
  ,t.trans_date
  ,t.amount
  ,(
     SELECT Sum(amount)
     FROM vt_amounts AS t2
     WHERE t2.account_id = t.account_id
       AND t2.trans_date BETWEEN t.trans_date - 27
                             AND t.trans_date
   ) AS sum_prev_28_days
  ,(
     SELECT Count(*)
     FROM vt_amounts AS t2
     WHERE t2.account_id = t.account_id
       AND t2.trans_date BETWEEN t.trans_date - 27
                             AND t.trans_date
   ) AS count_prev_28_days
FROM vt_amounts AS t
ORDER BY t.account_id, t.trans_date
;
```

Runtime doubled!

Lab 2 Solution (Join)
Volatile Tables

teradata.

Modify the previous report and add a column showing the transaction count of the previous four weeks.

Runtime doesn't change

```
SELECT
   t.account_id
  ,t.trans_date
  ,t.amount
  ,Sum(t2.amount) AS sum_prev_28_days
  ,Count(*) AS count_prev_28_days
FROM vt_amounts AS t
JOIN vt_amounts AS t2
  ON t2.account_id = t.account_id
 AND t2.trans_date BETWEEN t.trans_date - 27
                       AND t.trans_date
GROUP BY 1,2,3,t.trans_id
ORDER BY t.account_id, t.trans_date;
```

```
-- 24 AMP system
-- Using base tables
RunTime IO_logical    CPU   SpoolUsage
------- ---------- ------  -----------
   1.35     24,739  22.93   300,883,968
   4.33     46,356  80.16   404,602,880

   0.60      6,653   9.92    46,268,416
   0.62      6,742   9.60    47,333,376
```

```
-- Using Volatile tables
             RunTime IO_logical    CPU   SpoolUsage
             ------- ---------- ------  -----------
Create VT       0.19      2,159   2.22   13,910,016
SSQ             0.61      4,365   9.00   38,363,136
SSQ*2           1.16      8,398  18.23   48,476,160
SSQ*3           1.50     11,322  22.98   55,255,040
Join            0.36      2,648   5.28   46,268,416
Join*2          0.33      2,737   5.31   47,333,376
Join*3          0.33      2,860   5.24   48,402,432
```

Module 8: Data Manipulation – INSERT, UPDATE, DELETE

Teradata Vantage SQL Intermediate

Objectives

After completing this module, you will be able to:

- Insert rows into a table one-at-a-time.
- Insert rows using a bulk operation like "INSERT … SELECT".
- Update one or more values for one or more rows of a table.
- Delete one or more rows from a table.
- Determine the effect of CASESPECIFIC and NOT CASESPECIFIC on DML.
- Insert, update, or delete rows from tables using joins and subqueries.

Manipulating Data via SQL

The syntax in this module is referred to as **DML – Data Manipulation Language**.

	`INSERT`	Inserts a single row into a table.
	`INSERT-SELECT`	Inserts zero or more rows to a table using values selected from one or more existing source tables.
	`UPDATE`	Changes column values in existing rows of a table.
	`DELETE`	Removes rows from a table.
Covered in next module	`MERGE INTO`	Inserts, update or deletes rows in a target table based on a matching condition with a source table.

The title is using the word “manipulating” in a very general sense to mean “changing the data in a table.” In our instance we are saying that we can change the table’s data by either:

- Inserting new rows
- Updating existing rows
- Deleting existing rows

The additional syntax, that is referred to as not being covered in this module, is a feature that typically gets employed in ETL (Extract-Transform-Load) processing strategies.

Inserting a Single Row

Insert a new employee into the employee table.
Value order assumes table column order.

```
INSERT INTO employee
VALUES (1210, NULL, 401, 412101, 'Smith', 'James', DATE '2009-03-03', DATE '1966-04-21', 41000)
;
```

Insert a new employee with only partial data.
Column list may be any order.
Value list must match order of column list
Inserts default values (discussed later) for missing columns.

```
INSERT INTO employee
       (last_name, first_name, hire_date,         birthdate,         salary_amount, employee_number)
VALUES ('Garcia' , 'Maria'   , DATE '2006-10-27', DATE '1974-11-10', 76500.00     , 1291)
;
```

There is no typing shortcut facility for explicitly inserting multiple rows with this form.

This slide discusses the ability of SQL to insert a single row into a table. The two forms provide alternate methods for performing an insert.

The first method is used if you know the order of the columns in the table create statement, and assumes that you know their data types as well.

The second form also assumes that you know the data types for each column, but rather than knowing the table's column order, you reference the columns by name. This means that you need not know the order because the database can now match the columns by name. However, you must list the order of their values so they match the order in the column list.

As a Teradata Extension to Standard SQL:

- **INSERT** can be abbreviated as **INS**.
- **INTO** and **VALUES** are optional keywords.

Inserting an Apostrophe within a String

To insert an apostrophe into a character string, use the following notation:

```
INSERT INTO Department
VALUES(111, 'President''s Club', 400000.00, 222)
;
```

You can retrieve this row using the same notation like this:

```
SELECT *
FROM department
WHERE department_name = 'President''s Club'
;
```

```
department_number department_name    budget_amount manager_employee_number
----------------- ------------------ ------------- ------------------------
              111 President's Club       400000,00                      222
```

Remember: Strings are enclosed in single quotes
Names are enclosed in double quotes
Be aware of typographic characters such as ‘ “ ` when using cut and paste

Since the single quote is used to delimit a text string, and may also appear as a character in the string, a question might be asked, *“How do we include the single quote as a character in the text string during a single row insert?”* The answer to this question is what this slide discusses. This is not an issue in bulk row inserts, which follows on a later slide.

Inserting Default Values

```
CREATE TABLE test_defaults
 (
   c1 INTEGER
  ,c2 VARCHAR(10)  DEFAULT 'n/a'
  ,c3 CHAR(5)      WITH DEFAULT ---> DEFAULT '     '
  ,c4 INTEGER      DEFAULT 10
  ,c5 INTEGER      WITH DEFAULT ---> DEFAULT 0
  ,c6 VARCHAR(128) DEFAULT USER
  ,c7 DATE         DEFAULT Current_Date
  ,c8 TIME(2)      DEFAULT Current_Time(2)
  ,c9 TIMESTAMP(2) DEFAULT Current_Timestamp(2)
 )
;
INSERT INTO test_defaults
VALUES(1 , , , , , , , , )
;
INSERT INTO test_defaults
VALUES(2,DEFAULT,DEFAULT,DEFAULT,DEFAULT,DEFAULT,DEFAULT,DEFAULT,DEFAULT)
;
INSERT INTO test_defaults
VALUES(3,NULL,NULL,NULL,NULL,NULL,NULL,NULL,NULL)
;

SELECT *
FROM test_defaults
ORDER BY 1
;
```

Explicit NULLs do not trigger DEFAULTs

```
c1 c2   c3    c4   c5   c6         c7         c8           c9
-- ---- ----- ---- ---- ---------- ---------- ------------ ----------------------
 1 n/a          10    0 trainee_1  2022-10-12 15:42:31.77  2022-10-12 15:42:31.77
 2 n/a          10    0 trainee_1  2022-10-12 15:42:31.91  2022-10-12 15:42:31.91
 3 NULL NULL   NULL NULL NULL       NULL       NULL         NULL
```

`DEFAULT` is Standard SQL-compliant

The ANSI default values are:

- `NULL` for each columns

`WITH DEFAULT` is a Teradata extension to the SQL-2016 standard

- Character default is spaces `'  '`
- VarChar default is an empty string `''`
- Numeric default is zero

Default values are those which are inserted when a value isn't present. Each column has a default value, whether specified explicitly, in a CREATE TABLE or not. The ANSI default value is the same for character as it is for numeric, which is NULL, and is established simply by not defining your own default.

You may set your own default value in a create table, in which case, when a value for such a column is not present in an insert, the database puts that explicit (user defined) default value into the column so defined. This is exemplified on this slide by "DEFAULT USER" and "DEFAULT 10."

The "WITH DEFAULT" syntax is a Teradata extension to the ANSI syntax. It says to change the ANSI default, from NULL, to either spaces (for character values) or zero (for numeric values).

Methods for inserting defaulted values are shown on this slide.

Default Values and NOT NULL

Inserting a NULL into a non-NULL column fails:

3811 Column 'c1' is NOT NULL.
Give the column a value.

```
CREATE MULTISET TABLE test1
 (
  c1 INTEGER NOT NULL
 )
;
INSERT INTO test1 VALUES();         -- fails
INSERT INTO test1 VALUES(DEFAULT); -- fails
INSERT INTO test1 VALUES(NULL);     -- fails

SELECT *
FROM test1;
```

```
c1
---------
```

```
CREATE MULTISET TABLE test2
 (
  c1 INTEGER NOT NULL WITH DEFAULT
 )
;
INSERT INTO test2 VALUES();         -- inserts a zero
INSERT INTO test2 VALUES(DEFAULT); -- inserts a zero
INSERT INTO test2 VALUES(NULL);     -- fails

SELECT *
FROM test2;
```

```
c1
---------
        0
        0
```

```
CREATE MULTISET TABLE test3
 (
  c1 VARCHAR(128) NOT NULL DEFAULT USER
 )
;
INSERT INTO test3 VALUES();         -- inserts user name
INSERT INTO test3 VALUES(DEFAULT); -- inserts user name
INSERT INTO test3 VALUES(NULL);     -- fails

SELECT *
FROM test3;
```

```
c1
---------
TRAINEE_1
TRAINEE_1
```

INSERT … SELECT

INSERT … SELECT:

- Is used to copy rows from one table to another.
- Uses a SELECT statement to define a subset of rows and/or a subset of columns to be inserted.
- Number of columns in the Select and the target table must match.
  - Column name list may be used (omitted columns set to default)
  - Watch out for the correct order of columns!
- May need no spool (direct merge step) if source and target table share the exact same DDL.
  - Check Explain.

May be optimized: *"Fastpath Insert-Select"*

- No Transient Journal needed if
  - The target table is empty at the begin of the transaction.
  - And the Optimizer knows the transaction will be committed
    i.e., Explain shows *"Finally, we send out an END TRANSACTION step".*

```
INSERT INTO emp_copy
SELECT *
FROM employee_sales.employee
;
```

A more “complex” INSERT ... SELECT
1. Create the "birthdays" table

```
CREATE VOLATILE TABLE vt_birthdays
(
   empno  INTEGER   NOT NULL
  ,lname  CHAR(20)  NOT NULL
  ,fname  VARCHAR(30)
  ,birth  DATE FORMAT'yyyy-mm-dd'
)
UNIQUE PRIMARY INDEX (empno)
ON COMMIT PRESERVE ROWS;
```

2. Populate the "birthdays" table from employee

```
INSERT INTO vt_birthdays
SELECT
   employee_number
  ,last_name
  ,first_name
  ,birthdate
FROM employee_sales.employee
WHERE department_number = 403;
```

The “insert-select” syntax is commonly referred to as being a “bulk” operation. The term “bulk” meaning that we are inserting many rows into a table “in bulk” amounts.

As can be seen, this syntax can be used to initially load an unpopulated table, or incrementally load into a populated table.

CASESPECIFIC and SET Tables

SET tables disallow duplicate rows, even when the primary index is not unique.

```
CREATE SET TABLE t1
(
  c1 INTEGER
 ,c2 VARCHAR(20) NOT CASESPECIFIC
)
PRIMARY INDEX (c2)
;
INSERT INTO t1 VALUES(1,'ABC');
INSERT INTO t1 VALUES(2,'ABC');
INSERT INTO t1 VALUES(2,'abc'); ✗

SELECT * FROM t1       1 ABC
;                      2 ABC


-- trying to duplicate the data
INSERT INTO t1
SELECT * FROM t1
;
-- 0 rows processed.

SELECT *               1 ABC
FROM t1                2 ABC
WHERE c2 = 'abc'
;
```

```
CREATE SET TABLE t2
(
  c1 INTEGER
 ,c2 VARCHAR(20) CASESPECIFIC
)
PRIMARY INDEX (c2)
;
INSERT INTO t2 VALUES(1,'ABC');
INSERT INTO t2 VALUES(2,'ABC');
INSERT INTO t2 VALUES(2,'abc');

SELECT * FROM t2       1 ABC
;                      2 ABC
                       2 abc

-- trying to duplicate the data
INSERT INTO t2
SELECT * FROM t2
;
-- 0 rows processed.

SELECT *               2 abc
FROM t2
WHERE c2 = 'abc'
;
```

```
CREATE MULTISET TABLE t3
(
  c1 INTEGER
 ,c2 VARCHAR(20) NOT CASESPECIFIC
)
PRIMARY INDEX (c2)
;
INSERT INTO t3 VALUES(1,'ABC');
INSERT INTO t3 VALUES(2,'ABC');
INSERT INTO t3 VALUES(2,'abc');

SELECT * FROM t3       1 ABC
;                      2 ABC
                       2 abc

-- trying to duplicate the data
INSERT INTO t3
SELECT * FROM t3
;
-- 3 rows processed.

SELECT *               1 ABC
FROM t3                2 ABC
WHERE c2 = 'abc'       2 abc
;                      1 ABC
                       2 ABC
                       2 abc
```

```
CREATE MULTISET TABLE t4
(
  c1 INTEGER
 ,c2 VARCHAR(20) CASESPECIFIC
)
PRIMARY INDEX (c2)
;
INSERT INTO t4 VALUES(1,'ABC');
INSERT INTO t4 VALUES(2,'ABC');
INSERT INTO t4 VALUES(2,'abc');

SELECT * FROM t4       1 ABC
;                      2 ABC
                       2 abc

-- trying to duplicate the data
INSERT INTO t4
SELECT * FROM t4
;
-- 3 rows processed.

SELECT *               1 abc
FROM t4                2 abc
WHERE c2 = 'abc'
;
```

This slide show the effect of columns being defined as either CASESPECIFIC or NOT CASESPECIFIC within the context of a table defined as SET or MULTISET.

A single row insert into a SET table fails with a *2802 Duplicate row error in traine_1. t1*.
A SET table silently (does not fail) discards any duplicate rows into a SET table via an Insert-Select.

A MULTISET table retains any duplicate rows into a MULTISET table.

As a SELECT is concerned, case sensitivity always has an influence on a result when referenced in the predicate.

The examples on this slide where run in a Teradata Mode session.

Remember that in ANSI Mode session, all character literals are sensitive to case (i.e., they are CASESPECIFIC), thus the `WHERE c2 = 'abc'` will never return upper case `'ABC'`.

UPDATE

UPDATE modifies the content of rows in a table.

Darlene Johnson is changing to another department with a different job.
The new department has the following properties:

- Department: 403
- Job: 432101
- Manager: 1005

WARNING: No *Recycle Bin* on Teradata!
What has been updated & committed can only be restored from a backup.

```
SELECT
   employee_number          AS emp#
  ,last_name                AS lnm
  ,first_name               AS fnm
  ,manager_employee_number  AS mngr#
  ,department_number        AS dept#
  ,job_code
FROM  employee
WHERE employee_number = 1004
;

UPDATE employee
  SET department_number = 403
     ,job_code = 432101
     ,manager_employee_number = 1005
WHERE employee_number = 1004
;

SELECT
   employee_number          AS emp#
  ,last_name                AS lnm
  ,first_name               AS fnm
  ,manager_employee_number  AS mngr#
  ,department_number        AS dept#
  ,job_code
FROM  employee
WHERE employee_number = 1004;
```

```
emp# lnm      fnm     mngr# dept# job_code
---- -------- ------- ----- ----- --------
1004 Johnson  Darlene  1003   401   412101
```

```
emp# lnm      fnm     mngr# dept# job_code
---- -------- ------- ----- ----- --------
1004 Johnson  Darlene  1005   403   432101
```

The UPDATE clause is used to change data values for columns. The example shown is simple enough since the predicate is specifying that we are updating only a single row (referencing a UPI value like Employee_Number guarantees this).

Updating may also be done in bulk. In this case “bulk” means more than one row. An example of a bulk operation would be, for instance, if the WHERE condition was to be removed from this update so that all rows would be updated. Another example of a bulk operation is provided in the next discussion on the following slide.

UPDATE Based on Other Tables

Updates with joins and subqueries allow a table's rows to be updated based on information in another table.

Give everyone in all the support departments a 10% raise. (Assume we don't know the department numbers for all of the support departments.)

Three different approaches:

- Subquery
- Correlated subquery
- Inner join

Subquery

```
UPDATE employee
  SET salary_amount = salary_amount * 1.10
WHERE department_number IN
 ( SELECT department_number
   FROM  department
   WHERE department_name LIKE '%support%'
 );
```

Correlated subquery

```
UPDATE employee AS e
  SET salary_amount = salary_amount * 1.10
WHERE EXISTS
 ( SELECT *
   FROM   department AS d
   WHERE  e.department_number = d.department_number
     AND  d.department_name LIKE '%support%'
 );
```

Implicit inner join only (no explicit JOIN)

```
UPDATE e
FROM employee AS e, department AS d
  SET salary_amount = salary_amount * 1.10
WHERE e.department_number = d.department_number
  AND department_name LIKE '%support%'
;
```

UPDATE Based on Derived Tables

UPDATE with a FROM clause is used to update columns of a table with values from another table.

Increase the budget of each department by the sum of the salaries of its employees.

```
UPDATE department
FROM
 (
   SELECT department_number AS dept#
     ,Sum(salary_amount) AS sum_sal
   FROM employee_sales.employee
   GROUP BY 1
 ) AS src
SET budget_amount = budget_amount + src.sum_sal
WHERE department_number = src.dept#
;
```

Defining a table alias

```
UPDATE d
FROM department AS d,
 (
   SELECT department_number AS dept#
     ,Sum(salary_amount) AS sum_sal
   FROM employee_sales.employee
   GROUP BY 1
 ) AS src
SET budget_amount = d.budget_amount + src.sum_sal
WHERE d.department_number = src.dept#
;
```

DELETE

DELETE removes rows from a table.

When using a WHERE clause specifying which rows should be deleted, the rows will be placed as a “before image” in the “Transient Journal” for a possible rollback in case of an error.

Deleting all rows from a table will be optimized if all following conditions are met:

- Delete step in Explain shows "*full table deletion*".
- Delete is the final DML against the table.
- Optimizer knows the transaction will be committed, i.e., Explain shows "Finally, we send out an END TRANSACTION step".
- No Triggers or Foreign Keys based on this table exist.

Then *"Fastpath Delete"* skips the “Transient Journal” and will remove all rows from the table very quickly.

WARNING: No *Recycle Bin* on Teradata!
What has been deleted & committed
can only be restored from a backup.

```
-- no employees working in department anymore
DELETE
FROM employee
WHERE  department_number = 301
;

-- syntax variations to delete all rows
-- "truncate"

DELETE
FROM employee ALL
;

DELETE
FROM employee
;

-- Teradata extension (deprecated)
-- FROM keyword is optional
DELETE employee
;

-- Teradata extension (deprecated)
-- DELETE can be abbreviated to DEL
DEL employee
;
```

The DELETE clause can be used to remove rows from a table. When used conditionally, as in the first example, it can be used to remove targeted rows based upon the explicit values referenced in the predicate. The examples at the bottom are repeated from an earlier module, and remove all rows from the table very quickly.

The keyword “FROM” is noise, and is not required. The keyword DELETE may be abbreviated to “DEL”. (strongly discouraged).

DELETE Based on Other Tables

Remove all of the employees who are assigned to a temporary department (i.e., for which the department name is 'Temp').

Subquery

```
DELETE
FROM employee
WHERE department_number IN
 (
   SELECT department_number
   FROM   department
   WHERE  department_name  = 'temp'
 )
;
```

Correlated subquery

```
DELETE
FROM employee AS e
WHERE EXISTS
 (
   SELECT *
   FROM department AS d
   WHERE e.department_number = d.department_number
     AND d.department_name = 'Temp'
 )
;
```

Implicit join

```
DELETE
FROM employee As e
WHERE e.department_number = department.department_number
  AND department.department_name = 'Temp'
;
```

Summary

- DML syntax includes options for changing the data in a table.
- INSERT can be used to add rows to a table, either singly or in bulk.
- UPDATE can be used to change (update) column values either individually or in bulk.
- DELETE can be used to remove rows either individually or in bulk.
- Case sensitivity can affect the DML taught in this module.
- Default values may be referenced in inserts, updates, and deletes

Data Manipulation – INSERT, UPDATE, DELETE

Labs

Lab 1
Data Manipulation

teradata.

Create a MULTISET copy of the `finance_payroll.fin_trans` table without data in your *trainee_n* user.
Insert/Select *all rows* from the `finance_payroll.fin_trans` table.
The following statements should be based on the Primary Key (`trans_id`)
plus the Partitioned Primary Index (`account_id, trans_date`) of the fin_trans table.
Write an *update* to merge the rows which exist in the target table from the `finance_payroll.trans_staging_1` table.
Add the source table `amount` to the target table `balance.`
Write an *Insert-Select* to merge the rows which don't exist in the target table from the `finance_payroll.trans_staging_1` table.
Repeat the *Update* & *Insert-Select* steps using `finance_payroll.trans_staging_2` table.
Delete the rows from the target table which exist in the `finance_payroll.trans_staging_2` table.
Drop the table.

Repeat all steps, but add a USI on `(trans_id, balance)` and compare runtimes.

Repeat all steps, but add a NUSI on `(trans_date, balance)` and compare runtimes.

Data Manipulation – INSERT, UPDATE, DELETE

Lab Solutions

Lab 1 Solution
Data Manipulation

```
CREATE MULTISET TABLE fin_trans_copy AS finance_payroll.fin_trans WITH NO DATA;

CREATE UNIQUE INDEX (trans_id, balance) ON fin_trans_copy; -- added in 2nd run
CREATE INDEX (trans_date, balance) ON fin_trans_copy;      -- added in 3rd run
```

```
INSERT INTO fin_trans_copy
SELECT * FROM  finance_payroll.fin_trans;
```

```
UPDATE tgt
FROM fin_trans_copy AS tgt
    ,finance_payroll.trans_staging_1 AS src
SET amount = src.amount
   ,balance = tgt.balance + src.amount

WHERE tgt.account_id = src.account_id
  AND tgt.trans_date = src.trans_date
  AND tgt.trans_id   = src.trans_id;
```

```
INSERT INTO fin_trans_copy
SELECT *
FROM  finance_payroll.trans_staging_1 AS src
WHERE NOT EXISTS
 (
   SELECT 1
   FROM fin_trans_copy AS tgt
   WHERE tgt.account_id = src.account_id
     AND tgt.trans_date = src.trans_date
     AND tgt.trans_id   = src.trans_id
 );
```

```
DELETE FROM fin_trans_copy AS tgt
WHERE  EXISTS
 ( SELECT 1
   FROM finance_payroll.trans_staging_2 AS src
   WHERE tgt.account_id = src.account_id
     AND tgt.trans_date = src.trans_date
     AND tgt.trans_id = src.trans_id
 );
```

Lab 1 Solution
Data Manipulation (cont.)

teradata.

Resource usage data from DBC.QryLogV:
24-AMP system with Block Level Compression (BLC)

RunTime	IO_logical	IO_physical	CPU	SpoolUsage	InsCnt	UpdCnt	DelCnt	step	idx
0.33	4,304	1,228	4.76	50,561,024	1,056,320	0	0	ins_empty	no
1.37	10,123	3,415	6.26	50,561,024	1,056,320	0	0	ins_empty	USI
0.94	11,465	3,246	7.24	50,561,024	1,056,320	0	0	ins_empty	NUSI
0.31	3,663	375	4.91	11,812,864	0	305,946	0	update	no
1.31	18,309	4,237	12.93	14,954,496	0	305,946	0	update	USI
1.46	14,397	2,497	15.79	11,812,864	0	305,946	0	update	NUSI
1.37	129,116	2,824	21.01	100,696,064	307,550	0	0	insert	no
2.03	134,468	4,174	23.86	100,696,064	307,550	0	0	insert	USI
1.91	136,362	4,723	25.96	100,696,064	307,550	0	0	insert	NUSI
1.65	103,079	3,406	23.02	105,365,504	233,247	209,577	0	merge	no
2.55	124,833	8,543	33.59	105,365,504	233,247	209,577	0	merge	USI
2.48	126,090	6,439	31.65	105,365,504	233,247	209,577	0	merge	NUSI
0.48	10,165	2,133	6.71	13,496,320	0	0	442,824	delete	no
0.65	16,023	2,810	9.71	13,496,320	0	0	442,824	delete	USI
1.49	19,213	3,588	20.77	13,496,320	0	0	442,824	delete	NUSI

Lab 1 Solution
Data Manipulation (cont.)

teradata.

Resource usage data from DBC.QryLogV:
24-AMP system with Block Level Compression (BLC) switched off
SET QUERY_BAND = 'BLOCKCOMPRESSION=no;' UPDATE FOR SESSION VOLATILE;

RunTime	IO_logical	IO_physical	CPU	SpoolUsage	InsCnt	UpdCnt	DelCnt	step	idx
0.40	4,383	1,257	2.78	50,561,024	1,056,320	0	0	ins_empty	no
1.30	8,681	2,900	3.93	50,561,024	1,056,320	0	0	ins_empty	USI
0.81	10,582	2,995	3.99	50,561,024	1,056,320	0	0	ins_empty	NUSI
0.24	3,657	375	2.70	11,812,864	0	305,946	0	update	no
1.04	18,666	4,314	7.71	16,957,440	0	305,946	0	update	USI
1.13	14,477	2,339	14.22	11,812,864	0	305,946	0	update	NUSI
1.79	167,253	2,873	18.50	100,696,064	307,550	0	0	insert	no
2.11	171,519	4,355	20.17	100,696,064	307,550	0	0	insert	USI
2.22	189,635	4,870	23.13	100,696,064	307,550	0	0	insert	NUSI
1.74	103,268	3,580	16.14	105,365,504	233,247	209,577	0	merge	no
2.27	125,694	8,825	23.20	105,365,504	233,247	209,577	0	merge	USI
2.76	127,552	7,797	26.05	105,365,504	233,247	209,577	0	merge	NUSI
0.32	9,926	1,805	3.52	13,496,320	0	0	442,824	delete	no
0.70	15,852	2,953	5.88	13,496,320	0	0	442,824	delete	USI
1.35	19,010	3,610	17.63	13,496,320	0	0	442,824	delete	NUSI

Module 9: Data Manipulation – MERGE INTO

Teradata Vantage SQL Intermediate

Objectives

After completing this module, you will be able to:

- Insert, Update & Delete rows using MERGE INTO
- Describe the rules for matching a row in the target table.
- Use MERGE for inserts only.
- Use MERGE for updates only.
- Reference subqueries inside the MERGE.

MERGE Single Row

- Merges a source row into a target table based on whether the target row satisfies a specified matching condition with the source row.
- `WHEN MATCHED`: conditions match and the target row is UPDATEed.
- `WHEN NOT MATCHED`: conditions don't match and the source row is INSERTed into the target table.

- Restrictions:
  - Target table must be a table with a Primary Index.
  - Matching conditions must reference all Primary Index columns and for PPI tables all *partitioning* columns.
  - Multiple conditions must be ANDed.
  - Matching conditions must specify an *equality* constraint.
  - The UPDATE must not change a PI value.

```
MERGE INTO department AS tgt
USING VALUES (700, 'Shipping', 800000.00)
   AS src (dept#, name, budget)
ON src.dept# = tgt.department_number

WHEN MATCHED THEN UPDATE
SET department_name   = src.name
   ,budget_amount     = src.budget

WHEN NOT MATCHED THEN INSERT
VALUES
 ( src.dept#
  ,src.name
  ,src.budget
  ,NULL -- no manager_employee_number
 )
;
```

ANSI Compliance
The ANSI definition for this statement is MERGE INTO, while the Teradata definition is MERGE, with INTO being an optional keyword.

MERGE is mostly ANSI SQL:2016-compliant, some exceptions exist.

Explain shows both steps executed in parallel, but a single row MERGE will *either* UPDATE the target row *or* INSERT the source row.

```
1)  First, we execute the following steps in parallel.
    1)  We do a single-AMP UPDATE step from TRAINEE_1.department by way of the
        unique primary index "TRAINEE_1.department.department_number = 700"
        with no residual conditions.  The size is estimated with high
        confidence to be 1 row.  The estimated time for this step is 0.10
        seconds.
    2)  We do an INSERT step into TRAINEE_1.department.  The estimated time
        for this step is 0.08 seconds.
2)  Finally, we send out an END TRANSACTION step to all AMPs involved in
    processing the request.
->  No rows are returned to the user as the result of statement 1.  The total
    estimated time is 0.10 seconds.
```

MERGE Multiple Rows

- Merges a set of source rows into a target table.
- Performs much better than two UPDATE/INSERT statements.
  - Source might be a Table, View or Subquery

department

dept#	department_name	budget_amount	mgr#
100	president	400,000.00	801
201	technical operations	293,800.00	1,025
301	research and development	465,600.00	1,005
302	product planning	226,000.00	1,016
401	customer support	982,300.00	1,003
402	NULL	308,000.00	1,011
403	education	932,000.00	1,005
501	marketing sales	308,000.00	1,017
600	NULL	NULL	1,099
700	Shipping	800,000.00	NULL

department_staging

dept#	department_name	budget_amount	mgr#	
1	new department	0.00	801	INSERT
301	research and development	512,160.00	1,019	UPDATE
302	product planning	248,600.00	1,016	UPDATE
501	marketing sales	308,000.00	1,017	UPDATE (but not actually modified)
600	none	NULL	1,099	UPDATE

```
MERGE INTO department AS tgt
USING department_staging AS src
ON src.department_number = tgt.department_number

WHEN MATCHED THEN UPDATE
SET department_name = src.department_name
   ,budget_amount   = src.budget_amount
   ,manager_employee_number
                    = src.manager_employee_number

WHEN NOT MATCHED THEN INSERT
VALUES
 ( src.department_number
  ,src.department_name
  ,src.budget_amount
  ,src.manager_employee_number
 );
```

Department table after MERGE:

dept#	department_name	budget_amount	mgr#
1	new department	0.00	801
100	president	400,000.00	801
201	technical operations	293,800.00	1,025
301	research and development	512,160.00	1,019
302	product planning	248,600.00	1,016
401	customer support	982,300.00	1,003
402	NULL	308,000.00	1,011
403	education	932,000.00	1,005
501	marketing sales	308,000.00	1,017
600	none	NULL	1,099
700	Shipping	800,000.00	NULL

```
1) First, we lock TRAINEE_1.department_staging in TD_MAP1 for read on
      a reserved RowHash to prevent global deadlock.
   2) Next, we lock TRAINEE_1.department in TD_MAP1 for write on a
      reserved RowHash to prevent global deadlock.
   3) We lock TRAINEE_1.department_staging in TD_MAP1 for read, and we
      lock TRAINEE_1.department in TD_MAP1 for write.
   4) We do an all-AMPs merge with matched updates and unmatched inserts
      into TRAINEE_1.department from TRAINEE_1.department_staging with a
      condition of ("TRAINEE_1.department.department_number =
      TRAINEE_1.department_staging.department_number").  The number of
      rows merged is estimated with no confidence to be 6 rows.  The
      estimated time for this step is 5.37 seconds.
   5) Finally, we send out an END TRANSACTION step to all AMPs involved
      in processing the request.
   -> No rows are returned to the user as the result of statement 1.
      The total estimated time is 5.37 seconds.
```

MERGE Update Only

- Merge may be used to perform update-only results.
  - Missing `WHEN NOT MATCHED`
- May perform better than its equivalent UPDATE:
  - When USI maintenance is involved.
  - Source table not spooled when (P)PI matches target table

```
MERGE INTO department AS tgt
USING
 (                                  Subquery
   SELECT
      department_number AS dept#
     ,Sum(salary_amount) AS sum_sal
   FROM employee_sales.employee
   GROUP BY 1
 ) AS src
ON tgt.department_number = src.dept#
WHEN MATCHED THEN UPDATE
SET budget_amount = budget_amount + src.sum_sal;
```

```
UPDATE department
FROM
 (
   SELECT
      department_number AS dept#
     ,Sum(salary_amount) AS sum_sal
   FROM employee_sales.employee
   GROUP BY 1
 ) AS src
SET budget_amount = budget_amount + src.sum_sal
WHERE department_number = src.dept#
;
```

MERGE Insert Only

- Merge may be used to perform an insert-only result.
  - Missing `WHEN MATCHED`
  - For an unconditional Insert add a `1=0` condition
- May perform better than its equivalent INSERT.
  - When index maintenance is involved.
  - Source table not spooled when (P)PI matches target table

```
MERGE INTO department AS tgt
USING department_staging AS src
ON src.department_number = tgt.department_number
AND 1=0

WHEN NOT MATCHED THEN INSERT
VALUES
 ( src.department_number
  ,src.department_name
  ,src.budget_amount
  ,src.manager_employee_number
 );
```

```
INSERT INTO department
SELECT
   department_number
  ,department_name
  ,budget_amount
  ,manager_employee_number
FROM department_staging
;
```

MERGE Delete

- Merge may be used to delete matching rows.
  - No `WHEN NOT MATCHED` allowed
- Performs similar to its equivalent DELETE
  - Source table not spooled when (P)PI matches target table

```
MERGE INTO department AS tgt
USING department_staging AS src
ON src.department_number = tgt.department_number

WHEN MATCHED THEN DELETE
;
```

```
DELETE FROM department AS tgt
WHERE EXISTS
 (
   SELECT 1
   FROM department_staging AS src
   WHERE src.department_number = tgt.department_number
 ) ;
```

Error Handling

Errors such as NOT NULL and CHECK constraint violations result in a rollback.

```
ALTER TABLE department
ADD CHECK (budget_amount > 0);
```

```
MERGE INTO department AS tgt
USING department_staging AS src
ON src.department_number = tgt.department_number

WHEN MATCHED THEN UPDATE
SET department_name = src.department_name
   ,budget_amount   = src.budget_amount
   ,manager_employee_number = src.manager_employee_number

WHEN NOT MATCHED THEN INSERT
VALUES
 ( src.department_number
  ,src.department_name
  ,src.budget_amount
  ,src.manager_employee_number
 );
```

```
Failed [5317 : 23000] Check constraint violation:
Check error in field department.budget_amount.
```

Loading Data with Bulk SQL Requests

Data warehouses regularly source their data from other systems. Data is moved from the source systems into the data warehouse system by ETL (extract, transform, load) or ELT (extract, load, transform) processes. These data is seldom applied directly into target tables in the data warehouse. Instead, it is often more efficient to move the data into intermediate storage tables (staging tables) and apply them later in bulk to the target tables.

The over-all process flow of this load strategy would be:

1. Move source data into staging tables.
2. Create error tables for the target tables in the data warehouse.
3. Use INSERT-SELECT or MERGE requests that specify a new LOGGING ERRORS clause to apply bulk inserts and/or updates to the target tables.
   During the data load process, errors related to the data rows themselves are first detected and logged. These errors include NOT NULL and CHECK constraint violations, and other kinds of non-index related errors. Index maintenance begins after all data rows are inserted into the table, and all data row related errors are logged. All USI and RI violations detected during index maintenance will then be logged, and the request will abort and rollback at the end of index maintenance if one or more index violations are detected.
4. Analyze errors logged in the error tables and modify the source data and re-load if necessary. Error rows are tagged with an identifier that links them to specific requests. Error recovery may be done after each load, or after batches of loads.

Error Tables

An error table is a multiset copy of the data table without any constraints plus thirteen fixed columns, each prefixed with "ETC_" (for ***E**rror **T**able **C**olumn*).

Its Primary Index is changed to a NUPI without partitioning.

```
CREATE ERROR TABLE FOR department;

SHOW ERROR TABLE FOR department;
```

```
CREATE MULTISET TABLE TRAINEE_1.ET_department
    (
     department_number INTEGER,
     department_name CHAR(30) CHARACTER SET LATIN NOT CASESPECIFIC,
     budget_amount DECIMAL(10,2),
     manager_employee_number INTEGER,
     ETC_DBQL_QID DECIMAL(18,0) FORMAT '-(18)9' NOT NULL,
     ETC_DMLType CHAR(1) CHARACTER SET LATIN NOT CASESPECIFIC,
     ETC_ErrorCode INTEGER NOT NULL,
     ETC_ErrSeq INTEGER NOT NULL,
     ETC_IndexNumber SMALLINT,
     ETC_IdxErrType CHAR(1) CHARACTER SET LATIN NOT CASESPECIFIC,
     ETC_RowId BYTE(10),
     ETC_TableId BYTE(6),
     ETC_FieldId SMALLINT,
     ETC_RITableId BYTE(6),
     ETC_RIFieldId SMALLINT,
     ETC_TimeStamp TIMESTAMP(2) NOT NULL,
     ETC_Blob BLOB(2033152))
PRIMARY INDEX Department_Index ( department_number );
```

Alternatively, an error table name may also be specified:

```
CREATE ERROR TABLE my_db.my_error_table FOR department;
```

- **ETC_DBQL_QID** – The Database Query Logging (DBQL) query id. The query id will be used to uniquely identify all error rows for a particular request.
- **ETC_DMLType** - Denotes the type of request. Value is 'I', 'U', or 'D' for Insert, Update, or Delete.
- **ETC_ErrorCode** - DBC error code or a value of zero. A row with a value of zero is a special marker row that confirms the successful completion of a LOGGING ERRORS request.
- **ETC_ErrSeq** - Error sequence number. It starts with a value of 1 for the first error of a given request and increments by 1 for each subsequent error.
- **ETC_IndexNumber** - Contains the index-id that caused a USI or RI violation, null otherwise.
- **ETC_IdxErrType** - Indicates if the index id stored in the preceding column refers to an USI or RI; `'R'` or `'r'` for RI error, `'U'` for USI error. Upper-case `'R'` will represent the more common child-insert RI error, and lower-case `'r'` for the less common parent-update RI error.
- **ETC_RowId** - The row id of the target table row related to the error condition.
- **ETC_TableId** - Identifies the target table for the load.
- **ETC_FieldId** - Stores the id of the column that caused an error condition.
- **ETC_RITableId** - Identifies the other table involved in an RI violation.
- **ETC_RIFieldId** - Identifies the field in the table associated with an RI violation. For multi-column keys, this would only identify the first column, but that would usually identify the key uniquely.
- **ETC_TimeStamp** - Indicates the time an error is detected, not the time the error row is written.
- **ETC_Blob** – Currently not used.

LOGGING ERRORS Option

```
LOGGING [ALL] ERRORS [ WITH { NO LIMIT | LIMIT OF n } ]
```

- Logs some types of errors into the error table including
  5317 Check constraint violations
  3604 NOT NULL violation
- The default limit *n* is 10 errors
- NO LIMIT does not mean unlimited but the system defined limit of 16,000,000 errors
- If the request accumulates the specified number of errors:
  - The request aborts
  - All changes to the target table rows are rolled back but error table rows are preserved.
- When the system encounters USI or Referential Integrity errors, the following events occur:
  1. The transaction or request runs to completion
  2. The system writes all erring rows into the error table
  3. The system aborts the transaction or request

Error rows are logged row-at-a-time, it may still take a long time to log millions of errors. Most applications could probably use the default error limit of '10'.

A LOGGING ERRORS request will be aborted and rolled back, but logged error rows will be preserved, if one of the following occurs:

a) the error limit is reached
b) an error that cannot be handled is detected
c) RI or USI violations are logged

If none of the above occurs, the request will be committed after logging all errors.

Errors arising from building a spool row
Some errors may occur while spooling rows *before* the merge-step and fail the request without logging errors. These errors include:

2617 Numeric overflow
2618 Division by zero
2621 Bad character in format or data
2663 SUBSTR: string subscript out of bounds
2666 Invalid date supplied

MERGE Using an Error Table

```
MERGE INTO department AS tgt
USING department_staging AS src
ON src.department_number = tgt.department_number

WHEN MATCHED THEN UPDATE
SET department_name = src.department_name
   ,budget_amount   = src.budget_amount
   ,manager_employee_number = src.manager_employee_number

WHEN NOT MATCHED THEN INSERT
VALUES
 ( src.department_number
  ,src.department_name
  ,src.budget_amount
  ,src.manager_employee_number
 )
LOGGING ERRORS;
```

```
SELECT
  department_number AS dept#
 ,department_name
 ,budget_amount AS budget
 ,manager_employee_number  AS mgr#
 ,ETC_DBQL_QID
 ,ETC_DMLType
 ,ETC_ErrorCode
 ,ETC_ErrSeq
 ,ETC_FieldId
 ,ETC_TimeStamp
FROM ET_department
WHERE ETC_dbql_qid = 307190995672810431
ORDER BY ETC_ErrSeq, ETC_ErrorCode DESC
;
```

Data columns

ETC columns

Copied from Studio History Result Message

dept#	department_name	budget	mgr#
1	new department	0.00	801
NULL	NULL	NULL	NULL

ETC_DBQL_QID	ETC_DMLType	ETC_ErrorCode	ETC_ErrSeq	ETC_FieldId
307,190,995,672,810,431	I	5,317	1	1,027
307,190,995,672,810,431	NULL	0	1	NULL

A marker row with `ETC_ErrorCode 0` will be inserted if a request with logged errors completed successfully.

Summary

- Merge can be used to update and insert rows of a table conditionally.
- Merge may reference subqueries to source change data.
- Merge is most aptly used in active warehouses as a load strategy.

MERGE INTO

Labs

Lab 1
MERGE INTO

teradata.

Create a MULTISET copy of the `finance_payroll.fin_trans` table without data in your *trainee_n* user.

Write a merge to unconditionally insert all rows from the `finance_payroll.fin_trans` table.

Write an *update* only merge based on the `finance_payroll.trans_staging_1` table.
Add the source table `amount` to the target table `balance.`

Write an *insert* only merge based on the `finance_payroll.trans_staging_1` table.

Write a merge to *insert* and *update* based on the `finance_payroll.trans_staging_2` table.
Add the source table `amount` to the target table `balance.`

Write a merge to *delete* based on the finance_payroll.trans_staging_2 table.

Drop the table.

Repeat all steps, but add a USI on `(trans_id, balance)` and compare runtimes.

Repeat all steps, but add a NUSI on `(trans_date, balance)` and compare runtimes.

Lab 2
MERGE INTO

teradata.

Create a MULTISET copy of the `finance_payroll.fin_trans` table without data in your *trainee_n* user.

Add a check constraint and an error table:

```
ALTER TABLE your_copy_of_fin_trans ADD CHECK(balance <= 15000);
CREATE ERROR TABLE FOR your_copy_of_fin_trans;
```

Write an Insert/Select to insert all rows from the `finance_payroll.fin_trans` table and run it three times:
1. without `LOGGING ERRORS`
2. using `LOGGING ERRORS`
3. using `LOGGING ERRORS WITH NO LIMIT`

Write a merge to insert and update based on the `finance_payroll.trans_staging_2` table.
Add the source table `amount` to the target table `balance`.
Run it using `LOGGING ERRORS WITH NO LIMIT`.

Check the *Result* column in Studio's *SQL History* and analyze the errors.

Drop the table.

MERGE INTO

Lab Solutions

Lab 1 Solution
MERGE INTO

teradata.

```
CREATE MULTISET TABLE fin_trans_copy AS finance_payroll.fin_trans WITH NO DATA;

CREATE UNIQUE INDEX (trans_id, balance) ON fin_trans_copy; -- added in 2nd run
CREATE INDEX (trans_date, balance) ON fin_trans_copy;      -- added in 3rd run
```

```
MERGE INTO fin_trans_copy AS tgt
USING finance_payroll.xxxxx AS src
ON tgt.account_id = src.account_id
AND tgt.trans_date = src.trans_date
AND tgt.trans_id = src.trans_id
...
```

```
...
WHEN MATCHED THEN UPDATE
SET amount = src.amount
   ,balance = tgt.balance + src.amount
...
```

```
...
WHEN NOT MATCHED THEN DELETE
...
```

```
...
WHEN NOT MATCHED THEN INSERT
VALUES
 (
   src.Trans_Id,
   src.Account_Id,
   src.Trans_Date,
   src.Amount,
   src.Balance,
   src.Trans_Type,
   src.Operation,
   src.Category,
   src.Other_Bank_Id,
   src.Other_Account_Id
 )
...
```

Lab 1 Solution
MERGE INTO (BLC)

teradata.

Resource usage data from DBC.QryLogV:
24-AMP system with Block Level Compression (BLC)

RunTime	IO_logical	IO_physical	CPU	SpoolUsage	InsCnt	UpdCnt	DelCnt	step	idx
0.52	3,143	1,015	4.28	0	1,056,320	0	0	ins_empty	no
10.71	1,236,257	3,359	113.30	0	1,056,320	0	0	ins_empty	USI
1.01	9,100	2,682	6.05	0	1,056,320	0	0	ins_empty	NUSI
0.43	6,405	606	5.18	0	0	305,946	0	update	no
1.13	14,636	2,957	8.70	0	0	305,946	0	update	USI
1.50	17,950	3,041	17.00	0	0	305,946	0	update	NUSI
0.43	7,549	1,605	5.39	0	307,550	0	0	insert	no
0.93	12,305	3,688	7.76	0	307,550	0	0	insert	USI
0.81	14,603	3,107	10.35	0	307,550	0	0	insert	NUSI
0.65	10,016	2,250	7.89	0	233,247	209,577	0	merge	no
1.20	19,369	4,868	12.45	0	233,247	209,577	0	merge	USI
1.37	26,557	5,526	17.42	0	233,247	209,577	0	merge	NUSI
0.42	10,340	1,712	6.72	0	0	0	442,824	delete	no
0.74	15,731	2,975	9.30	0	0	0	442,824	delete	USI
1.47	19,399	3,615	21.50	0	0	0	442,824	delete	NUSI

Previous Results from Data Manipulation Lab Insert/Update/Delete (BLC)

teradata.

Resource usage data from DBC.QryLogV:
24-AMP system with Block Level Compression (BLC)

RunTime	IO_logical	IO_physical	CPU	SpoolUsage	InsCnt	UpdCnt	DelCnt	step	idx
0.33	4,304	1,228	4.76	50,561,024	1,056,320	0	0	ins_empty	no
1.37	10,123	3,415	6.26	50,561,024	1,056,320	0	0	ins_empty	USI
0.94	11,465	3,246	7.24	50,561,024	1,056,320	0	0	ins_empty	NUSI
0.31	3,663	375	4.91	11,812,864	0	305,946	0	update	no
1.31	18,309	4,237	12.93	14,954,496	0	305,946	0	update	USI
1.46	14,397	2,497	15.79	11,812,864	0	305,946	0	update	NUSI
1.37	129,116	2,824	21.01	100,696,064	307,550	0	0	insert	no
2.03	134,468	4,174	23.86	100,696,064	307,550	0	0	insert	USI
1.91	136,362	4,723	25.96	100,696,064	307,550	0	0	insert	NUSI
1.65	103,079	3,406	23.02	105,365,504	233,247	209,577	0	merge	no
2.55	124,833	8,543	33.59	105,365,504	233,247	209,577	0	merge	USI
2.48	126,090	6,439	31.65	105,365,504	233,247	209,577	0	merge	NUSI
0.48	10,165	2,133	6.71	13,496,320	0	0	442,824	delete	no
0.65	16,023	2,810	9.71	13,496,320	0	0	442,824	delete	USI
1.49	19,213	3,588	20.77	13,496,320	0	0	442,824	delete	NUSI

Lab 1 Solution
MERGE INTO (no BLC)

teradata.

24-AMP system with Block Level Compression (BLC) switched off
SET QUERY_BAND = 'BLOCKCOMPRESSION=no;' UPDATE FOR SESSION VOLATILE;

RunTime	IO_logical	IO_physical	CPU	SpoolUsage	InsCnt	UpdCnt	DelCnt	step	idx
0.60	3,169	1,030	2.41	0	1,056,320	0	0	ins_empty	no
1.65	1,234,899	2,594	19.21	0	1,056,320	0	0	ins_empty	USI
1.34	9,282	2,717	3.77	0	1,056,320	0	0	ins_empty	NUSI
0.48	11,845	714	3.38	0	0	305,946	0	update	no
1.51	14,064	2,947	6.67	0	0	305,946	0	update	USI
1.33	16,837	2,789	14.44	0	0	305,946	0	update	NUSI
0.61	7,824	2,079	3.26	0	307,550	0	0	insert	no
1.07	12,681	4,234	5.53	0	307,550	0	0	insert	USI
1.13	14,991	3,638	7.31	0	307,550	0	0	insert	NUSI
0.99	10,001	2,639	4.10	0	233,247	209,577	0	merge	no
2.08	20,783	5,550	8.03	0	233,247	209,577	0	merge	USI
2.02	27,772	6,115	11.93	0	233,247	209,577	0	merge	NUSI
0.34	10,244	1,704	4.16	0	0	0	442,824	delete	no
1.27	15,346	2,694	6.25	0	0	0	442,824	delete	USI
1.61	19,257	3,436	17.49	0	0	0	442,824	delete	NUSI

Previous Results from Data Manipulation Lab Insert/Update/Delete (no BLC)

teradata.

24-AMP system with Block Level Compression (BLC) switched off
SET QUERY_BAND = 'BLOCKCOMPRESSION=no;' UPDATE FOR SESSION VOLATILE;

RunTime	IO_logical	IO_physical	CPU	SpoolUsage	InsCnt	UpdCnt	DelCnt	step	idx
0.40	4,383	1,257	2.78	50,561,024	1,056,320	0	0	ins_empty	no
1.30	8,681	2,900	3.93	50,561,024	1,056,320	0	0	ins_empty	USI
0.81	10,582	2,995	3.99	50,561,024	1,056,320	0	0	ins_empty	NUSI
0.24	3,657	375	2.70	11,812,864	0	305,946	0	update	no
1.04	18,666	4,314	7.71	16,957,440	0	305,946	0	update	USI
1.13	14,477	2,339	14.22	11,812,864	0	305,946	0	update	NUSI
1.79	167,253	2,873	18.50	100,696,064	307,550	0	0	insert	no
2.11	171,519	4,355	20.17	100,696,064	307,550	0	0	insert	USI
2.22	189,635	4,870	23.13	100,696,064	307,550	0	0	insert	NUSI
1.74	103,268	3,580	16.14	105,365,504	233,247	209,577	0	merge	no
2.27	125,694	8,825	23.20	105,365,504	233,247	209,577	0	merge	USI
2.76	127,552	7,797	26.05	105,365,504	233,247	209,577	0	merge	NUSI
0.32	9,926	1,805	3.52	13,496,320	0	0	442,824	delete	no
0.70	15,852	2,953	5.88	13,496,320	0	0	442,824	delete	USI
1.35	19,010	3,610	17.63	13,496,320	0	0	442,824	delete	NUSI

Lab 2 Solution
MERGE INTO

teradata.

```
CREATE MULTISET TABLE fin_trans_copy
AS finance_payroll.fin_trans WITH NO DATA
;

--Add a check constraint and an error table:
ALTER TABLE fin_trans_copy ADD CHECK(balance <= 15000)
;

CREATE ERROR TABLE FOR fin_trans_copy
;

-- 1. without LOGGING ERRORS
INSERT INTO fin_trans_copy
SELECT * FROM finance_payroll.fin_trans
;

-- 2. using LOGGING ERRORS
INSERT INTO fin_trans_copy
SELECT * FROM finance_payroll.fin_trans
LOGGING ERRORS
;

-- 3. using LOGGING ERRORS WITH NO LIMIT
INSERT INTO fin_trans_copy
SELECT * FROM finance_payroll.fin_trans
LOGGING ERRORS WITH NO LIMIT
;
```

```
MERGE INTO fin_trans_copy AS tgt
USING finance_payroll.trans_staging_2 AS src
ON tgt.account_id = src.account_id
AND tgt.trans_date = src.trans_date
AND tgt.trans_id = src.trans_id

WHEN NOT MATCHED THEN
INSERT
VALUES(
   src.Trans_Id,
   src.Account_Id,
   src.Trans_Date,
   src.Amount,
   src.Balance,
   src.Trans_Type,
   src.Operation,
   src.Category,
   src.Other_Bank_Id,
   src.Other_Account_Id
 )

WHEN MATCHED THEN UPDATE
SET amount = src.amount
   ,balance = tgt.balance + src.amount

LOGGING ERRORS WITH NO LIMIT
;
```

Lab 2 Solution (cont.)
MERGE INTO

```
SELECT TOP 15
   ETC_DMLType
  ,ETC_ErrorCode
  ,ETC_ErrSeq
  ,ETC_TimeStamp
FROM ET_fin_trans_copy
WHERE ETC_dbql_qid = 307190995672811322
ORDER BY ETC_ErrSeq DESC, ETC_ErrorCode;
```

ETC_DMLType	ETC_ErrorCode	ETC_ErrSeq	ETC_TimeStamp
I	5.317	10	2022-11-08 06:42:33.61
I	5.317	9	2022-11-08 06:42:33.53
I	5.317	8	2022-11-08 06:42:33.47
I	5.317	7	2022-11-08 06:42:33.41
I	5.317	6	2022-11-08 06:42:33.13
I	5.317	5	2022-11-08 06:42:33.01
I	5.317	4	2022-11-08 06:42:32.95
I	5.317	3	2022-11-08 06:42:32.91
I	5.317	2	2022-11-08 06:42:32.91
I	5.317	1	2022-11-08 06:42:32.91

ETC_DMLType	ETC_ErrorCode	ETC_ErrSeq	ETC_TimeStamp
NULL	0	1.063	2022-11-08 07:20:10.05
I	5.317	1.063	2022-11-08 07:20:10.03
I	5.317	1.062	2022-11-08 07:20:10.03
I	5.317	1.061	2022-11-08 07:20:10.02
I	5.317	1.060	2022-11-08 07:20:10.02
I	5.317	1.059	2022-11-08 07:20:10.02
I	5.317	1.058	2022-11-08 07:20:09.41
I	5.317	1.057	2022-11-08 07:20:07.24
I	5.317	1.056	2022-11-08 07:20:07.23
I	5.317	1.055	2022-11-08 07:20:07.23
I	5.317	1.054	2022-11-08 07:20:07.23
I	5.317	1.053	2022-11-08 07:20:07.22
I	5.317	1.052	2022-11-08 07:20:07.22
I	5.317	1.051	2022-11-08 07:20:07.21
I	5.317	1.050	2022-11-08 07:20:07.20

An aborted transaction returns no `ErrorCode 0`

Module 10: Views

Teradata Vantage SQL Intermediate

Objectives

After completing this module, you will be able to:

- Create a view.
- Use a view to provide secure access to data.
- Modify (replace) a view.
- List several reasons for using views.

What is a View?

- A view can be compared to a window through which you can see selected portions of a database.
- Views retrieve portions of one or more tables or other views.
- Views look like tables to a user, but they are *virtual*, not physical, tables.
- They display data in columns and rows and, in general, can be used as if they were physical tables.
- A view does not contain data: it is a virtual table whose *definition* is stored in the Data Dictionary.
- The view is not materialized until it is referenced by a statement.
- The primary reason to use views is to simplify end user access to the database.
- Data can be accessed directly via a table or indirectly via a view, based on privileges.
- Views can contain complex SQL, thereby hiding it from users.

Views
A view can be compared to a window through which you can see selected portions of a database. Views retrieve portions of one or more tables or other views. Views may be nested up to 64 levels.

Views and Tables
Views look like tables to a user, but they are virtual, not physical, tables. They display data in columns and rows and, in general, can be used as if they were physical tables. However, only the column definitions for a view are stored: views are not physical tables.

A view does not contain data: it is a virtual table whose definition is stored in the Data Dictionary. The view is not materialized until it is referenced by a statement. Some operations that are permitted for the manipulation of tables are not valid for views, and other operations are restricted, depending on the view definition.

Using Views
The primary reason to use views is to simplify end user access to the database. Views provide a constant vantage point from which to examine and manipulate the database. Their perspective is altered neither by adding or nor by dropping columns from its component base tables unless those columns are part of the view definition.

From an administrative perspective, views are useful for providing an easily maintained level of security and authorization. For example, users in a Human Resources department can access tables containing sensitive payroll information without being able to see salary and bonus columns. Views also provide administrators with an ability to control read and update privileges on the database with little effort.

Creating and Using Views

- Every Select can be stored as a view based on following rules and restrictions:
  - The view name must be unique within the database
  - Any derived columns must be aliased
  - No ORDER BY allowed

```
CREATE VIEW engineers AS
SELECT
   e.employee_number AS emp#
  ,e.first_name || ' ' || e.last_name AS fullname
  ,j.description
  ,d.department_name
FROM employee AS e
JOIN job AS j
  ON e.job_code = j.job_code
JOIN department AS d
  ON e.department_number = d.department_number
WHERE j.description LIKE '%Engineer%'
;
```

- When selecting from the view the column aliases must be used

```
SELECT * FROM engineers
ORDER BY emp# DESC;
```

```
emp#  fullname        description       department_name
----- --------------- ----------------- ------------------------
1.001 William Hoover  Field Engineer    customer support
1.004 Darlene Johnson Field Engineer    customer support
1.006 John Stein      Software-Engineer research and development
1.008 Carol Kanieski  Hardware Engineer research and development
1.010 Frank Rogers    Field Engineer    customer support
```

Restrictions

Some operations that are permitted on base tables are not permitted on views—sometimes for obvious reasons and sometimes not.

The following set of rules outlines the restrictions on how views can be created and used.

- You cannot create an index on a view.
- A view definition cannot contain an ORDER BY clause (unless the TOP clause is present).
- Any derived columns in a view must explicitly specify view column names, for example by using an AS clause or by providing a column list immediately after the view name.

Alternative syntax to define column aliases

Column names can be assigned as a comma-delimited list after the view name.
This might introduce naming confusion/conflicts when the order of columns doesn't match the order in the Select or AS aliases are defined additionally.

```
CREATE VIEW trainee_1.engineers
 (emp#, full_name, description, department_name)
AS
SELECT
   e.employee_number
  ,e.first_name || ' ' || e.last_name
  ,j.description
  ,d.department_name
FROM ...
```

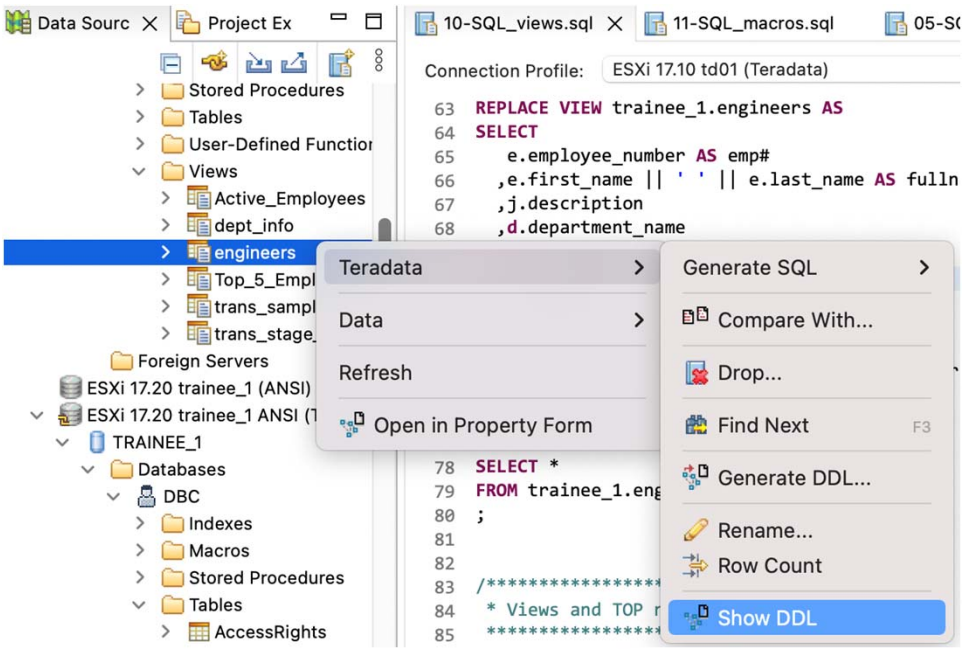
Replacing a View

- To modify the source code of a view:
  - Use the view's context menu in Studio: `Teradata > Show DDL` or submit a SHOW VIEW
  - Change `CREATE` to `REPLACE` and make the necessary changes.
  - Submit the replace statement.
  - Privileges that were granted directly on the original view are retained

```
SHOW VIEW trainee_1. engineers;

REPLACE VIEW trainee_1.engineers AS
SELECT
   e.employee_number AS emp#
  ,e.first_name || ' ' || e.last_name AS fullname
  ,j.description
  ,d.department_name
  ,Round(e.salary_amount, -4) AS sal_rounded
FROM employee AS e
JOIN job AS j
  ON e.job_code = j.job_code
JOIN department AS d
  ON e.department_number = d.department_number
WHERE j.description LIKE '%Engineer%'
;
```

- CREATE returns an error if the view already exists
- REPLACE *creates* the view if it doesn't exist
- REPLACE is NOT returning a warning when executed.
- Make sure you are replacing the correct view

REPLACE VIEW redefines an existing view or, if the specified view does not exist, creates a new view with the specified name. Privileges that were granted directly on the original view are retained for the replace view definition.



Required Privileges

- To create a view, you must have the CREATE VIEW privilege.
  - The creator receives all privileges on the newly created view WITH GRANT OPTION.
- To replace an existing view, you must have the DROP VIEW privilege.

- To Select from a view the SELECT privilege is required, either on the view or the database.
- The system checks access rights at creation time, and validates them again at execution time.
- By default every user has the CREATE VIEW privilege on itself.
  - A user can create a view within itself and use it.
- For other users to access a view, the *owner* must have the appropriate rights WITH GRANT OPTION.
  - A user can create a view within itself and grant Select to other users, but when they try to access it, it's probably failing due to missing WITH GRANT OPTION.

Privileges
The creator receives the DROP VIEW, INSERT, UPDATE, DELETE, and SELECT privileges on the newly created view WITH GRANT OPTION.

Privileges needed to use a view
The system checks access rights at creation time, and validates them again at execution time. Any database referenced by the view requires access rights on all objects accessed by the view.

For other users to access a view, the owner must grant the appropriate rights on the view and must have the appropriate rights WITH GRANT OPTION.

The system verifies that the creator has the appropriate right on the objects being referenced when a view is created. It also verifies that the creator has the rights needed to execute the statements defined in a macro. To grant to another user any privilege on a view or macro that references objects owned by a third user, the owner of the view or macro must have the appropriate rights with GRANT OPTION.

Teradata also verifies that the appropriate privileges exist on the objects being referenced for any user who attempts to access a view or execute a macro. This ensures that a change to a referenced object does not result in a violation of access rights when the view or macro referencing that object is invoked.

Views and TOP *n*

- You can specify an ORDER BY only in conjunction with the TOP clause.
- Otherwise, ORDER BY clauses are not valid within view definitions.

```
REPLACE VIEW trainee_1.Top_5_Employees AS
SELECT TOP 5
   first_name
  ,last_name
  ,hire_date
  ,salary_amount
  ,department_number AS dept#
  ,job_code
FROM  Employee
ORDER BY Salary_Amount DESC
;
```

```
SELECT *
FROM trainee_1.Top_5_Employees;
```

first_name	last_name	hire_date	salary_amount	dept#	job_code
I.B.	Trainer	2003-03-01	100.000,00	999	111.100
Irene	Runyon	2008-05-01	66.000,00	501	511.100
Nora	Rogers	2008-03-01	56.500,00	NULL	321.100
Larry	Ratzlaff	2008-07-15	54.000,00	501	512.101
Edward	Wilson	2008-03-01	53.625,00	501	512.101

Here a view references the TOP clause. Although a view is not allowed to reference an ORDER BY, this is an exception because "TOP *n*" may rely on this command in order to perform qualified ranking for a query, so it is allowed in this case.

An added recommendation might be to provide a view name that aptly describes the view as shown.

The optimizer replaces all View and Macro names with their underlying source code and resolves them down to table references. Explain shows if a table was accessed directly or via a view:

```
1) First, we lock EMPLOYEE_SALES.Employee in view
   trainee_1.Top_5_Employees in TD_MAP1 for read on a reserved
   RowHash to prevent global deadlock.
2) Next, we lock EMPLOYEE_SALES.Employee in view
   trainee_1.Top_5_Employees in TD_MAP1 for read.
3) We do an all-AMPs STAT FUNCTION step in TD_MAP1 from
   EMPLOYEE_SALES.Employee in view trainee_1.Top_5_Employees by way
   of an all-rows scan with no residual conditions into Spool 5 (Last
   Use), which is redistributed by hash code to all AMPs in TD_Map1.
   The result rows are put into Spool 1 (group_amps), which is built
   locally on the AMPs.  This step is used to retrieve the TOP 5 rows.
   The size is estimated with low confidence to be 5 rows (515 bytes).
   The estimated time for this step is 0.00 seconds.
4) Finally, we send out an END TRANSACTION step to all AMPs involved
   in processing the request.
-> The contents of Spool 1 are sent back to the user as the result of
   statement 1.  The total estimated time is 0.00 seconds.
```

View Column Info

When a view is created only column *names* are stored in the Data Dictionary.

Both `HELP VIEW` and `dbc.ColumnsV return NULL` for other metadata (besides the optional `Comment`).

```
HELP VIEW trainee_1.Top_5_Employees;
```

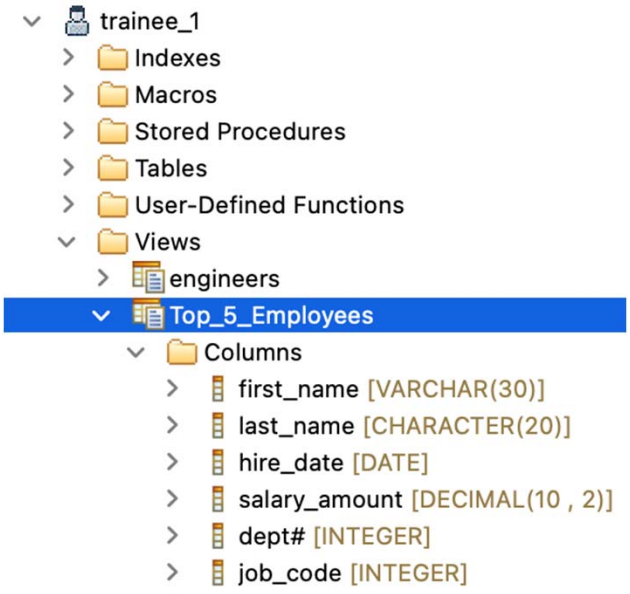
```
Column Name     Type Comment Nullable Format Title Max Length Deci
--------------- ---- ------- -------- ------ ----- ---------- ----
first_name      NULL NULL    NULL     NULL   NULL        NULL
last_name       NULL NULL    NULL     NULL   NULL        NULL
hire_date       NULL NULL    NULL     NULL   NULL        NULL
salary_amount   NULL NULL    NULL     NULL   NULL        NULL
dept#           NULL NULL    NULL     NULL   NULL        NULL
job_code        NULL NULL    NULL     NULL   NULL        NULL
```

```
HELP COLUMN trainee_1.Top_5_Employees.*;
```

```
Column Name     Type Nullable Format        Max Length Decimal Tot
--------------- ---- -------- ------------- ---------- -----------
first_name      CV   N        X(30)                 30
last_name       CF   N        X(20)                 20
hire_date       DA   N        YYYY-MM-DD             4
salary_amount   D    Y        ----------.99          8
dept#           I    Y        -(10)9                 4
job_code        I    Y        -(10)9                 4
```

HELP VIEW retrieves its info from the Data Dictionary while HELP COLUMN actually resolves the view's source code down to the base tables.

Studio submits a HELP COLUMN to display the data types of view columns:



Updatable Views

A view may be updatable, i.e., Data Manipulation operations can be performed against them.

It is *not updatable* if the view definition contains

- More than one table
- Expressions or derived columns
- DISTINCT
- TOP, GROUP BY, HAVING or QUALIFY clause
- Set operators

```
REPLACE VIEW high_sal_employees AS
SELECT
   employee_number
  ,last_name
  ,salary_amount
FROM emp_copy
WHERE Salary_Amount > 60000
WITH CHECK OPTION;
```

If an updatable view contains a WHERE-clause a DML statement may modify the data in a way that violates the WHERE conditions.

The WITH CHECK OPTION should be appended to the View definition to restricts the rows that can be affected by a DML request to those defined by the WHERE clause.

When WITH CHECK OPTION is …	Any DML made to the table through the view …
specified	only inserts, updates or deletes rows that satisfy the WHERE clause.
not specified	ignores the WHERE clause used in defining the view.

WITH CHECK OPTION Clause in Views

WITH CHECK OPTION pertains to updatable views. Views are typically created to restrict which base table columns and rows a user can access in a table. Base table column projection is specified by the columns named in the column_name list of a DELETE, INSERT, MERGE, SELECT, or UPDATE request, while base table row restriction is specified by an optional WHERE clause.

The WITH CHECK OPTION clause is an integrity constraint option that restricts the rows in the table that can be affected by an INSERT or UPDATE request to those defined by the WHERE clause. If you do not specify WITH CHECK OPTION, then the integrity constraints specified by the WHERE clause are not checked for updates. Base table constraints are not affected by this circumvention and continue to be enforced.

Unless you have compelling reasons not to honor the WHERE clause conditions specified by a view definition, you should always specify a WITH CHECK OPTION clause in all your updatable view definitions to protect the integrity of your databases.

```
EXPLAIN
DELETE FROM high_sal_employees;
```

```
   3) We do an all-AMPs DELETE step in TD_MAP1 from TRAINEE_1.emp_copy
      in view high_sal_employees by way of an all-rows scan with a
      condition of ("TRAINEE_1.emp_copy in view
      high_sal_employees.salary_amount > 60000.00").  The size is
      estimated with no confidence to be 10 rows.
```

Summary

- Views are virtual objects that are a "window" into the data contained in relational tables
- Views may be joined together
- Views may be used to provide summary information by including aggregations
- Views provide an additional level of security
- Views are unaffected if a column is added to a table
- Views are unaffected if a non-referenced column is dropped from a table
- Views cannot be indexed

Views

Labs

Lab 1
Views

teradata.

Create a view named `trainee_n.active_employees` returning all columns for *active* (`hire_end_date IS NULL`) employees only.

```
HELP TABLE hr_payroll;
```

Check the view's metadata using

- Studio's Data Source Explorer
- `SHOW VIEW`
- `HELP VIEW`
- `HELP COLUMN`

Replace the view to restrict the columns to

```
Employee_Number
Last_Name
First_Name
Department_Number
Division_Number
Job_Code
Hire_Date
Years_Service
Scheduled_Hours
Annual_Salary
```

Lab 2
Views

teradata.

Create another view based on `active_employees` calculating the number of active employees per department.
Add the minimum, average, median, maximum and sum of the `annual_salary`.

```
HELP VIEW active_employees;
```

Explain a select against the view.

Write a query based on this view to calculate the minimum/average/maximum of the maximum `annual_salary` per department for all department with over 30 employees and a department_number in the 40 to 80 range.

```
*** Query completed. 1 rows found. 5 columns returned.

min_max   avg_max    max_max    dept_count emp_count
--------- ---------- ---------- ---------- ---------
90,938.12 116,741.60 145,510.04         15     3,182
```

Explain this Select.

Why is it different from the previous explain?

Views

Lab Solutions

Lab 1 Solution
Views

teradata.

Create a view named `trainee_n.active_employees` returning all columns for *active* (`hire_end_date IS NULL`) employees only.

Check the view's metadata using

- Studio's Data Source Explorer
- `SHOW VIEW`
- `HELP VIEW`
- `HELP COLUMN`

Replace the view to restrict the columns to

Employee_Number
Last_Name
First_Name
Department_Number
Division_Number
Job_Code
Hire_Date
Years_Service
Scheduled_Hours
Annual_Salary

```
CREATE VIEW trainee_1.Active_Employees AS
SELECT *
FROM HR_Payroll
WHERE Hire_End_Date IS NULL;

SHOW VIEW trainee_1.Active_Employees;

HELP VIEW trainee_1.Active_Employees;

HELP COLUMN trainee_1.Active_Employees.*;

REPLACE VIEW trainee_1.Active_Employees AS
SELECT
   Employee_Number
  ,Last_Name
  ,First_Name
  ,Department_Number
  ,Division_Number
  ,Job_Code
  ,Hire_Date
  ,Years_Service
  ,Scheduled_Hours
  ,Annual_Salary
FROM HR_Payroll
WHERE Hire_End_Date IS NULL;
```

Lab 2 Solution
Views

teradata.

Create another view based on `active_employees` calculating the number of active employees per department.
Add the minimum, average, median, maximum and sum of the `annual_salary`.

Explain a select against the view.

Write a query based on this view to calculate the minimum/average/maximum of the maximum `annual_salary` per department for all departments with over 30 employees and a `department_number` in the 40 to 80 range.

Explain this Select.

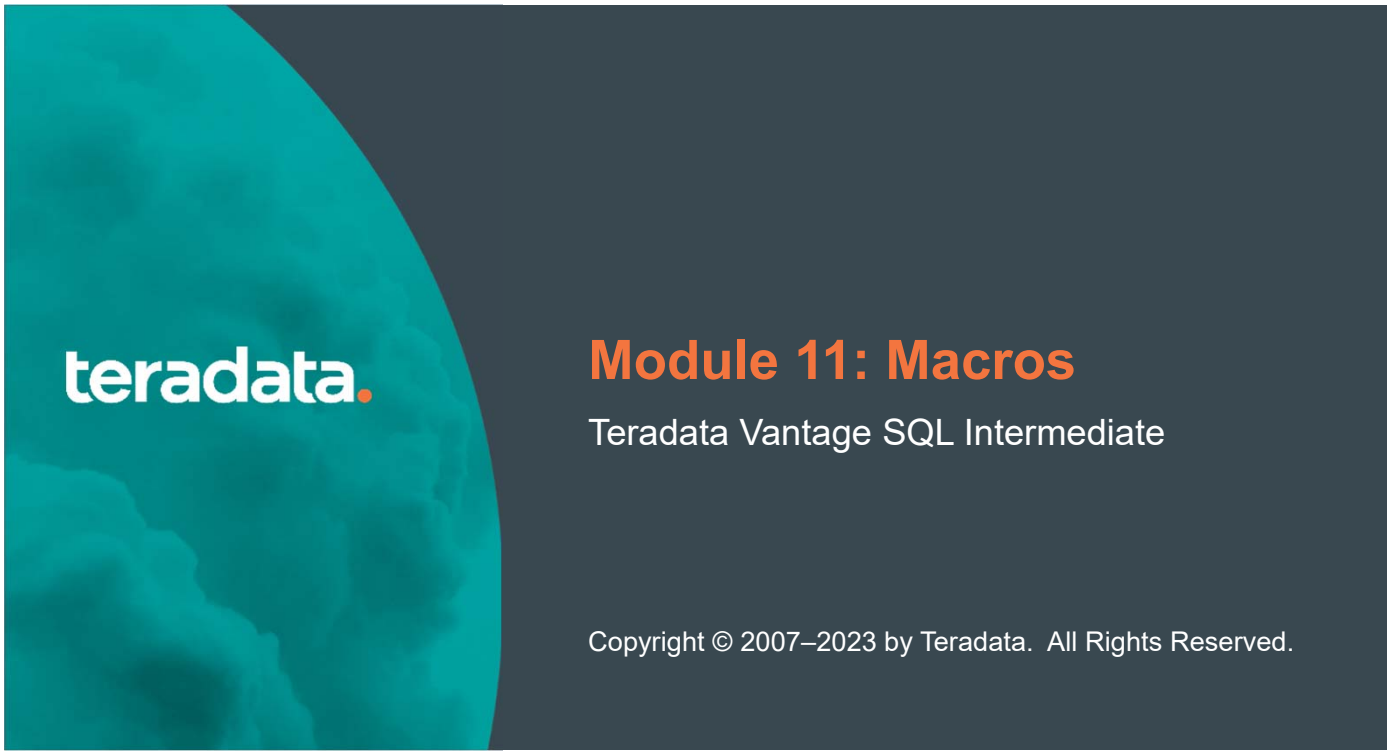
Optimizer tries to apply search conditions early.
Department_number is filtered before aggregation in the first SUM step.

Why is it so different from the previous explain?

Optimizer removed columns which are not used.
MEDIAN is a complex calculation requiring 3 steps: STATS-RETRIEVE-SUM)

```
REPLACE VIEW trainee_1.dept_info AS
SELECT
   department_number     AS dept#
  ,Count(*)              AS emp_cnt
  ,Min(annual_salary)    AS min_sal
  ,Avg(annual_salary)    AS avg_sal
  ,Median(annual_salary) AS median_sal
  ,Max(annual_salary)    AS max_sal
  ,Sum(annual_salary)    AS sum_sal
FROM trainee_1.Active_Employees
GROUP BY department_number
;
```

```
SELECT
   Min(max_sal) AS min_max
  ,Avg(max_sal) AS avg_max
  ,Max(max_sal) AS max_max
  ,Count(*)     AS dept_count
  ,Sum(emp_cnt) AS emp_count
FROM trainee_1.dept_info
WHERE emp_cnt > 30
  AND dept# BETWEEN 40 AND 80;
```



Module 11: Macros

Teradata Vantage SQL Intermediate

Objectives

After completing this module, you will be able to:

- Create and use a macro.
- Modify and drop a macro.
- Create a parameterized macro.
- Execute macros.

What is a Macro?

- A macro is a construct used to store and execute one or more SQL statements as one transaction (one request).
- A macro is an object owned by a database.
- A macro uses no owner database perm space.
- Rows are automatically inserted into dictionary tables to define the macro.
- Macros are executed from client SQL using the EXEC command.
- Macros can be the object of CREATE, REPLACE, DROP, SHOW, HELP, and EXEC (execute).
- A macro is a form of another SQL construct called a “multi-statement-request.”
- By combining statements, using them may become simpler.

A macro is a Teradata extension to ANSI SQL that contains prewritten SQL statements. The actual text of the macro is stored in a global repository called the Data Dictionary (DD). Macros are database objects and thus they belong to a specified user or database. They may contain one or more SQL statements. Macros have special efficiencies in the Teradata environment and are highly desirable for building reusable queries.

A macro allows you to name a set of one or more statements. When you need to execute those statements, simply execute the named macro. Macros provide a convenient shortcut for executing groups of frequently-run SQL statements.

Characteristics of a macro include the following:

- Are stored in the Data Dictionary
- Can contain one or more SQL statements
- Can contain comments
- Can contain certain BTEQ commands

CREATE and EXECUTE Macro

In the example:
- There is one statement.
- Any (all) statements must be enclosed in parentheses.
- Each statement must be separated (followed) by a semi-colon from the next.
- All statements must execute correctly or no statement executes (all-or-nothing).
- The creator must have the privileges necessary to perform all SQL within, plus the CREATE MACRO privilege.
- The executing user need only have "execute" privilege to execute it.

Create and execute a macro to display all employees working in department 401.

```
CREATE MACRO emp_401
AS
(
  SELECT
     employee_number
    ,last_name
    ,salary_amount
  FROM  employee
  WHERE department_number = 401
  ;
)
;
EXEC emp_401
;
```

```
employee_number last_name    salary_amount
--------------- ------------ -------------
          1,010 Rogers            50,600.00
          1,004 Johnson           39,930.00
          1,210 Smith             45,100.00
          1,002 Brown             47,410.00
          1,013 Phillips          26,950.00
          1,003 Trader            41,635.00
          1,022 Machado                NULL
          1,001 Hoover            28,077.50
```

Notice that there is a semi-colon before the closing parenthesis. Each statement inside the body of the macro must be delimited (separated from the other statements) with a semi-colon as shown. Later we shall see an example of a macro in which several statements exist.

The user creating the macro must have the appropriate privileges on the tables or views used in the macro. The user executing the macro requires only the EXEC privilege on the macro.

Note, the macro may be executed by either the EXEC or the EXECUTE command.

Running a Macro

Running a macro constitutes an implicit transaction. Therefore, in Teradata session mode, a macro need not be enclosed between BEGIN TRANSACTION and END TRANSACTION statements.
When the session performing a macro is in ANSI mode, the actions of the macro are uncommitted until a commit or rollback occurs in subsequent statements unless the macro body ends with a COMMIT statement. If you define a macro using a COMMIT statement, then it can be performed only in sessions running in ANSI mode.

Modifying a Macro with REPLACE

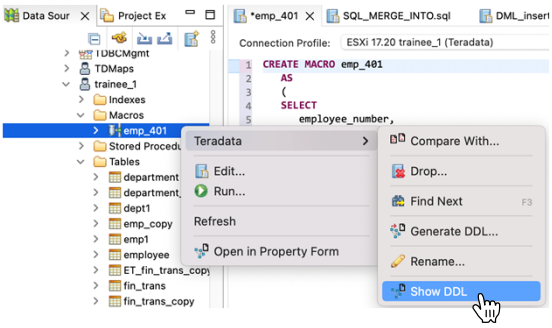
An existing Macro may be modified using `REPLACE MACRO`.

- Use a `SHOW MACRO` statement to get the DDL from the Data Dictionary.
- TD Studio is able to show the definition by using the macro's context menu.
- Change “CREATE” to “REPLACE” and make the necessary changes.
- Submit the replace statement.

Attention:

- CREATE returns an error if the macro exists.
- REPLACE is creating a macro if it doesn’t exist.
- REPLACE is NOT returning a warning when executed.
- Make sure you are replacing the correct Macro.

Therefore:
You may NOT always use REPLACE to create a macro.



Function of REPLACE MACRO Requests
REPLACE MACRO executes as a DROP MACRO request followed by a CREATE MACRO request except for the handling of privileges. Vantage retains all of the privileges that were granted directly on the original macro.

If the specified macro does not exist, a REPLACE MACRO request creates it. In this case, the REPLACE request has the same effect as a CREATE MACRO request.

If an error occurs during the replacement of the macro, the existing macro remains in place as it was prior to the performance of the REPLACE MACRO request (it is not dropped). This is analogous to a ROLLBACK on the operation.

Macros vs. Multi-Statement Requests

Teradata's analysis unit is the *request*.

Many requests contain only a single statement.

By using Multi-Statements, the optimizer may create plans saving resources.

Macros are a form of *multiple statement request* (MSR).

In a Teradata Mode session Macros and Multi-Statements are treated as *implizit* transactions, preventing interference by others.

To issue a Multi-Statement Request from Teradata Studio highlight the statements and press

- `Alt-C` (Windows)
- `Control-C` (MacOS)

Caution: Don't use the *Execute All* icon!

- Ignores the higlighting and submits *all* statements in the editor window in parallel.

```
CREATE MACRO dept_job
AS
(
  SELECT
    department_name
  FROM department
  WHERE department_number = 401
  ;
  SELECT
    description
  FROM job
  WHERE job_code = 412101
  ;
);

EXEC dept_job
;
```

Two result sets returned

```
department_name
----------------
customer support
```

```
description
--------------
Field Engineer
```

As stated earlier, macros may contain (and often enough do contain) multiple statements. When multiple statements exist within a macro, all statements must be successful or else none of them are executed. This is true for multi-statement requests as well. For this reason macros and multi-statement requests are different methods for accomplishing the very same result and are, in fact, considered as another form of multi-statement request.

Teradata Studio SQL Editor Context Menu

Execute All	Execute all statements in the editor as a single request (in parallel)
Execute Selected Text	Execute the highlighted statement(s) in the editor as individual requests (sequentially)
Execute as Individual Statements	Execute all statements in the editor as a single request (sequentially)
Execute Selected Text as One Statement	Execute the highlighted statement(s) in the editor as a single request (in parallel)
Execute Current Text	Execute the statement between the previous and the following semicolon based on the *cursor position*

Parameterized Macros

Macros may be parameterized to allow for more flexible control.

```
CREATE MACRO get_sal
 ( emp#  INTEGER )
AS
 (
   SELECT
      employee_number
     ,last_name
     ,salary_amount
   FROM  employee
   WHERE employee_number = :emp#
   ;
 )
;

EXEC get_sal(1018)
;



EXEC get_sal(1008)
;
```

Parameter name is **emp#**.
Its data type is integer.

Referenced parameter must be preceded with a ":" (colon), but a reference is not mandatory for a parameterized macro.

```
employee_number last_name  salary_amount
--------------- ---------- -------------
          1,018 Ratzlaff       54,000.00
```

```
employee_number last_name  salary_amount
--------------- ---------- -------------
          1,008 Kanieski       29,250.00
```

Parameterized macros allow substitutable variables. Values for these variables are supplied at runtime. In parentheses following the macro name is the parameter list. It names each parameter followed by its data type. When a parameter is used in the body of a macro, it is always preceded by a colon. This construct is referred to as a "host variable" or "parameterized variable."

Rules for Using Parameters in Macros

- You cannot pass the names of database objects to a macro as parameters. This refers to tables and views and their columns, databases, users, and similar database objects.
- If you supply a parameter value in the EXEC request that does not conform to the specified format, data type, or default value, the request aborts and returns a message to the requestor.
- The maximum number of parameters you can specify per macro is 2,048.

Macro Support for Global Temporary and Volatile Tables

You can reference both global temporary tables and volatile tables from a macro.

When you perform a macro that involves a global temporary table, then all references to the base temporary table are mapped to the corresponding materialized global temporary table within the current session. Note that this means the same macro, when performed from two different sessions, can have very different results depending on the content of the materialized global temporary tables
.
When you perform a macro that contains an INSERT request that inserts rows into a global temporary table, and that table is not materialized in the current session, then an instance of the global temporary table is created when the INSERT request performs.

When you perform a macro that contains a CREATE VOLATILE TABLE request, then the referenced table is created in the current session. If the macro includes a DROP on the named volatile table, then that table is dropped from the current session.

Parameterized Macros – Multiple Parameters

```
REPLACE MACRO new_dept
 (
   in_dept   INTEGER
  ,in_budget DECIMAL(10,2) DEFAULT 0
  ,in_name   CHAR(30)
  ,in_mgr    INTEGER
 )
AS
 ( -- check input parameters
   ROLLBACK 'in_name can''t be NULL'
   WHERE :in_name IS NULL
   ;
   ROLLBACK 'in_mgr doesn''t exist'
   WHERE :in_mgr IS NOT NULL
   AND NOT EXISTS
    ( SELECT 1
      FROM employee AS e
      WHERE e.employee_number = :in_mgr
    )
   ;
   -- inserting a new row
   INSERT INTO department
    ( department_number
     ,department_name
     ,budget_amount
     ,manager_employee_number
    )
   VALUES
    ( :in_dept
...
```

Defaults can be defined

ROLLBACK allows checking parameters

```
...
     ,:in_name
     ,:in_budget
     ,:in_mgr
    )
   ;
   SELECT -- echoing what was inserted
      department_number        AS dept#
     ,department_name          AS name
     ,budget_amount            AS budget
     ,manager_employee_number  AS mngr#
   FROM  department
   WHERE department_number = :in_dept
   ;
 );
```

Create a parameterized macro which inserts a new department row into the department table and then selects the same row for purposes of verification.

Macros may have more than one parameter. Each name and its associated data type are separated by a comma from the next name and its associated data type. The order is important. The first value specified in the EXEC of the macro will be associated with the first value in the parameter list. The second value in the EXEC is associated with the second value in the parameter list, and so on.

Recall from an earlier module that column names are optional in the INSERT statement provided the sequencing of elements in the VALUES clause corresponds to the column. The sequence of column names must agree with the sequence of items in the VALUES clause.

You can include a condition for halting execution of a macro by incorporating an ABORT or ROLLBACK request into its definition. If the specified condition is encountered during execution, the macro is aborted.

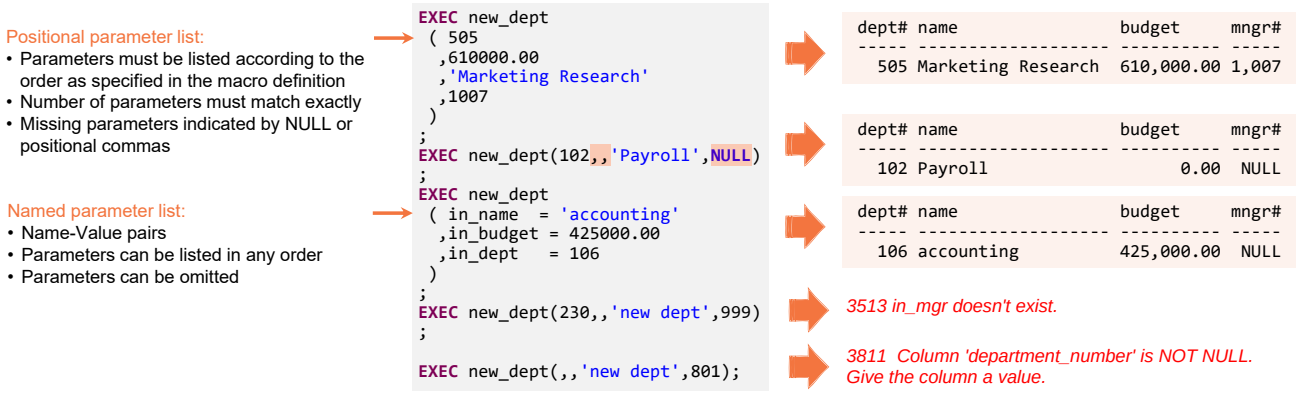
The transaction in process is concluded, locks on the tables are released, changes made to data are backed out, and any spooled output is deleted.

Specify the text of an optional error message following the ABORT keyword and enclose it in APOSTROPHE characters. This message displays on the terminal screen if the macro is aborted for the specified condition.

Executing the Parameterized Macro

The following events occur during macro execution:

- Data types are checked for compatibility with the CREATE TABLE data types.
- NULL inputs are checked for validity against the CREATE TABLE column definitions.
- Defaulting is applied if specified in the CREATE MACRO and no value is provided.
- Defaulting is applied if specified in the CREATE TABLE and no value or macro default is provided.



The following are requirements for executing the first two examples of a parameterized macro.

- Input data in the order specified in the Macro parameter list.
- Provide the exact number of parameters specified in the list.
- Use the keyword NULL to explicitly pass a null to the macro.
- Use positional commas to implicitly pass a null, or a specified default value.
- Check data types for compatibility with CREATE MACRO data types.
- Check that inserted nulls are permitted in the target columns.
- Check if defaulting is specified in the CREATE MACRO statement.

In the 2nd example, the value after 102 has been omitted, hence the two back-to-back commas. In such cases the parameter value will be set to the default specified for in the CREATE MACRO statement. In this case it is 0.

Another option available for passing parameters to a macro, is to explicitly name the parameters with an associated value as seen in the 3rd example. When this approach is selected, the parameters may be specified in any sequence. The use of parameter names eliminates the need to input data in the exact order as specified in the macro parameter list.

Beyond their user-friendliness, macros have certain performance advantages as well.

Parameterized macros may be submitted multiple times with different parameter values each time. Because the body of the query doesn't change, it permits execution of the same optimized “steps” which are cached for re-use. Not to mention that less data is being transferred to the database, and that the macros have already been syntax checked. They also provide a centralized location from which they can be maintained and executed. These features make them highly desirable from both a performance standpoint and maintenance standpoint.

Macros and DDL

You can use a macro to create a table.

A DDL statement must be the only statement in the macro and it must be immediately commited.

- Submit as implicit transaction in a Teradata Mode session.
- Next statement must be `COMMIT;` in an ANSI Mode session.

Object names and keywords cannot be passed as parameterized values.

```
CREATE MACRO vt_emp_copy
AS
(
  CREATE VOLATILE TABLE vt_employee
   (
     employee_number INTEGER
    ,manager_employee_number INTEGER
    ,department_number INTEGER
    ,job_code INTEGER
    ,last_name CHAR(20)
    ,first_name VARCHAR(30)
    ,hire_date DATE FORMAT'yyyy-mm-dd'
    ,birthdate DATE FORMAT'yyyy-mm-dd'
    ,salary_amount DECIMAL(10,2) NOT NULL
   )
  UNIQUE PRIMARY INDEX ( employee_number )
  ON COMMIT PRESERVE ROWS
  ;
)
;
-- submit whenever you need a copy
EXEC vt_emp_copy
;
-- verify what has been created
SHOW TABLE vt_employee
;
```

If a macro contains a data definition statement, it cannot contain other requests. A data definition statement in a macro is not fully resolved until the macro is performed. At that time, unqualified references to database objects are resolved using the default database for the user submitting the EXECUTE statement. It is therefore recommended that object references in a macro data definition statement be fully qualified in the macro body.

Required Privileges

- To create a macro, you must have the CREATE MACRO privilege.
  - The creator receives all privileges on the newly created macro WITH GRANT OPTION.
- To replace an exising macro, you must have the DROP MACRO privilege.
- To execute a macro the EXECUTE privilege is required, either on the macro or the database.
- Once a macro has been created, its immediate owner is the database or user in which it is contained, not the user who created it.
- The user creating a macro must have the privileges for all statements it performs.
- When a macro is executed the immediate owner of the macro must have the privileges for all statements it performs, including WITH GRANT OPTION.
- When a macro is executed by the immediate owner (i.e. the macro was created within the user) the `WITH GRANT OPTION` is not required.
- By default every user has the CREATE MACRO privilege on itself.
  - A user can create a macro within itself and use it.
  - The user can grant execute to other users, but when they try to access it, it's probably failing due to missing WITH GRANT OPTION.

Privileges
The creator receives the DROP VIEW, INSERT, UPDATE, DELETE, and SELECT privileges on the newly created view WITH GRANT OPTION.

Required Privileges to Execute a Macro
You must have EXECUTE privilege on the macro. The creator or any owner of the macro can grant the EXECUTE privilege to another. In addition, the immediate owner of the macro (the database in which the macro resides) must have the necessary privileges on objects named in the request set that are contained in the macro.

Summary

- Macros are a collection of stored SQL statements.
- Macros may be the created, replaced, dropped, and executed.
- Macros are another form of a multi-statement request and are executed.
- Macros may be parameterized.
- Macros may perform DDL, as well as INSERT, UPDATE, SELECT and DELETE.
- Macros are most useful for repetitive SQL.
- Macros can be the object of HELP and SHOW.
- When providing macros for general usage EXEC must be granted WITH GRANT OPTION.

Module 11: Review Questions

1. Which two are true about macros?
   a. Macros are Teradata's version of stored procedures
   b. Changes to macros are done by using ALTER MACRO
   c. Macros are an object which requires perm space from user DBC
   d. You can perform a HELP MACRO to look at parameter information on a macro

2. Which two are true about parameterized macros?
   a. They have two forms of execution
   b. You can use them to pass object names for creating objects
   c. In order to execute a parameterized macro, you must know the parameter names
   d. Parameters may be defined, but not referenced by the macro's statements

3. Which two are true about macros and DDL?
   a. Macros may be used to create objects
   b. Macros having multiple SQL statements may not contain DDL
   c. Macros containing DDL may not be parameterized
   d. Object referenced by macros may not be dropped

Check your understanding of the concepts discussed in this module by completing the review questions as directed by your instructor.

Module 11: Review Questions

4. Which two are true macros and privileges?
   a. If you can execute and macro, then you can REPLACE it
   b. You need REPLACE MACRO privileges to change a macro definition
   c. CREATE MACRO privileges are needed to create a macro
   d. Only EXECUTE privileges are needed to execute a macro
5. Which two are true about macro maintenance?
   a. You may replace a macro using CREATE
   b. You may create a macro using REPLACE
   c. You may update a macro using ALTER
   d. It is possible to create a macro in another database
6. Which two are true about macros?
   a. They are objects that have a “kind” value of “M”
   b. They may be located inside an “explorer tree” of a standard Teradata SQL tool
   c. Need not contain a semicolon within the body of it
   d. Are not typically used for repetitive access to objects

Check your understanding of the concepts discussed in this module by completing the review questions as directed by your instructor.

Macros

Labs

Lab 1
Macros

teradata.

Create a macro to parameterize this report:

```
HELP TABLE hr_salary_hist;
```

Write a query to report the employees with the highest overtime_pay per department in the year *YYYY*. Consider only employees with at least *xxx* overtime hours.

Order by descending overtime pay.

```
EXEC max_pay_macro( sal_yr = 2017, over_hrs = 200);
```

```
*** Query completed. 17 rows found. 8 columns returned.

Employee_Number First_Name Last_Name dept# Job_Code Total_Pay  Overtime_Pay Overtime_Hours
--------------- ---------- --------- ----- -------- ---------- ------------ --------------
        416,916 Arthur     Munoz        50    5,010 169,237.58   108,117.48       2,253.25
        347,191 Nicholas   Mcdonner     52  120,215 130,243.41    69,945.25       1,620.75
        371,432 Leroy      Perkins      70  151,515 109,918.06    63,830.34       2,010.50
        411,922 Charles    Lanehart     77  151,445  83,369.23    29,647.25         765.25
        357,111 Larry      Cooper        5  121,213  80,596.62    29,076.27         700.50
        347,531 Levert     Kemp         51  170,150 105,112.06    26,309.15         414.75
        327,255 Kyle       Russ         78  151,095  59,528.21    19,795.01         683.50
        442,771 Rhonda     Wright        9  183,115  65,546.41    19,406.99         541.00
        447,439 Jason      Widmeier     71  101,225  68,735.73    18,097.61         490.75
        357,472 Thomas     Wagner       21  182,118  82,232.54    17,994.34         355.25
        373,656 Terry      Hayes        53  124,265  62,546.48    13,819.36         393.75
```

Lab 2
Macros

teradata.

Modify the previous macro to add a default of 100 for `over_hrs` and some checks returning an error message for bad parameters:

- `sal_yr` should be within years 2008 to 2017
- `over_hrs` must be at least 100

```
EXEC trainee_?.max_pay_macro( sal_yr = 2020, over_hrs = 200);
3513  sal_yr must be between 2008 and 2017.
```

```
EXEC trainee_?.max_pay_macro( sal_yr = 2017, over_hrs = 20);
3513  over_hrs must be at least 100.
```

Macros

Lab solutions

Lab 1 Solution
Macros

teradata.

Create a macro to parameterize this report:

Write a query to report the employees with the highest overtime_pay per department in the year *YYYY*. Consider only employees with at least *xxx* overtime hours.

Order by descending overtime pay.

```
REPLACE MACRO max_pay_macro
 (
   sal_yr INT
  ,over_hrs INT
 )
AS
 (
   SELECT
      employee_number
     ,first_name
     ,last_name
     ,department_number AS dept#
     ,job_code
     ,total_pay
     ,overtime_pay
     ,overtime_hours
   FROM finance_payroll.hr_salary_hist
   WHERE sal_year = :sal_yr
     AND overtime_hours > :over_hrs
   QUALIFY
      overtime_pay
        = Max(overtime_pay)
          OVER (PARTITION BY dept#)
   ORDER BY overtime_pay DESC
   ;
 );
```

Lab 2 Solution
Macros

Modify the previous macro to add some checks returning an error message for bad parameters:

- `sal_yr` should be within years 2008 to 2017
- `over_hrs` must be at least 100

```
REPLACE MACRO trainee_?.max_pay_macro
 ( sal_yr INT
  ,over_hrs INT DEFAULT 100
 )
AS
 ( -- check the correct range of values for parameters
   ABORT 'sal_yr must be between 2008 and 2017'
   WHERE :sal_yr NOT BETWEEN 2008 AND 2017;

   ABORT 'over_hrs must be at least 100'
   WHERE NOT :over_hrs >= 100;

   SELECT
      employee_number
     ,first_name
     ,last_name
     ,department_number AS dept#
     ,job_code
     ,total_pay
     ,overtime_pay
     ,overtime_hours
```

Type Cast Functions

Teradata Vantage SQL Intermediate

Objectives

After completing this module, you will be able to:

- Convert data types using functions such as TO_CHAR, TO_DATE and TO_TIIMESTAMP
- Apply Format Strings to load or display numeric and datetime data types in a non-Standard format

Refresher
Data Type Conversion Using CAST

The `CAST` function allows you to convert a column or expression from one data type to another:

```
Cast(expression AS DataType)
```

```
Cast('127'      AS INTEGER)          127
Cast('127.9'    AS INTEGER)          127      Truncated
Cast('127.9'    AS DECIMAL(10,0))    128.     Rounded
Cast('127.2'    AS DECIMAL(10,2))    127.20
Cast('127.2'    AS DECIMAL( 4,2))    Error 2616 Numeric overflow occurred during computation.
Cast('127,2'    AS DECIMAL(10,2))    1272.00  Comma = 1000 seperator
Cast( 127.2     AS VARCHAR(10))      '127.2'
Cast( 127.2     AS VARCHAR(2))       '12'     Truncated! (Teradata mode)
                                     Error 3996 Right truncation of string data (ANSI mode)
Cast( 12        AS VARCHAR(3))       '12'
Cast(''         AS INTEGER)            0      ???
Cast(','        AS INTEGER)            0      ???
Cast(last_name AS INTEGER)           Error 2621 Bad character in format or data of employee.last_name.
```

CAST is a function that can be used to change the data type of a resulting column or expression. Data types may be changed in many ways: CHAR to CHAR, CHAR to NUMBER, NUMBER to CHAR, NUMBER to NUMBER, etc.

All in all, data type conversions can be quite complex and somewhat challenging. This slide discusses conversions using CAST.

Refresher
Data Type Conversion Using Teradata Syntax

Teradata extension to Standard SQL to convert a column or expression from one data type to another:

```
expression (DataType)
```

```
'127'     (INTEGER)          127
'127.9'   (INTEGER)          127      Truncated
'127.9'   (DECIMAL(10,0))    128.     Rounded
'127.2'   (DECIMAL(10,2))    127.20
'127.2'   (DECIMAL( 4,2))    Error 2616 Numeric overflow occurred during computation.
'127,2'   (DECIMAL(10,2))    1272.00  Comma = 1000 seperator
 127.2    (VARCHAR(10))      ' 127.2'
 127.2    (VARCHAR(2))       ' 1'     Truncated!

 12       (VARCHAR(3))       '  1'    Truncated!
''        (INTEGER)            0      ???
','       (INTEGER)            0      ???
last_name(INTEGER)           Error 2621 Bad character in format or data of employee.last_name.
```

Casting numeric to character using Teradata syntax applies a format before extracting the requested number of characters. The number is *right justified* within this format, which might result in *leading spaces*.

127.2 is a DECIMAL(4,1), formatted using *six* characters: a leading sign (a minus sign or a space), three digits, a period and one digit: `' 127.2'`

```
127.2 (VARCHAR(10)) ⟹ SUBSTRING(' 127.2' FROM 1 FOR 10) ⟹ ' 127.2'
127.2 (VARCHAR(2))  ⟹ SUBSTRING(' 127.2' FROM 1 FOR 2) ⟹ ' 1'
```

12 is a BYTEINT, formatted using *four* characters: a leading sign (a minus sign or a space) and three digits: `'  12'`

```
12 (VARCHAR(3)) ⟹ SUBSTRING('  12' FROM 1 FOR 3) ⟹ '  1'
```

In contrast, the ANSI CAST applies the same format, but actually trims the spaces in front of the number before acquiring the characters.

Refresher
TRYCAST

teradata.

Similar to `CAST` but does not fail on errors.

`TRYCAST` is limited to *character* input.

```
TryCast(string_expression AS DataType)
```

```
TryCast('127'      AS INTEGER)          127
TryCast('127.9'    AS INTEGER)          127      Truncated
TryCast('127.9'    AS DECIMAL(10,0))    128.     Rounded
TryCast('127.2'    AS DECIMAL(10,2))    127.20
TryCast('127.2'    AS DECIMAL( 4,2))    NULL
TryCast('127,2'    AS DECIMAL(10,2))    1272.00  Comma = 1000 seperator
TryCast(''         AS INTEGER)             0     ???
TryCast(','        AS INTEGER)             0     ???
TryCast(last_name AS INTEGER)           NULL
```

Teradata extension to SQL:2016

TRYCAST takes a string and tries to cast it to a data type specified after the AS keyword (similar to CAST). If the conversion fails, TRYCAST returns a NULL instead of failing.

The result of the conversion is returned unless there is an error, in which case a NULL is returned. The result data type is whatever data type was specified by the *data_type* input.

Refresher: Data Type Conversion Using TO_NUMBER

Converts a string expression to a NUMBER data type.

Similar to `CAST` but does not fail on errors.

```
TO_NUMBER(string_expression)
```

```
To_Number('127'   )          127
To_Number('127.9')           127.9
To_Number('127.2e+02')    12,720
To_Number('127,2')           NULL
To_Number(''      )          NULL
To_Number(','     )          NULL
To_Number(last_name)         NULL
```

Supports additional options to specify an input format.

Teradata extension to SQL:2016

Refresher
TO_CHAR: Numeric

Converts a numeric expression to a string exactly long enough to hold its significant digits.

Returns a `VARCHAR(40) CHARACTER SET UNICODE`.

```
TO_CHAR(numeric_expression)
```

```
To_Char(127  )          '127'
To_Char(127.9)          '127.9'
To_Char(127.2e+02)      '12720'
To_Char(-1.2345)        '-1.2345'
```

Supports additional options to specify an output format.

Teradata extension to SQL:2016

TO_NUMBER and TO_CHAR: Format Examples

- Numeric formats can be used for both `To_Char` and `To_Number`.

```
To_Number('$123.30')                                                    NULL
To_Number('$123.30','L999999D99')                                       123.3
To_Number('123.30€','999999D99L','NLS_CURRENCY=''€''')                  123.3
To_Number('€123.30','L999999D99','NLS_CURRENCY=''€''')                  123.3
To_Number('£123.30','U999999D99','NLS_DUAL_CURRENCY=''£''')             123.3
To_Number('Dollar123','C999','NLS_ISO_CURRENCY=''Dollar''')             123
To_Number('123.456,789','999G999G999D999', 'NLS_NUMERIC_CHARACTERS = '',.''') 123,456.789
To_Number(oTranslate('$123.456,789', ',.$', '.')) -- easier?            123,456.789

To_Char(123.30,'99.99')                                                 '######'
To_Char(123.30,'L999999D99')                                            '    $123.30'
To_Char(123.30,'FML999999D99')                                          '$123.30'
To_Char(123.30,'999G999G999D999', 'NLS_NUMERIC_CHARACTERS = '',.''')    '         123,300'
To_Char(123.30,'FM999G999G999D999', 'NLS_NUMERIC_CHARACTERS = '',.''')  '123,300'
```

- `NLS_`*`Params`* are only partially supported by *To_Number*, no `NLS_..._CURRENCY`

Teradata extension to SQL:2016

TO_CHAR: Numeric (cont.)

```
TO_CHAR(numeric_expression [, format_arg [, NLS_param ] ] )
```

format_arg	Description
numeric_expr	a numeric argument. If the conversion fails, '#' characters are returned.
format_arg	an optional character argument. *format_arg* is used to format the numeric values. If *format_arg* is omitted, *numeric_expr* is converted to a string exactly long enough to hold its significant digits.
NLS_param	an optional character argument to be used with *format_arg*.

NLS_param	Description
NLS_NUMERIC_CHARACTERS = ''*dg*''	***dg*** represents two characters, which are single-byte characters, used for the decimal marker **D** and thousands group marker **G**
NLS_CURRENCY = ''*text*''	currency string up to 10 characters, used for local currency marker **L**
NLS_DUAL_CURRENCY = ''*text*''	currency string up to 10 characters, used for dual currency marker **U**
NLS_ISO_CURRENCY = ''*text*''	currency string up to 10 characters, used for ISO currency marker **C**

TO_CHAR Formats: Numeric

Element	Example	Description
, (comma)	9,999	a comma in the specified position. A comma cannot begin a number format. A comma cannot appear to the right of a decimal character or period in a number format.
. (period)	9.99	a decimal point. You can only specify one period in a number format.
$	$9999	a value with a leading dollar sign.
0	09999 9990	leading zeros. trailing zeros.
9	9999	a value with the specified number of digits with a leading space if positive or with a leading minus if negative.
B	B9999	blanks for the integer part of a fixed point number when the integer part is zero.
C	C999	the ISO currency symbol as specified in the ISOCurrency element in the SDF file.
D	99D99	the character that separates the integer and fractional part of non-monetary values. This is specified in the RadixSeparator element in the SDF file.
EEEE	9.9EEEE	a value in scientific notation.
G	9G999	(group separator) the character that separates groups of digits in the integer part of non-monetary values. This is specified in the GroupSeparator element in the SDF file.
L	L999	(local currency) the string representing the local currency as specified in the Currency element in the SDF file.
MI	9999MI	a trailing minus sign if the value is negative. The MI format element can appear only in the last position of a number format.
PR	9999PR	a negative value in <angle brackets>, or a positive value with a leading and trailing blank. The PR format element can appear only in the last position of a number format.

TO_CHAR Formats: Numeric

Element	Example	Description
RN	RN rn	a value as Roman numerals in upper case. a value as Roman numerals in lower case. Valid values: a value between 1 and 3999.
S	S9999 9999S	a negative value with a leading or trailing minus sign. a positive value with a leading or trailing plus sign. The S format element can appear only in the first or last position of a number format.
TM	TM TM9 TME	(text minimum format) returns the smallest number of characters possible. This element is case insensitive. TM or TM9 return the number in fixed notation unless the output exceeds 64 characters. If the output exceeds 64 characters, the number is returned in scientific notation. TME returns the number in scientific notation with the smallest number of characters. You cannot precede this element with an other element. You can follow this element only with one 9 or one E (or e), but not with any combination of these.
U	U9999	(dual currency) the string that represents the dual currency as specified in the DualCurrency element in the SDF file.
V	999V99	a value multiplied by 10 to the n (and, if necessary, rounded up), where *n* is the number of 9's after the V.
X	XXXXX xxxxx	the hexadecimal value of the specified number of digits. If the specified number is not an integer, the function will round it to an integer. This element accepts only positive values or zero. Negative values return an error. You can precede this element only with zero (which returns leading zeroes) or FM. Any other elements return an error. If you do not specify zero or FM, the return always has one leading blank.
FM		Format Minimum mode. The value is returned with no leading or trailing blanks. This may be toggled on and off by adding an 'FM' before and after a section that should use minimum space. This is the default.

Refresher: DateTime Type Casts

teradata.

DateTime data types can be converted to other DataTime data types.

Time zones are automatically adjusted based on the current session time zone or a specified time zone.

```
SET TIME ZONE -4;

SELECT
   Cast(     DATE '2022-05-15' AS TIMESTAMP(0))                          2022-05-15 00:00:00          Midnight
  ,Cast(TIMESTAMP '2022-05-15 09:56:21.71' AS DATE)                      2022-05-15                   Time truncated
  ,Cast(TIMESTAMP '2022-05-15 09:56:21.71' AS TIME(2))                             09:56:21.71        Date truncated
  ,Cast(TIMESTAMP '2022-05-15 09:56:21.71' AS TIME(2) WITH TIME Zone)              09:56:21.71-04:00  Session time zone
  ,Cast(TIMESTAMP '2022-05-15 09:56:21.71' AS TIMESTAMP(2) WITH TIME Zone)  2022-05-15 09:56:21.71-04:00  Session time zone
  ,     TIMESTAMP '2022-05-15 09:56:21.71'       AT 0                    2022-05-15 13:56:21.71+00:00  Adjusted to time zone
  ,     TIMESTAMP '2022-05-15 09:56:21.71+01:00' AT 0                    2022-05-15 08:56:21.71+00:00  Adjusted to time zone
  ,     TIMESTAMP '2022-05-15 09:56:21.71+01:00' AT LOCAL                2022-05-15 04:56:21.71-04:00  Adjusted to time zone
  ,     TIMESTAMP '2022-05-15 09:56:21.71+01:00' AT 'america pacific'    2022-05-15 01:56:21.71-07:00  Adjusted to time zone
  ,Cast(TIMESTAMP '2022-05-15 09:56:21.71+01:00' AS TIME(2))                       04:56:21.71        Session time zone
  ,Cast(TIMESTAMP '2022-05-15 09:56:21.71+01:00' AS TIMESTAMP(2))          2022-05-15 04:56:21.71       Session time zone
  ,Cast(                TIME '09:56:21.71+01:00' AS TIMESTAMP(2));         2022-08-17 04:56:21.71       Current_Date added
```

Data Type Conversion Using TO_CHAR: DateTime

Converts a datetime expression to a Unicode character string applying an optional format.

```
TO_CHAR(datetime_expression [, format_arg ])
```

```
SELECT
   DATE '2022-08-15' AS dt                                  '2022-08-15'
  ,To_Char(dt)                                              '2022/08/15'
  ,To_Char(dt, 'yyyy-mm-dd')                                '2022-08-15'
  ,To_Char(dt, 'MON dd yyyy')                               'AUG 15 2022'
  ,To_Char(dt, 'Dy DY dy')                                  'Mon MON mon'
  ,To_Char(dt, 'FMDay, RM DdSP Year')                       'Monday, VIII Fifteen Twenty Twenty-Two'
  ,TIMESTAMP '2022-08-15 19:56:21.71-04:00' AS ts           '2022-08-15 19:56:21.71-04:00'
  ,To_Char(ts)                                              '2022-08-15 19:56:21.71-04:00'
  ,To_Char(ts, 'HH:mi:ss AM')                               '07:56:21 PM'
  ,To_Char(ts, 'HH24"h"mi"m"ss"s"')                         '19h56m21s'
  ,To_Char(ts, 'Day, MONTH dd. yyyy')                       'Monday   , AUGUST       15. 2022'
  ,To_Char(ts, 'FMday, Month dd. yyyy')                     'monday, August 15. 2022'
```

Teradata extension to SQL:2016

TO_CHAR Formats: DateTime

Element	Description
- / , . ; : "*text*"	Punctuation characters are ignored and text enclosed in quotation marks are inserted as-is.
CC **SCC**	Century. If the last 2 digits of a 4-digit year are between 01 and 99 inclusive, the century is 1 greater than the first 2 digits of that year. If the last 2 digits of a 4-digit year are 00, the century is the same as the first 2 digits of that year.
YYYY	4-digit year.
Y,YYY	Year with comma in this position.
YYY **YY** **Y**	Last 3, 2, or 1 digit of year. If the current year and the specified year are both in the range of 0-49, the date is in the current century.
IYYY	4-digit year based on the ISO standard
IYY IY I	Last 3, 2, or 1 digits of ISO year.
YEAR	Year, spelled out..
Q	Quarter of year (1, 2, 3, 4).
MM	Month (01-12).
MON	Abbreviated name of month.
MONTH	Name of month.
RM	Roman numeral month (I - XII).

Element	Description
WW	Week of year (1-53) where week 1 starts on the first day of the year and continues to the 7th day of the year.
W	Week of month (1-5) where week 1 starts on the first day of the month and ends on the seventh.
IW	Week of year (1-52 or 1-53) based on ISO model.
D	Day of week (1-7).
DAY	Name of day.
DY	abbreviated name of day.
DD	Day of month (1-31).
DDD	Day of year (1-366).
DL	Date Long. Equivalent to the format string 'FMDay, Month FMDD, YYYY'.
DS	Date Short. Equivalent to the format string 'FMMM/DD/YYYYFM'.
J	Julian day, the number of days since January 1, 4713 BC. Number specified with J must be integers. Teradata uses the Gregorian calendar in calculations to and from Julian Days.

TO_CHAR Formats: DateTime (cont.)

Element	Description
HH HH12	Hour of day (1-12).
HH24	Hour of the day (0-23).
MI	Minute (0-59).
SS	Second (0-59).
SSSSS	Seconds past midnight (0-86399).
TS	Time Short. Equivalent to the format string 'HH:MI:SS AM'.
TZH	Time zone hour.
TZM	Time zone minute.
TZR	Time zone region. Equivalent to the format string 'TZH:TZM'.
AM A.M. PM P.M.	Meridian indicator.
FF [1..9]	Fractional seconds. Use [1..9] to specify the number of fractional digits. FF without any number following it prints a decimal followed by digits equal to the number of fractional seconds in the input data type. If the data type has no fractional digits, FF prints nothing. Any fractional digits beyond 6 digits are truncated.

Element	Description
FM	Format Minimum mode. The value is returned with no leading or trailing blanks. This may be toggled on and off by adding an 'FM' before and after a section that should use minimum space..
SP	Spelled. Any numeric element followed by SP is spelled in English words. The words are capitalized according to how the element is capitalized. For example: 'DDDSP' specifies all uppercase, 'DddSP' specifies that the first letter is capitalized, and 'dddSP' specifies all lowercase.

TO_DATE and TO_TIMEZONE Formats

Element	Description
- / , . ; : "*text*"	Punctuation characters are ignored and text enclosed in quotation marks is ignored.
YYYY	4-digit year.
Y,YYY	Year with comma in this position.
YYY YY Y	Last 3, 2, or 1 digit of year. If the current year and the specified year are both in the range of 0-49, the date is in the current century.
RR	Stores 20th century dates in the 21st century using only 2 digits. If the current year and the specified year are both in the range of 0-49, the date is in the current century.
RRRR	Round year. Accepts either 4-digit or 2-digit input. 2-digit input provides the same return as RR.
MM	Month (01-12).
MON	Abbreviated name of month.
MONTH	Name of month.
RM	Roman numeral month (I - XII).
D	Day of week (1-7).
DAY	Name of day.
DY	abbreviated name of day.
DD	Day of month (1-31).

Element	Description
DDD	Day of year (1-366).
J	Julian day, the number of days since January 1, 4713 BC. Number specified with J must be integers. Teradata uses the Gregorian calendar in calculations to and from Julian Days.
HH HH12	Hour of day (1-12).
HH24	Hour of the day (0-23).
MI	Minute (0-59).
SS	Second (0-59).
SSSSS	Seconds past midnight (0-86399).
TZH	Time zone hour.
TZM	Time zone minute.
TZR	Time zone region. Equivalent to the format 'TZH:TZM'.
AM A.M. PM P.M.	Meridian indicator.
FF [1..9]	Fractional seconds. Use [1..9] to specify the number of fractional digits. FF without any number following it prints a decimal followed by digits equal to the number of fractional seconds in the input data type. Any fractional digits beyond 6 digits are truncated.

Data Type Conversion Using TO_DATE & TO_TIMESTAMP

teradata.

Convert a character string to a datetime expression applying an optional format.

`TO_DATE(datetime_expression [, format_arg ])`

`TO_TIMESTAMP(datetime_expression [, format_arg ])`

`TO_TIMESTAMP_TZ(datetime_expression [, format_arg ])`

Returned data type: `DATE` / `TIMESTAMP(6)`

```
SELECT
   To_Date('2022/05/15', 'yyyy/mm/dd')              2022-05-15
  ,To_Date('MAY 15 2022', 'MON dd yyyy')            2022-05-15
  ,To_Timestamp('2022-05-15 09:56:21.71')           2022-05-15 09:56:21.710000
  ,To_Timestamp_TZ('2022-05-15 09:56:21.71-04:00') 2022-05-15 09:56:21.710000-04:00
;
```

Teradata extension to SQL:2016

CAST Using FORMAT

FORMAT is used to mask data with predefined format specifications.

- Mainly used for casting strings to numeric or datetime data types while loading.

```
CAST('$12345.30' AS DECIMAL(10,2))                          12,345.30
CAST('12345.30$' AS DECIMAL(10,2))                          2620 The format or data contains a bad character.
CAST('12345.30$' AS DECIMAL(10,2) FORMAT 'GZ(I)D9(F)$')     12,345.30
CAST('08/24/2022' AS DATE FORMAT 'MM/DD/YYYY')              2022-08-24
CAST('24 AUG 2022' AS DATE FORMAT 'DDbMMMbYYYY')            2022-08-24
CAST('2022-05-15 09:56:21.71' AS TIMESTAMP(2))              2022-05-15 09:56:21.71
```

- Applying a format to a numeric or datetime data type for display works in BTEQ only.
  Other clients require a type cast to a varchar.

```
     Cast(127 AS FORMAT '9(10)')                            127             still numeric
Cast(Cast(127 AS FORMAT '9(10)') AS VarChar(10))            '0000000127'    explicit cast
Trim(Cast(127 AS FORMAT '9(10)'))                           '0000000127'    implicit cast
Trim(Cast(127 AS FORMAT '9(5)'))                            '*****'         value overflows mask!
 -- legacy syntax
      127 (FORMAT '9(10)')                                  127             still numeric
Cast((127 (FORMAT '9(10)')) AS VarChar(20))                 '0000000127'    explicit cast
 Trim(127 (FORMAT '9(10)'))                                 '0000000127'    implicit cast
```

Teradata extension to SQL:2016

FORMAT is a Teradata extension to the ANSI SQL-2003 standard.

Formats can be specified for columns that have character, numeric, byte, DateTime, or UDT data types. FORMAT pertains to data exported in report form, as is the case in BTEQ. FORMAT does not control internal storage representation of data or data returned in record or indicator variable mode.

Use the FORMAT phrase in a CREATE TABLE statement,or ALTER TABLE statement to define the display format for a column. Or, use it in a retrieval statement to override the default format of a column, or to define the display format of an expression. As such, it is both a Data Definition Language phrase and a Data Manipulation Language phrase.

A FORMAT specification can contain a maximum of 30 characters.
The output string that is produced as a result of a FORMAT phrase can have a maximum of 255 characters.

Implicit TRIM only works for numeric data types and DATE, but fails for data types auch as TIME, TIMESTAMP, INTERVAL and Period.

Summary

- You can change the data type for any expression by using the CAST function.
- FORMAT enables non-standard formats.
- Additional type conversion functions exist:
  - TO_CHAR
  - TO_DATE
  - TO_TIMESTAMP
  - TO_NUMBER
- Many date formatting options exist to tailor how dates can be displayed.

Module 13: Advanced String Functions

Teradata Vantage SQL Intermediate

Objectives

After completing this module, you will be able to:

- Combine multiple columns into *delimited-text* output
- Extract individual values from *delimited-text*
- Split *delimited-text* into columns
- Split *delimited-text* into rows
- Extract a value from a *Name-Value-Pair* (NVP) string

Creating Comma-Separated Values

Retrieve all columns as a delimited list

```
SELECT
              employee_number
   || ',' || manager_employee_number
   || ',' || last_name
   || ',' || first_name
   || ',' || hire_date
   || ',' || salary_amount
FROM employee;
```

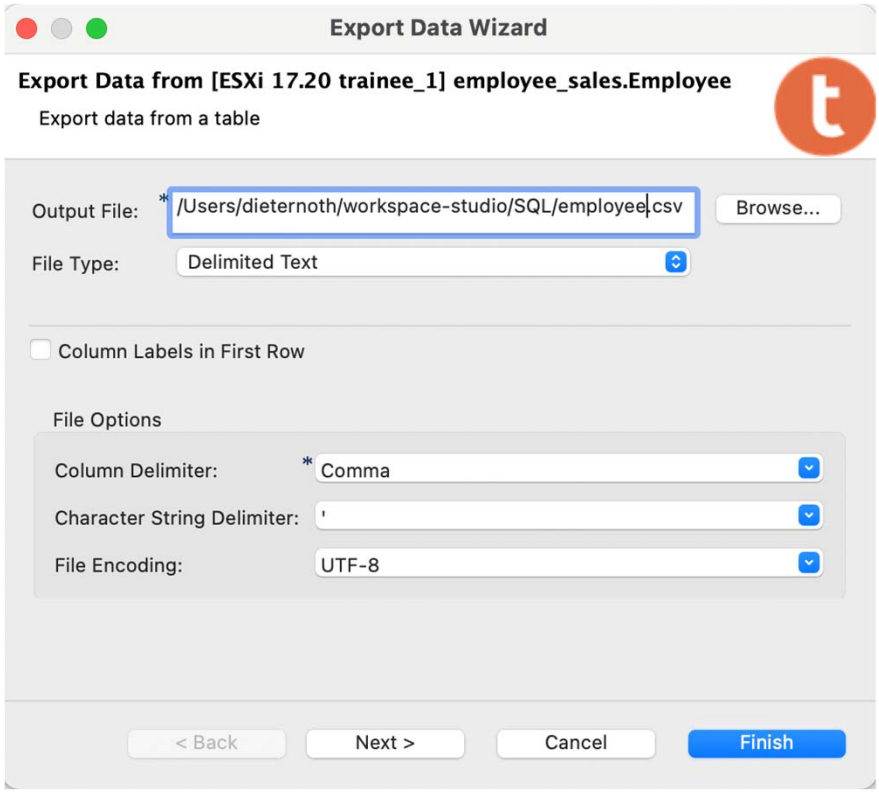
```
      1005,        801,Ryan              ,Loretta,2006-10-15,    31200.00
      1007,       1005,Villegas          ,Arnando,2007-01-02,    49700.00
      1009,       1005,Lombardo          ,Domingus,2007-02-01,    31000.00
      1012,       1005,Hopkins           ,Paulene,2007-03-15,    37900.00
NULL
      1024,       1005,Brown             ,Allen,2009-05-01,    43700.00
```

- NULLs propagate ⇒ COALESCE on Nullable column
- Implicit type cast for numeric columns: right aligned with leading spaces ⇒ TRIM
- Trailing spaces in CHARs ⇒ TRIM
- Optionally: quoted strings

```
SELECT
              Trim(employee_number)
   || ',' || Coalesce(Trim(manager_employee_number), '')
   || ',' || '''' || Trim(last_name)  || ''''
   || ',' || '''' || Trim(first_name) || ''''
   || ',' || hire_date
   || ',' || Coalesce(Trim(salary_amount), '')
FROM employee_sales.employee;
```

```
1005,801,'Ryan','Loretta',2006-10-15,31200.00
1007,1005,'Villegas','Arnando',2007-01-02,49700.00
1009,1005,'Lombardo','Domingus',2007-02-01,31000.00
1012,1005,'Hopkins','Paulene',2007-03-15,37900.00
1020,1005,'Charles','John',2008-10-01,
1024,1005,'Brown','Allen',2009-05-01,43700.00
```

CSV data can be be easily be exported using TPT (most efficient for large tables) or Terada Studio (smaller tables)



CSV Table Function

`CSV ( NEW VARIANT_TYPE ( value ), delimiter_char, quote_char )`

CSV returns input row column values in text format separated by a user-specified delimiter character.

```
WITH cte AS
 ( -- base select here
   SELECT *
   FROM employee_sales.employee AS e
)
SELECT *
FROM TABLE
 ( CSV
    ( NEW VARIANT_TYPE
       ( -- you need to list each column of your table
         cte.employee_number
        ,cte.manager_employee_number
        ,trim(cte.last_name) AS last_name
        ,cte.first_name
        ,cte.hire_date
        ,cte.salary_amount
       )
     , ','    -- delimiter character
     , '''')  -- quote character
   RETURNS (op VARCHAR(32000) CHARACTER SET UNICODE)
 ) AS dt
 ORDER BY Cast(StrTok(op, ',', 1) AS INT);
```

- Deals with NULLs
- Numeric columns without leading spaces
- Trailing spaces still need to be handled
- Optionally quoting strings
- Supports up to 1024 input columns
- Character Set based on input columns

```
1005,801,'Ryan','Loretta',2006-10-15,31200.00
1007,1005,'Villegas','Arnando',2007-01-02,49700.00
1009,1005,'Lombardo','Domingus',2007-02-01,31000.00
1012,1005,'Hopkins','Paulene',2007-03-15,37900.00
1020,1005,'Charles','John',2008-10-01,
1024,1005,'Brown','Allen',2009-05-01,43700.00
```

Teradata extension to SQL:2016

NEW VARIANT_TYPE

Supports up to 128 columns of

`BYTEINT, SMALLINT, BIGINT, INTEGER, DECIMAL, FLOAT, DATE, TIME, TIME WITH TIME ZONE, TIMESTAMP, TIMESTAMP WITH TIME ZONE, INTERVAL, CHAR, VARCHAR`

Up to 8 `NEW VARIANT_TYPE` can be used supporting up to 1024 columns (8 * 128)

Pack Table Operator (16.20 FU2)

The Pack operator packs data from multiple input columns into a single column.

```
WITH cte AS
 ( -- base select here
   SELECT
       employee_number
      ,manager_employee_number
      ,'''' || Trim(last_name) || '''' AS last_name
      ,'''' || first_name || '''' AS first_name
      ,hire_date
      ,salary_amount
   FROM employee_sales.employee
 )
SELECT * FROM Pack (
  ON cte
  USING
     Delimiter (',')
     OutputColumn ('packed_data')
     IncludeColumnName('no')
) AS dt;
```

- Deals with NULLs
- Numeric columns without leading spaces
- Trailing spaces still need to be handled
- Optionally quoting strings

```
1005,801,'Ryan','Loretta',2006-10-15,31200.00
1007,1005,'Villegas','Arnando',2007-01-02,49700.00
1009,1005,'Lombardo','Domingus',2007-02-01,31000.00
1012,1005,'Hopkins','Paulene',2007-03-15,37900.00
1020,1005,'Charles','John',2008-10-01,
1024,1005,'Brown','Allen',2009-05-01,43700.00
```

Teradata extension to SQL:2016

- **TargetColumns** Specify the names of the input table columns to pack into a single output column. If you specify this syntax element, but do not specify all input table columns, the function copies the unspecified input table columns to the output table.
- **Delimiter** Specify the delimiter - a single Unicode character. Default: ',' (comma)
- **IncludeColumnName** Specify whether to label each virtual column value with its column name (*name:value* pairs). Default: 'true'
- **OutputColumn** Specify the name to give to the packed output column.
- **Accumulate** Specify the input columns to copy to the output table.
- **ColCast** Specify whether to cast each numeric target_column to VARCHAR. Specifying 'true' decreases run time for queries with numeric target columns. Default: false

```
SELECT * FROM Pack (
  ON { table | view | (query) }
  USING
  [ TargetColumns ({ 'target_column' | target_column_range }[,...]) ]
  [ Delimiter ('delimiter') ]
  [ IncludeColumnName ({'true'|'t'|'yes'|'y'|'1'|'false'|'f'|'no'|'n'|'0'}) ]
  OutputColumn ('output_column')
  [ Accumulate ({ 'accumulate_column' | accumulate_column_range }[,...]) ]
  [ ColCast ({'true'|'t'|'yes'|'y'|'1'|'false'|'f'|'no'|'n'|'0'}) ]
) AS alias;
```

CSVLD Table Function

CSVLD (*value , delimiter_char, quote_char*)

CSVLD takes in a comma-separated string (as produced by the CSV table function), parses the string, and returns VARCHAR columns.

```
WITH cte AS
 (
   SELECT col FROM vt_csv
 )
SELECT *
FROM TABLE
 ( CSVLD
    (cte.col
    ,','  -- delimiter character
    ,'''' -- quote character
    )
   RETURNS
    ( emp# VarChar(11)
     ,mgr# VarChar(11)
     ,last_name VarChar(30)
     ,first_name VarChar(30)
     ,birthdate VarChar(10)
     ,salary VarChar(12)
    )
 ) AS t;
```

CSV string

- Returns up to 1024 VarChar columns
- Character Set based on input column
- Type cast can be added in the Select list

```
emp# mgr# last_name first_name birthdate  salary
---- ---- --------- ---------- ---------- --------
1009 1005 Lombardo  Domingus   2007-02-01 31000.00
1005 801  Ryan      Loretta    2006-10-15 31200.00
1012 1005 Hopkins   Paulene    2007-03-15 37900.00
1020 1005 Charles   John       2008-10-01 NULL
1007 1005 Villegas  Arnando    2007-01-02 49700.00
1024 1005 Brown     Allen      2009-05-01 43700.00
```

Teradata extension to SQL:2016

```
WITH cte AS
 ( -- base select here
   SELECT col
   FROM vt_csv
 )
SELECT – add type casts here
      Cast(employee_number AS INTEGER)
     ,Cast(manager_employee_number AS
INTEGER)
     ,last_name
     ,first_name
     ,Cast(birthdate AS DATE)
     ,Cast(salary_amount AS DEC(10,2))
FROM TABLE
 ( CSVLD
    (cte.col
    ,','
    ''''
    ,
    )
   RETURNS
    (
      employee_number VarChar(11)
     ,manager_employee_number VarChar(11)
     ,last_name VarChar(30)
     ,first_name VarChar(30)
     ,birthdate VarChar(10)
     ,salary_amount VarChar(12)
    )
 ) AS t;
```

STRTOK

STRTOK (*instring* [, *delimiter* [, *tokennum*]])

Splits *instring* into tokens based on the specified list of *delimiter* characters and returns the *nth* token, where *n* is specified by the *tokennum* argument.

```
SELECT
   email
  ,StrTok(email, '@', 1) AS email_user
  ,StrTok(email, '@', 2) AS email_domain
FROM vt_emails
;
```

```
email                       email_user email_domain
--------------------------- ---------- ------------------
jane@myemail.com            jane       myemail.com
johndoe@domain.com          johndoe    domain.com
jane.doe@subdomain.corp.com jane.doe   subdomain.corp.com
john.o.doe@gmail.com        john.o.doe gmail.com
jdoe@mycomp.com             jdoe       mycomp.com
```

```
StrTok('one,,three,four', ',', 2)              three
StrTok('one,,three,four', ',', 3)              four
StrTok('one,,three,four', ',', 4)              NULL
StrTok('this!! Is --#3????', ' ,.-:;!?', 3)    #3
```

- Non-existing tokens return NULL
- Multiple delimiter characters allowed
- Consecutive delimiters are treated as one

Teradata extension to SQL:2016

Expressions passed to this function must have the following data types:

- *instring* = *VARCHAR(32000)* or *CLOB*
- *delimiter* = *VARCHAR(64)*
- *tokennum* = *INTEGER*

Result Type

The result data type is VARCHAR and the character set is the same as that of the *instring* argument. For example, if the *instring* argument has a data type of VARCHAR CHARACTER SET UNICODE, the result data type is VARCHAR CHARACTER SET UNICODE.

The maximum length of any token returned is 256.

If *instring* contains fewer tokens than tokennum, the function returns NULL.

STRTOK_SPLIT_TO_TABLE Table Function

STRTOK_SPLIT_TO_TABLE (*inkey, instring, delimiter*)
RETURNS (*outkey, tokennum, token*)

Splits strings into a table of tokens based on the provided delimiter(s) string.

```
WITH cte AS
 (
   SELECT email_id, email
   FROM vt_emails
 )
SELECT *
FROM TABLE
 (
   StrTok_Split_To_Table(cte.email_id -- key
                        ,cte.email    -- data
                        ,'.@')        -- delimiters
   RETURNS (email_id INTEGER
           ,ord INTEGER
           ,token VARCHAR(128)
           )
 ) AS t
ORDER BY email_id, ord
;
```

```
email_id ord token
 -------- --- ---------
        1   1 jane
        1   2 myemail
        1   3 com
        2   1 johndoe
        2   2 domain
        2   3 com
        3   1 jane
        3   2 doe
        3   3 subdomain
        3   4 corp
        3   5 com
        4   1 john
        4   2 o
        4   3 doe
        4   4 gmail
        4   5 com
        5   1 jdoe
        5   2 mycomp
        5   3 com
```

Teradata extension to SQL:2016

Expressions passed to this table function must have the following data types:

- *inkey* = Numeric or *VARCHAR*
- *instring* = *VARCHAR(32000)* or *CLOB*
- *delimiter* = *VARCHAR(64)*

Result Type

- *outkey* = Numeric or *VARCHAR* (same as *inkey*)
- *tokennum* = *INTEGER*
- *token* = *VARCHAR(256)*

The result data type is VARCHAR and the character set is the same as that of the *instring* argument. For example, if the *instring* argument has a data type of VARCHAR CHARACTER SET UNICODE, the result data type is VARCHAR CHARACTER SET UNICODE.

The maximum length of any token returned is 256.

If *instring* contains fewer tokens than tokennum, the function returns NULL.

STRTOK_SPLIT_TO_TABLE Table Function (cont.)

```
WITH cte AS
 ( SELECT email_id, email
   FROM vt_emails
 )
,split AS
 (
   SELECT *
   FROM TABLE
    (
      StrTok_Split_To_Table(cte.email_id
                           ,cte.email
                           ,'.@')
      RETURNS (email_id INTEGER
              ,ord INTEGER
              ,token VARCHAR(128)
              )
    ) AS t
 )
SELECT
   e.email_id
  ,e.email
  ,split.ord
  ,split.token
FROM split
JOIN vt_emails AS e
ON e.email_id = split.email_id
ORDER BY split.email_id, split.ord;
```

- Table functions can't be directly joined

  3706 Syntax error: Joined table is not supported in conjuction with table operators or table function invoked with variable input argument.

- Workaround: Wrap it in a CTE or a Derived Table

email_id	email	token
1	jane@myemail.com	jane
1	jane@myemail.com	myemail
1	jane@myemail.com	com
2	johndoe@domain.com	johndoe
2	johndoe@domain.com	domain
2	johndoe@domain.com	com
3	jane.doe@subdomain.corp.com	jane
3	jane.doe@subdomain.corp.com	doe
3	jane.doe@subdomain.corp.com	subdomain
3	jane.doe@subdomain.corp.com	corp
3	jane.doe@subdomain.corp.com	com
4	john.o.doe@gmail.com	john
4	john.o.doe@gmail.com	o
4	john.o.doe@gmail.com	doe
4	john.o.doe@gmail.com	gmail
4	john.o.doe@gmail.com	com
5	jdoe@mycomp.com	jdoe
5	jdoe@mycomp.com	mycomp
5	jdoe@mycomp.com	com

NVP

NVP(*instring, name_to_search* [, *name_delimiters, value_delimiters* [, *occurrence*]])

Extracts the value of a *name-value pair* where the name in the pair matches the name and the number of the occurrence specified.

- Default *name_delimiters* is **&** (ampersand)
- Default *value_delimiters* is **=** (equal sign)
- Default *occurence* is **1**
- *name_to_search* is case sensitive
- Multibyte delimiters allowed, must match exactly

```
SELECT
   'ApplicationName=STUDIO;Version=17.20.0.202209150411;ClientUser=trainee_1;Source=MetadataQuery;Version=007' AS QB
  ,NVP(QB, 'Version', ';', '=')              17.20.0.202209150411
  ,NVP(QB, 'Version', ';', '=', 2)           007
  ,NVP(QB, 'Version', ';', '=', 3)           NULL
  ,NVP(QB, 'version', ';', '=', 1);          NULL
```

```
SELECT
   'name1==STUDIO#;#Version==17.20.0.202209150411#;#ClientUser==trainee_1#;#Source==MetadataQuery' AS QB
  ,NVP(QB, 'Version', '#;#', '==', 1);       17.20.0.202209150411
```

Teradata extension to SQL:2016

Delimiters can contain any characters. They are separated from each other in the string by spaces. If a space is used as part of a delimiter, it must be escaped using a backslash (\). The maximum length of any delimiter is 10, and the maximum size of this parameter is 32.

Summary

- CSV and the Pack create output in *Comma-Separated-Values* (CSV) format.
- STRTOK extracts a single value from CSV.
- CSVLD and STRTOK_SPLIT_TO_TABLE split CSV into multiple columns or rows.
- NVP extracts a value from a *Name-Value-Pair* (NVP) string.

Advanced String Functions

Labs

Lab 1
Advanced String Functions

teradata.

Write a macro returning some employee details.
Accept multiple employee numbers as a CSV parameter.
Return a row for each value in the input list.

```
HELP TABLE hr_payroll;
HELP TABLE hr_departments;
HELP TABLE hr_jobs;
```

```
EXEC employee_info('536792,301779,333824,445908,543446');
```

```
*** Query completed. 5 rows found. 5 columns returned.

emp#   last_name First_Name Job_Title                                  Department_Name
------ --------- ---------- ---------------------------------------- -------------------------
536792 Taylor    Gary       Juvenile Probation Officer/POST Certified Juvenile Services
301779 Macdonald Michael    Emergency Communications Officer          Emergency Medical Services
333824 Desalvo   Peter      College Student Intern/Contract           Emergency Medical Services
445908 Pope      Lou-Vanna  Maintenance Worker I                      Maintenance
543446 Shaffett  Jacob      College Student Intern/Contract           Parish Attorney
```

```
EXEC employee_info('111,347914,blah,236276');
```

```
*** Query completed. 4 rows found. 5 columns returned.

token  last_name First_Name Job_Title           Department_Name
------ --------- ---------- ------------------ ----------------------------------------------
111    n/a       NULL       NULL               NULL
347914 Raby      Kenya      Pest Control Worker Mosquito Abatement and Rodent Control District
blah   n/a       NULL       NULL               NULL
236276 St Romain Joseph     NULL               Parish Attorney
```

Advanced String Functions

Lab Solutions

Lab 1 Solution
Advanced String Functions

```
REPLACE MACRO employee_info(param VARCHAR(1000)) AS
 (
   WITH split AS
    (
      SELECT TryCast(token AS INTEGER) AS emp#, tokennum, token
      FROM TABLE (STRTOK_SPLIT_TO_TABLE(1, :param, ',')
           RETURNS (outkey INTEGER,  -- usually the PK of the table, here it's just a dummy
                    tokennum INTEGER, -- order of the token within the param string
                    token VARCHAR(128) CHARACTER SET UNICODE)
                 ) AS t
    )
   SELECT
      split.token AS emp# -- use token instead of emp#
     ,Coalesce(e.Last_Name, 'n/a') AS last_name
     ,e.First_Name
     ,j.Job_Title, d.Department_Name
   FROM finance_payroll.hr_payroll AS e
   LEFT JOIN finance_payroll.hr_jobs AS j
     ON e.job_code = j.job_code
   JOIN finance_payroll.hr_departments AS d
     ON e.department_number = d.department_number
   RIGHT JOIN split -- include all token
     ON e.employee_number = split.emp#
   ORDER BY tokennum
   ;
 )
;
```

Module 14: Data Types and Functions – Binary

Teradata Vantage SQL Intermediate

Objectives

After completing this module, you will be able to:

- Use Teradata Binary Data Types and Functions.

Topics

- Overview
- Binary Data Types and Functions

Current Topic – Overview

- **Overview**
  - **Examples for binary data**
  - **Historical Evolution**
  - **From Bits to Bytes**
- Binary Data Types and Functions

Warning

The following slides may be

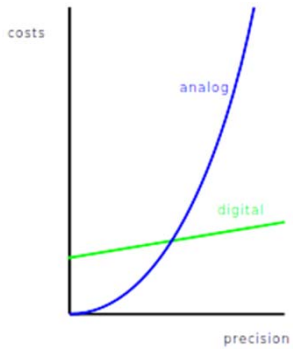
- boring
- a repetition
- not interesting
- something new

„Keep Calm
and
Carry On“

Overview
Examples for Binary Data

teradata.

The world is analog.



Why do we digitize analog values?

It becomes cheaper to calculate with digital values if an accuracy of less than 1‰ is to be achieved.

Disadvantage:

Computers do not know „infinity“ (∞).

They cannot estimate what may happen when dividing by zero.

Computers may only be usable within limits:

-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807

or

from $-(2^{63})$ to $2^{63} - 1$

when using a 64-bit integer.

Overview
Examples for Binary Data (cont.)

teradata.

The world is analog.

Why should we struggle with binary data?

Normally in the business world we are confronted with values based on decimals, because humans do have ten fingers.

But there are also objects which may be described with two values only, 0 and 1 or true and false.

- A lamp may be turned or off
- A door may be open or closed
- An alarm may be triggered or not
- A message may be true or false

And here we are now approaching the area where we need to deal with binary data. Since a couple of years, new buzzwords ramped up:

- IoT ⇒ the "Internet of Things"
- Smart Homes ⇒ control electronic devices remotely using a mobile

Overview
Examples for Binary Data (cont.)

teradata.

Some areas are using binary data even though the majority of human beings don't care about it.

- The internet
- Mobile communication
- TV/Radio broadcasting
- Plane Guidance and Control
- Satellite navigation
- Wind energy parks
- …

Therefore, you may "need to know“ a little bit about binary data.

Some Words on the History

The earliest sources about binary data go back to 3rd century BC in India (*Pingala*) and the 11th century in China (*I Ching*).

Not so long ago we should recognize as pioneers:

- Abu Dschaʿfar Muhammad ibn Musa al-Chwārizmī
  (→ *algorithm, making the Indian '0' popular)*
- Gottfried Wilhelm Leibniz
- Charles Babbage and Ada Lovelace
- George Boole
- Konrad Zuse
- Alan Turing
- and many more

The internet can be helpful in learning more about the history of computing.

Comparing the Decimal System with the Binary System

teradata.

We are so used to decimal values that maybe nobody thinks about them anymore.

Decimal numbers use the numerals {0,1,2,3,4,5,6,7,8,9}

We can use these to describe any number if we use them like this:

When counting units, start with 1, continue with 2 … and when you reach 9 and keep counting, use 0 and remember the first 10 values have been counted.

1, 2, 3, … , 9, 10, 11, 12, … , 19, 20, 21, … , 99, 100, …

Comparing the Decimal System with the Binary System (cont.)

teradata.

This can be expressed mathematically using a formula, a polynomial.

(simplified)

$$a * 10^{n} + a * 10^{(n-1)} + \ldots + a * 10^{3} + a * 10^{2} + a * 10^{1} + a * 10^{0}$$
$$a \in \{0, \ldots, 9\}$$

and 10^{0} as *units*, 10^{1} as *tens*, 10^{2} as *hundreds*, 10^{3} as *thousands* etc.

A value of 1234 can be expressed as a polynomial:

$$1 * 10^{3} + 2 * 10^{2} + 3 * 10^{1} + 4 * 10^{0} = 1234$$

Comparing the Decimal System with the Binary System (cont.)

teradata.

On the way towards binary numbers a more general form of the former polynomial may help.

$$a * B^n + a * B^{(n-1)} + \ldots + a * B^3 + a * B^2 + a * B^1 + a * B^0$$
$$B = 10;\ a \in \{0, \ldots, 9\}:\ decimalsystem$$
$$B = 2;\ a \in \{0,1\}:\ binarysystem$$

$$a * 2^3 + a * 2^2 + a * 2^1 + a * 2^0$$
$$2^0 = 1$$
$$2^1 = 2$$
$$2^2 = 4$$
$$2^3 = 8$$
$$1 * 2^3 + 0 * 2^2 + 1 * 2^1 + 0 * 2^0 = 1010 = 12$$

Can you answer why this is true? `15⁰ = 1`

Comparing the Decimal System with the Binary System (cont.)

$$1 * 2^3 + 0 * 2^2 + 1 * 2^1 + 0 * 2^0 = 1010 = 12$$

With four bits only it is possible to represent all decimal digits, 0 to 9.

But there are 16 possible combinations, therefore today we may also be confronted with a system based on 16, hexadecimal numbers.

By the way, every calculator is still using a 4-bit decimal system with an overflow to the next position when 9 is exceeded.

decimal	binary	hex
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

From Bits to Bytes

In the early 1970s, the first microprocessors reached the developers.

Still only 4 bits, but soon expanded to 8 bits or 1 byte.

Today we still use the unit „byte“ for the size of RAM or disks.

No longer KByte, but GByte, TByte or PByte.

The structure of a byte:

MSB							LSB
7	6	5	4	3	2	1	0
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
0/128	0/64	0/32	0/16	0/8	0/4	0/2	0/1

MSB: most significant bit
LSB: least significant bit

the exponent only from

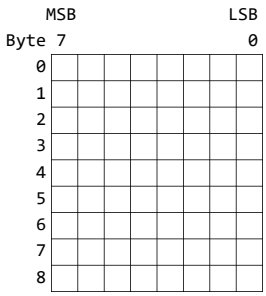
←

the decimal value per bit

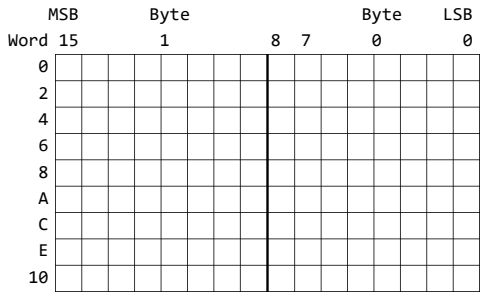
From Bytes to Words and DoubleWords and Beyond ("little endian")

teradata.

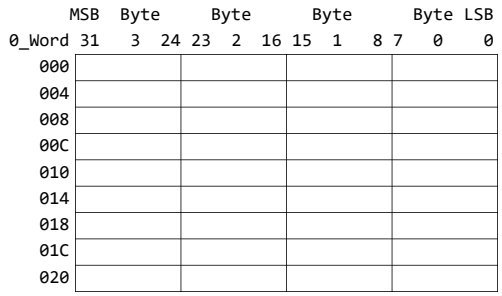
The 8 bit world



The 16 bit world



The 32 bit world



The 64 bit world … left for your fantasy

From Theory to Practical Use

Based on works of Leibniz, Boole, Venn and others, the „Boolean Algebra“ evolved.

It uses three operators:

- the unary NOT (¬, **x**)
- AND (∧, •) and
- **OR** (∨, **+**)

The functionality is represented with truth tables.

A	NOT A (¬A)
0	1
1	0

A	B	A AND B (A ∧ B)
0	0	0
0	1	0
1	0	0
1	1	1

A	B	A OR B (A ∨ B)
0	0	0
0	1	1
1	0	1
1	1	1

From Theory to Practical Use (cont.)

Using two binary input variables, there are four possible combinations using 0 and 1, but 16 possible outputs.

Not all of them are „useful“.

A	B	NEVER	NOR		NOT A		NOT B	XOR	NAND	AND	XNOR	B		A		OR	ALWAYS
0	0	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
0	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	0	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1

Current Topic – Binary Data Types and Functions

- Overview
- **Binary Data Types and Functions**
  - **Binary Data Types**
  - **Bit Functions**

Byte Data Types

- Three data types for holding character data:
  - `BYTE(n)` - fixed-length binary string up to 64,000 bytes.
    Left justified, right padded with binary zeroes to fill the length.
  - `VARBYTE(n)` - variable-length binary string up to 64000 bytes.
  - `BLOB(n)` - *B*inary *L*arge *OB*ject to store binary objects, such as graphics, video clips, PDFs, etc. up to 2 gigabytes

- Never translated by the Teradata Database
  - Stored “as is” from the client

```
DatabaseId BYTE(4)
EncryptedPassword VARBYTE(1024)
Histogram BLOB(10285760)
```

Data Storage of Byte and BLOB Types

The BYTE, VARBYTE, and BLOB data types are stored in the client system format—they are never translated by Vantage, but stored “as is” from the client source.

The BYTE, VARBYTE, and BLOB data types store raw data as logical bit streams. For any machine, BYTE, VARBYTE, and BLOB data is transmitted directly from the memory of the client system. The sort order is logical, and values are compared as if they were *n*-byte, unsigned binary integers suitable for digitized images.

The BLOB data type is just a much larger VARBYTE.

BLOB is ANSI SQL:2016 compliant.
BYTE and VARBYTE are Teradata extensions to the ANSI standard.

Byte Strings

- Hexadecimal Byte Literals
  *'hexadecimal digits'*XB[V]
  - A *hexadecimal digit* is a character from 0 to 9, a to f, or A to F
  - Up to 62000 characters surrounded by single quotation marks
    '546572616461746120'XB ⇒ BYTE(9)
    '56616E74616765AE'XBV ⇒ VARBYTE(8)
  - Even number of hexadecimal digits required
    - Automatically extended on the right with zeros when required
      '546'XB ⇒ '5460'XB
- Byte concatenation
  - Concatenation operator: || (Standard SQL)
  - Concatenation function: CONCAT (Teradata 16.20)

```
SELECT CONCAT('546572616461746120'XB, '56616E74616765AE'XB);
SELECT '00'XB || '010203'XB;
```

Syntax

'hexadecimal digits' **XB** [**V** | **F**]

Syntax Elements

hexadecimal digits

A string of hexadecimal digits, where a hexadecimal digit is a character from 0 to 9, a to f, or A to F.

V

The hexadecimal literal is in byte variable format.

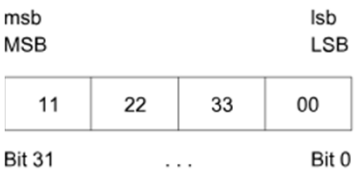
F

The hexadecimal literal is in byte fixed format. This is the default if F or V is not specified.

Usage Notes

Hexadecimal byte literal is the only form for entering a byte string.
Hexadecimal byte literals are represented by an even number of hexadecimal digits. Hexadecimal literals are extended on the right with zeros when required.

For example:
A 7-char hexadecimal byte literal, '1122330'XB, becomes a 4-byte hexadecimal byte literal: '11223300'XB



Teradata Binary Functions – td_sysfnlib

`BITNOT ( target_arg )`	The bitwise NOT, or complement, is a unary operation which performs logical negation on each bit, forming the ones' complement of the specified binary value.
`BITAND ( target_arg, bit_mask_arg )`	This function takes two bit patterns of equal length and performs the logical AND operation on each pair of corresponding bits.
`BITOR ( target_arg, bit_mask_arg )`	This function takes two bit patterns of equal length and performs the logical OR operation on each pair of corresponding bits.
`BITXOR ( target_arg, bit_mask_arg )`	The bitwise exclusive OR takes two bit patterns of equal length and performs the logical XOR operation on each pair of corresponding bits.
`GETBIT ( target_arg, target_bit_arg )`	Returns the value of the bit specified by *target_bit_arg* from the *target_arg* byte expression.
`SETBIT ( target_arg, target_bit_arg [, 0|1 ] )`	Sets the value of the bit specified by *target_bit_arg* to the value of (0 or 1) in the *target_arg* byte expression.
`COUNTSET( target_arg [, 0|1 ] )`	Returns the count of the binary bits within the *target_arg* expression that are either set to 1 or set to 0.
`SUBBITSTR ( target_arg, position_arg, num_bits_arg )`	Extracts a bit substring from the *target_arg* string expression starting at the bit position specified by *position_arg*.
`SUBSTR ( target_arg, position_arg [, length_arg ] )`	Extracts a substring from a byte string starting on *position_arg*

Bit and Byte Numbering Model

The following diagrams show the logical bit and byte numbering model employed by the byte/bit manipulation functions described in these sections.

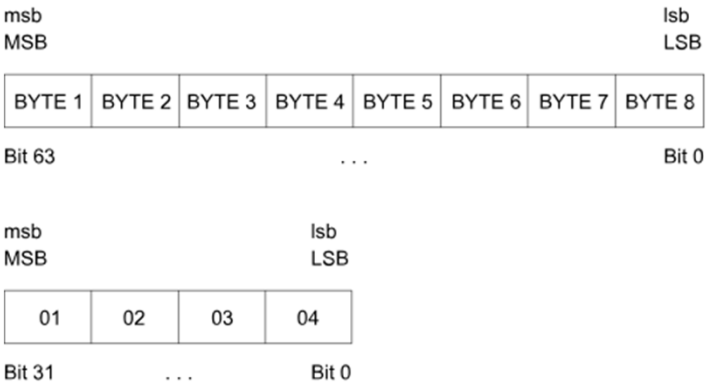
The model is big endian or little endian independent. The numbering system used for numeric data types is consistent with that used for byte strings. This simplifies the development of appropriate bit masks.

Users of the byte/bit manipulation functions must visualize the numeric and byte data types as shown when contemplating what masks (*bit_mask_arg*) need to be applied to the target data (*target_arg*).

Example: A VARBYTE Value with 8 bytes

HEXADECIMAL BYTE LITERALS

With respect to byte-bit system functions, hexadecimal byte literals are interpreted as follows:
A 4-byte hexadecimal byte literal: '01020304'XB



Teradata Binary Functions – td_sysfnlib (cont.)

SHIFTLEFT (*target_arg*, *num_bits_arg*)	Returns the expression *target_arg* shifted by the specified number of bits (*num_bits_arg*) to the left. The bits in the most significant positions are lost, and the bits in the least significant positions are filled with zeros.
SHIFTRIGHT (*target_arg*, *num_bits_arg*)	Returns the expression *target_arg* shifted by the specified number of bits (*num_bits_arg*) to the right. The bits in the least significant positions are lost, and the bits in the most significant positions are filled with zeros.
ROTATELEFT(*target_arg*, *num_bits_arg*)	Returns an expression rotated to the left by the number of bits you specify, with the most significant bits wrapping around to the right.
ROTATERIGHT (*target_arg*, *num_bits_arg*)	Returns an expression rotated to the right by the number of bits you specify, with the least significant bits wrapping around to the left.
BYTES (*byte_expression*)	Returns the number of bytes contained in the specified byte string.
TO_BYTES (*in_string*, *in_encoding*)	Decodes a sequence of characters in a given encoding into a sequence of bits.
FROM_BYTES (*in_string*, *out_encoding*)	Encodes a sequence of bits into a sequence of characters representing its encoding
TO_BYTE (*target_arg*)	Converts a numeric data type to Vantage byte representation (byte value) of the input value.

Byte Conversion Functions

To_Bytes(*instring, in_encoding*) decodes a sequence of characters in a given *encoding* into a sequence of bytes. Returns a VARBYTE (31750).

```
To_Bytes('01b69b4ba630f34e', 'base16')                                                    01-b6-9b-4b-a6-30-f3-4e
To_Bytes('0000000110110110100110110100101110100110001100001111001101001110', 'base2')   01-b6-9b-4b-a6-30-f3-4e
To_Bytes('06664664564614171516', 'base8')                                                 01-b6-9b-4b-a6-30-f3-4e
To_Bytes('00FF', 'base16')                                                                00-ff
To_Bytes('FF',   'base16')                    Leading zeros added if msb = 1             00-ff
To_Bytes('Hello world!', 'ascii')                                                         48-65-6c-6c-6f-20-77-6f-72-6c-64-21
```

From_Bytes(*inbytes, out_encoding*) decodes a sequence of bytes into a sequence of characters representing its *encoding*. Returns a VARCHAR(31750) CHARACTER SET Unicode.

```
From_Bytes('FF'xb,       'base10')      '-1'        Leading zeros
From_Bytes('00FF'xb,     'base10')      '255'       always omitted
From_Bytes('FFFF'xb,     'base10')      '-1'
From_Bytes('00FFFF'xb,   'base10')      '65535'
From_Bytes('000000FF'xb, 'base10')      '65535'
From_Bytes('000000FF'xb, 'base16')      'FF'
From_Bytes('01b69b4ba630f34e'xb, 'base10')          '123456789012345678'
From_Bytes('48656c6c6f20776f726c6421'xb, 'ascii')   'Hello world!'
From_Bytes('0001'xb, 'base2')           '1'
Lpad(From_Bytes('0001'xb, 'base2'), 16, '0')        '0000000000000001'
From_Bytes('01b69b4ba630f34e'xb, 'base8')           '66646645646141715 16'
```

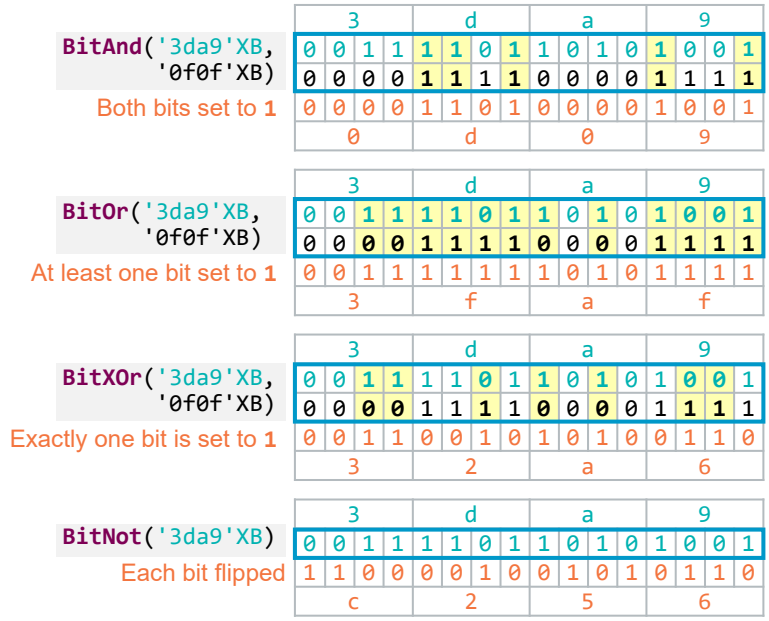
The following encodings are supported:

- Base2
- Base8
- Base10
- Base16
- Base36
- Base64M (MIME)
- ASCII

```
-- Convert VARBYTE(n) to a binary string with leading zeros
SELECT
   '0001b69b4ba630f34e'xb AS in_byte
  ,Translate(LPad ( -- without the leading zero byte a minus sign
                    -- will be returned if MSB = 1
                    From_Bytes('00'xb || in_byte, 'base2')
                    -- calculate the number of digits needed to display
                  , BYTES(in_byte) * 8
                  , '0' -- add back leading zeroes
                  )
             -- No UNICODE needed, switch to LATIN to save space
             USING Unicode_to_Latin) AS byte2bin
;
-- Should be created as a SQL_UDF
REPLACE FUNCTION byte2binary2(in_byte VARBYTE(128))
-- RETURN VarChar must be 8*in_byte
RETURNS VARCHAR(1024) CHARACTER SET Latin
LANGUAGE SQL
CONTAINS SQL
DETERMINISTIC
RETURNS NULL ON NULL INPUT
SQL SECURITY DEFINER
COLLATION INVOKER
INLINE TYPE 1
RETURN
   -- From_Bytes truncates leading zeroes
   LPad ( -- without the leading byte a minus sign
          -- will be returned if MSB = 1
          From_Bytes('00'xb || in_byte, 'base2')
          -- calculate the number of digits needed to display
```

```
        , Bytes(in_byte) * 8
        , '0' -- add leading zeroes
        )
;
```

BitAnd, BitOr, BitXOr, and BitNot



If *target* and *bit mask* arguments passed to these functions differ in length:

- The *target* and *bit mask* arguments are aligned on their least significant byte/bit.
- The smaller argument is padded with zeros *to the left* until it becomes the same size as the larger argument.
- Vantage pads to the left (instead of to the right) so that the hexadecimal byte literals, serving as bit masks, will not have to be changed every time the size of a byte string is changed.

All bit functions return `VARBYTE(8192)`

- Type cast to `[VAR]BYTE(n)` may be required

Performing Bit-Byte Operations against Arguments with Non-Equal Lengths
This topic applies only to the BITOR, BITXOR, and BITAND functions.

If the target_arg and bit_mask_arg arguments passed to these functions differ in length:

- The target_arg and bit_mask_arg arguments are aligned on their least significant byte/bit.
- The smaller argument is padded with zeros to the left until it becomes the same size as the larger argument.

Vantage pads to the left (instead of to the right) so that the hexadecimal byte literals, serving as bit masks, will not have to be changed every time the size of a byte string is changed.

Byte ⇄ Integer Conversion

`To_Byte( in_numeric )` converts an integer data type into its byte representation.

```
To_Byte(-1)                                 ff                   = 11111111
To_Byte(255)                                00-ff                = 0000000011111111
To_Byte(CAST(-1 AS SMALLINT))               ff-ff                = 1111111111111111
To_Byte(65535)                              00-00-ff-ff          = 00000000000000001111111111111111
To_Byte(15785)                              3d-a9                = 0011110110101001
To_Byte(16711935)                           00-ff-00-ff          = 00000000111111110000000011111111
To_Byte(CAST(123456789012345678 AS BIGINT)) 01-b6-9b-4b-a6-30-f3-4e = 0000000110110110100110110100101110100110001100001111001
```

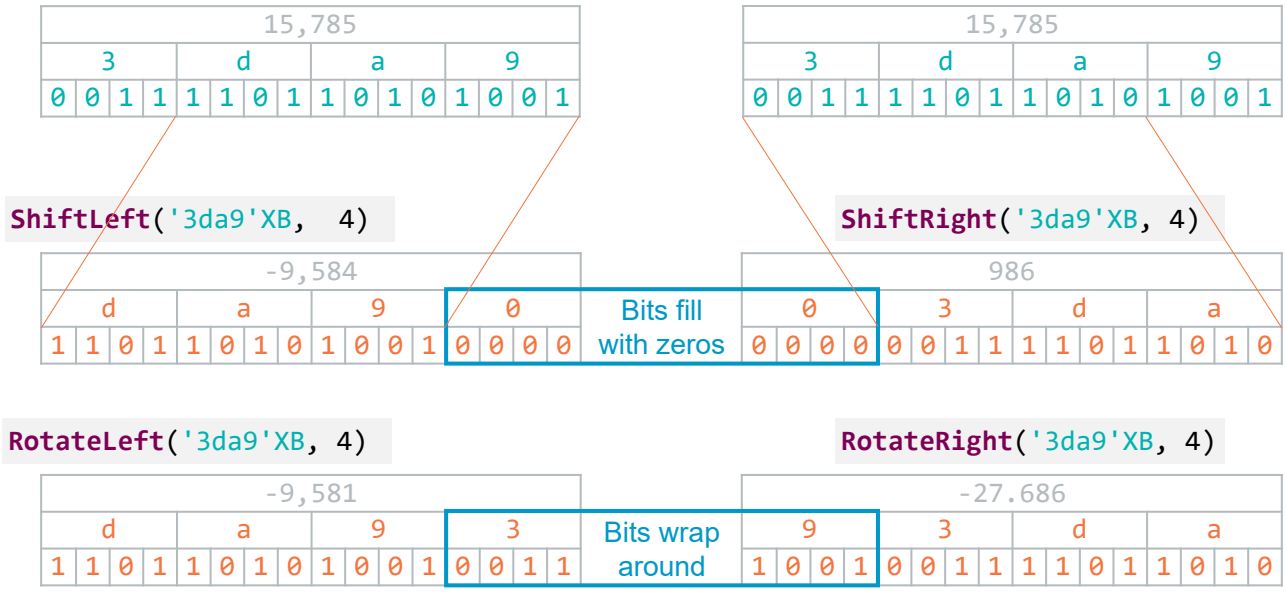
msb set to 1 indicates negative value

There's no function to reverse this calculation, but bit operations work on integers, too:

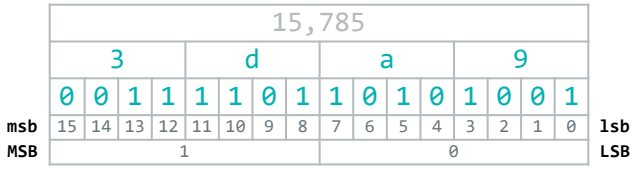
```
To_Byte(-1) AS b1                                 ff
BitOr(Cast(0 AS ByteInt), b1)                     -1

To_Byte(15785) AS b2                              3d-a9
BitOr(Cast(0 AS SmallInt), b2)                    15,785

To_Byte(16711935) AS b4                           00-ff-00-ff
BitOr(Cast(0 AS Int), b4)                         16,711,935

To_Byte(CAST(123456789012345678 AS BIGINT)) AS b8 01-b6-9b-4b-a6-30-f3-4e
BitOr(Cast(0 AS BigInt), b8)                      123,456,789,012,345,678
```

```
BYTEINT  ⇄ BYTE(1)
SMALLINT ⇄ BYTE(2)
INTEGER  ⇄ BYTE(4)
BIGINT   ⇄ BYTE(8)
```

Shift and Rotate



Other Bit Functions




```
GetBit('3da9'XB, 0)                 1
GetBit('3da9'XB, 2)                 0
SetBit('3da9'XB, 2)                 15,789 = 3d-ad = 0011110110101101
SetBit(15785, 0, 0)                 15,784 = 3d-a8 = 0011110110101000
CountSet('3da9'XB)                  9
CountSet('3da9'XB, 0)               7
SubBitStr('3da9'XB, 4, 4)           0a = 00001010
SubBitStr('3da9'XB, 0, 8)           a9 = 10101001
```

```
SubStr('3da9'XB, 2, 1)              a9
SubString('3da9'XB FROM 2 FOR 1)    a9
```

SubString function works on Bytes, too

```
BYTES('01b69b4ba630f34e'xb)         8
BYTES('003da900'xbv)                4
```

Bit functions number bits & bytes from
- Right to Left
- lsb to msb
- LSB to MSB

Bytes numbered from
- Left to Right

Similar to `Char_Length`
- Trailing zeros count

Summary

In this module, you learned about the binary world:

- Digital systems may represent analog values
- Increasing the precision in analog control systems becomes extremely costly
- Digital systems can reproduce values more precisely

Data Types and Functions: Binary Data

Labs

Lab Preparation

```
CREATE VOLATILE TABLE vt_ip
 (
   id INT NOT NULL PRIMARY KEY
  ,ip VARCHAR(18) CHARACTER SET Latin NOT NULL
  ,ip_b BYTE(4) NOT NULL
 )
ON COMMIT PRESERVE ROWS
;

INSERT INTO vt_ip VALUES(1, '127.0.0.1',           '7f000001'xb);
INSERT INTO vt_ip VALUES(2, '110.206.245.243',     '6ecef5f3'xb);
INSERT INTO vt_ip VALUES(3, '110.206.245.243/20', '6ecef5f3'xb);
INSERT INTO vt_ip VALUES(4, '192.168.1.100',       'c0a80164'xb);
```

Lab 1
Data Types and Functions: Binary

Use bit functions to convert the text IP-address to binary.

```
*** Query completed. 4 rows found. 4 columns returned.

ip                 ip2bin
-----------------  -----------
110.206.245.243/20  6e-ce-f5-f3
127.0.0.1           7f-00-00-01
192.168.1.100       c0-a8-01-64
110.206.245.243     6e-ce-f5-f3
```

Use bit functions to convert the binary IP-address to text.

```
*** Query completed. 4 rows found. 4 columns returned.

ip_b         bin2ip
-----------  ---------------
6e-ce-f5-f3  110.206.245.243
7f-00-00-01  127.0.0.1
c0-a8-01-64  192.168.1.100
6e-ce-f5-f3  110.206.245.243
```

Data Types and Functions: Binary Data

Lab Solutions

Lab 1 Solution
Data Types and Functions: Binary

teradata.

Use bit functions to convert the text IP-address to binary.

```
SELECT
   ip
  ,To_Byte( ShiftLeft(Cast(StrTok(ip, '.',  1) AS INT), 24)
          + ShiftLeft(Cast(StrTok(ip, '.',  2) AS INT), 16)
          + ShiftLeft(Cast(StrTok(ip, '.',  3) AS INT), 8)
          +           Cast(StrTok(ip, './', 4) AS INT)
          ) AS ip2bin
FROM vt_ip
;
```

Use bit functions to convert the binary IP-address to text.

```
SELECT
   ip_b
  ,Trim(ShiftRight(BitAnd('FF000000'xi, ip_b), 24) (FORMAT 'zz9.')) ||
   Trim(ShiftRight(BitAnd('00FF0000'xi, ip_b), 16) (FORMAT 'zz9.')) ||
   Trim(ShiftRight(BitAnd('0000FF00'xi, ip_b),  8) (FORMAT 'zz9.')) ||
   Trim(           BitAnd('000000FF'xi, ip_b)      (FORMAT 'zz9' )) AS bin2ip
FROM vt_ip
;
```

SQL UDFs for Handling IP4

SQL UDFs for handling IP4-addresses

```
-- Convert text IP-address to binary
REPLACE FUNCTION ip2byte(ip VARCHAR(15) CHARACTER SET Latin)
RETURNS BYTE(4)
LANGUAGE SQL
CONTAINS SQL
DETERMINISTIC
RETURNS NULL ON NULL INPUT
SQL SECURITY DEFINER
COLLATION INVOKER
INLINE TYPE 1
RETURN
   To_Byte( ShiftLeft(Cast(StrTok(ip, '.',  1) AS INT), 24)
          + ShiftLeft(Cast(StrTok(ip, '.',  2) AS INT), 16)
          + ShiftLeft(Cast(StrTok(ip, '.',  3) AS INT), 8)
          +           Cast(StrTok(ip, './', 4) AS INT)
          )
;
```

SQL UDFs for Handling IP4 (cont.)

SQL UDFs for handling IP4-addresses

```
-- Convert binary IP-address to text
REPLACE FUNCTION byte2ip(ip_b BYTE(4))
RETURNS VARCHAR(20) CHARACTER SET Latin
LANGUAGE SQL
CONTAINS SQL
DETERMINISTIC
RETURNS NULL ON NULL INPUT
SQL SECURITY DEFINER
COLLATION INVOKER
INLINE TYPE 1
RETURN
   Trim(ShiftRight(BitAnd('FF000000'xi, ip_b), 24) (FORMAT 'zz9.')) ||
   Trim(ShiftRight(BitAnd('00FF0000'xi, ip_b), 16) (FORMAT 'zz9.')) ||
   Trim(ShiftRight(BitAnd('0000FF00'xi, ip_b),  8) (FORMAT 'zz9.')) ||
-- fails when used in UDF: Optimizer bug?
-- Trim(           BitAnd('000000FF'xi, ip_b)     (Format 'zz9'))
   Trim(      BitAnd(Cast('000000FF'xi AS INT), ip_b)  (FORMAT 'zz9'))
;
```

SQL UDFs for Handling IP4 (cont.)

Additional calculations for
- Netmask
- Network Address
- Broadcast Address

```
SELECT
   ip                                                                                  110.206.245.243/20
  ,ip_b                                                                                6e-ce-f5-f3
  ,Cast(StrTok(ip, './', 5) AS INT) AS netmask_bits                                    20
  ,Cast(ShiftLeft( 'ffffffff'xb, 32 - netmask_bits ) AS BYTE(4)) AS netmask            ff-ff-f0-00
  ,Cast(ShiftRight('ffffffff'xb,      netmask_bits ) AS BYTE(4)) AS netmask_inv        00-00-0f-ff
  ,Cast(BitNot(netmask) AS BYTE(4))                                 AS netmask_inv2    00-00-0f-ff
  ,Cast(BitAnd(ip_b, netmask)     AS BYTE(4)) AS network_address                       6e-ce-f0-00
  ,Cast( BitOr(ip_b, netmask_inv) AS BYTE(4)) AS broadcast_address                     6e-ce-ff-ff
FROM vt_ip
WHERE netmask_bits IS NOT NULL
;
```

Appendix A: Review Question Solutions

This Appendix contains answers to the review questions for the course modules.

Module 1: Review Questions

1. Which two are true about PARTITION BY?
   a. It cannot reference an aggregate function.
   b. It can reference a column by name or alias.
   c. It must reference a projected column.
   d. PARTITION BY and GROUP BY may both be present within the same projection.
2. There is an EMPLOYEE table with columns name, dept and salary.
   Which SQL statement produces a report that shows only employees having a salary greater than their departmental average salary?

```
SELECT name, dept, salary
FROM EMPLOYEE AS e
```

a.
```
GROUP BY dept
HAVING  salary > AVG(salary);
```

b.
```
WHERE   salary > AVG(salary);
```

c.
```
QUALIFY salary > AVG(salary) OVER (PARTITION BY dept);
```

d.
```
HAVING  salary > AVG(salary) OVER (PARTITION BY dept);
```

Module 1: Review Questions (cont.)

3. What is the correct order of query clauses?
   a. SELECT-FROM-WHERE-QUALIFY-GROUP BY-HAVING-SAMPLE-ORDER BY
   b. SELECT-FROM-WHERE-GROUP BY-HAVING-SAMPLE-QUALIFY-ORDER BY
   c. SELECT-FROM-WHERE-GROUP BY-HAVING-QUALIFY-SAMPLE-ORDER BY
   d. SELECT-FROM-WHERE-SAMPLE-GROUP BY-HAVING-QUALIFY-ORDER BY

4. What is the logical query processing order?
   a. SELECT-FROM-WHERE-GROUP BY-HAVING-QUALIFY-SAMPLE-ORDER BY
   b. FROM-WHERE-GROUP BY-HAVING-QUALIFY-SELECT-SAMPLE-ORDER BY
   c. FROM-WHERE-GROUP BY-HAVING-SAMPLE-QUALIFY-SELECT-ORDER BY
   d. SELECT-FROM-WHERE-SAMPLE-GROUP BY-HAVING-QUALIFY-ORDER BY

5. Which of the following is invalid syntax?
   a. `SUM(col) OVER (PARTITION BY 1))`
   b. `SUM(col) OVER (PARTITION BY col2 / 10))`
   c. `SUM(SUM(col)) OVER ()`
   d. `SUM(SUM(col) OVER ())`

Module 11: Review Questions

1. Which two are true about macros?
   a. Macros are Teradata's version of stored procedures
   b. Changes to macros are done by using ALTER MACRO
   c. Macros are an object which requires perm space from user DBC
   d. You can perform a HELP MACRO to look at parameter information on a macro

2. Which two are true about parameterized macros?
   a. They have two forms of execution
   b. You can use them to pass object names for creating objects
   c. In order to execute a parameterized macro, you must know the parameter names
   d. Parameters may be defined, but not referenced by the macro's statements

3. Which two are true about macros and DDL?
   a. Macros may be used to create objects
   b. Macros having multiple SQL statements may not contain DDL
   c. Macros containing DDL may not be parameterized
   d. Object referenced by macros may not be dropped

Module 11: Review Questions

4. Which two are true macros and privileges?
   a. If you can execute and macro, then you can REPLACE it
   b. You need REPLACE MACRO privileges to change a macro definition
   c. CREATE MACRO privileges are needed to create a macro
   d. Only EXECUTE privileges are needed to execute a macro
5. Which two are true about macro maintenance?
   a. You may replace a macro using CREATE
   b. You may create a macro using REPLACE
   c. You may update a macro using ALTER
   d. It is possible to create a macro in another database
6. Which two are true about macros?
   a. They are objects that have a “kind” value of “M”
   b. They may be located inside an “explorer tree” of a standard Teradata SQL tool
   c. Need not contain a semicolon within the body of it
   d. Are not typically used for repetitive access to objects