



Teradata Vantage MasterClass

Version 17.20.0

95044
Student Guide

Trademarks

The product or products described in this book are licensed products of Teradata Corporation or its affiliates.

Teradata, Applications-Within, Aster, BYNET, Claraview, DecisionCast, Gridscale, MyCommerce, QueryGrid, SQL-MapReduce, Teradata Decision Experts, "Teradata Labs" logo, Teradata ServiceConnect, Teradata Source Experts, WebAnalyst, and Xkoto are trademarks or registered trademarks of Teradata Corporation or its affiliates in the United States and other countries.

Adaptec and SCSISelect are trademarks or registered trademarks of Adaptec, Inc.

Amazon Web Services, AWS, [any other AWS Marks used in such materials] are trademarks of Amazon.com, Inc. or its affiliates in the United States and/or other countries.

AMD Opteron and Opteron are trademarks of Advanced Micro Devices, Inc.

Apache, Apache Avro, Apache Hadoop, Apache Hive, Hadoop, and the yellow elephant logo are either registered trademarks or trademarks of the Apache Software Foundation in the United States and/or other countries. Apple, Mac, and OS X all are registered trademarks of Apple Inc.

Axeda is a registered trademark of Axeda Corporation.

Axeda Agents, Axeda Applications, Axeda Policy Manager, Axeda Enterprise, Axeda Access, Axeda Software Management, Axeda Service, Axeda ServiceLink, and Firewall-Friendly are trademarks and Maximum Results and Maximum Support are servicemarks of Axeda Corporation.

CENTOS is a trademark of Red Hat, Inc., registered in the U.S. and other countries.

Cloudera, CDH, [any other Cloudera Marks used in such materials] are trademarks or registered trademarks of Cloudera Inc. in the United States, and in jurisdictions throughout the world.

Data Domain, EMC, PowerPath, SRDF, and Symmetrix are registered trademarks of EMC Corporation.

GoldenGate is a trademark of Oracle.

Hewlett-Packard and HP are registered trademarks of Hewlett-Packard Company.

Hortonworks, the Hortonworks logo and other Hortonworks trademarks are trademarks of Hortonworks Inc. in the United States and other countries.

Intel, Pentium, and XEON are registered trademarks of Intel Corporation.

IBM, CICS, RACF, Tivoli, and z/OS are registered trademarks of International Business Machines Corporation.

Linux is a registered trademark of Linus Torvalds.

LSI is a registered trademark of LSI Corporation.

Microsoft, Active Directory, Windows, Windows NT, and Windows Server are registered trademarks of Microsoft Corporation in the United States and other countries.

NetVault is a trademark or registered trademark of Dell Inc. in the United States and/or other countries.

Novell and SUSE are registered trademarks of Novell, Inc., in the United States and other countries.

Oracle, Java, and Solaris are registered trademarks of Oracle and/or its affiliates.

QLogic and SANbox are trademarks or registered trademarks of QLogic Corporation.

Quantum and the Quantum logo are trademarks of Quantum Corporation, registered in the U.S.A. and other countries.

Red Hat is a trademark of Red Hat, Inc., registered in the U.S. and other countries. Used under license.

SAP is the trademark or registered trademark of SAP AG in Germany and in several other countries.

SAS and SAS/C are trademarks or registered trademarks of SAS Institute Inc.

SPARC is a registered trademark of SPARC International, Inc.

Symantec, NetBackup, and VERITAS are trademarks or registered trademarks of Symantec Corporation or its affiliates in the United States and other countries.

Unicode is a registered trademark of Unicode, Inc. in the United States and other countries.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other product and company names mentioned herein may be the trademarks of their respective owners.

The information contained in this document is provided on an "as-is" basis, without warranty of any kind, either express or implied, including the implied warranties of merchantability, fitness for a particular purpose, or non-infringement. Some jurisdictions do not allow the exclusion of implied warranties, so the above exclusion may not apply to you. In no event will Teradata Corporation be liable for any indirect, direct, special, incidental, or consequential damages, including lost profits or lost savings, even if expressly advised of the possibility of such damages.

The information contained in this document may contain references or cross-references to features, functions, products, or services that are not announced or available in your country. Such references do not imply that Teradata Corporation intends to announce such features, functions, products, or services in your country. Please consult your local Teradata Corporation representative for those features, functions, products, or services available in your country.

Information contained in this document may contain technical inaccuracies or typographical errors. Information may be changed or updated without notice. Teradata Corporation may also make improvements or changes in the products or services described in this information at any time without notice.

Teradata Vantage MasterClass

Version 17.20.0

Module 1 – Analyze Primary Index

Objectives	1-2
How Goes Data Get Stored?	1-3
Why This Approach Isn't Effective?.....	1-4
Parallelism – The Solution.....	1-5
Vantage – Parallelism and How We Do It!	1-6
Primary Index – Distribution of Rows.....	1-7
Row Distribution Using a Primary Index – Case 1.....	1-8
Row Distribution Using a Primary Index – Case 2.....	1-9
Row Distribution Using a Highly Non-Unique Primary Index (NUPI) – Case 3	1-10
Primary Index Types.....	1-11
What if We Don't Specify the Primary Index?	1-12
What Makes a Good Primary Index?.....	1-13
Viewing a Tables Distribution.....	1-14
Tables in Vantage	1-15
NoPI Table – Why?	1-16
Using SQL to Predict Row Distribution	1-17
Creating a Table Without a PI	1-18
Loading Data into a NoPI Table (SQL).....	1-19
Multiple NoPI Tables at the AMP Level	1-20
NoPI Table Options	1-21
Summary	1-22

Module 2 – Analyze Secondary Index

Objectives	2-2
Why Secondary Indexes	2-3
Secondary Indexes	2-4
Defining Secondary Indexes	2-5
Secondary Index Subtables	2-6
USI Subtable General Row Layout.....	2-7
USI Hash Mapping	2-8
NUSI Subtable General Row Layout.....	2-9
NUSI Hash Mapping.....	2-10
Secondary Index Usage.....	2-11
Secondary Index Properties	2-12
When Creating a Secondary Index	2-13

NUSI vs. Full Table Scan (FTS).....	2-14
Secondary Index Overhead.....	2-15
Secondary Index Considerations.....	2-16
Row Selection.....	2-17
NUSI Bit Mapping.....	2-18
Single and Dual NUSI Access.....	2-19
Value-Ordered NUSIs.....	2-20
Hash-Ordered NUSIs.....	2-22
Covering Indexes.....	2-24
Full Table Scans – Sync Scans.....	2-25
Summary.....	2-26
Module 2: Analyze Secondary Index Bring Up JupyterHub.....	2-27

Module 3 – Row Partitioning

Objectives.....	3-2
Why Row Partitioning?.....	3-3
Why Partition a Table?.....	3-4
What is Row Partitioning?.....	3-5
How is Partitioning Implemented?.....	3-6
Primary Index Access – Row Partitioned Table.....	3-7
Access Using Partitioned Data.....	3-8
Access Using Primary Index.....	3-9
Place a USI on NUPI.....	3-10
Place a NUSI on NUPI.....	3-11
Implementing PPI – PARTITION BY Option.....	3-12
Partitioning with CASE_N and RANGE_N.....	3-13
Partitioning with RANGE_N – Example.....	3-14
RANGE_N – Example with Varying Intervals.....	3-15
Special Partitions with CASE_N and RANGE_N.....	3-16
Partitioning with CASE_N – Example.....	3-17
SQL Use of PARTITION Key Word.....	3-18
ALTER TABLE – Row Partitioned Tables.....	3-19
ALTER TABLE – NO RANGE is Defined.....	3-20
ALTER TABLE – NO RANGE is Not Defined.....	3-21
ALTER TABLE.....	3-22
Multilevel Partitioning.....	3-23
Multilevel Partitioning – Example.....	3-24
ADD Partitions Option – Example.....	3-25
Character Partitioning.....	3-26
Summary.....	3-27
Module 3: Partitioned Primary Index Bring Up JupyterHub.....	3-28

Module 4 – Join Processing

Objectives	4-2
Revision – Join Syntax.....	4-3
Teradata – How Joins Work	4-4
Join Process – How the Optimizer Minimizes Spool Usage	4-5
Join Process – How the Optimizer Minimizes Row Selection	4-6
Join Technique 1 – Row Redistribution – Matching Indexes.....	4-7
Join Technique 1 – Row Redistribution – Non-Matching Indexes.....	4-8
Join Technique 2 – Duplicating a Table in Spool.....	4-9
Join – Duplicate the Smaller Table.....	4-10
General Join Distribution Strategies	4-11
Merge Join	4-12
Merge Join Strategy	4-13
Hash Join.....	4-14
Nested Joins	4-15
Product Join	4-16
Cartesian Product.....	4-17
Exclusion Joins	4-18
Exclusion Join Example.....	4-19
Inclusion Joins	4-20
n-Table Joins	4-21
Join Considerations with Partitioned Tables.....	4-22
Additional Join Options with Partitioned Tables.....	4-23
Non-partitioned to Partitioned Table Join (Few Partitions).....	4-24
Non-partitioned to Partitioned Table Join (Many Partitions)	4-25
Partitioned to Partitioned Table Join (Rowkey Match Scan).....	4-26
Join Processing Highlights.....	4-27
Summary	4-28
Module 4: Joins Bring Up JupyterHub	4-29

Module 5 – Optimizer and Statistics Collection

Objectives	5-2
Why Collect Statistics?	5-3
Who Collects Statistics?	5-4
Optimizer – Statistics.....	5-5
Teradata Optimizer	5-6
Optimizer – Random AMP Samples.....	5-7
Optimizer – Heuristics	5-8
Optimizer's Search for Statistics	5-9
Example of an Optimizer Estimate without Collected Statistics	5-10
Statistics Collection – The Impact	5-11
Statistics Data – What is Collected?	5-12
Statistics Collection – Table Level	5-14
Table Level Summary Statistics	5-15
COLLECT STATISTICS Command.....	5-17

COLLECT STATISTICS Command (Index Format)	5-18
Collecting Statistics Example	5-19
Refresh or Re-Collect Statistics	5-20
Viewing Statistics	5-21
Collect Statistics Using Sample	5-22
Collecting Statistics (Additional Options)	5-23
Collecting Statistics on PARTITION	5-24
Copying STATISTICS.....	5-25
Teradata AutoStats Features	5-26
Statistics Highlight.....	5-27
Summary	5-28
Module 5: Optimizer and Statistics Collection Bring Up JupyterHub	5-29

Module 6 – Additional Index Options

Objectives	6-2
Why Join Index	6-3
Additional Index Choices	6-4
Join Indexes	6-5
Join Index Considerations.....	6-8
Compressed and Non-Compressed Join Indexes.....	6-10
Compressed Multi-Table Join Index.....	6-11
Non-Compressed Multi-Table Join Index	6-12
Example 1 – Does a Join Index Help?	6-13
Example 2 – Does a Join Index Help?	6-14
Example 3 – Partitioning a Join Index.....	6-15
Join Index – Single Table	6-16
Creating a Join Index – Single Table.....	6-18
Example 4 – Does Single Table Join Index Help?	6-19
Why Use Aggregate Join Indexes?	6-20
Aggregate Join Index Properties	6-21
Aggregation Without an Aggregate Index.....	6-22
Creating an Aggregate Join Index.....	6-23
Example 5: Aggregation With an AJI.....	6-24
Sparse Join Indexes.....	6-25
Creating a Sparse Join Index.....	6-26
Global (Join) Indexes.....	6-27
Global Index – Multiple Tables	6-28
Highlights.....	6-29
Summary	6-30
Module 6: Additional Indexes Bring Up JupyterHub	6-31

Module 7 – Compression

Objectives	7-2
What is Compression!	7-3
Compression in Vantage	7-4
MVC Considerations	7-5
Example 1 – MVC List of Values	7-6
Example 2 – MVC Alter Table	7-7
Algorithmic Compression Considerations (ALC)	7-8
Example 1 – ALC	7-9
Block Level Compression Considerations (BLC)	7-10
Example – BLC	7-11
Teradata Compression Comparison	7-12
Key Highlights	7-13
Summary	7-14
Module 7: Compression Bring Up JupyterHub	7-15

Module 8 – 4D Analytics

Objectives	8-2
Customer 360 View	8-3
A Business is Multi-Dimensional	8-4
Vantage – 4D Analytics	8-5
Importance of Geospatial Analytics	8-6
Use Cases of Geospatial Analytics	8-7
Questions Spatial Analysis Can Answer	8-8
Geospatial Data – The Library	8-9
Geospatial – Where are the Profitable Customers Located?	8-10
Geospatial Example	8-11
Teradata Temporal	8-12
Temporal Time	8-13
Bi-Temporal Table DDL (ANSI Temporal)	8-14
Temporal Data and Analysis	8-15
Temporal Example	8-16
Characteristics of Time Series Data	8-17
What Constitutes Time Series Data?	8-18
What was Measured and When It was Logged	8-19
Teradata Time Aware Capabilities	8-20
Time Aware Functions	8-21
Time-Aware Aggregate Functions – GROUP BY TIME	8-22
Time Aware Example – How Does it Look?	8-23
Time Aware Example – How Does it Work?	8-24
Time Aware – Fill Clause	8-25
Primary Time Index	8-26
Time-series Data – Time Bucket	8-27
Key Aspects of PTI	8-28
Distribution and Ordering Strategies	8-29

PTI Table Definition – How It Looks	8-30
GROUP BY TIME Rules and Restrictions	8-31
4D Analytics in Action	8-32
Summary	8-33

Module 9 – Teradata Columnar

Objectives	9-2
Why Columnar?	9-3
Row vs. Columnar	9-4
Teradata Columnar.....	9-5
No Primary Index Table DDL	9-6
No Primary Index Table.....	9-7
Column Partition Table DDL (without Auto-Compression)	9-8
Column Partition Container (No Automatic Compression).....	9-9
CP Table Query #1 (without Auto-Compression)	9-12
Column Partitioning – 2nd Example.....	9-14
Column Partition Table DDL (with Auto-Compression)	9-15
Auto-Compression for CP Tables	9-16
Auto-Compression Techniques for CP Tables	9-17
User-Defined Compression Techniques	9-19
Column Partition Container (Automatic Compression)	9-20
Column Partition Table (with Auto-Compression)	9-21
CP Table Query #2 (with Auto-Compression)	9-22
CP Table with Row Partitioning DDL.....	9-23
Column Partition Table (with Row Partitioning)	9-24
CP Table with Multi-Column Container DDL	9-25
CP Table with Multi-Column Container.....	9-26
CP Table Hybrid Row and Column Store DDL	9-27
CP Table (with Hybrid Row and Column Store).....	9-28
Populating a CP Table	9-29
DELETE Considerations	9-30
UPDATE and USI/NUSI Considerations	9-31
CP Table Restrictions	9-32
Key Highlights	9-33
Summary	9-34
Module 9: Columnar Bring Up JupyterHub	9-35

Module 10 – Native Object Store

Objectives	10-2
Why Object Storage?	10-3
What is Object Storage?.....	10-5
Pros and Cons of Object Stores	10-6
What is Native Object Store?.....	10-7
Native Object Store Capabilities.....	10-8
How NOS is Modernizing Data Management	10-9
NOS Feature Description.....	10-10
Vantage NOS On-Premises.....	10-11
Exploring External Data in Place – Use Case 1	10-12
Simplified Data Load – Use Case 2	10-13
Performance Considerations – Use Case 3	10-14
READ_NOS Table Operator Overview.....	10-15
READ_NOS Functions	10-16
CREATE FOREIGN TABLE – JSON	10-17
Bad Data in Numeric Fields.....	10-19
Data Dictionary – Foreign Table Views	10-20
WRITE_NOS Table Operator Overview.....	10-21
WRITE_NOS High Level Overview	10-23
WRITE_NOS Table Operator.....	10-24
Summary	10-25
Module 10: Native Object Store Bring Up JupyterHub.....	10-26

Appendix A: Lab Solutions



Module 1: Analyze Primary Index

Teradata Vantage MasterClass

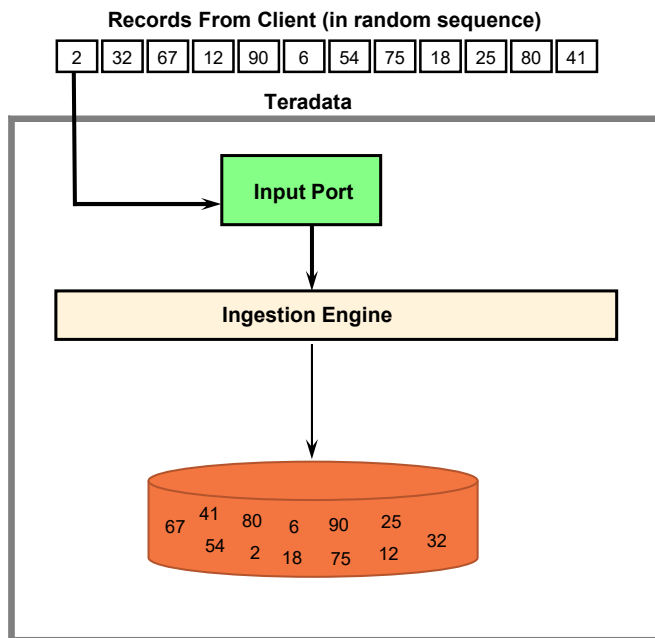
Copyright © 2007–2023 by Teradata. All Rights Reserved.

Objectives

After completing this module, you will be able to:

- Compare Non-Parallel vs Parallel Systems
- Describe the need for a Primary Index
- Describe how a Primary Index distributes data
- List types of Primary Indexes
- List types of Teradata Tables
- Describe the need for a No Primary Index (NoPI) table
- Describe how data is loaded into a NoPI table
- List NoPI Limitations

How Goes Data Get Stored?

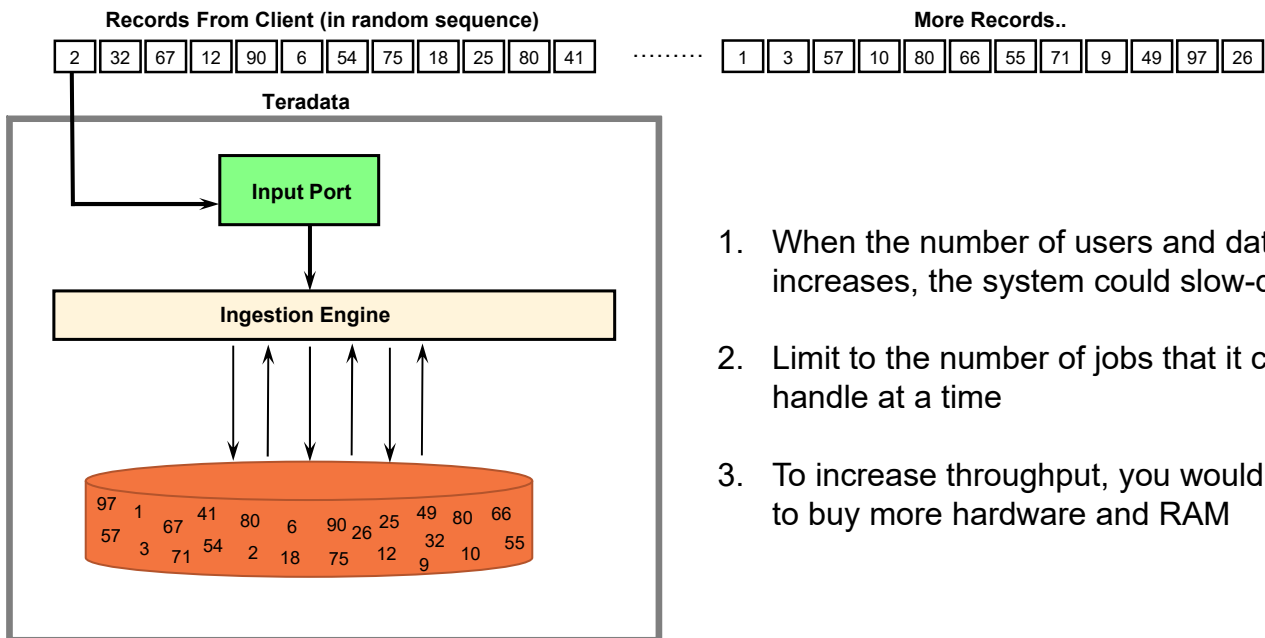


1. Traditional database engines used to insert data into the disks directly depending upon where space was available on the disk
2. Access was not parallel but sequential resulting in ineffective data access paths and longer wait times
3. Although this approach worked for OLTP systems, it was ineffective for Decision Support Systems (DSS)

Traditional database engines such as Oracle and MS SQL did not have parallelism built in. They relied on more infra when workloads increased.

However, throwing more infrastructure at a problem was never going to solve it, as the I/O incurred was still the same this was just kicking the can down the road.

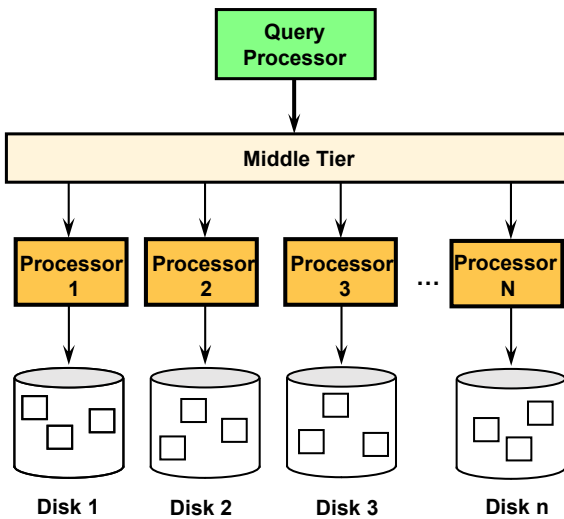
Why This Approach Isn't Effective?



1. When the number of users and data increases, the system could slow-down
2. Limit to the number of jobs that it can handle at a time
3. To increase throughput, you would need to buy more hardware and RAM

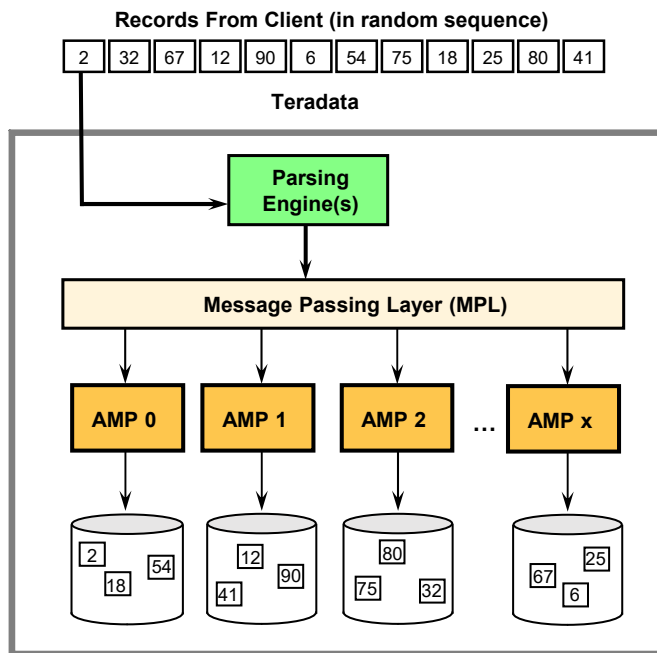
As the load increases the capacity for non-parallel databases is clearly hampered. Since workloads are not parallelized, access to data is constrained and this introduces a drag on the system.

Parallelism – The Solution



1. Each processor has its own memory and disk storage
2. Communication happens via messaging between layers
3. Easy to scale this architecture
4. Isolation is easily implemented for each transaction

Vantage – Parallelism and How We Do It!



- The **Parsing Engine** dispatches requests to insert a row
- The **Message Passing Layer** ensures that a row gets to the appropriate Access Module Processor (AMP)
- The **AMP** stores the row on its associated (logical) disk
- An AMP manages a logical or **virtual disk** that is mapped to multiple physical disks in a disk array

PEs and AMPs are actually implemented as virtual processors (vprocs) in the system. A vproc is effectively a group of processes that represents a Teradata software component.

The **Parsing Engine** (PE) interprets the SQL command and converts the data record from the host into an AMP message. The PE is the software component that interprets SQL requests, receives input records, and passes data. To do that it sends the messages through the Message Passing Layer to the AMPs.

The **Message Passing Layer** (MPL) distributes the row to the appropriate Access Module Processor (AMP). The MPL is implemented as hardware and/or software, depending on the platform used. It determines which vprocs should receive a message.

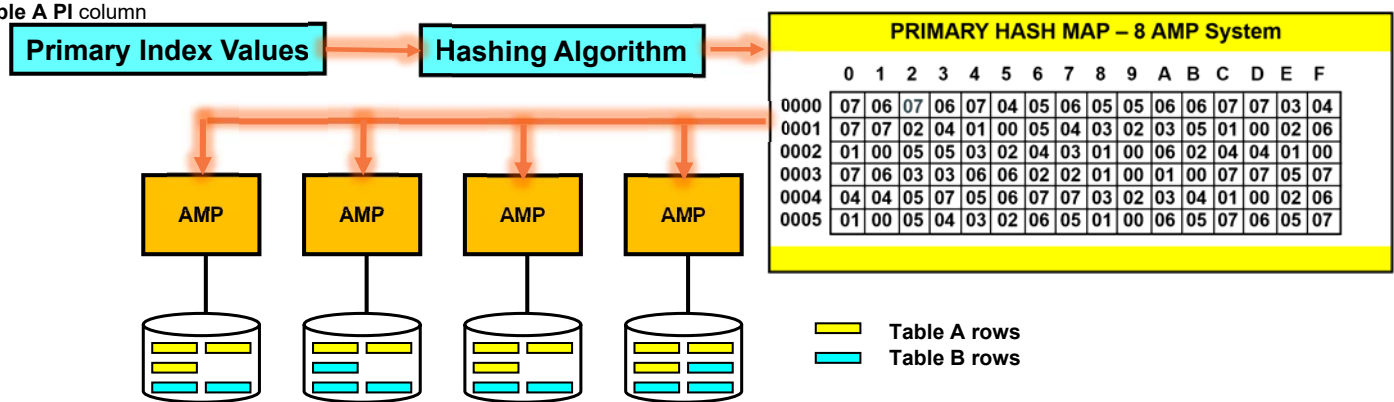
The **AMP** formats the row and writes it to its associated disks (Vdisks) which are assigned to physical disks in a disk array. The physical disk holds the row for subsequent access.

The **Client system** supplies the records. These records are the raw data from which the database will be constructed.

Think of the **AMP (Access Module Processor)** as an independent computer designed for and dedicated to managing a portion of the entire database. It performs all the database management functions – such as sorting, aggregating, and formatting the data. It receives data from the PE, formats the rows, and distributes the rows to the disk storage units it controls. It also retrieves the rows requested by the parsing engine.

Primary Index – Distribution of Rows

Table A PI column



- The rows of **every** table are distributed among all AMPs
- Each AMP is responsible for a **subset** of the rows of each table
 - Ideally, each table will be evenly distributed among all AMPs
 - Evenly distributed tables result in evenly distributed workloads

Ideally, the rows of every table will be distributed among all of the AMPs. There may be some circumstances where this is not true. What if there are fewer rows than AMPs? Clearly, in this case, at least some AMPs will hold no rows from that table. This should be considered an exceptional situation, and not the rule. Each AMP is designed to hold a portion of the rows of each table. The AMP is responsible for the storage, maintenance, and retrieval of the data under its control.

More ideally, the rows of each table will be evenly distributed across all of the AMPs. This is desirable because in operations involving all rows of the table (such as a full table scan); each AMP will have an equal portion of the work to do. When workload times are not evenly distributed, the desired response will only be as fast as the slowest AMP.

Controlling the distribution of the rows of a table is done by the selection of the Primary Index. The relative uniqueness of the Primary Index will determine the uniformity of distribution of the rows of this table among the AMPs.

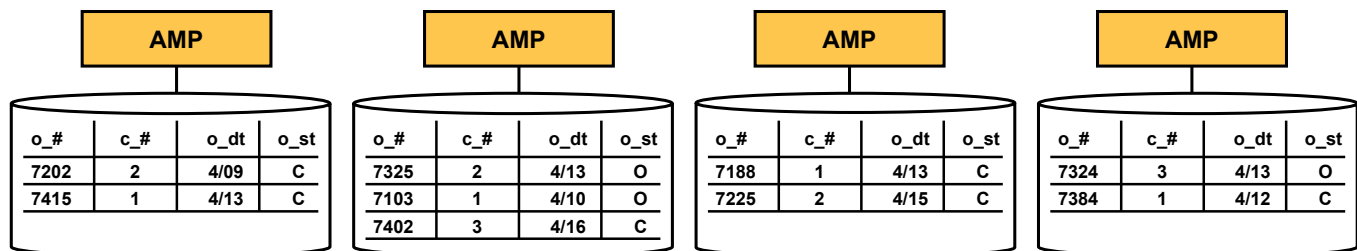
Row Distribution Using a Primary Index – Case 1

Orders

Order Number	Customer Number	Order Date	Order Status
PK			
UPI			
7325	2	4/13	O
7324	3	4/13	O
7415	1	4/13	C
7103	1	4/10	O
7225	2	4/15	C
7384	1	4/12	C
7402	3	4/16	C
7188	1	4/13	C
7202	2	4/09	C

Notes:

- Often, but not always, the PK column(s) will be used as a UPI
 - Order_Number can be a Unique Primary Index (UPI) since all the values are unique
- Teradata will distribute different UPI values evenly across all AMPs
 - Resulting row distribution among AMPs is very uniform
 - Assures maximum efficiency for parallel operations



At the heart of the Vantage Advanced SQL Engine is a way of predictably distributing and retrieving rows across AMPs. The same value stored in the same data type will always produce the same hash value. If the Primary Index is unique, Teradata can distribute the rows evenly. If the Primary Index is slightly non-unique, that is, there are only four or five rows per index value; the table will still distribute evenly. But if there are hundreds or thousands of rows for some index values the distribution will probably be lumpy.

In this example, the Order_Number is used as a unique primary index. Since the primary index value for Order_Number is unique, the distribution of rows among AMPs is very uniform. This assures maximum efficiency because each AMP is doing approximately the same amount of work. No AMPs sit idle waiting for another AMP to finish a task.

This way of storing the data provides for maximum efficiency and makes the best use of the parallel features of the Teradata system.

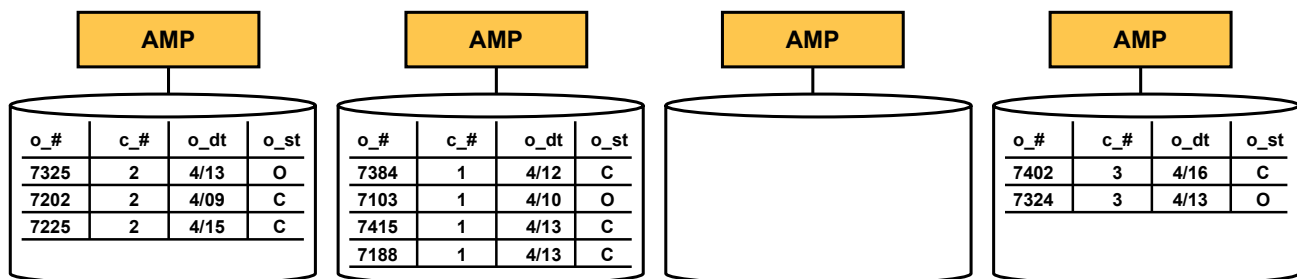
Row Distribution Using a Primary Index – Case 2

Orders

Order Number	Customer Number	Order Date	Order Status
PK			
	NUPI		
7325	2	4/13	O
7324	3	4/13	O
7415	1	4/13	C
7103	1	4/10	O
7225	2	4/15	C
7384	1	4/12	C
7402	3	4/16	C
7188	1	4/13	C
7202	2	4/09	C

Notes:

- Customer_Number may be the preferred access column for this table, thus a good index candidate
 - Since a customer can have multiple orders, **Customer_Number will be a Non-Unique Primary Index (NUPI)**
- Rows with the same PI value distribute to the same AMP
 - Row distribution is less uniform or skewed.



In the example on this slide, **Customer_Number** has been used as a non-unique Primary Index (NUPI). Note that row distribution among AMPs is uneven. All rows with the same primary index value (in other words, with the same customer number) are stored on the same AMP.

Customer_Number has three possible values, so all the rows are hashed to three AMPs, leaving the fourth AMP without rows from this table. While this distribution will work, it is not as efficient as spreading all the rows among all the AMPs.

AMP 2 has a disproportionate number of rows and AMP 3 has none. In an all-AMP operation, AMP 2 will take longer than the other AMPs. The operation cannot complete until AMP 2 completes its tasks. The overall operation time is increased and some of the AMPs are under-utilized.

NUPIs can create irregular distributions, called “skewed distributions”. AMPs that have more than an average number of rows will take longer for full table operations than the other AMPs will. Because an operation is not complete until all AMPs have finished, this will cause the operation to finish less quickly due to being underutilized.

Row Distribution Using a Highly Non-Unique Primary Index (NUPI) – Case 3

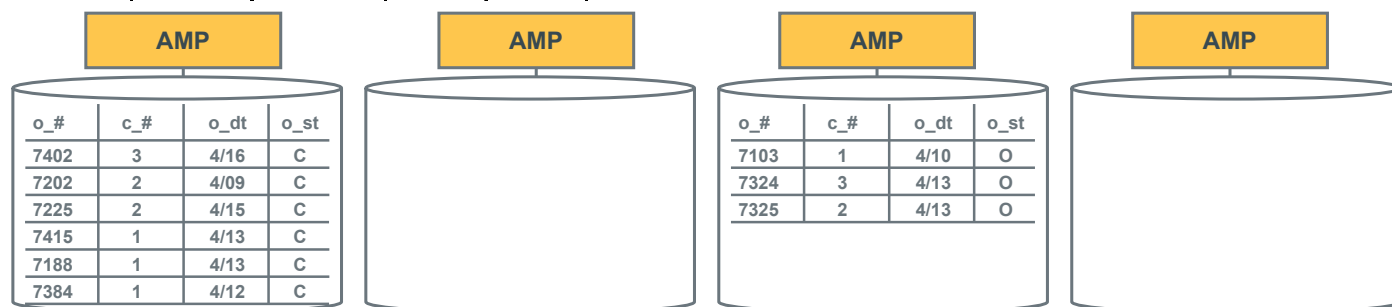
teradata.

Orders

Order Number	Customer Number	Order Date	Order Status
PK			
			NUPI
7325	2	4/13	O
7324	3	4/13	O
7415	1	4/13	C
7103	1	4/10	O
7225	2	4/15	C
7384	1	4/12	C
7402	3	4/16	C
7188	1	4/13	C
7202	2	4/09	C

Notes:

- Values for Order_Status are “highly” non-unique
 - Order_Status would be a NUPI
 - If only two values exist, then only two AMPs will be used for this table
- Highly non-unique columns are generally poor PI choices
 - The degree of uniqueness is critical to efficiency



This example uses **Order_Status** as a **NUPI**. Order_Status is a poor choice, because it yields the most uneven distribution. Because there are only two possible values for Order_Status, all of the rows are placed on two AMPs.

"Order Status is an example of a **highly non-unique Primary Index**.

When choosing a Primary Index, you should never choose a column with such a severely limited value set. The degree of uniqueness is critical to efficiency. Choose NUPI's that allow all AMPs to participate fairly equally.

The degree of uniqueness of a NUPI is critical to efficiency.

Primary Index Types

No	Common Traits – UPI & NUPI
1	Only 1 Index per table (specified during table creation)
2	Index cannot be modified
3	Always 1 AMP access
4	64 column limit

No	Unique Primary Index (UPI)	Non-Unique Primary Index (NUPI)
1	Only 1 Table row	Multiple Table Rows
2	No Spool file used	Spool file may be used
3	Only 1 NULL allowed	Multiple Nulls allowed
4	No Duplicates	Duplicates Reside on same Amp

Each table has one and only one Primary Index. A Primary Index may be different than a Primary Key.

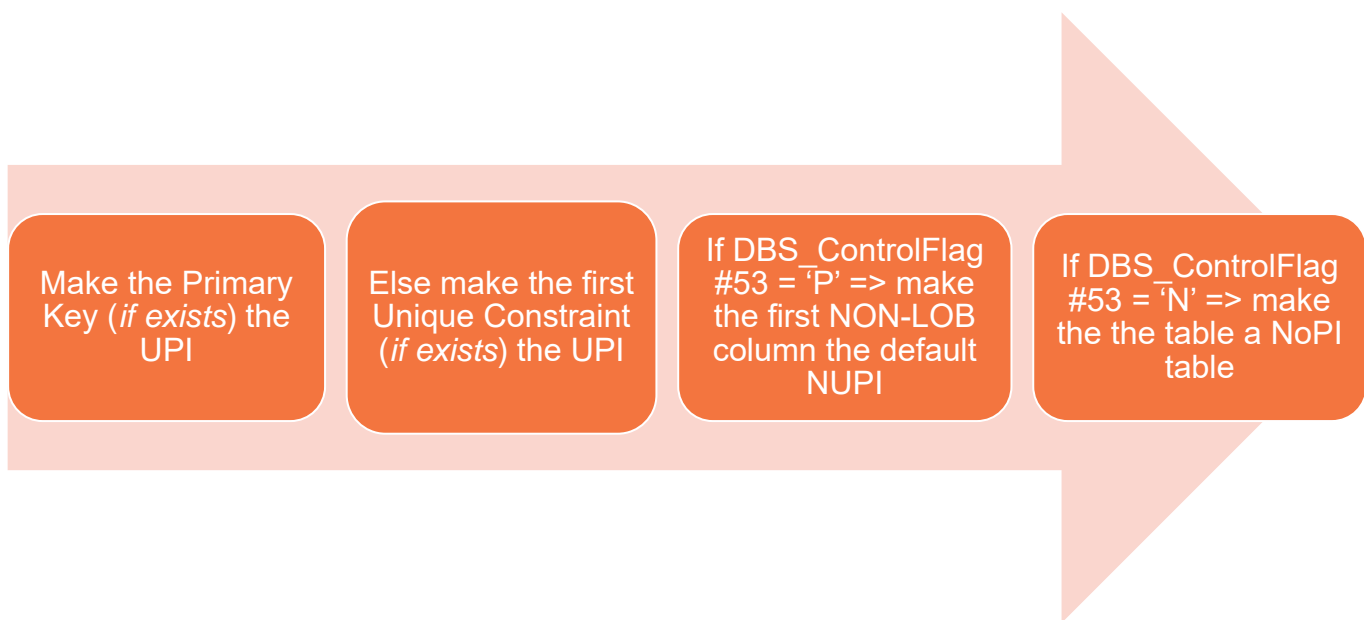
UPI = Best Performance, Best Distribution

- UPIs offer the best performance possible for several reasons. They are:
- A Unique Primary Index involves a single base table row at most
- No Spool file is ever required
- Single value access via the Primary Index is a one-AMP operation and uses only one I/O

NUPI = Good Performance, Good Distribution

- NUPI performance differs from UPI performance because:
- Non-Unique Primary Indexes may involve multiple table rows.
- Duplicate values go to the same AMP and the same data block, if possible.
- Multiple I/Os are required if the rows do not fit in a single data block.
- Spool files are used when necessary.
- A duplicate row check is required on INSERT and UPDATE for a SET table.

What if We Don't Specify the Primary Index?



This chart assumes the system default is to create tables with a Primary Index. The index implementation schedule is as follows:

Is a PI specified? Yes

Any PK specified or UNIQUE constraints PK = USI, UNIQUE = USI

Is a PI specified? No

PK specified? PK = UPI; UNIQUE = USI

PK not specified

1st UNIQUE column level constraint = UPI; other UNIQUE constraints = USI(s)

Neither specified?

1st non-LOB column = NUPI

What Makes a Good Primary Index?

ACCESS	Maximize one-AMP operations: Choose the column(s) most frequently used for access
DISTRIBUTION	Optimize parallel processing: Choose the column(s) that provides good distribution
VOLATILITY	Reduce maintenance resource overhead (I/O): Choose the column(s) with stable data values

Note: Data distribution has to be balanced with access usage in choosing a PI.

General Notes:

- A good logical model identifies the Primary Key for each table or entity
- Do not assume that the Primary Key will become the Primary Index
- The Primary Index is used by Teradata to access data only when all its columns are provided in SQL statements

There are three Primary Index Choice Criteria: **Access Demographics**, **Distribution Demographics**, and **Volatility**.

- Access demographics are the first of three Primary Index Choice Criteria. Access columns are those that would appear (with a value) in a WHERE clause in an SQL statement. Choose the column most frequently used for access to maximize the number of one-AMP operations.
- Distribution demographics are the second of the Primary Index Choice Criteria. The more unique the index, the better the distribution. Optimizing distribution optimizes parallel processing.
- In choosing a Primary Index, there is a trade-off between the issues of access and distribution. The most desirable situation is to find a PI candidate that has good access and good distribution. Many times, however, index candidates offer great access and poor distribution or vice versa. When this occurs, the physical designer must balance these two qualities to make the best choice for the index.
- The third of the Primary Index Choice Criteria is volatility, or how often the data values will change. The Primary Index should not be very volatile. Any changes to Primary Index values may result in heavy I/O overhead, as the rows themselves may have to be moved from one AMP to another. Choose a column with stable data values.

Viewing a Tables Distribution

Provides AMP Vproc disk space usage at table level.

DBC.TableSize[V][X]

Vproc	DatabaseName	AccountName
TableName	CurrentPerm	PeakPerm

Example: Display table distribution across AMPs of a NoPI table.

```
SELECT Vproc
      ,CAST (TableName AS CHAR(15))
      ,CurrentPerm
FROM DBC.TableSizeV
WHERE DatabaseName = USER
AND   TableName = 'Customer_NoPI'
ORDER BY Vproc;
```

Result:

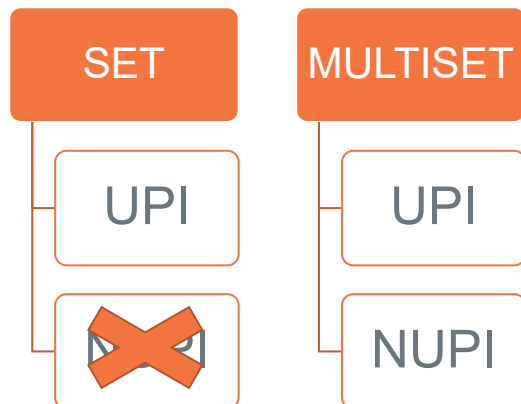
Vproc	TableName	CurrentPerm
0	Customer_NoPI	783,360
1	Customer_NoPI	812,032
2	Customer_NoPI	792,576
3	Customer_NoPI	803,840
4	Customer_NoPI	796,672
5	Customer_NoPI	802,304
6	Customer_NoPI	790,016
7	Customer_NoPI	793,088
8	Customer_NoPI	796,160
9	Customer_NoPI	800,768
10	Customer_NoPI	805,888
11	Customer_NoPI	801,792
12	Customer_NoPI	806,400
13	Customer_NoPI	781,212
:	:	:
29	Customer_NoPI	806,400

The TableSize[V][X] views are Data Dictionary views that provides AMP Vproc information about disk space usage at a table level, optionally for tables the current User owns or has SELECT privileges on.

Example

The SELECT statement on this slide looks for poorly distributed tables by displaying the CurrentPerm figures for a single table on all AMP vprocs.

Tables in Vantage



Types of Teradata Tables: -

- Set table – No FULL Duplicates allowed
 - Full row checking overhead
- Multiset table – FULL duplicates allowed
 - No overhead for row checking
- Any table with a UPI avoids FULL row duplicate check
- SET table with a NUPI incurs FULL row duplicate checking overhead

Each table has one and only one Primary Index. A Primary Index may be different from a Primary Key.

UPI = Best Performance, Best Distribution

- UPIs offer the best performance possible for several reasons. They are:
- A Unique Primary Index involves a single base table row at most
- No Spool file is ever required
- Single value access via the Primary Index is a one-AMP operation and uses only one I/O

NUPI = Good Performance, Good Distribution

- NUPI performance differs from UPI performance because:
- Non-Unique Primary Indexes may involve multiple table rows.
- Duplicate values go to the same AMP and the same data block, if possible.
- Multiple I/Os are required if the rows do not fit in a single data block.
- Spool files are used when necessary.
- A duplicate row check is required on INSERT and UPDATE for a SET table.

NoPI Table – Why?

What is a No Primary Index (NoPI) Table?

- It is simply a table without a primary index
- Rows will still be distributed between AMPs almost equally

Benefits

- Can be used as a staging table/sandbox
- Used to help determine ideal Primary Index distribution for a PI table (*upcoming slide*)
- Preferable than using the first column default

Drawbacks

- All access to NoPI tables are full table scans

A NoPI Table is simply a table without a primary index. NoPI stands for No Primary Index.

Without a PI, the hash value as well as AMP ownership of a row is arbitrary. Within the AMP, there are no row-ordering constraints and therefore rows can be appended to the end of the table as if it were a spool table. Each row in a NoPI table has a hash bucket value that is internally generated. A NoPI table is internally treated as a hashed table; it is just that typically all the rows on one AMP will have the same hash bucket value.

This slide identifies various reasons to consider using NoPI tables.

Why is a NoPI table useful?

- A NoPI can be very useful in those situations when the default primary index (first column) causes skewing of data between AMPs and performance degradation.
- This type of table provides a performance advantage in that data can be loaded and stored quickly into a NoPI table using the TPT Load operator (FastLoad) or TPT Stream operator (TPump) Array INSERT.

Using SQL to Predict Row Distribution

Given a NoPI Customer table, the Row Hash functions can be used to predict the distribution of data rows for a specific column or columns.

Ex: Display the distribution of Customer by custnumber which is unique within the table.

```
SELECT      HASHAMP (HASHBUCKET
                (HASHROW (custnumber))) AS "AMP #"
            , COUNT(*)
FROM        Customer_NoPI
GROUP BY    1
ORDER BY    1;
```

AMP #	Count(*)
0	5,806
1	5,811
2	5,775
3	5,750
4	5,692
5	5,765
6	5,808
:	:
29	5,845

Ex: Display the distribution of Customer by zipcode which is not unique within the table.

```
SELECT      HASHAMP (HASHBUCKET
                (HASHROW (zipcode))) AS "AMP #"
            , COUNT(*)
FROM        Customer_NoPI
GROUP BY    1
ORDER BY    1;
```

AMP #	Count(*)
0	6,211
1	4,845
2	5,237
3	6,031
4	6,139
5	5,243
6	7,186
:	:
29	5,496

This slide contains simple examples of SQL that can be used to determine the actual row distribution for a table.

Creating a Table Without a PI

To create a NoPI table, specify the *NO PRIMARY INDEX* clause in the CREATE TABLE statement.

```
CREATE TABLE <table_name> (<column1>    <column1_datatype>,  
                             <column2>    <column2_datatype>,  
                             ... )  
  
NO PRIMARY INDEX;
```

Considerations:

- When a table is created with no primary index, the TableKind column is set to 'O' instead of 'T' and appears in the DBC.TVM table
- If PRIMARY KEY or UNIQUE constraints are also defined, these will be implemented as Unique Secondary Indexes
- A NoPI table is automatically created as a MULTiset table

This slide identifies the syntax to create a table without a primary index.

If you attempt to include the key word SET (set table) and NO PRIMARY INDEX in the same CREATE TABLE statement, you will receive a syntax error.

Loading Data into a NoPI Table (SQL)

Case – 1: Simple INSERTs

For a simple INSERT, the PE selects a random AMP (based on queryID) and sends the row to it. That AMP then turns the row into a proper internal format and appends it to the end of the NoPI table.

Case – 2: INSERT-SELECT

When inserting data from a source table (PI or NoPI) table into a NoPI target table, data from the source table will NOT be redistributed and will be locally appended into the target table.

Example: `INSERT INTO target_nopi_table SELECT * FROM skewed_table;`

With INSERT-SELECT, the NoPI table can be skewed if the source table is skewed.

If the source table is skewed, the "HASH BY" option can be used to randomly (more evenly) distribute the data between the AMPs for a NoPI table.

This slide summarizes various techniques for inserting data into a NoPI table using SQL INSERT or INSERT-SELECT statements.

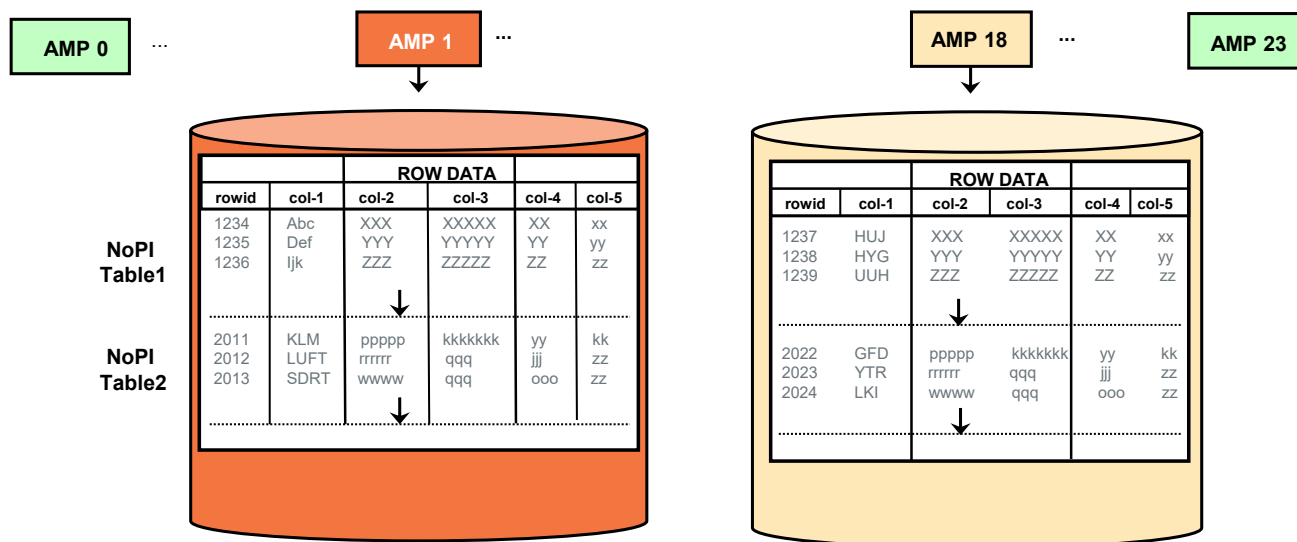
However, the HASH BY and/or LOCAL ORDER BY options can be used to redistribute the data before copying the data into the NoPI table.

Note: You can only specify HASH BY and LOCAL ORDER BY clauses if the target table or the underlying target view table has no primary index (for example, a column-partitioned table). If the table has a primary index, the request is aborted and an error is returned.

If you specify a HASH BY RANDOM clause, data blocks of selected rows are first randomly redistributed. If you also specify a LOCAL ORDER BY clause, then the rows are locally ordered and inserted locally into the target table or underlying target view table.

HASH BY RANDOM (1, 2000000000) is useful to redistribute each individual selected row when there is no particular column set on which to hash distribute the rows, and when a more even distribution is needed than the HASH BY RANDOM clause provides.

Multiple NoPI Tables at the AMP Level



Data within an AMP is logically stored in Row ID sequence.

Other NoPI considerations include:

Archive/Recovery Issues

Archive/Restore will be supported for NoPI table. Archiving a table or a database and restoring or copying that to the same system or a different system should work out fine with the existing scheme for NoPI table when no data redistribution takes place (same number of AMPs). Data redistribution takes place when there is a difference in configuration between the source system and the target system. In the case of a difference in configuration, each row in a table will be looked at and if its hash bucket belongs to some other AMP using the new configuration, that row will be redistributed to its hash-owning AMP.

This means that data in a NoPI table can be skewed after a Restore or Copy. This is because permanent space is divided equally among the AMPs whether or not any of them get any data. As some AMPs not getting any data from a Restore or Copy, some other AMPs will get more data compared to what it was in the source system and this will require more space allocated overall. However, as a staging table, NoPI table is not intended to stay around for too long so it is not expected to have many NoPI tables being restored or copied.

Reconfig Issues

Reconfig will be supported for NoPI table. The issue with Reconfig is very similar to that of Restore or Copy to a different configuration.

NoPI Table Options

Options Available with NoPI Tables

- FALLBACK
- Secondary indexes – USI and NUSI
- Join and reference indexes
- Primary Key and Foreign Key constraints are allowed
- LOBs are allowed on a NoPI table
- INSERT, UPDATE, and DELETE trigger actions are allowed on a NoPI table
- Can be a Global Temporary or Volatile table
- COLLECT/DROP STATISTICS are allowed
- TPT Load (FastLoad) – note that duplicate rows are loaded and not deleted with a NoPI table
- Can be column partitioned

Limitations of NoPI Tables

- SET tables are not allowed
- Partitioned primary index is not allowed
- Permanent journaling is not allowed
- Identity column is not allowed
- Cannot be an error table
- Hash index is not allowed on a NoPI table
- TPT Update (MultiLoad) cannot be used on a NoPI table
- UPSERT and MERGE-INTO operations using the NoPI table as the target table are not allowed

Summary

Now that you have completed this course, you will be able to:

- Compare Non-Parallel vs Parallel Systems
- Describe the need for a Primary Index
- Describe how a Primary Index distributes data
- List types of Primary Indexes
- List types of Teradata Tables
- Describe the need for a No Primary Index (NoPI) table
- Describe how data is loaded into a NoPI table
- List NoPI Limitations

Thank you.

teradata.

©2023 Teradata



Module 2: Analyze Secondary Index

Teradata Vantage MasterClass

Copyright © 2007–2023 by Teradata. All Rights Reserved.

Objectives

After completing this module, you will be able to:

- Describe USI and NUSI implementations
- Describe dual NUSI access
- Explain NUSI and Aggregate processing
- Compare NUSI vs. full table scan (FTS)
- Describe Composite Secondary Indexes
- Choose columns as candidate Secondary Indexes
- Analyze Value Access and Range Access



Why Secondary Indexes

- **Alternate Access Path**
 - Provides alternate access for range and equality-based access
 - Provided faster single row or set selection of records
 - Can often improve system performance, particularly in decision-support environments, and standardized and repetitive queries
- **Uniqueness Constraints**
 - Uniqueness if not possible via Primary index definition, then a Unique Secondary Index is an option

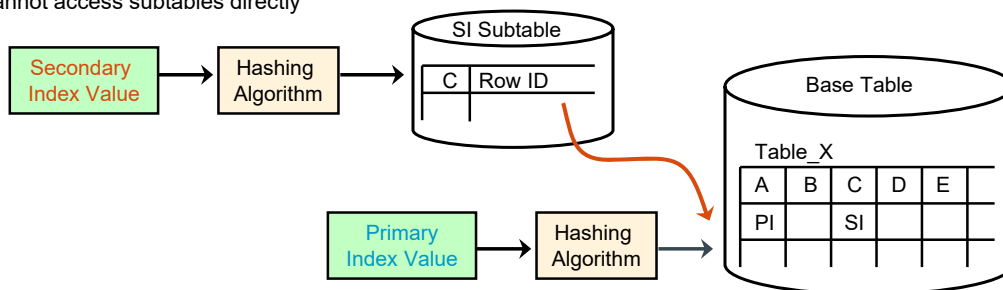
Secondary Indexes

Secondary indexes provide faster set selection.

- They may be unique (USI) or non-unique (NUSI)
- A USI may be used to maintain uniqueness on a column
- The system maintains a separate subtable for each secondary index
- A secondary index can consist of 1 to 64 columns

Subtables keep the base table secondary index row hash, column values, and RowID (which point to the row(s) in the base table with that value).

- The implementation of a USI is different from a NUSI
- Users cannot access subtables directly



Secondary Indexes are generally defined to provide faster set selection. Teradata allows up to 32 Secondary Indexes per table. Teradata handles Unique Secondary Indexes (USIs) and Non-Unique Secondary Indexes (NUSIs) very differently.

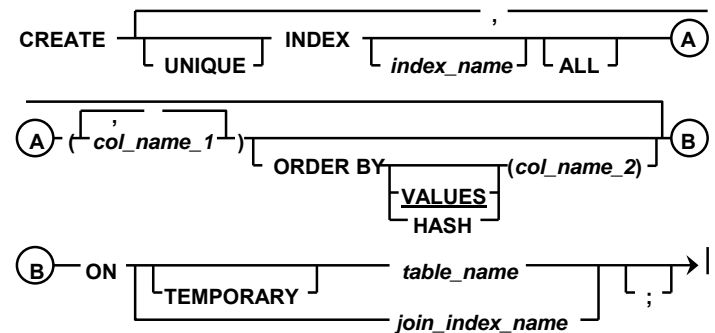
The diagram illustrates how Secondary Index values are stored in subtables. Secondary Index values, like Primary Index values, are input to the Hashing Algorithm. As with Primary Indexes, the Hashing Algorithm takes the Secondary Index value and outputs a Row Hash. These Row Hash values point to a subtable which stores index rows containing the base table SI column values and Row IDs which point to the row(s) in the base table with the corresponding SI value.

Teradata can determine the difference between a base table and a SI subtable by checking the Subtable ID, which is part of the Table ID.

Defining Secondary Indexes

Secondary indexes can be defined as ...

- when a table is created (CREATE TABLE).
- for an existing table (CREATE INDEX).



Examples:

Unnamed USI:

```
CREATE UNIQUE INDEX
(itemid, storeid, salesdate)
ON DailySales;
```

Named NUSI:

```
CREATE INDEX ds_nusi
(salesdate)
ON DailySales;
```

Use the CREATE INDEX statement to create a secondary index on an existing table or join index. The index can be optionally named.

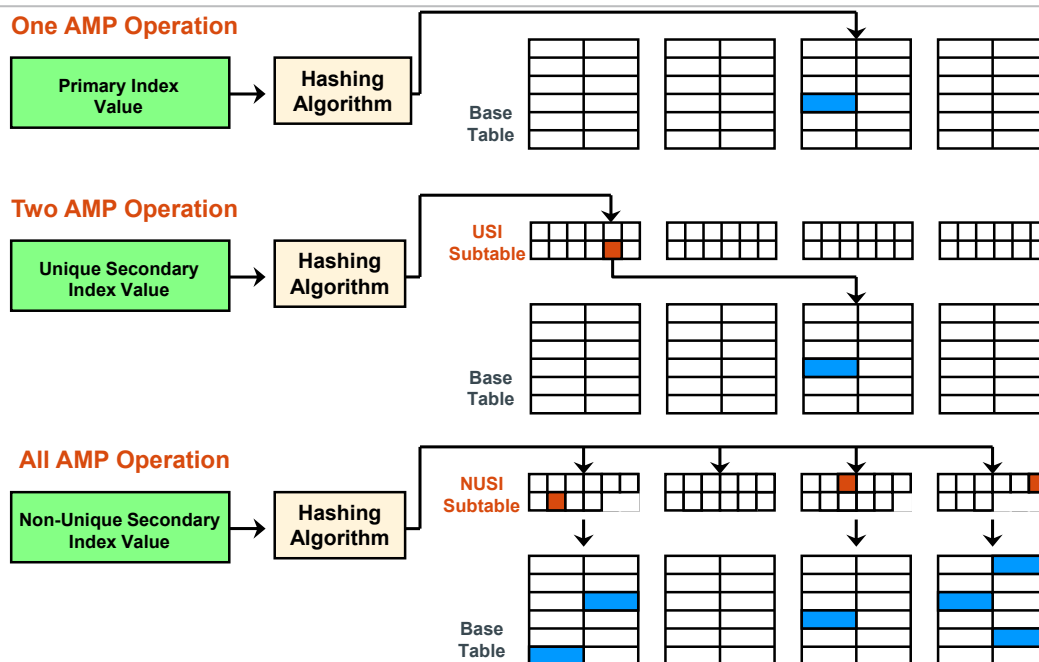
Notes on ORDER BY option:

- If the ORDER BY option is not used, the default is to order by hash.
- If the ORDER BY option is specified and neither of the keywords (HASH or VALUES) is specified, then the default is to order by values.

Notes on the ALL option:

- The ALL option indicates that a NUSI should retain row ID pointers for each logical row of a **join index** (as opposed to only the compressed physical rows).
- ALL also ignores the NOT CASESPECIFIC attribute of data types so a NUSI can include case-specific values.
- ALL enables a NUSI to cover a join index, enhancing performance by eliminating the need to access a join index when all values needed by a query are in the secondary index. However, ALL might also require the use of additional index storage space.
- Use this keyword when a NUSI is being defined for a join index and you want to make it eligible for the Optimizer to select when covering reduces access plan cost. ALL can also be used for an index on a table, however.
- You cannot specify multiple indexes that differ only by the presence or absence of the ALL option.
- The use of the ALL option for a NUSI on a data table does not cause a syntax error.

Secondary Index Subtables



Primary Indexes (UPIs and NUPIs)

In the case of a PI, Teradata hashes the value and uses the Row Hash to find the desired row. This is always a one-AMP operation and is shown in the top diagram on this slide.

Unique Secondary Indexes (USIs)

The middle diagram illustrates the process of a USI row retrieval. An index subtable contains index rows, which in turn point to base table rows matching the supplied index value. USI rows are globally hash-distributed across all AMPs, and are retrieved using the same procedure for Primary Index data row retrieval. Since the USI row is hash-distributed on different columns than the Primary Index of the base table, the USI row typically lands on an AMP other than the one containing the data row. Once the USI row is located, the base table RowID effectively "points" to the corresponding data row. This requires a second access and usually involves a different AMP.

Non-Unique Secondary Indexes (NUSIs)

NUSIs are implemented on an AMP-local basis. Each AMP is responsible for maintaining only those NUSI subtable rows that correspond to base table rows located on that AMP. Since NUSIs allow duplicate index values and are based on different columns than the PI, data rows matching the supplied NUSI value could appear on any AMP.

In a NUSI access, a message is sent to all AMPs to see if they have an index row for the supplied value. Those that do use the "reference identifiers" in the index row to access their corresponding base table rows. Any AMP that does not have an index row for the NUSI value will not access the base table.

USI Subtable General Row Layout

USI Subtable
Row Layout

Row Length	Row ID of USI		Secondary Index Value	Base Table Row Identifier		
	Row Hash	Uniq. Value		Part. #	Row Hash	Uniq. Value
	4	4		2 or 8	4	4

Notes:

- USI subtable rows are distributed by the Row Hash, like any other row
- The Row Hash is based on the unique secondary index value
- The subtable row includes the secondary index value and a second Row ID which identifies a single base table row (usually on a different AMP)
- There is one index subtable row for each base table row
- For row partitioned tables, the partition number (2 or 8 bytes) is included in the base table row identifier *(to be covered in the PPI module)*

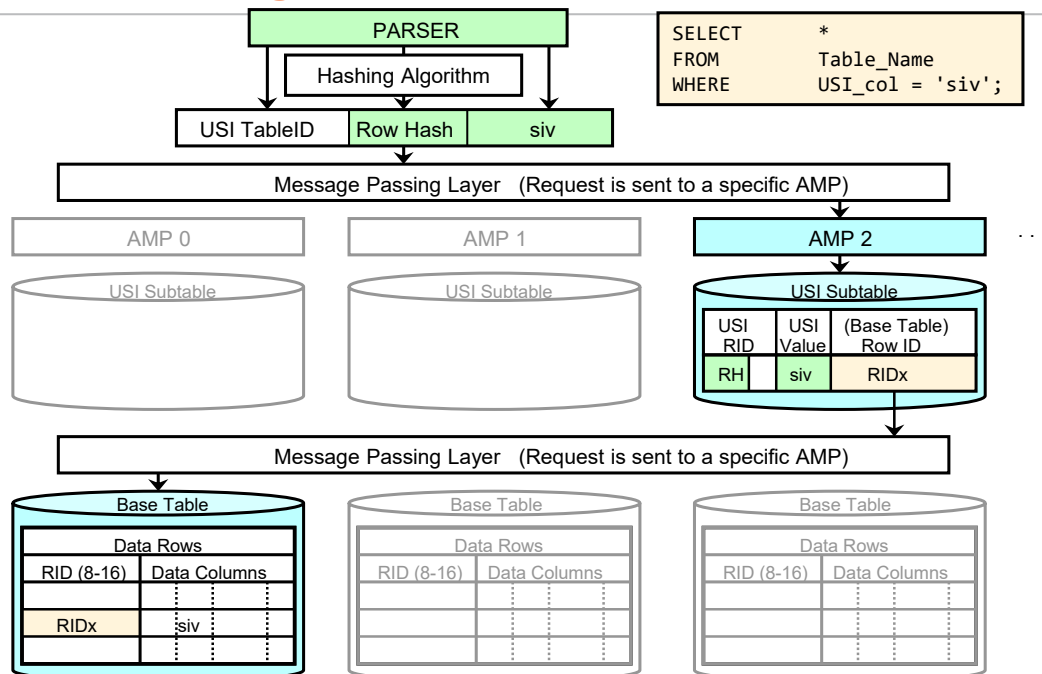
The layout of a USI subtable row is illustrated at the top of this slide. It is composed of several sections:

- The first two bytes designate the **row length**.
- The next 8 bytes contain the **Row ID of the row**. Within this Row ID, there are 4 bytes of Row Hash and 4 bytes of Uniqueness Value.
- The following 2 bytes are **additional system bytes** that will be explained later as will be the 7 bytes for row offsets.
- The next section contains the **SI value**. This is the value that was used by the Hashing Algorithm to generate the Row Hash for this row. This section varies in length depending on the index.
- Following the SI value are 8 bytes containing the **Row ID of the base table row**. The base table Row ID tells the system where the row corresponding to this particular USI value is located. If the table is partitioned, then the USI subtable row needs 10 or 16 bytes to identify the **Row ID of the base table row**. The Row ID (of the base table row) is combination of the Partition Number, Row Hash, and Uniqueness Value.
- The last two bytes contain the **reference array pointer** at the end of the block.

Teradata creates one index subtable row for each base table row.

For tables defined with a PPI, a two-byte or optionally eight-byte partition number is embedded in the data row. Therefore, the unique row identifier is comprised of the Partition Number, the Row Hash, and the Uniqueness Value.

USI Hash Mapping



This slide shows the three-part message that is put onto the Message Passing Layer for USI access.

- The only difference between this and the three-part message used in PI access (previously discussed) is that the Subtable ID portion of the Table ID references the USI subtable not the data table. Using the DSW for the Row Hash, the Message Passing Layer (a.k.a., Communication Layer) directs the message to the correct AMP which uses the Table ID and Row Hash as a logical index block identifier and the Row Hash and USI value as the logical index row identifier. If the AMP succeeds in locating the index row, it extracts the base table Row ID ("pointer"). The Subtable ID portion of the Table ID is then modified to refer to the base table and a new three-part message is put onto the Communications Layer.
- Once again, the Message Passing Layer uses the DSW to identify the correct AMP. That AMP uses the Table ID and Row ID to locate the correct data block and then uses the Row ID to locate the correct row.

NUSI Subtable General Row Layout

NUSI Subtable
Row Layout

Row Length	Row ID of NUSI		Secondary Index Value	Table Row ID List					
	Row Hash	Uniq. Value		P	RH	U	P	RH	U
	4	4		2/8	4	4	2/8	4	4

Notes:

- The Row Hash is based on the base table secondary index value
- The NUSI subtable row contains Row IDs that identify the base table rows on the same AMP that carry the Secondary Index Value
- The Row IDs reference (or "point") to base table rows on this AMP only.

The layout of a **NUSI subtable row** is illustrated on this slide. It is almost identical to the layout of a USI subtable row. There are, however, two major differences:

- First, NUSI subtable rows are not distributed across the system via AMP number in the Hash Map. NUSI subtable rows are built from the base table rows found on that particular AMP and refer only to the base rows of that AMP.
- Second, NUSI rows may point to or reference more than one base table row. There can be many base table Row IDs (8, 10, or 16 bytes) in a NUSI subtable row. Because NUSIs are always AMP-local to the base table rows, it is possible to have the same NUSI value represented on multiple AMPs.

NUSI Change for Row Partitioned Tables

For tables defined with row partitioning, the 2- or 8-byte partition number is embedded in the data row. Therefore, the unique row identifier is comprised of the Partition Number, Row Hash, and Uniqueness Value.

Geospatial Index (TD 14.10)

A NUSI that is defined on a column that has a geospatial data type. Geospatial indexes are implemented using GeoGrids for their population component and Hilbert R-trees for their summary component. Scalar NUSIs use Index Key values that are equal to their base table values, while geospatial NUSIs use index keys that only approximate their base table values. For geospatial indexes, the Secondary Index Value field contains the Minimum Bounding Rectangle (MBR) for the object being indexed.

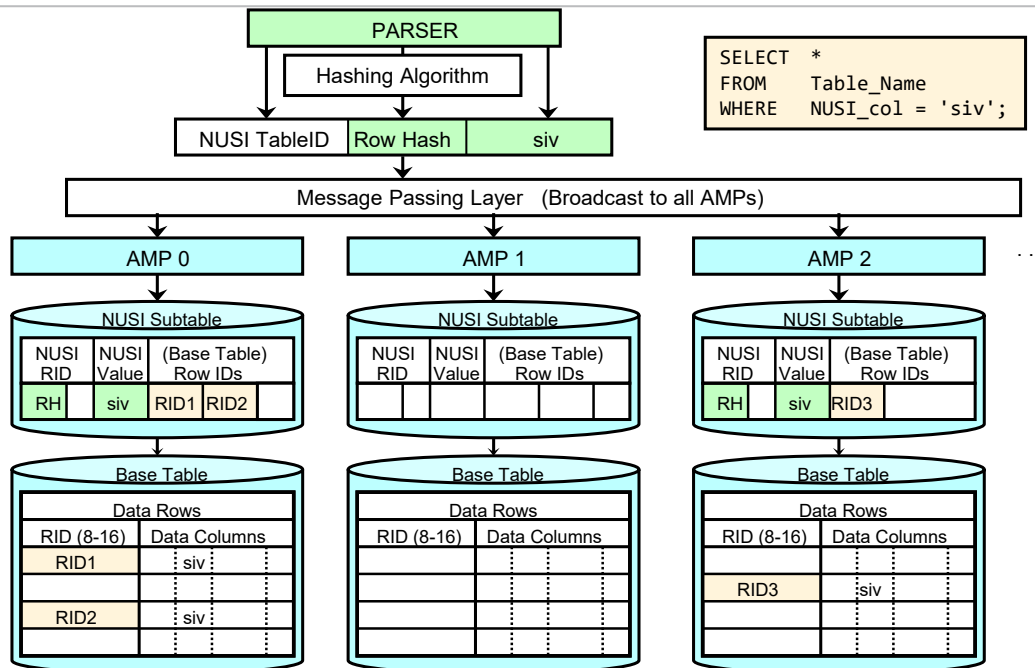
One NUSI subtable row can hold 60,000 (or more) RowIDs; assuming a NUSI data type less than 200 characters (CHAR(200)).

There are one (or more) subtable rows for each secondary index value on the AMP.

If a NUSI subtable row has more RowIDs that will fit in a data row (with the same NUSI value), then

another NUSI subtable row is created for the same NUSI value.
The maximum size of a single NUSI row is 1 MB.

NUSI Hash Mapping



When a secondary index value is provided in an equality condition, the specified secondary index value is hashed and then each NUSI subtable is probed for rows with the same row hash. For each matching NUSI entry, the corresponding Row IDs are used to access the base rows on the same AMP. Because the NUSI rows are stored in row hash order, searching the NUSI subtable for a particular row hash is very efficient.

The example on this slide shows the standard, three-part Message Passing Layer row-access message. Because NUSIs are AMP-local indexes, this message gets broadcast to all AMPs. Each AMP uses the values to search the appropriate index block for a corresponding NUSI row. Only those AMPs with one or more of the desired rows use the base table Row IDs to access the proper data blocks and data rows.

In the example, the SELECT statement is designed to find those rows with a NUSI value of 'siv'. Examination of the NUSI subtables on each AMP shows that AMPs 0, 2 and 3 (not shown) all have a subtable index row, and, therefore, base table rows satisfying this condition. These AMPs then participate in the base table access. The NUSI subtable on AMP 1, on the other hand, shows that there are no rows with a NUSI value of 'siv' located on this AMP. AMP 1 does not participate in the base table access process.

Secondary Index Usage

Unique Secondary Index (USI) Usage

- A USI is used to maintain uniqueness in a column or columns
- Usage is determined by specifying the USI value in an equality condition in the WHERE clause or ON clause

Non-unique Secondary Index (NUSI) Usage

- Usage is determined by specifying the NUSI value in an equality condition in the WHERE clause or ON clause
- In some cases, it may be better to use multiple single-column NUSIs (City, State) instead of a single composite NUSI
- Can significantly reduce base table I/O during value and join operations

This slide lists common usage for a USI and a NUSI.

Secondary Index Properties

- Secondary Indexes consume disk space for their subtables
- A table may have up to 32 secondary indexes and can be created and dropped dynamically
- INSERTs, DELETEs, and UPDATEs (sometimes) cost double the I/Os for tables that have Secondary Indexes
- Choose Secondary Indexes on frequently used set selections
 - Secondary Index use is typically based on an Equality search
 - A NUSI may have multiple rows per value
- Avoid choosing Secondary Indexes with volatile data values

Data demographics change over time. Revisit ALL index (Primary and Secondary) choices regularly.
Make sure they are still serving you well.

This slide describes key considerations involved in decisions regarding the use of Secondary Indexes. It is important to weigh the costs of Secondary Indexes against the benefits.

- Some of these costs are increased use of disk space and increased I/O.
- The main benefit of Secondary Indexes is faster set selection. Choose them on frequently used set selections.
- A NUSI subtable row is AMP local and the base table UNDO row can be used to rollback the base table row and NUSI subtable row. A USI subtable row is usually on a different AMP than the base table row.

REMEMBER

- Data demographics change over time.
- Revisit all index choices regularly to make sure that they remain appropriate and serve you well.

When Creating a Secondary Index

- The Optimizer may, or may not, use a NUSI, depending on its selectivity
 - Without COLLECT STATISTICS, the Optimizer often chooses to do an FTS
- The following approach is recommended:
 - CREATE Index
 - COLLECT STATISTICS on the index (or column)
 - Use EXPLAIN to see if the index is being used

As mentioned at the beginning of this module, a table may have up to 32 Secondary Indexes that can be created and dropped dynamically. It is probably not a good idea to create 32 SIs for each table just to speed up set selection because SIs consume the following extra resources:

- SIs require additional storage to hold their subtables. In the case of a Fallback table, the SI subtables are Fallback also. Twice the additional storage space is required.
- SIs require additional I/O to maintain these subtables.

When deciding whether or not to define a NUSI, there are other considerations. The Optimizer may choose to do a Full Table Scan rather than utilize the NUSI in two cases:

- When the NUSI is not selective enough.
- When no COLLECTed STATISTICS are available.

As a guideline, choose only those rows having frequent access as NUSI candidates. After the table has been loaded, create the NUSI indexes, COLLECT STATISTICS on the indexes, and then do an EXPLAIN referencing each NUSI. If the Parser chooses a Full Table Scan over using the NUSI, drop the index.

REMEMBER

The only way to determine for certain whether an index is being used is to utilize the EXPLAIN facility.

NUSI vs. Full Table Scan (FTS)

The Optimizer generally chooses an FTS (or partition scan if possible) over a NUSI when:

- The index is too weakly selective – effectively means that many rows have that value or fit within the range of values specified
- If statistics have **not** been collected on the NUSI, a NUSI may or may not be used

Access Method	Physical I/Os per AMP
NUSI	1 Index Subtable Row(s) 0 – Many Data Row(s)
Partition or Full Table Scan	0 Index Subtable Rows N or ALL Data Row(s)

General Rules:

- **COLLECT STATISTICS** on all NUSIs
- USE EXPLAIN to see whether a NUSI is being used
- Do not define NUSIs that will not be used

The Optimizer generally chooses an FTS over a NUSI when one of the following occurs:

- Rows per value is greater than data blocks per AMP.
- It does not have COLLECTed STATISTICS on the NUSI.
- The index is too weakly selective. The Optimizer determines this by using COLLECTed STATISTICS.

Example

The table on this slide shows how the access method chosen affects the number of physical I/Os per AMP.

In the case of a NUSI, there is ONE I/O necessary to read the Index Block on each AMP plus 0-ALL (where ALL = Number of Data Blocks) I/Os required to read the Data Blocks for a possible total ranging from the Number of AMPs - (Number of AMPs + ALL) I/Os.

In general, the NUSI is not used if the estimated number of rows to be read in the base table is equal to or greater than the estimated number of data blocks in the base table; in this case, a full table scan is done, or, if appropriate, partition scans are done.

In the case of a Full Table Scan, there are no I/Os required to read any Index Blocks, but the system reads ALL Data Blocks. The only way to tell whether or not a NUSI is being used is by using EXPLAIN.

Secondary Index Overhead

- **Secondary indexes have two types of additional overhead**
 - Additional I/O maintenance – SI subtable rows are updated as base table rows are inserted, updated, or deleted
 - Secondary index subtables require additional disk space (covered later)
- **Many factors influence the number of physical I/Os in a transaction**
 - Cache hits, rows per block, BLOBs/CLOBs, etc.
 - Number of spool files and spool file sizes
- **"Logical I/O counts" are used to indicate the relative cost of a transaction**
 - A given I/O operation may not cause any actual physical I/O

Secondary Indexes has two type of overhead – additional I/O maintenance and disk space for the secondary index subtables.

Understanding the cost of various Teradata transactions in terms of I/O will help you avoid unnecessary I/O overhead when doing your physical design. Many factors can influence the number of physical I/Os in a transaction. Some are listed on this slide.

The main concept of the next few pages is to help you understand the relative cost of doing INSERT, DELETE, and UPDATE operations. This understanding enables you to detect subsequent problems when doing performance analysis on a troublesome application. When counting I/O, it is important to remember that all such calculations give you a relative – not the absolute – cost of the transaction. Any given I/O operation may or may not cause any actual physical I/O.

- Normally, when making a change to a table (INSERT, UPDATE, and DELETE), not only does the actual table have to be updated, but before-images have to be written in the Transient Journal to maintain transaction integrity. Transient Journal space is automatically allocated and is integrated with the WAL (Write-Ahead-Logic) Log, which has its own cylinders and file system.

Additional I/O

A table may also have Join Indexes, Hash indexes, or a Permanent Journal associated with it. Join Indexes can also have secondary indexes.

- Data and index subtable block I/O may or may not require Cylinder Index I/O.
 - I/Os may be done serially or in parallel.
- A table may also have Join/Hash indexes, secondary indexes on Join Indexes, or a Permanent Journal associated with it.

- These options will result in additional I/O.

Secondary Index Considerations

- A NUSI will be used depending on the percentage of table rows that will be accessed (selectivity)



All PI candidates are SI candidates. Columns that are not PI candidates have to also be considered as NUSI candidates. A NUSI will be used by the Optimizer to select data if it is strongly selective.

A guideline to use in initially selecting NUSI candidates is the following:

The optimizer does not only look at selectivity of a column to determine if a FTS or an indexed access will be used in a given plan. The decision is made based after comparing the total cost of both approaches, after considering multiple factors, including row size, block size, number of rows in the table, and also the I/O and CPU cost (based on the current hardware cost factors).

In this course, we are going to 5% as a guideline for NUSI selectivity.

- Example 1: Assume a table has 100M rows and a column has 50 distinct values that are evenly distributed (each value has the same number of rows). Therefore, each value has 2M rows and effectively represents 2% of the rows. The NUSI would be used.
- Example 2: Assume a table has 100M rows and a column has 20 distinct values that are evenly distributed (each value has the same number of rows). Therefore, each value has 5M rows and effectively represents 5% of the rows. The NUSI would be used.
- Example 3: Assume a table has 100M rows and a column has 10 distinct values that are evenly distributed (each value has the same number of rows). Therefore, each value has 10M rows and effectively represents 10% of the rows. The NUSI would not be used.

The greater the discrepancy between typical rows per value and max rows per value, the higher the probability the NUSI would not be used based on the max value used to qualify the rows.

- Example:
 - If the number of rows accessed via a NUSI is $\leq 5\%$, the NUSI will be used.
 - If the number of rows accessed via a NUSI is 5 – 10%, the NUSI may or may not be used.

- If the number of rows accessed via a NUSI is $> 10\%$, the NUSI will not be used.
- If 5% is used as the guideline, then any column with 20 or more distinct values is considered a NUSI candidate.
 - The optimizer (based on statistics) will decide to use (or not) the NUSI for specific values.
 - The Optimizer may, or may not, use a NUSI, depending on its selectivity.
- Validate (via Explain and testing) that the NUSI will be chosen AND that it will provide better performance.

Row Selection

WHERE clause conditions that may use indexing if available.

- Access methods for the above depend on whether the column(s) are indexed, the type of index, and the selectivity of the index

colname = value colname IS NULL colname IN (subquery)	colname IN (explicit list of values) t1.col_x = t1.col_y t1.col_x = t2.col_x	condition1 AND condition2 condition1 OR condition2 colname = ANY, SOME or ALL
---	--	---

WHERE clause conditions that typically cause a Partition or Full Table Scan

non-equality comparisons colname IS NOT NULL colname NOT IN (explicit list of values) colname NOT IN (subquery) colname BETWEEN ... AND ... Join condition1 OR Join condition2 t1.col_x [computation] = value t1.col_x [computation] = t1.col_y	colname IN (explicit list of values) t1.col_x = t1.col_y t1.col_x = t2.col_x	condition1 AND condition2 condition1 OR condition2 colname = ANY, SOME or ALL
--	--	---

Poor relational models severely limit physical design choices and generally force more Full Table Scans.

When TERADATA processes an SQL statement with a WHERE clause, it examines the clause and builds an execution plan and access method to satisfy the clause conditions. Note that poor relational models severely limit physical design choices and generally force more Full Table Scans.

Certain conditions contained in the WHERE clause take advantage of indexing (assuming that the appropriate index is in place). These conditions are shown in the upper box on this slide. Notice that these conditions all ask the RDBMS to locate a specific value or set of values. Application programmers should use these conditions whenever possible as they offer the best performance.

Other WHERE clause conditions are not able to take advantage of indexing and will always cause a Full Table Scan of either the Base Table or a SI subtable. Though they benefit from the Teradata distributed architecture, they are less desirable from a performance standpoint. These kind of conditions are listed in the middle box on the opposite page and do not focus on a specific value or set of values, thus forcing the system to do a Full Table Scan to find all the values to satisfy them.

Maximum number of ORed conditions or IN list values per request can't exceed 1,048,576. There really no fixed limit on the number of entries in an IN list; however, the maximum SQL text size is 1MB and this places a request-specific upper bound on this number.

The following functions affect the output only, not base row selection.

GROUP BY
HAVING
WITH
WITH ... BY ...
ORDER BY
UNION
INTERSECT
EXCEPT

NUSI Bit Mapping

- NUSI Bit Mapping is an optimizer technique to determine common Row IDs between multiple NUSI values.
 - Note: Bit mapping is used only when equality or range constraints involving multiple nonunique secondary indexes are applied to very large tables
- All NUSI conditions must be linked by the AND operator
- Requires at least two NUSI equality conditions
- The Optimizer is more likely to consider bit mapping if you COLLECT STATISTICS
- Use EXPLAIN to see if bit mapping is being used

Total Records in Employee table = 1 Million		
Individual Condition	Record Count	Common RowID (Bitmap)
SalaryAmount > 75K	70,000 (7%)	3000 (0.3%)
CountryCode = 'USA'	400,000 (40%)	
JobCode = 'IT'	120,000 (12%)	

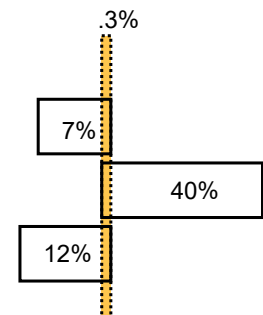
```

SELECT *
FROM Employee
WHERE salaryamount > 75000

AND countrycode = 'USA'

AND jobcode = 'IT';

```



NUSI Bit Mapping is a process that determines common Row IDs between multiple NUSI values by a process of intersection:

- It is much faster than copying, sorting and comparing the Row ID lists.
- It dramatically reduces the number of base table I/Os.

NUSI bit mapping can be used with conditions other than equality if all of the following conditions are satisfied:

- All conditions must be linked by the AND operator.
- At least two NUSI equality conditions must be specified.
- The Optimizer is more likely to consider if you have COLLECTed STATISTICS on the NUSIs.

Even when the above conditions are satisfied, the only way to be absolutely certain that NUSI bit mapping is occurring is to use the EXPLAIN facility.

Single and Dual NUSI Access

A separate NUSI is created on each column:

```
CREATE INDEX (deptnumber) ON Employee;
CREATE INDEX (jobcode) ON Employee;
```

Single NUSI Access:

```
SELECT lastname, firstname, ...
FROM Employee
WHERE deptnumber = 500;
```

AND with Equality Conditions:

```
SELECT lastname, firstname, ...
FROM Employee
WHERE deptnumber = 500
AND jobcode = 2147;
```

OR with Equality Conditions:

```
SELECT lastname, firstname, ...
FROM Employee
WHERE deptnumber = 500
OR jobcode = 2147;
```

Optimizer options for single NUSI access:

- Use the index if it is strongly selective
- Perform an FTS of index subtable if it covers the query or an FTS of the base table

Optimizer options with AND:

- Use one of the two indexes if it is strongly selective
- If the two indexes together are strongly selective, optionally do a bit-map intersection (not common)
- Perform an FTS of the base table if both indexes are weakly selective

Optimizer options with OR:

- If each of the NUSIs is strongly selective, it may use each of the NUSIs to return the appropriate rows
- Do an FTS of the two NUSI subtables and retrieve Rows IDs of qualifying rows into spool and eliminate duplicate Row IDs from spool
- Do an FTS of the base table

In the example on this slide, two NUSIs are CREATED on separate columns of the Employee Table. Teradata decides how to use these NUSIs based on their selectivity. It is always recommended to COLLECT STATISTICS on any NUSIs.

If only one of the two columns joined by the OR is indexed, Teradata always does an FTS of the base table.

Value-Ordered NUSIs

A Value-Ordered NUSI is limited to a single numeric (4-byte) or DATE column.

Some benefits of using value-ordered NUSIs:

- Index subtable rows are sorted (sequenced) by data value rather than hash value
- Optimizer can search only a portion of the index subtable for a given range of values
- Can provide major advantages in the performance of range queries
- Even with a row-partitioned table, the Value-Ordered NUSI is still a valuable index selection for other columns in a table

Example of creating a Value-Ordered NUSI:

```
CREATE INDEX (shipdate) ORDER BY VALUES (shipdate)
ON Orders;
```

```
SELECT      shipdate
            ,SUM(ordertotal)
FROM        Orders
WHERE       shipdate
            BETWEEN DATE '2021-01-20' AND DATE '2021-01-26'
GROUP BY    1
ORDER BY    1;
```

→ The optimizer may choose to transverse the NUSI using a range constraint rather than do an FTS.

NUSIs are maintained as separate subtables on each AMP. Their index entries point to base table or Join Index rows residing on the same AMP as the index. The row hash for NUSI rows is based on the secondary index column(s). Unlike row hash values for base table rows, this row hash does not determine the distribution of subtable rows; only the local sort order of each subtable.

Enhancements have been made to support the user-specified option of sorting the index rows by data value rather than by hash code. This is referred to as "value ordered" indexes and is presented to the user in the form of new syntax options in the CREATE INDEX statement.

By using the "value-ordered" indexes feature, this option can be specified to sort the index rows by data value rather than by hash code. Value-ordered NUSIs, on the other hand, are useful for processing range conditions and conditions with an inequality on the secondary index column set.

By sorting the NUSI rows by data value, it is possible to search only a portion of the index subtable for a given range of key values. The major advantage of a value-ordered NUSI is in the performance of range queries.

Value-ordered NUSIs have the following limitations.

- The sort key is limited to a single numeric column.
- The sort key column must be four or fewer bytes.

Value-Ordered NUSIs

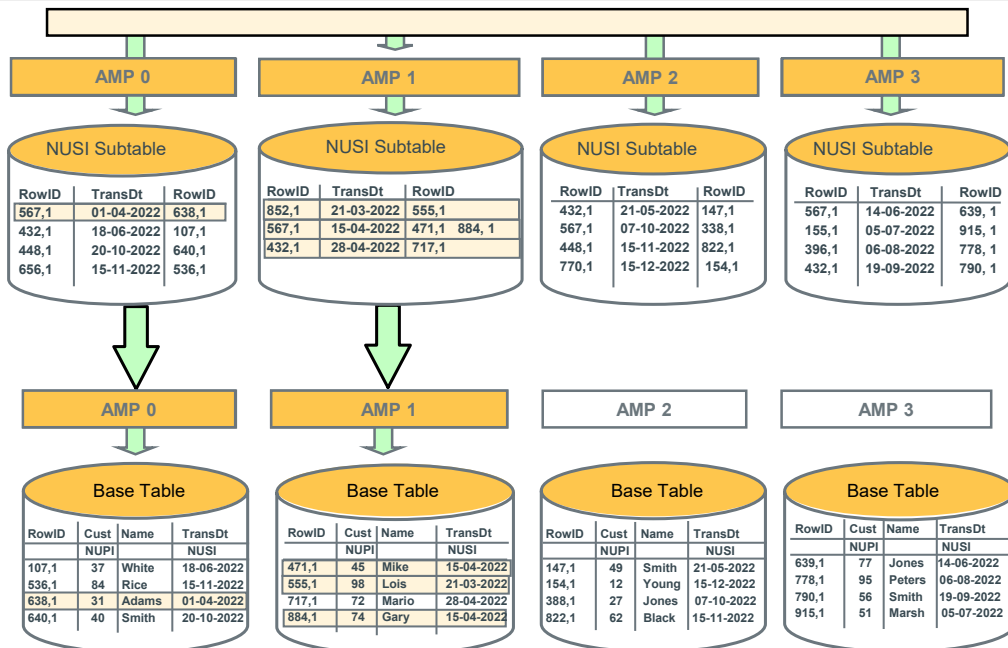
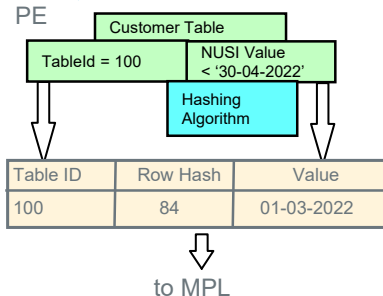
Create VONUSI

```
CREATE INDEX (TransDt) Order by
Values (TransDt) ON Sales_Txn;
```

Access via VONUSI

```
SELECT *
FROM Sales_Txn
WHERE
TransDt between Date '01-03-
2022' and Date '30-04-2022';
```

PE



The row hash is only used for distributing the row to the AMPs, but it's replaced by the actual value within AMP for sorting.

That's why Order By Values is restricted to Int or Date, for the file system it's just another 4-byte value like the RowHash.

Hash-Ordered NUSIs

- If a NUSI consists of multiple columns (and is not hash ordered), then the hash order of the NUSI subtable is based on the composite hash of the set of columns
- The 'ORDER BY HASH' clause provides the ability to create a multi-column index
- The NUSI hashed is based on a single attribute within the index, not the entire composite value
 - A Hash-Ordered NUSI is useful for equality searches based on a single attribute and can be used to also cover queries
- Optimizer is likely to use a hash-ordered NUSI if you have collected statistics on the hash-ordered columns

When a multi-column NUSI is created (without using the Order by Values or Hash option), the hash of the NUSI is based on the set of NUSI values. This is effectively a composite hash. Therefore, in order the hash to be used, an *equality* condition is needed on all of the secondary index column(s).

The Hash-Ordered NUSI provides the ability to create a multi-column index, but have the NUSI hashed based on a single attribute within the index, not the entire composite value.

When a multi-column NUSI is ordered by hash on a single column, each AMP creates a subtable row with a local row hash based on the hash of the single column. When the table is accessed via this column, the value can be hashed and this hash is used to access its portion of the index subtable to see if a corresponding row(s) exists.

This is effectively an example of vertical partitioning with a NUSI.

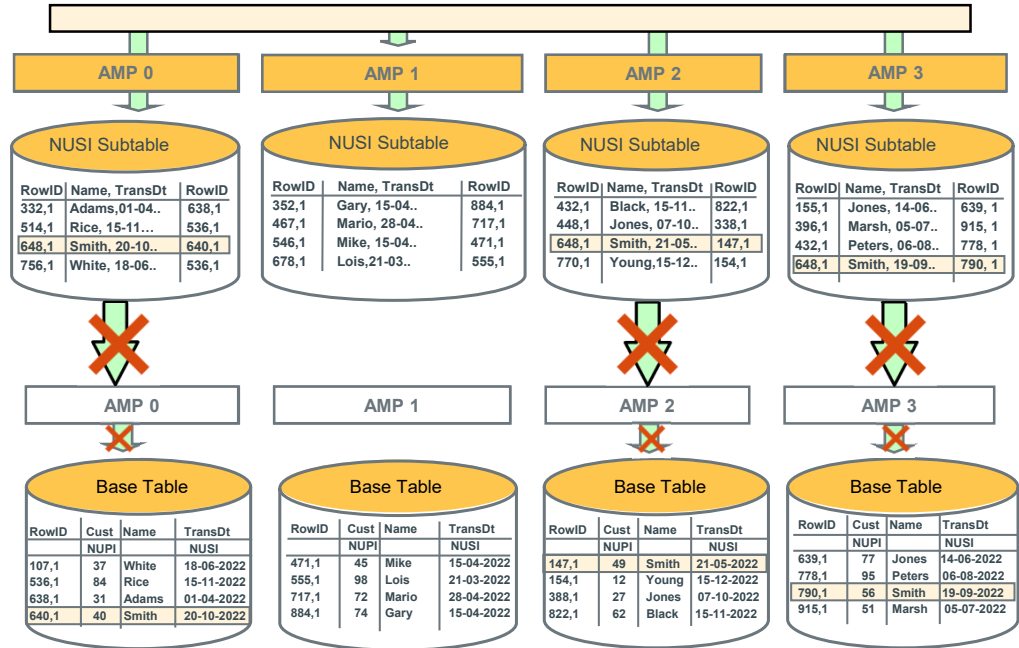
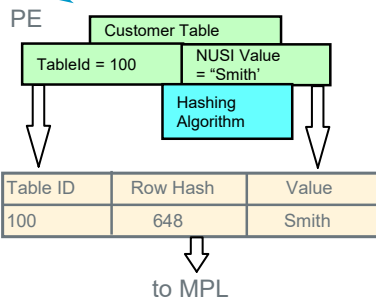
Hash-Ordered NUSIs

Create HONUSI

```
CREATE INDEX (Name, TransDt)
Order by Hash (Name) ON
Sales_Txn;
```

Access via VONUSI

```
SELECT Name, TransDt
FROM Sales_Txn
WHERE
Name = 'Smith' Group BY Name;
```



Covering Indexes

- A NUSI subtable is said to “cover” any query that references only columns defined within NUSI
- These columns can be specified anywhere in the query including:
 - SELECT list
 - WHERE clause
 - GROUP BY clauses and AGGREGATE functions
 - Expressions
- A covering NUSI is **considered by the optimizer in join plans** and can join to other tables too

```
CREATE INDEX IdxOrder2
  (orderid, custid, orderdate, shipdate, ordertotal)
  ORDER BY VALUES (orderdate)
ON Orders;
```

This NUSI is considered for covering the first query and for covering and ordering on the second query.

```
SELECT  custid, SUM(ordertotal)
FROM    Orders
GROUP BY 1;
```

```
SELECT  orderdate, AVG(ordertotal)
FROM    Orders
WHERE   orderdate = DATE '2021-03-31'
GROUP BY 1;
```

If the query references only columns of that table that are fully contained within a given index, the index is said to “cover” the table in the query. In these cases, it is often more efficient to access only the index subtable and avoid accessing the base table rows altogether.

Covering will be considered for any table in the query that references only columns defined in a given NUSI. These columns can be specified anywhere in the query including the:

- SELECT list
- WHERE clause
- Aggregate functions
- GROUP BY expressions

The presence of a WHERE condition on each indexed column is not a prerequisite for using the index to cover the query. The optimizer will consider the legality and cost of covering versus other alternative access paths and choose the optimal plan. Many of the potential performance gains from index covering require no user intervention and will be transparent except for the execution plan returned by EXPLAIN.

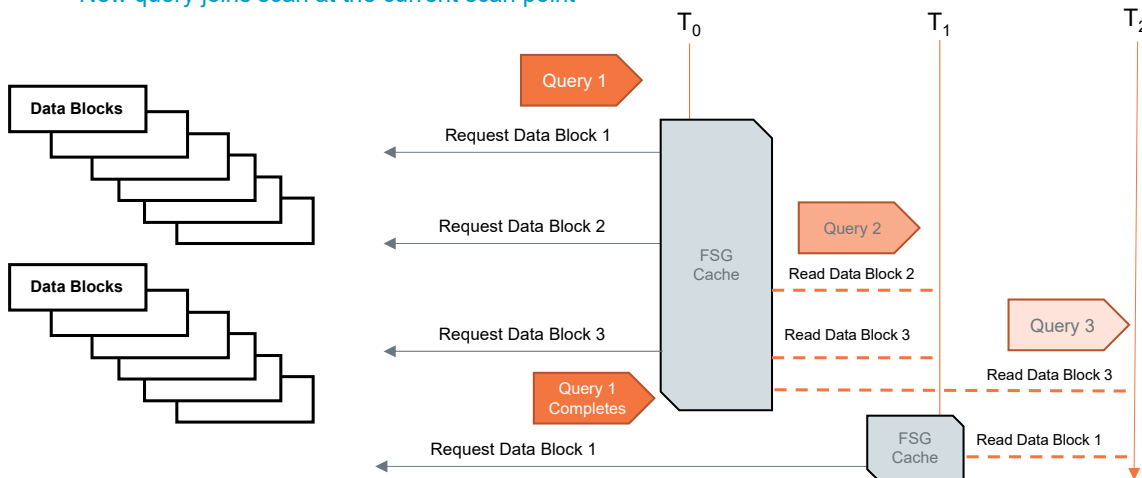
Join Index Note:

This course hasn’t covered Join Indexes to this point, but it is possible to create a NUSI on top of a Join Index. The CREATE INDEX has a special option of ALL which is required if these columns will be potentially used for covering.

Full Table Scans – Sync Scans

In the case of multiple users that access the same table at the same time

- The system can do a synchronized scan (sync scan) i.e., multiple simultaneous scans share reads – this is a sync scan at the block level
- New query joins scan at the current scan point



In the case of multiple users that access the same table at the same time, the system can do a synchronized scan (sync scan) on the table.

Summary

Now that you have completed this course, you will be able to:

- Describe USI and NUSI implementations
- Describe dual NUSI access
- Explain NUSI and Aggregate processing
- Compare NUSI vs full table scan (FTS)
- Describe Composite Secondary Indexes
- Choose columns as candidate Secondary Indexes
- Analyze Value Access and Range Access



Module 2: Analyze Secondary Index Bring Up JupyterHub

teradata.

Let's now do the lab together



Thank you.

teradata.

©2023 Teradata



Module 3: Row Partitioning

Teradata Vantage MasterClass

Copyright © 2007–2023 by Teradata. All Rights Reserved.

Objectives

After completing this module, you will be able to:

- Describe the components that comprise a Row ID in a row partitioned table
- List two advantages of partitioning a table
- Identify partition definition on Numeric and Character data types
- Create single-level and multilevel partitioned tables
- Use the PARTITION key word to display partition information
- Use ALTER TABLE to add and drop ranges from a row partitioned table

NOTE: It is recommended that students go through the official publications to get a deeper understanding of various use cases.

Why Row Partitioning?

Orders table defined without Row Partitioning across four AMPs.

```
SELECT ...
WHERE Order# = 1043;
```

RH	Order#	OrderDate	RH	Order#	OrderDate	RH	Order#	OrderDate	RH	Order#	OrderDate
'01'	1028	2021/01/03	'06'	1009	2021/01/01	'04'	1008	2021/01/01	'02'	1024	2021/01/02
'03'	1016	2021/01/02	'07'	1017	2021/01/02	'05'	1048	2021/01/04	'08'	1006	2021/01/01
'12'	1031	2021/01/03	'10'	1034	2021/01/03	'09'	1018	2021/01/02	'11'	1019	2021/01/02
'14'	1001	2021/01/01	'13'	1037	2021/01/04	'15'	1042	2021/01/04	'18'	1041	2021/01/04
'17'	1013	2021/01/02	'16'	1021	2021/01/02	'19'	1025	2021/01/03	'20'	1005	2021/01/01
'23'	1040	2021/01/04	'21'	1045	2021/01/04	'24'	1004	2021/01/01	'22'	1020	2021/01/02
'28'	1032	2021/01/03	'26'	1002	2021/01/01	'27'	1014	2021/01/02	'25'	1036	2021/01/03
'30'	1038	2021/01/04	'29'	1033	2021/01/03	'32'	1003	2021/01/01	'31'	1026	2021/01/03
'35'	1007	2021/01/01	'34'	1029	2021/01/03	'33'	1039	2021/01/04	'38'	1046	2021/01/04
'39'	1011	2021/01/01	'36'	1012	2021/01/01	'40'	1035	2021/01/03	'41'	1044	2021/01/04
'42'	1047	2021/01/04	'36'	1043	2021/01/04	'44'	1022	2021/01/02	'43'	1010	2021/01/01
'48'	1023	2021/01/02	'45'	1015	2021/01/02	'47'	1027	2021/01/03	'46'	1030	2021/01/03

Orders table defined with Row Partitioning on OrderDate across four AMPs.

```
SELECT ...
WHERE OrderDate =
DATE '2021-01-03';
```

RH	Order#	OrderDate	RH	Order#	OrderDate	RH	Order#	OrderDate	RH	Order#	OrderDate
'14'	1001	2021/01/01	'06'	1009	2021/01/01	'04'	1008	2021/01/01	'08'	1006	2021/01/01
'35'	1007	2021/01/01	'26'	1002	2021/01/01	'24'	1004	2021/01/01	'20'	1005	2021/01/01
'39'	1011	2021/01/01	'36'	1012	2021/01/01	'32'	1003	2021/01/01	'43'	1010	2021/01/01
'03'	1016	2021/01/02	'07'	1017	2021/01/02	'09'	1018	2021/01/02	'02'	1024	2021/01/02
'17'	1013	2021/01/02	'16'	1021	2021/01/02	'27'	1014	2021/01/02	'11'	1019	2021/01/02
'48'	1023	2021/01/02	'45'	1015	2021/01/02	'44'	1022	2021/01/02	'22'	1020	2021/01/02
'01'	1028	2021/01/03	'10'	1034	2021/01/03	'19'	1025	2021/01/03	'25'	1036	2021/01/03
'12'	1031	2021/01/03	'29'	1033	2021/01/03	'40'	1035	2021/01/03	'31'	1026	2021/01/03
'28'	1032	2021/01/03	'34'	1029	2021/01/03	'47'	1027	2021/01/03	'46'	1030	2021/01/03
'23'	1040	2021/01/04	'13'	1037	2021/01/04	'05'	1048	2021/01/04	'18'	1041	2021/01/04
'30'	1038	2021/01/04	'21'	1045	2021/01/04	'15'	1042	2021/01/04	'38'	1046	2021/01/04
'42'	1047	2021/01/04	'36'	1043	2021/01/04	'33'	1039	2021/01/04	'41'	1044	2021/01/04

This slide provides a logical example of an Orders table implemented with a NPPI (Non-Partitioned Primary Index) and the same table implemented with row partitioning (or PPI – Partitioned Primary Index). Only the Order# and the OrderDate are shown as user columns in the example.

The column headings in this example represent the following:

- RH – Row Hash – the two-digit row hash is used for simplification purposes. A true table would contain a Row ID for each row (Row Hash + Uniqueness Value). Note that as just in a real implementation, two different order numbers happen to hash to the same row hash value. Order numbers 1012 and 1043 on AMP 2 both hash to '36'.
- Order# – this example assumes that Order Number is the Primary Index and the data rows are hash distributed based on this value.
- Order Date – the order date for each order. This example only illustrates contains orders for 4 days.

Important points to understand from this example:

- All of the rows in the NPPI table are stored in logical Row ID sequence (row hash + uniqueness value) within each AMP.
- The rows in the row partitioned table are first ordered by Partition Number, and then by Row Hash (actually Row ID) sequence within the Partition.
- This example illustrates 4 partitions – one for each of the 4 days shown in the example.
- The first query that requests "order information" for a specific date – WHERE condition that specifies a specific date.
- The second query requests "order information" for a specific order number (e.g., #1043).

Why Partition a Table?

- Increase query efficiency by avoiding full table scans without the overhead and maintenance costs of secondary indexes
 - **Partition Elimination** – the key advantage to partitioning a table is that the optimizer can eliminate partitions for queries
- Deleting large volumes of rows in entire partitions can be extremely fast

One key reason to partition a table is to reduce the number of full table scans of the table. To determine which columns to use for the Primary Index and for the partitioning depends on how its rows are most frequently accessed. Row partitioned tables are designed to optimize range queries while also providing efficient primary index join strategies. For range queries, only rows of the qualified partitions need to be accessed.

Queries that request a subset of the data (some number of months) only need to access the required partitions instead of the entire table. For example, a query that requests two days of sales data only needs to read 2 partitions of the data from each AMP. This is about 1/365 of the table. Without row partitioning or any secondary indexes, this query has to perform a full table scan.

The more partitions there are, the greater the potential benefit.

Some of the performance opportunities available by using row partitioning:

- Get more efficiency in querying against a subset of large volumes of transactional detail data as well as to manage this data more effectively.
- Allow “instantaneous” dropping of “old” data and simple addition of “new” data.

The term “**partition elimination**” refers to an automatic optimization in which the optimizer determines, based on query conditions, that some partitions can't contain qualifying rows, and causes those partitions to be skipped. Partitions that are skipped for a particular query are called excluded partitions. Generally, the greatest benefit of a row partitioned table is obtained from partition elimination.

What is Row Partitioning?

What is Row Partitioning – a.k.a. Partitioned Primary Index (PPI)?

- An indexing mechanism in Teradata for use in physical database design
- Data rows are grouped into partitions at the AMP level – partitioning is simply an ordering of the rows within a table on an AMP

What advantages does partitioning provide?

- Increases the available options to improve the performance of certain types of queries – specifically range-constrained queries
- Only the rows of the qualified partitions in a query need to be accessed – avoid full table scans

How is a row partitioning created and managed?

- Row partitioning is easy to create and manage
 - The CREATE TABLE and ALTER TABLE statements contain options to create and/or alter partitions
- As always, data is distributed among AMPs and automatically placed within partitions

As part of implementing a physical design, Teradata provides numerous indexing options that can improve performance for different types of queries and workloads. For example, secondary indexes, join indexes, or hash indexes may be utilized to improve performance for known queries. Teradata provides additional new indexing options to provide even more flexibility in implementing a Teradata database. One of these new indexing options is the ability to row partition a table. This feature was initially known as Partitioned Primary Index (PPI). Key characteristics of row partitioned tables are listed on this slide.

Tables with a Primary index can be partitioned or non-partitioned. A non-partitioned table (a.k.a., (no-partitioned primary index – NPPI) is the traditional primary index by which rows are assigned to AMPs. Apart from maintaining their storage in row hash order, no additional assignment processing of rows is performed once they are hashed to an AMP.

A row partitioned table permits rows to be assigned to user-defined data partitions on the AMPs, enabling enhanced performance for range queries that are predicated on primary index values.

The row partitioning feature allows a class of queries to access a portion of a large table, instead of the whole table. The traditional uses of the Primary Index (PI) for data placement and rapid access of the data when the PI values are specified are retained.

How is Partitioning Implemented?

Provides an additional level of data distribution and ordering.

- Rows are distributed between the AMPs based on the Primary Index value
- At the AMP level and within the table, rows are first ordered by their partition number
- Within the partition, data rows are logically stored in Row ID sequence

If a table is partitioned, rows are placed into partitions.

In a partitioned table, each row is uniquely identified by the following:

- Row ID = Partition # + Row Hash + Uniqueness Value
- Row Key = Partition # + Row Hash (e.g., Row Key will appear in Explain plans)
 - In a partitioned table, data rows will have the Partition # included as part of the data row

Combined Partition	Bytes
<= 65,535	2
> 65,535	8
Maximum Combined Partitions is 9.223 Quintillion	

To help understand how partitioning is implemented, this presentation will include examples of data access using [non-partitioned tables](#) and [row partitioned tables](#).

The PRIMARY INDEX clause (part of the CREATE TABLE statement) has been extended to include a PARTITION BY clause. This new partition expression definition is the only thing that needs to be done to create a partitioned table. Advantages to this approach are:

- No separate partition layout
- No disk layout for partitions
- No definition of location in the system for partition
- No need to define/manage separate tables per segment of the table that needs to be accessed
- Even data distribution and even processing of a logical partition is automatic due to the PI distribution of the rows

No query has to be modified to take advantage of a row partitioned table.

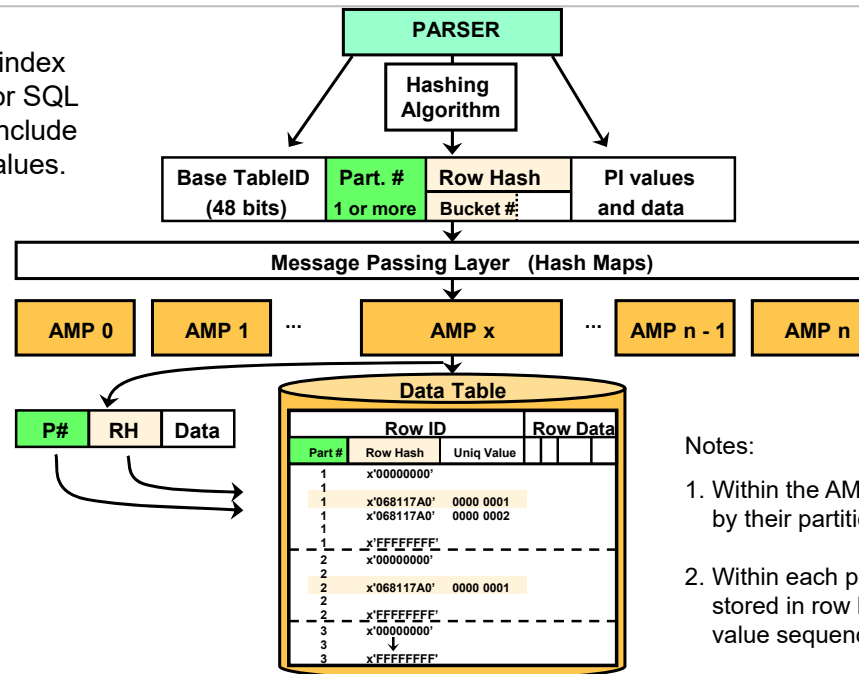
For tables that are row partitioned, Teradata utilizes a 3-level scheme to distribute and later locate the data. The 3 levels are:

- Rows are distributed across all AMPs (and accessed via the Primary Index) based upon HBN (Hash Bucket Number) portion of the Row Hash.
- At the AMP level, rows are first ordered by their partition number.
- Within the partition, data rows are logically stored in Row ID sequence.

A new term is associated with row partitioned tables. The **Row Key** is a combination of the Partition # and the Row Hash. The term Row Key will appear in EXPLAIN reports.

Primary Index Access – Row Partitioned Table

SQL with primary index values and data, or SQL expressions that include partition related values.



Notes:

1. Within the AMP, rows are ordered first by their partition number
2. Within each partition, rows are logically stored in row hash and uniqueness value sequence

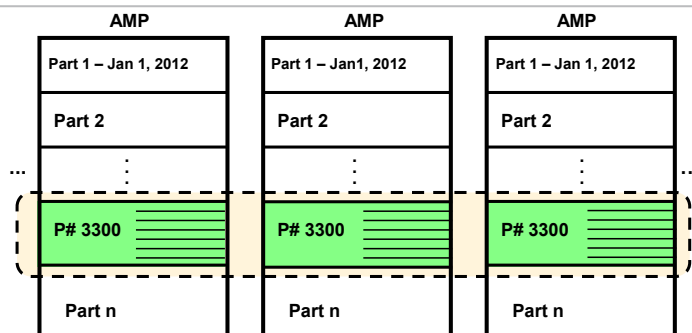
The process to locate a data row(s) via in a row partitioned table is similar to the process in retrieving data rows with a table without row partitioning – a process described earlier. If the SQL request provides data about columns associated with the partitions, then the PARSER will include specific partition information in the request.

- The key to remember is that a specific Row Hash value can be found in different partitions on the AMP. The Partition Number, Row Hash, and Uniqueness Value are needed to uniquely identify a row in a PPI-based table.
- A Row Hash and Uniqueness Value combination is only unique within a partition of a row partitioned table. The same Row Hash and Uniqueness Value combination can be present in different partitions (e.g., x'068117A0').

You can have the same hash and uniqueness values in different partitions. **The uniqueness value within a partition is unique, not across partitions.** When a row hash is added to a partition, the uniqueness value is incremented by 1 and Teradata only looks at that partition. The important concept to understand is that for a partitioned table, each row is uniquely identified by the combination of **part#**, **row hash**, and **uniqueness value**.

Assume a row partitioned table with a NUPI. In this case, assume the NUPI is (orderid) and the table has RANGE_N partitioning by orderdate. Even though the values of orderid may be unique, you cannot create the table with orderid as a UPI. The reason is this – if Teradata allowed you to define orderid as unique, then each time you inserted a new order, Teradata would have to check every partition to make sure the new orderid did not already exist and this would kill performance (I/O to check every partition).

Access Using Partitioned Data



QUERY – row partitioned

```
SELECT *
FROM   Claim_RP
WHERE  claimdate =
       DATE '2021-01-13';
```

Explain plan

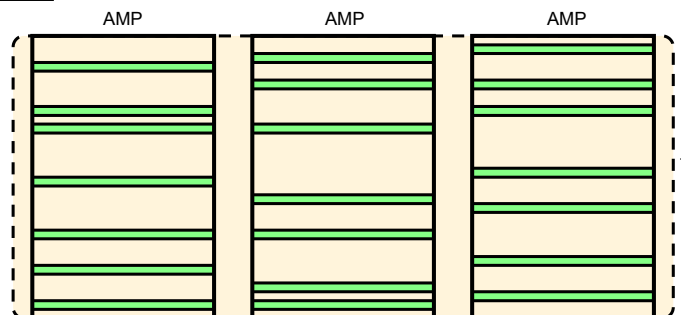
ALL-AMPs – Single Partition Scan with a Partition Read Lock. This date (partition) has 17,293 rows.
EXPLAIN estimated cost – 0.04 sec.

QUERY – not row partitioned

```
SELECT *
FROM   Claim
WHERE  claimdate =
       DATE '2021-01-13';
```

Explain plan

ALL-AMPs – Full Table Scan with a Table Read Lock – table has 30M rows.
EXPLAIN estimated cost – 16.11 sec.



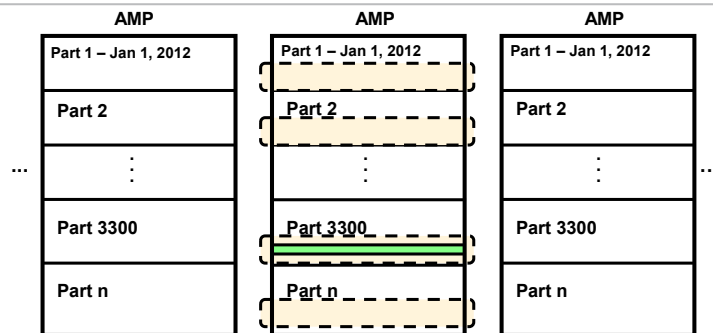
Both Claim_RP and Claim tables have 30,000,000 rows. Statistics for custid and claimdate were collected on both tables. In addition, claimid and PARTITION statistics were collected on the Claim_RP table.

The EXPLAIN text for these queries is shown below.

```
EXPLAIN      SELECT *
              FROM   Claim_RP
              WHERE  claimdate = DATE '2021-01-13';
```

- 1) First, we lock TFACT.Claim_RP in TD_MAP1 for read on a reserved RowHash in a single partition to prevent global deadlock.
 - 2) Next, **we lock TFACT.Claim_RP in TD_MAP1 for read on a single partition.**
 - 3) We do an all-AMPs RETRIEVE step in TD_MAP1 from a single partition of TFACT.Claim_RP with a condition of ("TFACT.Claim_RP.claimdate = DATE '2021-01-13'") with a residual condition of ("TFACT.Claim_RP.claimdate = DATE '2021-01-13'") into Spool 1 (group_amps), which is built locally on the AMPs. The size of Spool 1 is estimated with high confidence to be 17,293 rows (4,029,269 bytes). The estimated time for this step is 0.04 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 0.04 seconds.**

Access Using Primary Index



QUERY – row partitioned

```
SELECT *
FROM   Claim_RP
WHERE  claimid = 1028111;
```

Explain plan

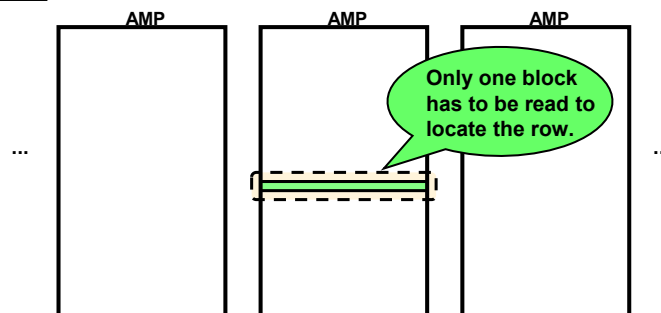
One AMP – All Partitions are probed
EXPLAIN estimated cost – 0.82 sec.

QUERY – not row partitioned

```
SELECT *
FROM   Claim
WHERE  claimid = 1028111;
```

Explain plan

One AMP – UPI Access
EXPLAIN estimated cost – 0.00 sec.



```
EXPLAIN SELECT * FROM Claim_RP
WHERE claimid = 1028111;
```

- 1) First, we do a single-AMP RETRIEVE step from all partitions of TFACT.Claim_RP by way of the primary index "TFACT.Claim_RP.claimid = 1028111" with a residual condition of ("TFACT.Claim_RP.claimid = 1028111") into Spool 1 (one-amp), which is built locally on that AMP. The size of Spool 1 is estimated with high confidence to be 1 row (233 bytes). The estimated time for this step is 0.82 seconds.
- > The contents of Spool 1 are sent back to the user as the result of statement 1. **The total estimated time is 0.82 seconds.**

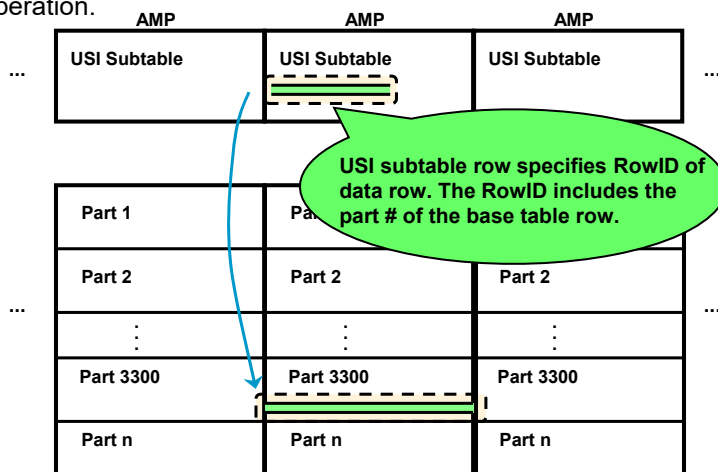
The table named Claim is similar to Claim_RP except it does not have a Partitioned Primary Index, but does have "claimid" as a UPI.

```
EXPLAIN SELECT * FROM Claim
WHERE claimid = 1028111;
```

- 1) First, we do a **single-AMP RETRIEVE step** from TFACT.Claim by way of the unique primary index "TFACT.Claim.claimid = 1028111" with no residual conditions. The estimated time for this step is 0.00 seconds.
- > The row is sent directly back to the user as the result of statement 1. **The total estimated time is 0.00 seconds.**

Place a USI on NUPI

- If the partitioning column(s) are not part of the Primary Index, the Primary Index cannot be unique (e.g., claimdate is not part of the PI).
- To maintain uniqueness on the Primary Index, you can create a USI on the PI (e.g., claimid). This is a two-AMP operation.



```
CREATE UNIQUE INDEX
  (claimid) ON Claim_RP;
```

```
SELECT  *
FROM    Claim_RP
WHERE   claimid = 1028111;
```

USI Considerations:

- Eliminate partition probing
- Row-hash locks
- 2-AMP operation
- Can only be used if values in PI column(s) are unique
- Will maintain uniqueness

```
CREATE UNIQUE INDEX (claimid) ON Claim_RP;
```

```
EXPLAIN SELECT *
FROM    Claim_RP
WHERE   claimid = 1028111;
```

- First, we do a **two-AMP RETRIEVE** step in TD_MAP1 from TFACT.Claim_RP by way of **unique index # 4** "TFACT.Claim_RP.claimid = 1028111" with no residual conditions. The estimated time for this step is 0.00 seconds.
- > The row is sent directly back to the user as the result of statement 1. The total estimated time is 0.00 seconds.

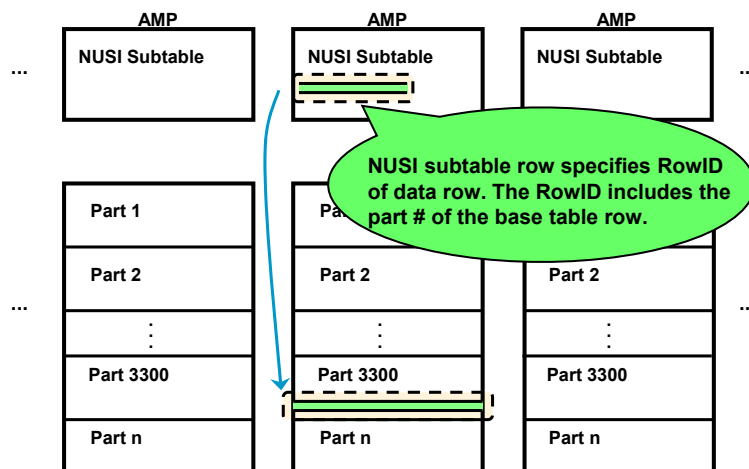
As an alternative, the SELECT can include the Primary Index values and the partitioning information. If you knew that the claimid of 1028111 occurred on 2021-01-13, you could include both claimid and claimdate in the query. This allows the PE to build a request that has the AMP scan a specific partition.

```
EXPLAIN SELECT *
FROM    Claim_RP
WHERE   claimid = 1028111
AND    claimdate = DATE '2021-01-13';
```

However, the user may not know the claimdate in order to include it in the query.

Place a NUSI on NUPI

- You can optionally create a NUSI on the same columns as the Primary Index (e.g., claimid). The PI may be unique or not.
- Optimizer generates a plan for a **single-AMP NUSI access** with **row-hash locking** (instead of table-level locking).



```
CREATE INDEX (claimid)
ON Claim_RP;
```

```
SELECT *
FROM Claim_RP
WHERE claimid = 1028111;
```

NUSI Considerations:

- Eliminate partition probing
- Row-hash locks
- 1-AMP operation
- Can be used with unique or non-unique PI columns
- Must be equality condition
- NUSI Single-AMP operation are only supported on row partitioned tables

You can use a NUSI on the same columns that make up the PI and actually get a single-AMP access operation. This feature only applies to a NUSI created on the same columns as a PI on row partitioned table. Additionally, instead of table level locks (typical NUSI), row hash locks will be used.

Reasons to choose a NUSI for your PI may include:

- The primary index is non-unique (cannot use a USI) and you need faster access than do a look-up or a probe into multiple partitions on a single AMP.
- TPT Update (MultiLoad protocol) can be used to load a table with a NUSI without using the MultiLoadX protocol.
- The access time for a USI and NUSI will be similar (each will access a subtable block) – however, the USI is a 2-AMP operation and requires BYNET message passing.

```
CREATE INDEX (claimid) ON Claim_RP;
EXPLAIN SELECT * FROM Claim_RP
WHERE claimid = 1028111;
```

- 1) First, we do a **single-AMP RETRIEVE step** from TFACT.Claim_RP by way of **index # 4** "TFACT.Claim_RP.claimid = 1028111" with no residual conditions into Spool 1 (one-amp), which is built locally on that AMP. The size of Spool 1 is estimated with high confidence to be 1 row (233 bytes). The estimated time for this step is 0.00 seconds.
- > 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.

Implementing PPI – PARTITION BY Option

The PRIMARY INDEX definition portion of a CREATE TABLE statement has a optional **PARTITION BY** option.

```
CREATE TABLE ...  
  [UNIQUE] PRIMARY INDEX (col1, col2, ...)  
    PARTITION BY <partitioning-expression>
```

Options for the *<partitioning-expression>* include:

- Range partitioning
- Conditional partitioning, modulo partitioning, and general expression partitioning
- Partitioning columns do not have to be columns in the primary index. If they aren't, then the primary index cannot be unique

Column(s) included in the partitioning expression are called the “partitioning column(s)”.

- Two functions, **CASE_N** and **RANGE_N**, are provided to simplify the creation of common partitioning schemes

Primary indexes can be partitioned or non-partitioned. A primary index is defined as part of the CREATE TABLE statement. The PRIMARY INDEX definition has a new option to create partitioned primary indexes.

A partitioned primary index (PPI) permits rows to be assigned to user-defined data partitions on the AMPs, enabling enhanced performance for range queries that are predicated on partitioning columns(s) values. The *<partitioning_expression>* is evaluated and Teradata determines the appropriate partition number or assignment.

The *<partitioning-expression>* is a general expression, allowing wide flexibility in tailoring the partitioning scheme to the unique characteristics of the table.

Limitations on PARTITION BY option include:

- Multiple columns from the table may be specified in the expression
- UNIQUE only allowed if all partitioning columns are included in the PI.
- Partitioning expression limited to approximately 16,000 characters.

One or more columns can make up the partitioning expression, although it is anticipated that for most tables one column will be specified. The partitioning column(s) can be part of the primary index, but are not required to be. The result of the partitioning expression must be a scalar value that is INTEGER or can be cast to INTEGER. Most deterministic functions can be used within the expression. The expression must not require character or graphic comparisons, although character or graphic columns can be used in some circumstances.

Partitioning with CASE_N and RANGE_N

The <partitioning expression> may use one of the following functions to help define partitions.

- CASE_N
- RANGE_N

Use of CASE_N results in the following:

- Evaluates a list of conditions and returns the position of the first condition that evaluates to TRUE
- Result is the data row being placed into a partition associated with that condition
- Note: Patterned after SQL CASE expression

Use of RANGE_N results in the following:

- The expression is evaluated and is mapped into one of a list of specified ranges
- Ranges are listed in increasing order and must not overlap with each other
- Result is the data row being placed into a partition associated with that range

NO CASE, NO RANGE, and UNKNOWN options are also available.

For many tables, there is no suitable column that lends itself to direct usage as a partitioning column. For these situations, the CASE_N and RANGE_N functions can be used to concisely define partitioning expressions.

The PARTITION BY phrase requires a partitioning expression that determines the partition assignment of a row. You can use the CASE_N function to construct a partitioning expression such that a row with any value or NULL for the partitioning column is assigned to a partition.

The CASE_N function is patterned after the SQL CASE expression. It evaluates a list of conditions and returns the position of the first condition that evaluates to TRUE, provided that no prior condition in the list evaluates to UNKNOWN. The returned value will map directly into a partition number.

Another option is to use the RANGE_N function to construct a partitioning expression with a list of ranges such that a row with any value or NULL for the partitioning column is assigned to a partition.

Partitioning with RANGE_N – Example

For example, partition a Claim table by "claimdate".

```
CREATE TABLE Claim_RP
( claimid      INTEGER NOT NULL
, custid       INTEGER NOT NULL
, claimdate    DATE     NOT NULL
... )
```

```
PRIMARY INDEX (claimid)
PARTITION BY RANGE_N (claimdate BETWEEN
DATE '2012-01-01' AND DATE '2021-12-31' EACH INTERVAL '1' DAY, NO RANGE);
```

This table has 3654 logically defined partitions. As INSERTs place new rows into the Claim table, the date is evaluated and the rows are placed into the appropriate partitions.

```
INSERT INTO Claim_RP VALUES (1028111,1009,'2012-01-13',...); placed in partition #13
INSERT INTO Claim_RP VALUES (24286111,1020,'2021-01-13',...); placed in partition #3300
INSERT INTO Claim_RP VALUES (45451111,1025,'2022-01-01',...); placed in partition #3654
INSERT INTO Claim_RP VALUES (100039,1009, NULL,...); Error 3811 – NOT NULL violation
```

If the table did not have the NO RANGE partition defined, then the following error occurs:

```
INSERT INTO Claim_RP VALUES (45451111,1025,'2022-01-01',...);
(5728 – Partitioning violation)
```

Note: claimid must be defined as a NUPI because claimdate is not part of PI.

One of most common partitioning expression is to use RANGE_N partitioning to partition the table based on a group of dates (e.g., month partitions). A range is defined by a starting boundary and an optional ending boundary. If an ending boundary is not specified, the range is defined by its starting boundary, inclusively, up to but not including the starting boundary of the next range. The list of ranges must specify ranges in increasing order, where the ending boundary of a range is less than the starting boundary of the next range.

RANGE_N limitations include:

- Multiple test values are not allowed in a RANGE_N function.
- Range value and range size in a RANGE_N function must be constant.
- Ascending ranges only and ranges must not overlap with each other.

For example, this CREATE TABLE statement can be used to establish the monthly partitioning.

```
CREATE SET TABLE Claim_RP
(claimid      INTEGER NOT NULL
, custid       INTEGER NOT NULL
, claimdate    DATE     NOT NULL
:
PRIMARY INDEX (claimid)
PARTITION BY RANGE_N (claimdate
BETWEEN DATE '2012-01-01' AND DATE '2021-12-31'
EACH INTERVAL '1' MONTH, NO RANGE);
```

RANGE_N – Example with Varying Intervals

- This example places current and history sales data into one table
- Current year data is partitioned on a more granular basis (daily) while historical sales data is placed into monthly partitions
- Partitions of varying intervals can be created for a table

```
CREATE TABLE Sales_and_SalesHistory
  (storeid      INTEGER NOT NULL,
   itemid       INTEGER NOT NULL,
   salesdate    DATE FORMAT 'YYYY-MM-DD',
   totalrevenue DECIMAL(9,2),
   totalsold    INTEGER,
   note         VARCHAR(256))
PRIMARY INDEX (storeid, itemid)
PARTITION BY RANGE_N (salesdate BETWEEN
  DATE '2012-01-01' AND DATE '2020-12-31' EACH INTERVAL '1' MONTH,
  DATE '2021-01-01' AND DATE '2021-12-31' EACH INTERVAL '1' DAY);
```

To partition by week, the following partitioning can be used.

```
PARTITION BY RANGE_N (salesdate BETWEEN
  DATE '2012-01-01' AND DATE '2012-12-31' EACH INTERVAL '7' DAY,
  DATE '2013-01-01' AND DATE '2013-12-31' EACH INTERVAL '7' DAY,
  :
  :
```

This example illustrates that a table can be partitioned with different size intervals. The current Sales data and Sales History data are placed in the same table. It typically is not practical to create a partitioning expression as shown in example #2, but the example is included to show the flexibility that you have with the partitioning expression.

For example, you may decide to partition the Sales History by month and the current sales data by day. You may want to do this if users frequently access the Sales History data with range constraints, resulting in full table scans. It may be that users access the current year data frequently looking at data for a specific day. The example on this slide partitions the years 2012 to 2020 by month and the year 2021 by day.

One option may be to partition by week as follows:

```
PARTITION BY RANGE_N (salesdate BETWEEN
  DATE '2012-01-01' AND DATE '2012-12-31' EACH INTERVAL '7' DAY,
  DATE '2013-01-01' AND DATE '2013-12-31' EACH INTERVAL '7' DAY,
  :
  :
  DATE '2021-01-01' AND DATE '2021-12-31' EACH INTERVAL '7' DAY);
```

Special Partitions with CASE_N and RANGE_N

The CASE_N and RANGE_N can place rows into specific-use partitions when ...

- the expression doesn't meet any of the CASE and RANGE expressions
- the expression evaluates to UNKNOWN
- two partition numbers are reserved even if the above options are not used

The keywords used to define two specific-use partitions are:

- **NO CASE (or NO RANGE) [OR UNKNOWN]**
 - If this option is used, then a specific-use partition is used when the expression isn't true for any case (or is out of range)
 - If OR UNKNOWN is included with the NO CASE (or NO RANGE), then UNKNOWN expressions are also placed in this partition
- **UNKNOWN**
 - If this option is specified, a different specific-use partition is used for unknowns
- **NO CASE (or NO RANGE), UNKNOWN**
 - If this option is used, then two separate specific-use partitions are used when the expression isn't true for any case (or is out of range) and different special partition is used for NULLs

The keywords, NO CASE (or NO RANGE) [OR UNKNOWN] and UNKNOWN are used to define the specific-use partitions. Even if these options are not specified with the CASE_N (or RANGE_N) expressions, these two specific-use partitions are still reserved in the event the ALTER TABLE command is later used to add these options.

If it is necessary to test a CASE_N condition directly as NULL, it needs to be the first condition listed.

```
PARTITION BY CASE_N
(col3 IS NULL,
 col3 < 10,
 col3 < 100, NO CASE OR UNKNOWN)

INSERT INTO TabA_RP VALUES (1, 'A', NULL, DATE);
INSERT INTO TabA_RP VALUES (2, 'B', 5, DATE);
INSERT INTO TabA_RP VALUES (3, 'C', 500, DATE);

SELECT PARTITION AS "Part #", COUNT(*) FROM TabA_RP
GROUP BY 1 ORDER BY 1;
```

<u>Part #</u>	<u>Count(*)</u>
1	1
2	1
4	1

Partitioning with CASE_N – Example

- Partition the data based on total revenue for the products
- The NO CASE and UNKNOWN options allow for totalrevenue >=100,000 or “unknown revenue”
- A UPI is **NOT** allowed because the partitioning columns are **NOT** part of the PI

```
CREATE TABLE SalesRevenue
(storeid          INTEGER NOT NULL,
itemid           INTEGER NOT NULL,
salesdate        DATE FORMAT 'YYYY-MM-DD',
totalrevenue     DECIMAL(9,2),
totalsold        INTEGER,
note             VARCHAR(256))
PRIMARY INDEX (storeid, itemid, salesdate)
PARTITION BY CASE_N
( totalrevenue < 2000 ,
  totalrevenue < 4000 ,
  totalrevenue < 6000 ,
  totalrevenue < 8000 ,
  totalrevenue < 10000 ,
  totalrevenue < 20000 ,
  totalrevenue < 50000 ,
  totalrevenue < 100000 ,
  NO CASE ,
  UNKNOWN );
```

This example illustrates the capability of partitioning based upon conditions (CASE_N).

For example, assume a table has a total revenue column, defined as decimal. The table could be partitioned on that column, so that low revenue products are separated from high revenue products. The partitioning expression could be written as shown on this slide. In this example, 8 partitions are defined for total revenue values up to 100,000. Two additional partitions are defined – one for revenues greater than 100,000 and another for unknown revenues (e.g., NULL).

Note: Starting with Teradata 13.10, CURRENT_DATE and/or CURRENT_TIMESTAMP within partitioning expressions is allowed. However, it is recommended to NOT use these in a CASE expression for a row partitioned table. Why? In this case, all rows are scanned during reconciliation.

SQL Use of PARTITION Key Word

The **PARTITION** SQL key word can be used to return partition numbers that have rows and a count of rows that are currently located in partitions of a table.

SQL:

```
SELECT      PARTITION AS "Part #",
            COUNT(*) AS "Row Count"
FROM        SalesRevenue
GROUP BY    1
ORDER BY    1;
```

Result:

Part #	Row Count
1	169690
2	163810
3	68440
4	33490
5	18640
6	27520
7	1760

```
totalrevenue < 2,000
totalrevenue < 4,000
totalrevenue < 6,000
totalrevenue < 8,000
totalrevenue < 10,000
totalrevenue < 20,000
totalrevenue < 50,000
```

SQL - insert two rows:

```
INSERT INTO SalesRevenue VALUES (1003, 5052, CURRENT_DATE, 102000, 113, NULL);
INSERT INTO SalesRevenue VALUES (1003, 5053, CURRENT_DATE, NULL, NULL, NULL);
```

SQL (same as above):

```
SELECT      PARTITION AS "Part #",
            COUNT(*) AS "Row Count"
FROM        SalesRevenue
GROUP BY    1
ORDER BY    1;
```

Result:

Part #	Row Count
1	169690
2	163810
3	68440
4	33490
5	18640
6	27520
7	1760
9	1
10	1

same as above

NO CASE
UNKNOWN

This slide contains an example of using the key word **PARTITION** to determine the number of rows there are in physical partitions. This example is based on the **Sales_Revenue** table as defined on the previous page.

The following table shows the same result as this slide, but also identifies the internal partition #'s as allocated.

Part #	Row Count	
1	169690	internally mapped to partition #3
2	163810	internally mapped to partition #4
3	68440	internally mapped to partition #5
4	33490	internally mapped to partition #6
5	18640	internally mapped to partition #7
6	27520	internally mapped to partition #8
7	1760	internally mapped to partition #9

Note that this table does not have any rows with a **total_revenue** value greater than 50,000 and less than 100,000. Partition #8 was not assigned. Also, there are no rows with a **total_revenue** $\geq 100,000$ or **NULL** because the **NO CASE** and **UNKNOWN** partitions are not used.

ALTER TABLE – Row Partitioned Tables

The ALTER TABLE statement has enhancements for a partitioned table to modify the partitioning properties of the primary index for a table.

For populated tables, ...

- You are permitted to drop and/or add ranges at the “ends” of existing partitions on a range-partitioned table
 - ALTER TABLE includes ADD / DROP RANGE options
 - You can also [add or drop special partitions](#) (NO RANGE or UNKNOWN)
 - You [cannot](#) drop all the ranges
- Possible use – drop ranges for the oldest dates and prepare additional ranges for future dates
- The set of primary index columns [cannot](#) be altered for a populated table

To use ALTER TABLE for any purpose other than the above situations, the table must be empty.

The permitted changes for populated tables are to drop ranges at the ends or to add ranges at the ends. For example, a common use of this capability would be to drop ranges for the oldest dates, and to prepare additional ranges for future dates, among other things.

Limitations with ALTER TABLE:

- Primary Index of a non-empty table may not be altered
- Partitioning of a non-empty table is generally limited to altering the “ends”.
- If a table has **Delete triggers**, they must be disabled if the WITH DELETE option is specified.
- If a save table has **Insert triggers**, they must be disabled if the WITH INSERT option is specified.

For empty row partitioned tables, the ALTER TABLE statement can be used to do the following:

- Remove or add partitioning for a partitioned table
- Change the columns that comprise the primary index
- Change a unique primary index to non-unique or vice-versa

Assume you have a populated data table (and the table is quite large) defined with a “non-unique partitioned primary index” and all of the partitioning columns are part of the PI. You realize that the table should have been defined with a “unique partitioned primary index”, but the table is already loaded with data. Here is a technique to convert this NUPI into a UPI:

- CREATE a USI on the columns making up the PI. ALTER the table, effectively changing the NUPI to a UPI, and the software will automatically drop the USI.

ALTER TABLE – NO RANGE is Defined

Assume a table is partitioned as follows:

```
PARTITION BY RANGE_N (salesdate BETWEEN
DATE '2012-01-01' AND DATE '2021-12-31' EACH INTERVAL '1' MONTH, NO RANGE);
```

To drop/add partitions from the table definition:

```
ALTER TABLE Sales MODIFY
DROP RANGE BETWEEN DATE '2012-01-01' AND DATE '2012-12-31'
ADD RANGE BETWEEN DATE '2022-01-01' AND DATE '2022-12-31' EACH INTERVAL '1' MONTH
;
```

If WITH DELETE or WITH INSERT is included, it is effectively ignored.

```
ALTER TABLE Sales MODIFY
DROP RANGE BETWEEN DATE '2012-01-01' AND DATE '2012-12-31'
ADD RANGE BETWEEN DATE '2022-01-01' AND DATE '2022-12-31' EACH INTERVAL '1' MONTH
WITH INSERT INTO SalesHistory;
```

Notes assuming NO RANGE is defined:

- If dropped partitions have data, then the data is moved to the NO RANGE partition
- If WITH DELETE or WITH INSERT syntax is included, it is effectively ignored
- The WITH DELETE or WITH INSERT syntax is not needed and is confusing if included
- If new partitions are added, the NO RANGE partition is checked and rows that apply to the newly added partitions are moved from NO RANGE into the new partitions
- If you want to delete the data, use SQL DELETE before using ALTER TABLE
- General recommendation is to not use NO RANGE unless it is specifically needed

When using the ALTER TABLE to drop existing ranges, if the table has NO RANGE defined, it is not necessary to include the WITH DELETE or WITH INSERT syntax.

If NO RANGE is defined and the WITH DELETE or WITH INSERT syntax is used, the WITH DELETE or WITH INSERT is effectively ignored.

This simply means the rows are not deleted or copied to another table.

In this case (table has a NO RANGE partition), rows are copied from dropped partitions into the NO RANGE partition. If new partitions are added, the NO RANGE partition is checked and rows that apply to the newly added partitions are moved from NO RANGE into the new partitions.

Note: When dropping Ranges, the word "Interval" is optional.

```
ALTER TABLE Sales MODIFY
DROP RANGE BETWEEN DATE '2012-01-01' AND DATE '2012-12-31'
EACH INTERVAL '1' MONTH
ADD RANGE BETWEEN DATE '2021-01-01' AND DATE '2021-12-31'
EACH INTERVAL '1' MONTH;
```

You can add the NO RANGE and/or UNKNOWN partitions to an already partitioned table.

```
ALTER TABLE Sales MODIFY ADD RANGE NO RANGE OR UNKNOWN;
```

ALTER TABLE – NO RANGE is Not Defined

Assume a table is partitioned as follows:

To drop/add partitions **and DELETE** the old data:

```
PARTITION BY RANGE_N (salesdate BETWEEN
    DATE '2012-01-01' AND DATE '2021-12-31' EACH INTERVAL '1' MONTH);
```

```
ALTER TABLE Sales MODIFY PRIMARY INDEX
DROP RANGE BETWEEN DATE '2012-01-01' AND DATE '2012-12-31'
ADD RANGE BETWEEN DATE '2022-01-01' AND DATE '2022-12-31' EACH INTERVAL '1' MONTH
WITH DELETE;
```

To drop/add partitions **and COPY** the old data to another table:

```
ALTER TABLE Sales MODIFY
DROP RANGE BETWEEN DATE '2012-01-01' AND DATE '2012-12-31'
ADD RANGE BETWEEN DATE '2022-01-01' AND DATE '2022-12-31' EACH INTERVAL '1' MONTH
WITH INSERT INTO SalesHistory;
```

Notes assuming **NO RANGE** is not defined:

- The WITH DELETE or WITH INSERT syntax is required if the partitions have data
- Dropping partition ranges means the data in the dropped partitions is also deleted from the table
- The MODIFY syntax does not require you to also include the words PRIMARY INDEX
- The SalesHistory table must already exist when using the WITH INSERT option
- SalesHistory may optionally be partitioned; recommended for the table to be empty and have a matching PI as the Sales table for performance

The DROP RANGE option is used to drop a range set from the RANGE_N function on which the partitioning expression for the table is based. You can only drop ranges if the partitioning expression for the table is derived only from a RANGE_N function.

You can drop empty partitions without specifying the WITH DELETE or WITH INSERT option. However, if the partitions to be dropped have data, you must include either the WITH DELETE or WITH INSERT syntax.

Some of the ALTER TABLE statement options include:

DROP RANGE WHERE *conditional_expression* – a conditional partitioning expression used to drop a range set from the RANGE_N function on which the partitioning expression for the table is based.

You can only drop ranges if the partitioning expression for the table is derived only from a RANGE_N function.

You must base *conditional_partitioning_expression* on the system-derived PARTITION column.

ALTER TABLE

ALTER TABLE Sales MODIFY

Statement	No Range Partition Defined	No Range Partition Not Defined
<code>DROP RANGE BETWEEN DATE '2012-01-01' AND DATE '2012-12-31'</code>	If dropped partitions have data, then the data is moved to the NO RANGE partition	The WITH DELETE or WITH INSERT syntax is required if the partitions have data.
<code>DROP RANGE BETWEEN DATE '2012-01-01' AND DATE '2012-12-31' WITH DELETE;</code>	Same as above and the WITH DELETE syntax is effectively ignored	Dropping partition ranges means the data in the dropped partitions is also deleted from the table
<code>DROP RANGE BETWEEN DATE '2012-01-01' AND DATE '2012-12-31' WITH INSERT INTO SalesHistory;</code>	Same as above and the WITH INSERT syntax is effectively ignored	Partition definition is dropped and data is moved into the SalesHistory table The table must already exist when using the WITH INSERT option.
<code>ADD RANGE BETWEEN DATE '2022-01-01' AND DATE '2022-12-31' EACH INTERVAL '1' MONTH</code>	If new partitions are added, the NO RANGE partition is checked Rows that apply to the newly added partitions are moved from NO RANGE into the new partitions.	New Partition is added

- If you want to delete the data, use SQL DELETE before using ALTER TABLE

This slide summarizes up both the NO_RANGE case up when it is defined and not defined

Multilevel Partitioning

- From Teradata 14.0 onwards we allow a maximum of 9.223 quintillion partitions and a total of 62 levels
Query – Compare District 25 revenue for Week 6 vs. same period last year?



You can use a multilevel row partitioned table to improve query performance via partition elimination, either at each of the partition levels or by combining all of them. A multilevel partitioned table provides multiple access paths to the rows in the base table.

This example shows the benefit of using a multilevel partitioned table. Assume that there are 50 districts and the single partitioning is by month for 2 years and the multilevel partitioning is first by month (24 months) and sub-partitioned by 50 districts. In the single level partitioning, there are a total of 24 partitions and in the multilevel, there are a total of 1200 partitions.

The following list describes the various access methods with multilevel partitioning:

- If there is an equality constraint on the primary index and there are constraints on the partitioning columns such that access is limited to a single partition at each level, access is as efficient as with a non-partitioned table.
- With constraints defined on the partitioning columns, performance of a primary index access can approach the performance of a non-partitioned table depending on the extent of partition elimination that can be achieved.
- Access by means of equality constraints on the primary index columns that does not also include all the partitioning columns, and without constraints defined on the partitioning columns, might not be as efficient as access with a non-partitioned table. The efficiency of the access depends on the number of non-empty sub-partitions at the lowest level of the partition hierarchy.
- With constraints on the partitioning columns of a partitioning expression such that access is limited to a subset of, say n percent, of the partitions for that level, the scan of the data is reduced to about n percent of the time required by a full-table scan.

Multilevel Partitioning – Example

For example, partition Claim table by "claimdate" and "stateid".

```
CREATE TABLE Claim_MLRP
( claimid      INTEGER NOT NULL
, custid       INTEGER NOT NULL
, claimdate    DATE NOT NULL
, stateid     BYTEINT NOT NULL
, ... )
PRIMARY INDEX (claimid)
PARTITION BY (
/* First level of partitioning */
  RANGE_N (claimdate BETWEEN
    DATE '2012-01-01' AND DATE '2021-12-31' EACH INTERVAL '1' DAY),
/* Second level of partitioning */
  RANGE_N (stateid BETWEEN 1 AND 75 EACH 1) )
UNIQUE INDEX (claimid);
```

```
SELECT ...
FROM   Claim_MLRP C
JOIN   States S
ON     C.stateid = S.stateid
WHERE  S.statename = 'California'
AND    C.claimdate = DATE '2021-01-13';
```

Notes:

- For multi-level row partitioned table, the set of partitioning expressions must be enclosed in parentheses
- Each level must define at least two partitions for a multi-level row partitioned table
- The number of defined partitions in this example is (3653 * 75) or 273,975

You create a multilevel partitioned table by specifying two or more partitioning expressions, where each expression must be defined using either a `RANGE_N` function or a `CASE_N` function exclusively. The system combines the individual partitioning expressions internally into a single partitioning expression that defines how the data is partitioned on an AMP.

The first partitioning expression is the highest level partitioning. Within each of those partitions, the second partitioning expression defines how each of the highest-level partitions is sub-partitioned. Within each of those second-level partitions, the third-level partitioning expression defines how each of the second level partitions is sub-partitioned. Within each of these lowest level partitions, rows are ordered by the row hash value of their primary index and their assigned uniqueness value.

You define the ordering of the partitioning expressions in your `CREATE TABLE` SQL text, and that ordering implies the logically ordering by RowID. Because the partitions at each level are distributed among the partitions of the next higher level in the hierarchy, scanning a partition at a certain level requires skipping some internal partitions.

Partition expression order does *not* affect the ability to eliminate partitions, but *does* affect the efficiency of a partition scan. As a general rule, this should not be a concern if there are many rows, which implies multiple data blocks, in each of the partitions.

There are two levels of partitioning defined in this example. The first level defines 3653 partitions and the second defines 75 partitions. Therefore, the total number of partitions for the combined partitioning expression is the product of $3653 * 75$, or 273,975.

ADD Partitions Option – Example

The ADD option allows you to define additional partitions for a sub-level – these partitions can be added at a later time via the ALTER table command.

```
CREATE TABLE Claim_MLRP2
( claimid      INTEGER NOT NULL,
  custid       INTEGER NOT NULL,
  claimdate    DATE NOT NULL FORMAT 'YYYY-MM-DD',
  :
  district     SMALLINT,
  :
  PRIMARY INDEX (claimid)
  PARTITION BY (
    RANGE_N (claimdate BETWEEN DATE '2012-01-01' AND DATE '2021-12-31'
              EACH INTERVAL '1' DAY, NO RANGE),
    RANGE_N (district BETWEEN 1 AND 100 EACH 1,
              NO RANGE OR UNKNOWN) ADD 200)
  UNIQUE INDEX (claimid);
```

An INSERT of a new claim into district 101 will cause the new row to be placed in the NO RANGE partition.

The following ALTER TABLE can be used to add an additional 75 district partitions.

```
ALTER TABLE Claim_MLRP2 MODIFY ADD RANGE#L2 BETWEEN 101 AND 175 EACH 1;
```

Note the syntax to add or drop ranges for sub-levels (RANGE#Ln).

If the table or join index is defined with multiple partitioning levels, you can (and possibly should consider) explicitly specify the number of partitions per partitioning level using the ADD option.

This slide has an example of using the ADD option for the "district" sub-partition.

Note that the following ALTER TABLE will fail and return the 5734 error message:

```
ALTER TABLE Claim_MLRP2 MODIFY
  ADD RANGE#L2 BETWEEN 1 AND 175 EACH 1;
```

This fails because the ranges (i.e., districts) 1 to 100 already exist in the table.

If you want to change the special partitions for District from NO RANGE OR UNKNOWN to just NO RANGE, the following SQL can be used:

```
ALTER TABLE Claim_MLRP2 MODIFY
  DROP RANGE#L2 NO RANGE OR UNKNOWN
  ADD RANGE#L2 NO RANGE
  WITH DELETE;
```


Character Partitioning

In this example, three levels of partitioning are defined.

```
CREATE TABLE Claim_MLRP3
(claimid      INTEGER      NOT NULL,
 custid       INTEGER      NOT NULL,
 claimdate    DATE          NOT NULL,
 city         VARCHAR(30)   NOT NULL,
 statecode    CHAR(2)       NOT NULL,
 :            )
PRIMARY INDEX (claimid)
PARTITION BY
( RANGE_N
  (claimdate BETWEEN DATE '2012-01-01' AND DATE '2021-12-31' EACH INTERVAL '1' DAY, NO RANGE)
, RANGE_N
  (statecode BETWEEN 'A', 'D', 'I', 'N', 'T' AND 'ZZ', NO RANGE)
, RANGE_N
  (city BETWEEN 'A', 'C', 'E', 'G', 'I', 'K', 'M', 'O', 'Q', 'S', 'U', 'W' AND 'ZZ', NO RANGE)
)
UNIQUE INDEX (claimid);
```

The following queries will benefit from this type of partitioning.

- SELECT * FROM Claim_MLRP3 WHERE statecode = 'OH';
- SELECT * FROM Claim_MLRP3 WHERE statecode = 'GA' AND city LIKE 'a%';
- SELECT * FROM Claim_MLRP3 WHERE claimdate = DATE '2021-08-24' AND city LIKE 'b%';

In this example, the Claim table is first partitioned by claimdate (daily intervals). Claimdate is then sub-partitioned by state codes. State codes are then sub-partitioned by the first two letters of a city name. The special partitions of NO RANGE and UNKNOWN are defined at the claimdate, statecode, and city levels.

The following queries will benefit from this type of partitioning.

```
SELECT * ... MLPP13 WHERE statecode = 'GA' AND city LIKE 'a%';
SELECT * ... WHERE claimdate = '2021-08-24' AND city LIKE 'b%';
```

The session mode when these tables were created and when these queries were executed was Teradata mode (BTET). Teradata mode defaults to "not case specific". The session collation in effect when the character partitioning was created determines the ordering of data used to evaluate the partitioning expression.

The default case sensitivity in effect when the character partitioning is created will also affect the ordering of character data for the table.

- Default case sensitivity of comparisons involving character constants is influenced by the **session mode**.
 - Teradata Mode (BTET) is NOT CASESPECIFIC
 - ANSI mode is CASESPECIFIC
- If any expression in the comparison is case specific, then the comparison is case sensitive.

Summary

Now that you have completed this course, you will be able to:

- Describe the components that comprise a Row ID in a row partitioned table
- List two advantages of partitioning a table
- Identify partition definition on Numeric and Character data types
- Create single-level and multilevel partitioned tables
- Use the PARTITION key word to display partition information
- Use ALTER TABLE to add and drop ranges from a row partitioned table

NOTE: It is recommended that students go through the official publications to get a deeper understanding of various use cases.

The customer (e.g., DBA, Database Designer, etc.) has a flexible and powerful tool to structure tables to allow automatic optimization of frequently used queries. This feature is effectively “row partitioning”. This feature allows tables to be partitioned on columns of interest, while retaining the traditional use of the primary index (PI) for data distribution and efficient access when the PI values are specified in the query.

This slide contains a summary of the key customer benefits that can be obtained by using row partitioning.

Whether and how to partition a table is a physical design choice.

A well-chosen partitioning scheme may be able to turn many frequently run queries into partial-table scans instead of full-table scans, with much improved performance.

However, understand that there are trade-off considerations that must be understood and carefully considered to get the most benefit from the row partitioning feature.



Module 3: Partitioned Primary Index Bring Up JupyterHub

teradata.

Let's now do the lab together



Thank you.

teradata.

©2023 Teradata



Module 4: Join Processing

Teradata Vantage MasterClass

Copyright © 2007–2023 by Teradata. All Rights Reserved.

Objectives

After completing this module, you will be able to:

- Identify and describe different kinds of join plans
- Describe join plan strategies
 - Merge Join
 - Nested Join
 - Hash Join
 - Product Join
 - Exclusion Merge Join
- Describe different types of join strategies that may be used with row partitioned tables



Revision – Join Syntax

Teradata supports the ANSI join syntax which provides outer join options.

```
SELECT      cname [, cname , ...]
FROM        tname [aname]
           [INNER] JOIN
           LEFT [OUTER] JOIN
           RIGHT [OUTER] JOIN
           FULL [OUTER] JOIN
           CROSS JOIN
           tname [aname]
           ON  condition ;
```

- A two-table join condition references two different tables
- A self-join condition references two alias names for one table

Where:

cname	Column or expression name
tname	Table or view name
aname	Alias for table or view name
condition	Criteria for the join

INNER JOIN	All matching rows
LEFT OUTER JOIN	Table to the left is used to qualify; table on the right has nulls when rows do not match
RIGHT OUTER JOIN	Table to the right is used to qualify; table on the left has nulls when rows do not match
FULL OUTER JOIN	Both tables are used to qualify and extended with nulls
CROSS JOIN	Product join or Cartesian product join

This slide shows all syntax options for joins using the ANSI standard join conventions.

Teradata – How Joins Work

Join Main Rule

As part of a JOIN, the rows that need to be compared between 2 or more tables must lie on the same AMP to be joined.

- If the rows to be joined are not on the same AMP, Teradata uses one of the following approaches to make it happen: -
 - Redistribution of one or more tables in spool
 - Duplicate the smaller table on all amps
- Join Processing never physically moves rows in tables, it does all its work in spool

Typical Relational (or SQL) join types:

- Inner
- Outer
 - Left, Right, or Full
- Self-Join
- NOT IN (Exclusion)
- Cross (Product)
 - Cartesian

Typical Teradata Join Plans:

- Merge Join
- Product Join
- Hash Join
- Nested Join
- Exclusion Join
- Inclusion Join
- RowID Join
- Self-Join

At a high level, for Joins in Teradata to be executed, rows that have to be joined (or compared) need to be on the same AMP.

This could sometimes be confusing for people to understand. The important thing to recall is that you join 2 or many tables on – “**Joining columns**”. When comparing data across the columns from 2 tables, the data from these rows from 2 tables need to lie on the same AMP for join comparisons to be able to be executed in parallel.

There are 2 ways we get rows to lie on the same AMP, as mentioned above – Redistribution / Duplication.

Join Process – How the Optimizer Minimizes Spool Usage

teradata.

1st Example: Select from Dept #1025

```
SELECT    E.empnum  
          E.lastname,  
          E.firstname,  
          P.checknum  
FROM      Employee E  
JOIN      Paycheck P  
ON        E.empnum = P.empnum  
WHERE     E.deptnum = 1025;
```

Projection List

Explicit Join

Join Condition

Set Condition

The Optimizer minimizes spool size before the join.

- Applies SET conditions first (WHERE).
- Only the necessary columns are used in Spool

2nd Example: Select from all departments

```
SELECT    E.*, P.checknum  
FROM      Employee E  
INNER JOIN Paycheck P  
ON        E.empnum = P.empnum;
```

In this example, all columns from Employee, but only two columns from Paycheck are needed

Assume the following:

- Employee (100,000 Rows, each row is 300 bytes)
- Paycheck (1,000,000 rows, each row is 250 bytes)

Therefore, the following applies to spool space needed. Employee – 100,000 x 300 bytes = 30 MB

- Employee – 100,000 x 300 bytes = 30 MB
- Paycheck – empnum (INTEGER), checknum (INTEGER) – 8 x 1,000,000 = 8 MB

Teradata often has to utilize Spool space to hold the copies of rows redistributed to do a Join. The Optimizer minimizes the amount of Spool required by:

- Projecting (copying) only those columns which the query requires.
- Doing single-table Set Selections first (qualifying rows).
- Putting only the Smaller Table into Spool whenever possible.

Teradata determines the Smaller Table by multiplying the number of qualified rows by the number of bytes in the columns to be projected (qualifying rows * projected column bytes). Which table this turns out to be is not always obvious.

Note: Non-equality Join Operators produce a (partial) Cartesian Product. Join operators should always be equality conditions. Set selection operators may be any condition.

Join Process – How the Optimizer Minimizes Row Selection

teradata.

- Column projection always precedes a join and row selection usually precedes a join
- Tip:** Reduce the rows that participate to improve join performance

3rd Example:

SHIPMENT		
5,000 Rows	SHIP#	...
PK/FK	PK,SA,NN	
Distinct Values	5000	
Max Rows/Value	1	
Max Rows/NULL	0	
Typical Rows/Value	1	
PI/SI	UPI	

PART			
30,000,000 Rows	PART#	...	SHIP#
PK/FK	PK,SA,NN		FK
Distinct Values	30M		5001
Max Rows/Value	1		200
Max Rows/NULL	0		29.5M
Typical Rows/Value	1		100
PI/SI	UPI		

```
SELECT ...
FROM Shipment S
INNER JOIN Part P
ON S.ship# = P.ship#;
```

```
SELECT ...
FROM Shipment S
INNER JOIN Part P
ON S.ship# = P.ship#
WHERE P.ship# > 150183;
```

Assuming Shipment.Ship# is defined as NOT NULL:

- The Optimizer automatically eliminates all NULLs for INNER joins in the Part table before doing join
- Even though the Part table has 30,000,000 rows, only 500,000 rows have values (match) and are joined to the Shipment table and output
- To further eliminate additional rows, add a WHERE condition that reduces the number of rows that participate in the join

Row Selection is a very important part of Join Processing. It is dependent on the conditions specified in the WHERE clause. If no Set Selection conditions exist, then all rows of both tables will participate in the Join.

Reducing the number of rows that participate will improve Join performance. The two examples on this slide illustrate this. The first SELECT statement has no Set Selection conditions; therefore, there is no Row Selection, and all rows from both tables will participate in the Join.

The second SELECT statement differs from the first one in that a Set Selection condition has been added. By specifying “WHERE Part.Ship# IS NOT NULL,” the performance of the Join will be greatly improved. The effect of this new condition is to reduce the number of participating Part Table rows from 30,000,000 to 500,000.

The Optimizer automatically eliminates NULLs for INNER Joins.

Note: The Part Number represents a “serial number” and is unique for a part. Therefore, a particular part number or serial number can only be on 1 shipment.

Join Technique 1 – Row Redistribution – Matching Indexes

teradata.

Employee

Enum	Name	Dept
PK		FK
UPI		
1	BROWN	200
2	SMITH	310
3	JONES	310
4	CLAY	400
5	PETERS	150
6	FOSTER	200
7	GRAY	310
8	BAKER	310
9	TYLER	450
10	CARR	450

Employee_Phone

Enum	Area_Code	Phone
	PK	
FK		
NUPI		
1	213	3241576
1	213	4950703
3	408	3628822
4	415	6347180
5	312	7463513
6	203	8337461
8	301	6675885
8	301	2641616

Primary Indexes match: no duplication or sorting needed

Example:

```

SELECT  E.Enum, E.Name, ...
FROM    Employee E
JOIN    Employee_Phone P
ON      E.Enum = P.Enum;

```

Employee rows hash distributed on Enum (UPI)

6 FOSTER 200	4 CLAY 400	1 BROWN 200	5 PETERS 150
8 BAKER 310	3 JONES 310	7 GRAY 310	2 SMITH 310
	9 TYLER 450		10 CARR 450

Employee_Phone rows hash distributed on Enum (NUPI)

6 203 8337461	4 415 6347180	1 213 3241576	5 312 7463513
8 301 2641616	3 408 3628822	1 213 4950703	
8 301 6675885			

The example on this slide illustrates joining two tables together with matching primary indexes. This strategy does not require any duplication or sorting of rows.

This example is pictured on a 4 AMP System. Teradata merely compares the rows already on the proper AMPs.

This join strategy will occur when the Join Column is the Primary Index of both tables. This is also referred to an AMP Local Join.

This Join is the most efficient as it does not involve any row redistribution.

Join Technique 1 – Row Redistribution – Non-Matching Indexes

Employee

Enum	Name	Dept
PK		FK
UPI		
1	BROWN	200
2	SMITH	310
3	JONES	310
4	CLAY	400
5	PETERS	150
6	FOSTER	200
7	GRAY	310
8	BAKER	310
9	TYLER	450
10	CARR	450

REDISTRIBUTE one side and SORT on join column row hash.

Example:

```
SELECT E.Enum, E.Name, D.Dept, D.Name FROM
Employee E
JOIN Department D
ON E.Dept = D.Dept;
```

Employee rows hash distributed on Enum (UPI)

6 FOSTER 200	4 CLAY 400	1 BROWN 200	5 PETERS 150
8 BAKER 310	3 JONES 310	7 GRAY 310	2 SMITH 310
	9 TYLER 450		10 CARR 450

Spool file after redistribution on Employee.Dept row hash

5 PETERS 150	7 GRAY 310	1 BROWN 200	4 CLAY 400
	3 JONES 310	6 FOSTER 200	
	8 BAKER 310	9 TYLER 450	
	2 SMITH 310	10 CARR 450	

Department rows hash distributed on Department.Dept (UPI)

150 PAYROLL	310 MFG.	200 FINANCE	400 EDUCATION
		450 ADMIN	

Department

Dept	Name
PK	
UPI	
150	PAYROLL
200	FINANCE
310	MFG.
400	EDUCATION
450	ADMIN

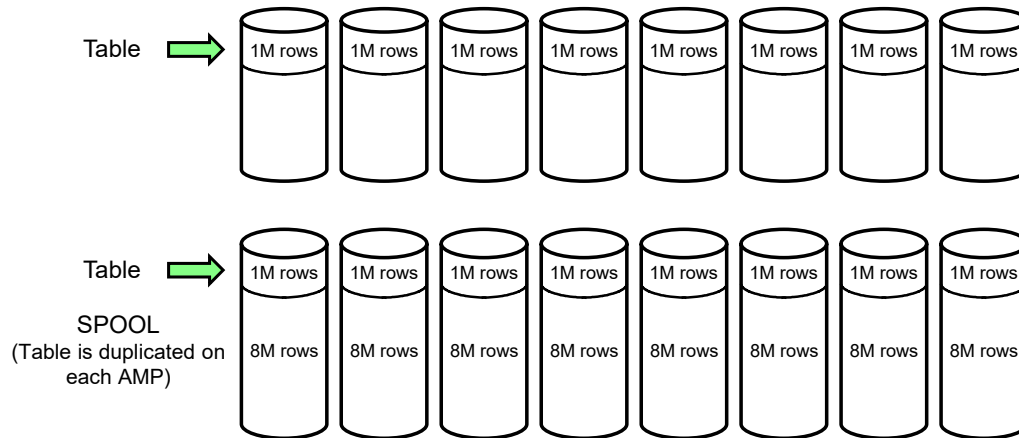
The example on this slide illustrates an example of redistributing one of the tables. This strategy consists of redistributing the rows of one table and sorting them on the Row Hash of the Join Column.

This example is pictured on a 4 AMP System. Teradata copies the employee rows into Spool and redistributes them on Employee.Dept Row Hash. The Join then occurs with the rows to be joined located on the same AMPs.

This strategy occurs when one of the tables is already distributed on the Join Column Row Hash. The Join Column is the PI of one, not both, of the tables.

Join Technique 2 – Duplicating a Table in Spool

- For merge joins, the optimizer may choose to **duplicate a small table** on each AMP
- For product joins, the optimizer **always duplicates** one table across all AMPs
- In either case, each AMP must have enough spool space for a complete copy



The Explain plan will indicate that 64M rows are needed for spool.

This slide highlights the fact that Joins can require considerable Spool space. Take this into consideration when calculating Spool requirements.

The top diagram shows an 8 million row table distributed evenly across 8 AMPs so that there are 1 million rows on each AMP.

The bottom diagram shows what happens when the table is duplicated in Spool across all the AMPs. You will notice that there are now 9 million rows on each AMP – 1 million for the actual table and 8 million in Spool.

This example should convince you of the importance of using the EXPLAIN facility so that you don't do unnecessary Product Joins.

Join – Duplicate the Smaller Table

Employee

Enum	Name	Dept
PK		FK
UPI		
1	BROWN	200
2	SMITH	310
3	JONES	310
4	CLAY	400
5	PETERS	150
6	FOSTER	200
7	GRAY	310
8	BAKER	310
9	TYLER	450
10	CARR	450

DUPLICATE and SORT the Smaller Table on all AMPs. LOCALLY BUILD a copy of the Larger Table and SORT.

Department

Dept	Name
PK	
UPI	
150	PAYROLL
200	FINANCE
310	MFG.
400	EDUCATION
450	ADMIN

Example:

```
SELECT E.Enum, ...
FROM Employee E
JOIN Department D
ON E.Dept = D.Dept;
```

Spool file after duplicating the Department rows

150 PAYROLL	150 PAYROLL	150 PAYROLL	150 PAYROLL
200 FINANCE	200 FINANCE	200 FINANCE	200 FINANCE
310 MFG.	310 MFG.	310 MFG.	310 MFG.
400 EDUCATION	400 EDUCATION	400 EDUCATION	400 EDUCATION
450 ADMIN	450 ADMIN	450 ADMIN	450 ADMIN

Employee rows hash distributed on Employee.Enum (UPI)

6 FOSTER 200	4 CLAY 400	1 BROWN 200	5 PETERS 150
8 BAKER 310	3 JONES 310	7 GRAY 310	2 SMITH 310
	9 TYLER 450		10 CARR 450

Spool file after locally building and sorting on Employee.Dept row hash

6 FOSTER 200	3 JONES 310	1 BROWN 200	5 PETERS 150
8 BAKER 310	4 CLAY 400	7 GRAY 310	2 SMITH 310
	9 TYLER 450		10 CARR 450

The example on this slide illustrates duplicating the smaller table.

This strategy consists of duplicating and sorting the smaller table on all AMPs and locally building a copy of the Larger Table and sorting it.

This example is pictured on a 4 AMP System. Teradata duplicates the department table and sorts it on the department column for all AMPs. The employee table is built locally and sorted on the department Row Hash.

This example is the same as the previous example. If the Parser determines from statistics that it would be less expensive to duplicate and sort the smaller table than to hash redistribute the larger table, it will choose this strategy.

General Join Distribution Strategies

MERGE JOIN

Do nothing if Primary Indexes match and are the join columns.

OR

REDISTRIBUTE one or both sides (depending on the Primary Indexes used in the join).

OR

DUPLICATE the smaller table (or data set) in spool on all AMPs.

After redistributing or duplicating the data, the optimizer may choose to:

- Perform a Hash Join if the redistributed or duplicated table can be held in memory
- SORT the duplicated table on join column row hash, create a local spool copy of the larger table and sort it on join column hash, and perform a merge join

NESTED JOIN – Special join case

Equijoin with a constant value for a unique index in one table.

Only join expression that generally does not require all AMPs.

PRODUCT JOIN – Rows do not have to be in any sequence.

DUPLICATE the Smaller Table on all AMPs.

The Optimizer has many join methods, or modes, to choose from to ensure that any join operation is fully optimized. This module discusses some of the more common join methods, but certainly does not cover all of the join methods available to the Optimizer.

The particular processing described for a given type of join (for example, duplication or redistribution of spooled data) might not apply to all joins of that type.

Some of the common join methods include:

- Merge or Hash
 - When done on matching primary indexes, do not require any data to be redistributed.
 - Hash joins are often better performers and are used whenever possible. They can be used for equijoins *only*.
- Nested
 - Only join expression that generally does not require all AMPs.
 - Preferred join expression for OLTP applications.
- Product
 - Always selected by the Optimizer for WHERE clause inequality conditions.
 - High cost because of the number of comparisons required.

Merge Join

Merge joins

- Require rows to be on the same AMP to be joined
- Are usually chosen for an equality join condition
- Blocks from both tables are read only once
- Are generally more efficient than a product join
- Compare matching join column row hash values for the rows
- Cause significantly fewer comparisons than a product join

Merge join process:

- If the tables do not have matching PI on the join columns, identify the smaller table
- If necessary:
 - Put qualifying data of one or both tables into spool(s)
 - Move the spool rows to AMPs based on the join column hash
 - Sort the spool rows into join column hash sequence
- Compare the rows with matching join column row hash values

General considerations:

- Join costs rise with the number of rows that are moved and sorted
- Join plans for the same tables may change as the demographics change

The Merge Join retrieves rows from two tables and then puts them onto a common AMP based on the row hash of the columns involved in the join. The system sorts the rows into join column row hash sequence, then joins those rows that have matching join column row hash values.

Merge joins is commonly done when the Join condition is based on equality. They are generally more efficient than product joins because the number of rows comparisons is smaller. The general merge join strategy consists of the following steps:

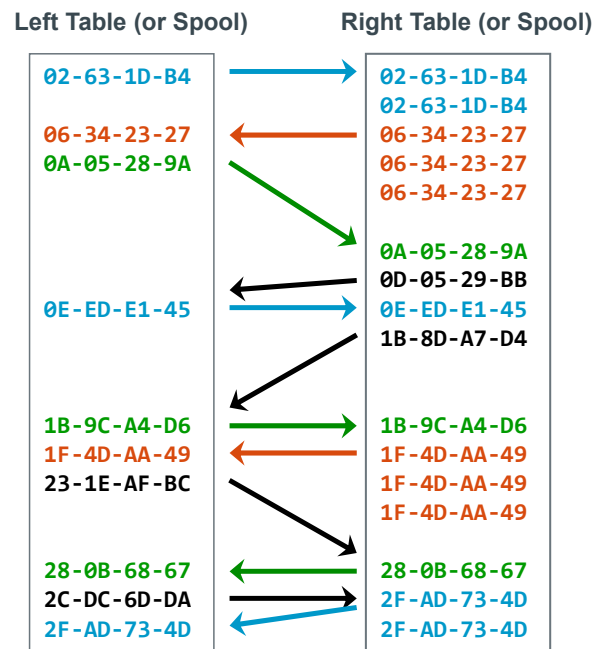
- Identify the smaller table to be joined
- The Optimizer only pursues the following steps if it is necessary to place qualified rows into a spool file.
 - Place the qualifying rows from one or both relations into a spool file.
 - Relocate the qualified spool rows to their target AMPs based on the hash of the join column set
 - Sort the qualified spool rows on their join column row hash values.
- Compare those rows with matching Join Column Row Hash values.

Merge Join Strategy

There are several versions of merge join. This illustration is for the row hash match scan.

With a row hash match scan, both tables (or spool) are sorted on join column row hash sequence.

Note that unnecessary comparisons are ignored, and that each data block is touched only once.



Two different general Merge Join algorithms are available:

- Slow Path – the slow path is used when the left table is accessed using a read mode other than an all rows scan. The determination is made in the AMP, not by the Optimizer.
- Fast Path – the fast path is used when the left table is accessed using the all-row scan reading mode.

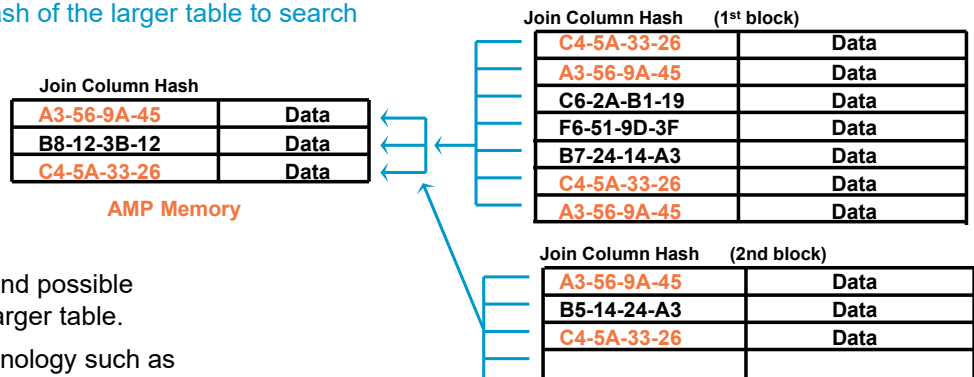
The illustration on this slide gives you a graphical representation of how a Merge Join compares only those rows with matching Row Hash values.

Hash Join

This optimizer technique effectively places the smaller table in AMP memory and joins it to the larger table in unsorted spool. A Hash Join is applicable *only* to equijoins.

Row Hash Join Process:

- Redistribute or duplicate the smaller table in memory across the AMPs
- Usually sort the AMP memory into join column row hash sequence
- Use the join column row hash of the larger table to search memory for a match



This join eliminates the sorting, and possible redistribution or copying, of the larger table.

EXPLAIN plans will contain terminology such as "Single Partition Hash Join".

The Merge Join requires that both sides of the join have their qualifying row in join column row hash sequence. In addition, if the join column(s) are not the Primary Index, then some redistribution or duplication of rows precedes the sort.

Hash Join

In a Row Hash Join, the smaller table is sorted into join column row hash sequence and then redistributed or duplicated on all AMPs. The larger table is then processed a row at a time and the rows in this table do NOT have to be sorted into join column hash sequence. For those rows that qualify for joining (WHERE or ON), the Row Hash of the join column(s) is used to do a binary search of the smaller table (in memory) for a match. The Optimizer can choose this join plan when the qualifying rows of the small table can be held AMP memory resident.

Miscellaneous Hash Join Notes:

- The standard hash join does require the same spooling and the same relocation of rows before the join, whether a table is being duplicated or redistributed. In this regard, it is similar to the merge join.
- However, if the small table is small enough to fit into one hash partition and if it is duplicated, the redistribution of the large table can be eliminated by doing a hash join on the fly, a variant of the standard hash join. In such a case, the large table is read directly, without spooling or redistributing, and the hash join is performed between the small table spool and the large table rows.
- Both types of hash joins eliminate sorting, and the on the fly version eliminates redistribution.
- A **Dynamic Hash Join** provides the ability to do an equality join directly between a small table and a large table on non-primary index columns without placing the large table into a spool file. For Dynamic Hash Join to be used, the left table must be small enough to fit in a single partition.

Nested Joins

- This is a special join case
- This is the only join that doesn't always use all of the AMPs
- It is the most efficient in terms of system resources
- It is the best choice for OLTP applications
- To choose a Nested Join, the Optimizer must have:
 - An equality value for a unique index (UPI or USI) on Table1
 - A join on a column of that single row to any index on Table2
- The system retrieves the single row from Table1
- It hashes the join column value to access matching Table2 row(s)

Example:

```
SELECT    E.Name
          ,D.Name
FROM      Employee E
JOIN      Department D
ON        E.Dept = D.Dept
WHERE     E.Enum = 5;
```

Employee			Department	
Enum	Name	Dept	Dept	Name
PK		FK	PK	
UPI			UPI	
1	BROWN	200	150	PAYROLL
2	SMITH	310	200	FINANCE
3	JONES	310	310	MFG.
4	CLAY	400	400	EDUCATION
5	PETERS	150		
6	FOSTER	400		
7	GRAY	310		
8	BAKER	310		

Nested Joins are the most efficient types of Join. For a Nested Join to be done between Table 1 and Table 2, the Optimizer must be provided with both of the following:

- An equality value for a unique index on Table 1 (this will retrieve a single row).
- A Join on a column of that single row to any index on Table 2

The system will retrieve the single row from Table 1 based on the UPI or USI value, then determine the hash of the value in the Join Column to access matching Table 2 rows.

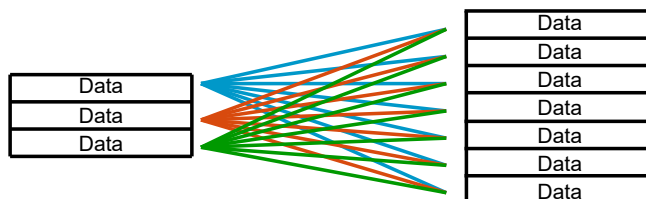
Nested Joins are the only types of Join that don't always use all of the AMPs. The number of AMPs involved in a Nested Join will vary.

The query on this slide can also be coded as follows:

```
SELECT    E.Name
          ,D.Name
FROM      Employee E, Department D
WHERE     E.Dept = D.Dept
AND       E.Enum = 5;
```

Product Join

- Does not sort the rows.
- May re-read blocks from one table if AMP memory size is exceeded
- It compares every qualifying Table1 row to every qualifying Table2 row
- Those that match the WHERE condition are saved in spool
- It is called a Product Join because:



Rows must be on the same AMP to be joined.

$$\text{Total Compares} = \# \text{ Qualified Rows Table1} * \# \text{ Qualified Rows Table2}$$

- The internal compares become very costly when there are more rows than AMP memory can hold at one time
- They are generally unintentional and often give meaningless output
- Product Join process:
 - Identify the Smaller Table and duplicate it in spool on all AMPs
 - Join each spool row for smaller table to every row for larger table

Product Joins are the most general forms of Join. In a product Join, every qualifying row of one table is compared to every qualifying row in the other table. Rows that match on WHERE conditions are saved.

Product Joins are caused by any of the following:

- The WHERE clause is missing.
- A Join condition is not based on equality (NOT =, LESS THAN, GREATER THAN).
- Join conditions are ORed together.
- There are too few Join conditions.
- A referenced table is not named in any Join condition.
- Table aliases are incorrectly used.
- The Optimizer determines that it is less expensive than the other Join types.

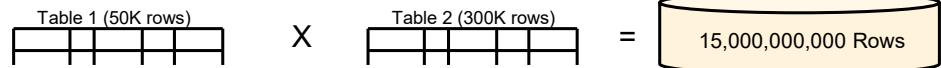
Product joins get their name from the fact that the number of comparisons required is the “product” of the number of qualifying rows of both tables. A product Join between a table of 1,000 rows and a table of 50 rows would require 50,000 comparisons and a potential answer set of 50,000 rows.

Because all rows of one side must be compared with all rows of the other, the smaller table is always duplicated on all AMPs. Its rows then are compared with the AMP local rows of the other table. If the entire table cannot fit into memory, blocks will have to be read in more than once. Comparisons that qualify are written to spool. While legitimate cases exist for these joins, they should be avoided whenever possible.

Cartesian Product

- This is an unconstrained Product join
- Each row of Table1 is joined to every row in Table2
- Cartesian Product Joins consume significant system resources
- Cartesian Product Joins are occasionally used in some business cases
 - Example: Join a single column table (with limited rows) to a second table to populate a third table
 - These are also supported for ANSI compatibility
- Cartesian Product Joins may occur when:
 - A join condition is missing or there are too few join conditions
 - Join conditions are not based on equality
 - A referenced table is not named in any join condition
 - Table aliases are incorrectly used
- The transaction aborts if it exceeds the user's spool limit

Table row count is critical:



Number of tables is even more critical:



Cartesian Products are unconstrained Product Joins. Every row of one table is joined to every row of another table.

Since there is rarely any practical business use of Cartesian Product Joins, they generally will occur when an error is made in coding the SQL query. Some reasons why Cartesian Products are caused are shown on this slide.

Running your queries through EXPLAIN will enable you to avoid unintentional Cartesian Product Joins and thus save a lot of system resources. You should always EXPLAIN a Join before it goes production.

An example of using an alias incorrectly:

```
SELECT    TableA.col1
FROM      TableA A;
```

The result will be a product join. Teradata will rename the TableA as A and then in the SELECT clause when it sees a reference to TableA, it will treat it as a separate instance of TableA.

Teradata assumes that you want to join the table to itself. It will look for a join condition, but as there is none; Teradata will carry out a PRODUCT join.

Exclusion Joins

- Finds rows that **DON'T** have a match
- May be done as merge or product joins
- SQL that frequently causes exclusion joins are NOT IN subqueries and EXCEPT or MINUS operations
- Uses 3-value logic (=, <>, unknown) on nullable columns
- To avoid NULL result sets:
 - If possible, define columns (used with NOT IN) as NOT NULL on the CREATE TABLE
 - In queries, include WHERE column_name IS NOT NULL against nullable join columns

Set_A	NOT IN	Set_B	=	Result
1		1		2
2		3		4
3		5		
4				

Set_A	NOT IN	Set_B	=	Result
1		1		NULL
2		3		
3		5		
4		NULL		

Exclusion Joins are based on set subtraction and used for finding rows that don't have a matching row in the other table. Queries with the NOT IN and EXCEPT operator lead to Exclusion Joins.

Exclusion Join is a Product or Merge Join where only the rows that do not satisfy (are NOT IN) any condition specified in the request are joined. In other words, Exclusion Join finds rows in the first table that do *not* have a matching row in the second table.

Exclusion Join is an implicit form of the outer join.

The appearance of a null (unknown) join value will return an answer set of NULL. Null join values must be prohibited to get a result other than NULL. Another way to view exclusion operations, is to look at the 3 rules that are applied from the selected set to the NOT IN set.

1. Any True – disqualifies the row
2. Any Unknown – disqualifies the row
3. All False – qualifies the row.

Therefore, in the first example, rows 1 and 3 are matched (true) so they are disqualified. Rows 2 and 4 are all false (no matches), so they qualify.

In the second example, row 2 hits the unknown (NULL) and is disqualified, so in essence all of the rows that do not get disqualified by the match (true) will get disqualified by rule number 2 (Any Unknown).

Exclusion Join Example

Employee

Enum	Name	Job Code
PK		FK
UPI		
1	BROWN	2101
2	SMITH	2101
3	JONES	3100
4	CLAY	1201
5	PETERS	3100
6	FOSTER	3100
7	GRAY	1302
8	BAKER	3100
9	TYLER	3100
10	CARR	1302

Customer

Cust_Num	Sales_Emp_Number
PK	FK
UPI	
23	6
24	3
25	8
26	1
27	6
28	8
29	1
30	3
31	8

This is an example of an Exclusion Merge Join.

Example: `SELECT Enum, L_Name
FROM Employee
WHERE Job_Code = 3100
AND Enum NOT IN (SELECT Sales_Emp_Number FROM Customer);`

Employee rows hash distributed on Employee.Enum (UPI)

6 FOSTER 3100	4 CLAY 1201	1 BROWN 2101	5 PETERS 3100
8 BAKER 3100	3 JONES 3100	7 GRAY 1302	2 SMITH 2101
	9 TYLER 3100		10 CARR 1302

Customer rows hash distributed on Cust_Num (UPI).

30 6	23 6	28 8	25 8
24 3	29 1	27 6	26 1
31 8		30 3	

Spool file after redistributing and duplication on Customer.Sales_Emp_Number

6	3	1	
8			

Spool file after locally building and filtering(Job Code) on Employee rows

6 FOSTER 3100	3 JONES 3100		
8 BAKER 3100			

The example on this slide illustrates an Exclusion Merge Join. The SQL query is designed to list those salespeople who don't have any customers.

This example is pictured on a 4 AMP System. As you can see, Teradata does the following:

- Performs the subquery (SELECT Sales_Emp_Number FROM Customer).
- Hash redistributes these rows on all AMPs.
- Eliminates duplicate values.
- Returns the name column of those rows in the employee table which do not match the rows in the sub-query.

In this case, the salespersons whose names would be returned would be Peters and Tyler.

Inclusion Joins

Inclusion Join – an inclusion join is a Merge or Product Join where the first right table row that matches the left row is joined.

An example of a query that may cause an inclusion merge join:

```
SELECT      Name, Emp#  
FROM        Employee  
WHERE       Emp# IN (SELECT Emp# FROM Charges)  
ORDER BY    Name ;
```

There are two types of Inclusion Join.

- **Inclusion Merge Join**
 - Read each row from the left table
 - Join each left table row with the first right table row having the same hash value
- **Inclusion Product Join**
 - For each left table row read all right table rows from the beginning until one is found that can be joined with it
 - Return the left row if a matching right row is found for it

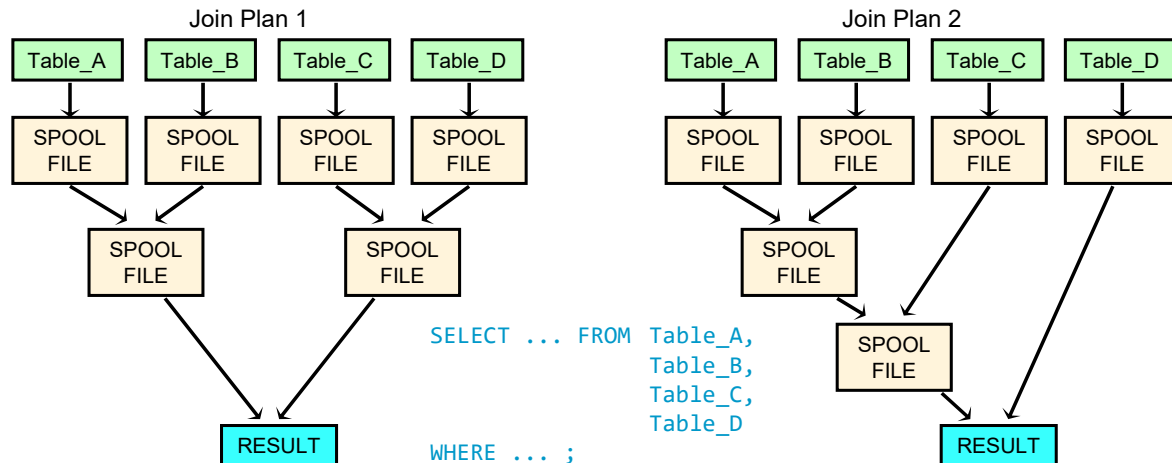
There are two types of Inclusion Join.

- **Inclusion Merge Join**
 - Read each row from the left table.
 - Join each left table row with the first right table row having the same hash value.
 - End of process.
- **Inclusion Product Join**
 - For each left table row read all right table rows from the beginning until one is found that can be joined with it.
 - Return the left row if a matching right row is found for it.
 - End of process.

This slide illustrates an example of an Inclusion Merge Join. Explain plan terminology will indicate either “Inclusion Merge or Inclusion Product” in the explain text.

n-Table Joins

- All n-Table joins are reduced to a series of two-table joins
- The Optimizer attempts to determine the best join order
- Collected Statistics on Join columns help the Optimizer choose wisely



It is not uncommon to have Joins that involve more than two tables. The Optimizer will decide which tables it processes first based on its own decision algorithms. The Optimizer can only work with two tables at a time. The results of that Join operation will then be applied to a third table (or another Join result).

“Smaller” means the size of the table both as a function of selection and projection. “Selection” refers to the number of rows that qualify from that table in the answer set. “Projection” refers to the “row size” as a function of the number of column bytes selected. Each reduces the amount of data carried to subsequent Join steps.

Join Considerations with Partitioned Tables

Row Partitioning (e.g., PPI) is based on modular extensions to the existing implementation of Teradata. Therefore, all join algorithms support partitioned tables without requiring a whole new set of join code.

Performance Note

- Performance for Row Hash Merge Joins may be worse with partitioned table (as compared to non-partitioned tables) if a large number of partitions are not eliminated via query constraints

Why?

- Row Hash Merge Join algorithm is more complicated and requires more resources with partitioned tables than with non-partitioned tables (assuming an equal number of qualified data blocks) **since rows are not in hash order, but rather in partition/hash order**

Direct merge joins (in which the table of interest doesn't have to be spooled in preparation for a merge join) are available as an optimizer choice when two non-partitioned tables have the same PI, and all PI columns are specified as equality join terms (the traditional merge join). A direct merge join, in the traditional sense, is not available when one table is partitioned and the other is not, or when both tables are partitioned, but not in the same manner, as the rows of the two tables will not be ordered in the same way. However, the traditional merge join algorithms have been extended to provide a partitioned-aware merge join, called a "sliding window" join.

The optimizer has three general avenues of approach when joining a partitioned table to a non-partitioned table, or when joining two partitioned tables with different partitioning expressions.

- One option is to spool the partitioned table (or both partitioned tables) into a non-partitioned spool file in preparation for a traditional merge join.
- A second option (not always available) is to spool the non-partitioned table (or one of the two partitioned tables) into a partitioned spool file, with identical partitioning to the remaining table, in preparation for a rowkey-based merge join.
- The third approach is to use the sliding window join of the tables without spooling either one. The optimizer will consider all reasonable join strategies, and pick the one that has the best-estimated performance.

Sliding window joins may be slower than the traditional merge join or the rowkey-based merge join when there are many non-excluded partitions. Sliding window joins can give roughly similar elapsed-time performance when the number of non-excluded partitions is small (but with greater CPU utilization and memory consumption).

Additional Join Options with Partitioned Tables

The optimizer has other options than the “sliding window” join.

If one table is partitioned and the other isn't, alternatives to the “sliding window” join approach.

- One is to spool the partitioned table to a non-partitioned spool file, after which the join is between two non-partitioned relations
- Spool the NPPI table to a partitioned spool file (matching the partitioning of the PPI table), then directly join two identically-partitioned relations
 - This option is available only if the query specifies an equality condition on every column that is referenced in the partitioning expression

If both tables are partitioned, possibilities are ...

- Spool one of the tables to a spool file with partitioning identical to the other table
- Spool both tables to non-partitioned spool files

As usual, the optimizer estimates the cost of all reasonable alternatives and chooses the least expensive solution.

The optimizer has other options than the “sliding window” join. As usual, the optimizer estimates the cost of all reasonable alternatives, and chooses the least expensive solution. Given the importance of the term p/k in the previously mentioned formulas, it is important that the optimizer have a realistic estimate of the number of non-excluded partitions.

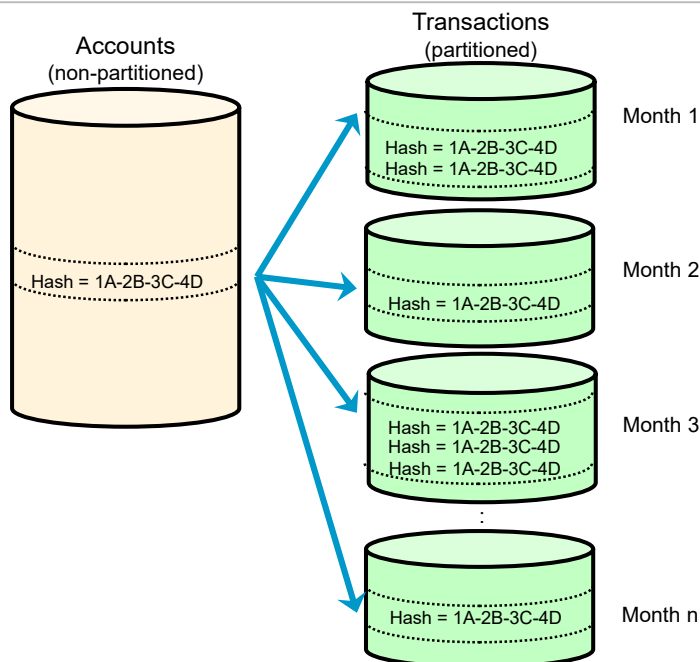
This is the reason that the earlier examples that partitioned on “store_id” indicated that the RANGE_N example was better than the example that used the column directly. In the RANGE_N example, the optimizer knows that there are a maximum of ten “store_id” partitions with rows, and will use ten or a smaller number as the value of p . When the column is used directly, the optimizer may use a value much larger than ten when estimating p , especially if statistics haven't been collected on the “store_id” column.

If one table is partitioned and the other isn't, the optimizer has two viable alternatives in addition to the “sliding window” join approach. One is to spool the partitioned table to a non-partitioned spool file, after which the join is between two non-partitioned relations. The other option, not always available, is to spool the non-partitioned table to a partitioned spool file, then directly join two identically partitioned relations. The partitioned spool file option is available only if the query specifies an equality condition on every column that is referenced in the partitioning expression.

If both tables are partitioned, it may be possible to spool one of the tables to a spool file with partitioning identical to the other table. It may also be cost-effective to spool both tables to non-partitioned spool files.

Non-partitioned to Partitioned Table Join (Few Partitions)

teradata.



Teradata can keep one block from the non-partitioned table and one block per partition in memory from a partitioned table to facilitate efficient join execution

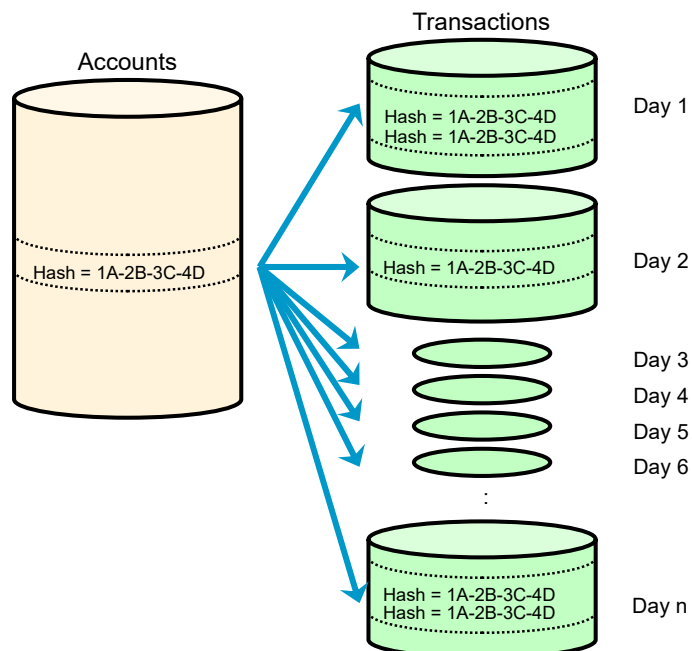
The following discussion assumes that two tables have the same PI and all PI columns are specified as equality join terms.

When joining two tables together (one is non-partitioned and the other has row partitioning) and after applying constraints, if there are a “small” number of surviving partitions, then the following applies:

- Teradata can keep one block from the non-partitioned table and one block per partition in memory from the row partitioned table to facilitate efficient join execution.
- Performance is similar to non-partitioned table to non-partitioned table join even when no partitions are eliminated (assumes that the number of partitions is relatively small).
 - Same number of disk I/Os (except in anomalous cases - large number of rows in a hash)
 - Higher memory requirements
 - Slightly higher CPU utilization
 - You can get significantly better performance when query constraints allow partition elimination.

Non-partitioned to Partitioned Table Join (Many Partitions)

teradata.



One option available to Teradata is to keep one block from a non-partitioned table and one block for as many partitions from the partitioned table as it can fit into memory using a "sliding window" technique

The basic enhancement is the use of a "sliding window" technique, which can be slower than the performance of a direct join. This is true as long as the number of partitions participating in the join is small. Usually, CPU utilization will probably be somewhat higher for a row partitioned table, and more memory will be used.

When joining two tables together (one with non-partitioned and the other is row partitioned) and after applying constraints, if there are a "large" number of surviving partitions, then the following applies:

- Teradata can keep one block from a non-partitioned table and one block for as many partitions as it can fit (k) into memory from a row partitioned table to facilitate efficient join execution using a "sliding window" technique.
- This type of join will usually have worse performance than non-partitioned to non-partitioned join unless partition elimination can reduce total amount of work performed.

- ✓ Higher number of disk I/Os - the data blocks in non-partitioned table will have to be rescanned multiple times.

d1 = # of data blocks in non-partitioned table

d2 = # of data blocks in row partitioned table

p = # of partitions participating in the join

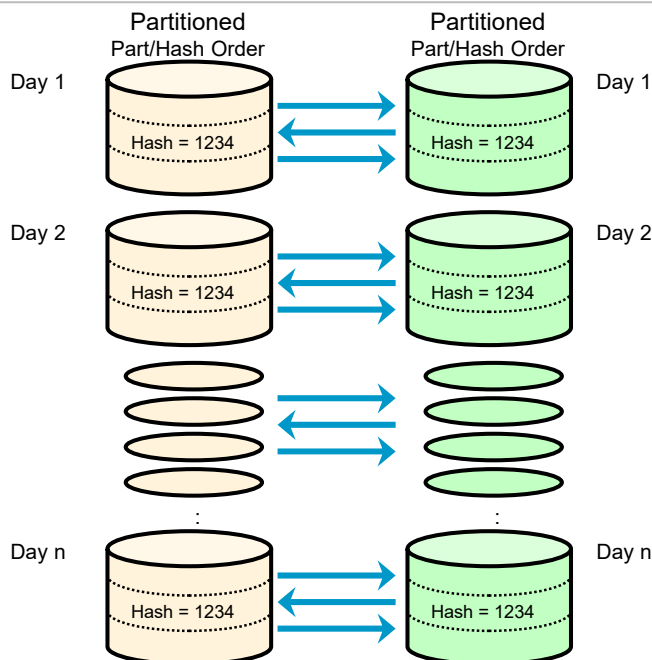
k = # of partitions that can fit into memory from partitioned table

The number of I/Os for non-partitioned to partitioned join is: $(p/k * d1) + d2$

The number of I/Os for non-partitioned to non-partitioned join is: $d1 + d2$

Partitioned to Partitioned Table Join (Rowkey Match Scan)

teradata.



When joining two partitioned tables that have equivalent partition definitions, a **Row Key Match Scan within partitions** can be used to join the tables together

Hash of 1A-2B-3C-4D is shown as Hash of 1234.

Direct merge joins of two partitioned tables are available as an optimizer choice when the tables have the same PI and identical partitioning expressions, and all PI columns and all partitioning columns are specified as equality join terms. This is referred to as a **rowkey-based merge join**. In this case, the rows of the two tables will be ordered in the same way, allowing a merge join without redistribution or sorting of the rows.

If the partitioning column is part of the Primary Index (PI), then it is possible, and usually advantageous, to define all the tables with the same PI with identical partitioning expressions. This allows for a rowkey-based merge join.

If non-partitioned table is placed into spool (redistributed or duplicated) and this spool is joined with a partitioned table, then spool can optionally be partitioned the same as the partitioned table allowing for this fast join.

The performance characteristics of a traditional merge join (on matching primary indexes) and a rowkey-based merge join will be approximately the same.

Join Processing Highlights

Inefficient joins result from:

- Poor physical design choices
 - Lack of indexes
 - Inappropriate indexes
- Stale or missing Collected Statistics
- Inefficient SQL code
 - Poor SQL code can degrade performance on a good database design
 - Good SQL code cannot compensate for a poor database design

The system bases join planning on:

- Primary and Secondary Indexes
- Estimated number of rows in each subtable
- Estimated ratio of table rows per index value

COLLECTed STATISTICS may improve join performance.

The fastest merge joins are based on matching Primary Indexes.

Data demographics change over time.

Join plans for the same tables change as demographics changes.

Revisit ALL index (Primary and Secondary) choices regularly. Make sure they are still serving you well.

This slide summarizes many of the key points regarding Join Processing. There are three key factors:

1. Physical design choices
2. Availability of COLLECTed STATISTICS for design
3. Quality of SQL coding

Database design is the key to efficient Join Processing because the Optimizer bases its plans on Primary and Secondary Indexes.

COLLECTed STATISTICS are also vital since the Optimizer needs to know table Row Counts as well as Rows per Value. Make sure that the Optimizer always has fresh STATISTICS since data demographics change over time.

MAKE SURE you write efficient SQL code. But remember, even the best SQL code cannot compensate for poor database design choices.

Summary

Now that you have completed this course, you will be able to:

- Identify and describe different kinds of join plans
- Describe join plan strategies
 - Merge Join
 - Nested Join
 - Hash Join
 - Product Join
 - Exclusion Merge Join
- Describe different types of join strategies that may be used with row partitioned tables



Module 4: Joins Bring Up JupyterHub

teradata.

Let's now do the lab together



Thank you.

teradata.

©2023 Teradata



Module 5: Optimizer and Statistics Collection

Teradata Vantage MasterClass

Copyright © 2007–2023 by Teradata. All Rights Reserved.

Objectives

After completing this course, you will be able to:

- Explain how the Optimizer acquires statistics
- Describe random AMP sampling
- State a method for viewing statistics
- Learn how to utilize both the standard and index formats to collect statistics

Why Collect Statistics?

My queries take very long to execute even on columns from a single table

I thought as long as I have access to all tables, I can join and filter on any columns I wish

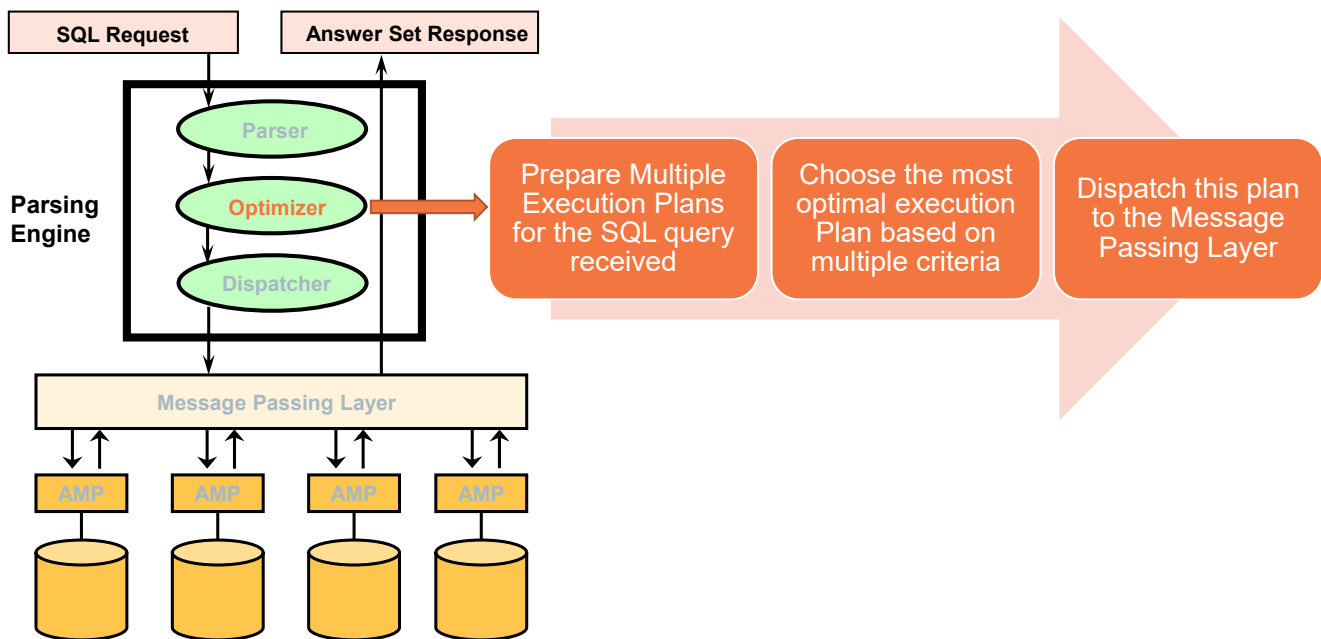


Is there a way that I can tell Teradata that
“This column is very important for my business, so go and gather as much information (or Statistics) as you can about it, so my queries can run effectively”

What is the need to collect statistics?

Data is after all loaded in tables and a query is supposed to pick the data up? What is the real need to collect statistics?

Who Collects Statistics?



Parsing Engines (PEs) are made up of the following software components: session control, the Parser, the Optimizer, and the Dispatcher.

Once a valid session has been established, the PE is the component that manages the dialogue between the client application and the Teradata database. The major functions performed by **session control** are logon and logoff.

When a PE receives an SQL request from a client application, the **Parser** interprets the statement, checks it for proper SQL syntax and evaluates it semantically. The PE accesses the Data Dictionary/Directory to ensure that all objects and columns exist and that the user has authority to access these objects.

The **Optimizer's** role is to develop the least expensive plan to return the requested response set. Processing alternatives are evaluated and the fastest alternative is chosen. This alternative is converted to executable steps, to be performed by the AMPs, which are then passed to the dispatcher.

The **Dispatcher** controls the sequence in which the steps are executed and passes the steps on to the Message Passing Layer. It is composed of execution control and response control tasks. Execution control receives the step definitions from the Parser, transmits the step definitions to the appropriate AMP or AMPs for processing, receives status reports from the AMPs as they process the steps, and passes the results on to response control once the AMPs have completed processing. Response control returns the results to the user. The Dispatcher sees that all AMPs have finished a step before the next step is dispatched.

Optimizer – Statistics

Environment Information

- ☐ Number of nodes
- ☐ Number of AMPs
- ☐ Number and type of CPUs
- ☐ Disk Array information
- ☐ Interconnect (BYNET) information
- ☐ Memory available

Data Demographics

- ☐ Number of rows in the table
- ☐ Row size
- ☐ Column demographics
 - Range of values for the column
 - Number of rows per value
 - Number of NULLs for the column
- ☐ Index demographics



The Optimizer plans an execution strategy for every SQL query submitted to it. We have also seen that the execution strategy for any query may be subject to change depending on various factors. The best way to assure that the Optimizer has all the information it needs to generate optimum execution strategies is to **COLLECT STATISTICS**.

Optimizer and Block-Level Compression

The optimizer is not sensitive to block-level compression. Neither will you see any additional steps added to the explain text when a query accesses a compressed table, as the decompression processes are performed transparently at the file system level. The optimizer does not take into account the extra time or CPU required for decompression when estimated processing times are established. However, the average row size that is calculated during random AMP sampling will be different for compressed tables. When random AMP sampling is performed, one or more cylinder indexes are read. While the cylinder indexes are never compressed, the information they carry is based on physical characteristics of the underlying data.

Even if full statistics have been collected, random AMP sampling is relied upon for determining the average row size as input to query optimization. Because the table's row size is based on the compressed image of the data, estimated processing times, which are influenced by row size, may be slightly less in queries accessing compressed tables. While it is not expected that this discrepancy will be large enough to cause the optimizer to make different decisions in most cases, the row size under-estimation with compression might lead to some query plan changes when block level compression is implemented.

Teradata Optimizer

Teradata uses a “cost-based optimizer”.

- The Optimizer evaluates the “costs” of all reasonable execution plans and the best choice is used.
 - Effectively finds the optimal plan
 - Optimizes resource utilization - maximizes system throughput
- What does the “optimizer” optimize?
 - Access Path (Use index, table scan, dynamic bitmap, etc.)
 - Join Method (How tables are joined – merge join, product join, hash join, nested join)
 - Join Geography (How rows are relocated – redistribute, duplicate, AMP local, etc.)
 - Join Order (Sequence of table joins)
- The Optimizer is Parallel-aware

Teradata’s optimizer is cost based and will actually generate several plans and choose the best plan based on analyzing the lowest cost numbers. This is critical to performance in supporting mixed workloads. A cost based optimizer requires statistical information about the data as well as being aware of the machine resources (CPU, disk, memory, etc.). The other type of optimizer is rules based. This is good for transactional workloads where the queries are well known and the data has been physically structured to support this known workload. It is based on a set of rules that have been defined and only performs well in a highly structured transactional environment.

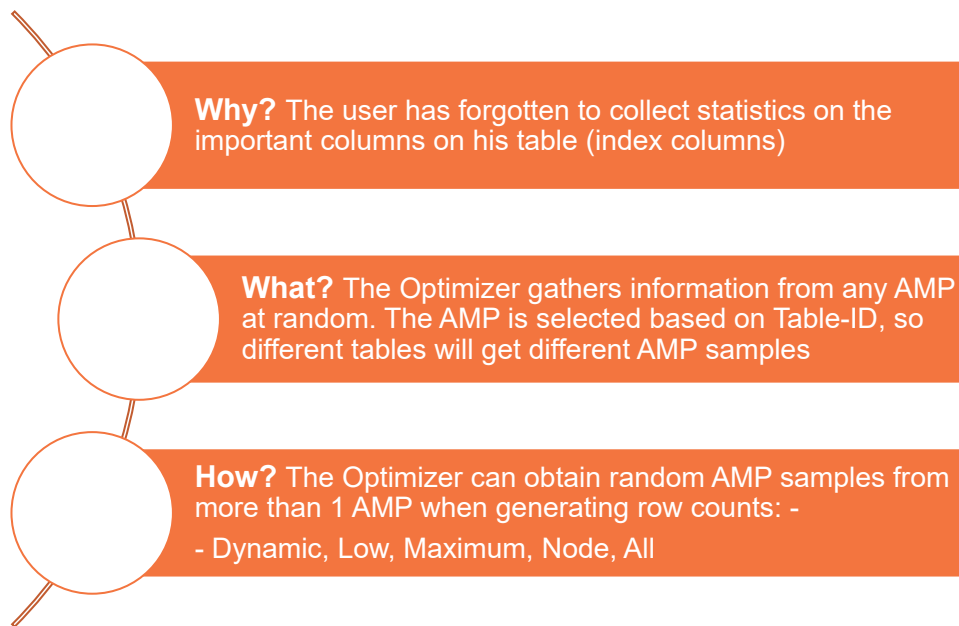
Cost Based Optimizer (Teradata)

- Best for complex DSS queries
- Requires statistics on tables and individual columns within tables
- Plans are independent of table ordering in FROM clause & predicate ordering in WHERE clause
- Should understand available resources within the system (e.g., CPU, Memory, etc.)
- Generates several plans and chooses best plan based on smallest cost factors

Rules Based Optimizer

- Used in OLTP environments with structured and known access paths
- Rules are defined for access paths and types of SQL statements (e.g., simple, join, complex, etc.)
- Plans are chosen based on access paths available, the ranks of these access paths and type of query
- Not a good choice in DSS environments
- Users can provide hints to help influence a rules based optimizer.

Optimizer – Random AMP Samples



The Optimizer does Random AMP Sampling with information it gathers from a random AMP. This AMP is selected based on the Table ID. Different tables will get their samples from different AMPs. This assures that no single AMP will be overloaded with Random AMP Sample requests.

Since this method uses only a single AMP for sampling purposes, it is sensitive to the evenness of the data distribution. Badly distributed data gives skewed samples that impact optimization by misleading the Optimizer into making improper choices. The Optimizer will be more conservative in the choices it makes if it has to rely upon Random AMP Sampling.

DBSControl Options

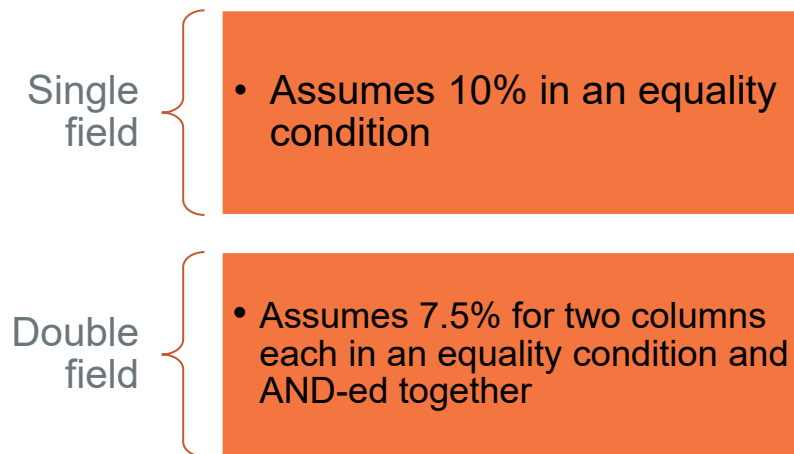
An internal parameter (#65) within DBSControl can be set by the Teradata Customer Engineer to specify the type of Random AMP Sampling that will be used in V2R6.

65. RandomAmpSampling – this field determines the number of AMPs to be sampled for getting the row estimates of a table. The valid values are D, L, M, N or A.
- D - The default is one AMP sampling (D is the default unless changed.)
 - L - Maximum of two AMPs sampling
 - M - Maximum of five AMPs sampling
 - N - Node Level sampling - i.e., all the AMPs in a node would be sampled.
 - A - System Level sampling - i.e., all the AMPs in a system would be sampled.

Note that a higher number of AMPs sampled will provide better estimates, but can cause short running queries to run slower and long running queries to run faster.

Optimizer – Heuristics

For non-indexed columns without statistics, the optimizer uses **heuristics** or fixed formulas to estimate the number of rows

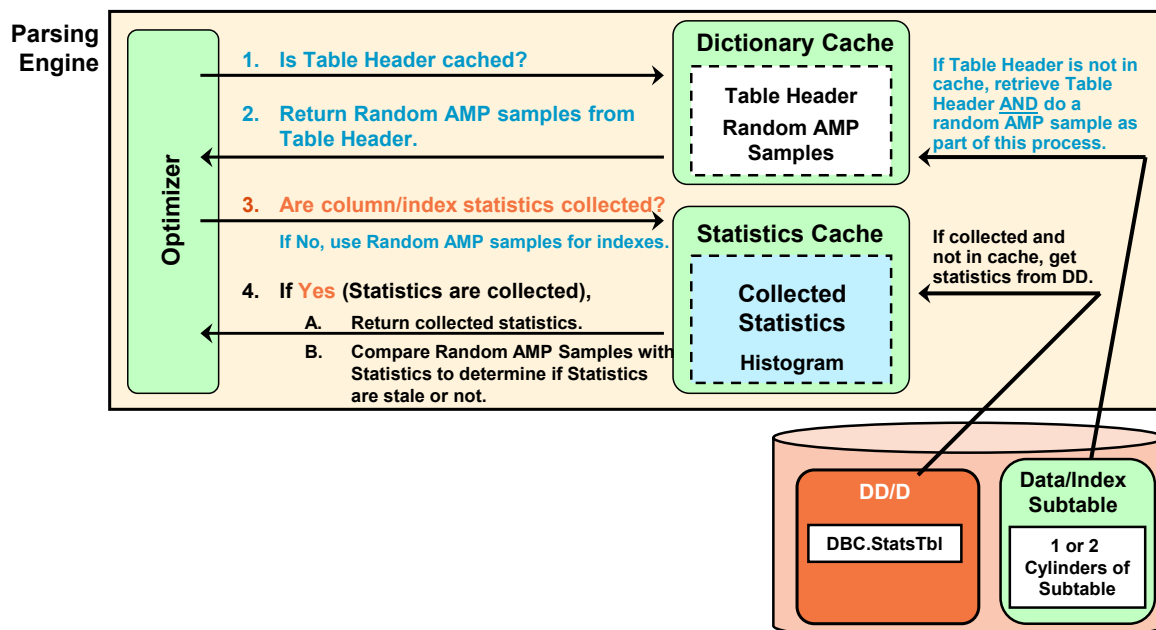


Sampling Efficiency for Non-indexed Predicate Columns

The heuristic is to estimate the cardinality to be 10% of the table rows. With two selection criteria, the Optimizer assumes that about 7.5% ($10\% \times .75$) of the rows will be returned.

For additional equality criteria, each is 75% of the previous level. For example, for 3 columns, the Optimizer assumes 5.2% ($7.5\% \times .75$) of the rows will be returned.

Optimizer's Search for Statistics



When there are no statistics available to quantify the demographics of a table or an index, the Optimizer selects (by default) a single AMP to sample for statistics using an algorithm based on the table ID. Note that the statistics collected by a random AMP sample only apply to indexed columns.

Step Wise:

1 & 2. Get the random AMP statistics from table header in DD cache.

3. If statistics are collected for the index or column, get the statistics from cache or DD.

Random AMP sample row counts and collected statistics row counts are compared to determine if collected statistics are stale or if extrapolation is needed.

The process of the optimizer searches for statistics as follows:

- First, random AMP samples are kept in the table header that resides in memory. The optimizer first looks for the table header in the dictionary cache in order to extract the random AMP samples. If the table header is not in the cache, it is read from disk. As part of the process of reading the table header from disk, random AMP samples are collected for that table and moved into a field in the table header when the table header is placed in the cache.
- After the table header has been located and the random AMP samples have been accessed, a routine that looks for collected statistics is called. This routine will be called once for each index and/or column on a table for which the optimizer would like statistics. As part of that routine, the dictionary cache is searched for the relevant histogram.
- If statistics have been collected for the column/index of interest but the statistics are not cached, an express request is issued that retrieves the collected statistics from the data dictionary tables on disk.
- If no collected statistics exist and the column is a primary or non-unique secondary index (NUSI), then the random AMP samples that are stored in the table header will be used. Non-indexed columns that have no statistics collected will not use random AMP samples, but will rely on static formulas to determine selectivity estimates.
- Statistics are removed from the cache periodically to make sure that what is cached is

reasonably current.

Example of an Optimizer Estimate without Collected Statistics

Table information:

- Claim table has 30,000,000 rows and **only** summary statistics have been collected.
- The column Custid is not indexed and does not have collected statistics.

Query

```
EXPLAIN SELECT * FROM Claim WHERE custid = 1038994;
```

Explanation (partial)

3) We do an all-AMPs RETRIEVE step in TD_MAP1 from TFACT.Claim by way of an **all-rows scan** with a condition of ("TFACT.Claim.custid = 1038994") into Spool 1 (group_amps), which is built locally on the AMPs. **The size of Spool 1 is estimated with no confidence to be 3,000,000 rows** (429,000,000 bytes). The estimated time for this step is 19.73 seconds.

Question 1

Why does the optimizer estimate 3,000,000 rows will be retrieved for a custid of 1038994?

Question 2

```
EXPLAIN SELECT * FROM Claim WHERE custid = 1038994 AND claimdate = CURRENT_DATE;
```

Why does the optimizer estimate 2,250,000 rows with these two conditions?

This Claim table has only Summary Statistics.

The answer to the first question is 10%. The second question answer is "The Optimizer will assume 7.5% for two non-indexed equality selection criteria".

With two equality conditions, the approximate estimate is 7.5% (10% x .75).

```
EXPLAIN      SELECT      *
              FROM        Claim
              WHERE        custid = 1038994 AND  claimtype = 99;
```

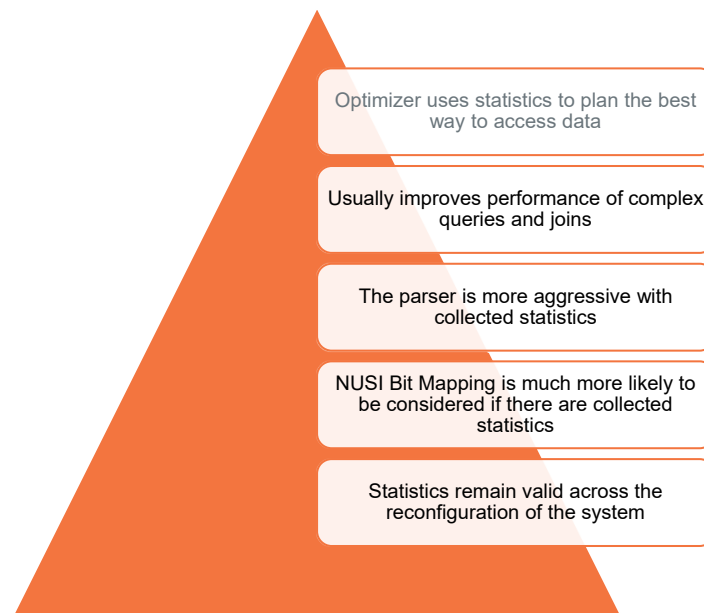
The size of Spool 1 is estimated with no confidence to be 2,250,000 rows.

With three equality conditions, the approximate estimate is 5.2% (10% x .75 x .75).

```
EXPLAIN      SELECT      *
              FROM        Claim
              WHERE        custid = 1038994
              AND          claimtype = 99
              AND          agentname = 'LC123456';
```

The size of Spool 1 is estimated with no confidence to be 1,658,500 rows.

Statistics Collection – The Impact



The primary purpose of STATISTICS is to tell the Optimizer how many rows/values there are.

- Helpful in accessing a column or index with uneven value distribution.
- The Optimizer uses statistics to decide whether a query plan should use a secondary, hash, or join index instead of performing a partition or full-table scan.
- Improve the performance of complex queries and joins.
- The Optimizer uses statistics to estimate the cardinalities of intermediate spool files based on the qualifying conditions specified by a query. The estimated cardinality of intermediate results is critical for the determination of both optimal join orders for tables and the kind of join method that should be used to make those joins.
- For example, assume 2 tables or spool files are redistributed and then merge joined, or assume one of the tables or spool files is duplicated and then product joined with the other. Depending on how accurate the statistics are, the generated join plan can vary so greatly that the same query can take only seconds to complete using one join plan, but take hours to complete using another.
- Enable the Optimizer to utilize NUSI Bit Mapping.

Additional Considerations

- COLLECT STATISTICS is a DDL statement and should be scheduled during off-hours. It should not be done during production hours. The operation holds a row-hash Write Lock on DBC.StatsTbl which means that no new SQL requests involving the affected table can be parsed.
- Remain valid if the system is reconfigured.

Statistics Data – What is Collected? (1 of 2)

If you **ONLY** collect (or refresh) **SUMMARY** statistics, then a Table Summary row is created (or refreshed).

Table Summary Information (row count, average row size, sample estimates, etc.)	Previous Table Summary Records	...
--	--------------------------------	-----

To view table summary row information,

SHOW SUMMARY STATISTICS VALUES ON tablename;

If you collect (or refresh) statistics for a column/index, then a Table Summary record is created (or refreshed) as well as a "histogram row" with details for the column/index for which you are collecting statistics.

Table Summary Information (row count, average row size, sample estimates, etc.)	Previous Table Summary Records	...
Column/Index SummaryInfo	<div> <div>←</div> <div>Detail Intervals (Max Value, mode value, mode frequency, ...)</div> <div>→</div> </div>	Previous SummaryInfo Records

Bias Values

To view column/index detail information,

SHOW STATISTICS VALUES COLUMN column_name ON tablename;

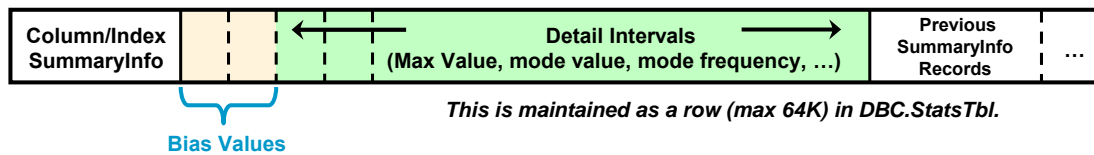
A **COLLECT STATISTICS** request divides the rows in ranges of values such that each range has approximately the same number of rows, but it never splits rows with the same value across intervals. To achieve constant interval cardinalities, the interval widths, or value ranges, must vary.

Statistics have significantly changed starting with Teradata 14.0. To start with, an option called **SUMMARY** is introduced to collect only the table-level statistical information such as row count, average block size, average row size, etc. without the histogram detail. This option can be used to provide up-to-date summary information to the optimizer in a quick and efficient way. This runs very quickly with negligible impact to the system resources.

Note that **SUMMARY** statistics are different from regular column or index statistics and doesn't contain any histogram. Also, this should not be confused with summary information for a column or index specific histogram (also referred as interval[0]). When **SUMMARY** option is specified in a collect statistics statement, no column or index specification is allowed.

The **SUMMARY** statistics are automatically collected when statistics are collected or recollected on any column or index either with index-style syntax. Therefore, it is not required to explicitly submit a request to collect summary statistical information when statistics are being collected or recollected for individual columns or indexes.

Statistics Data – What is Collected? (2 of 2)



Column/Index Summary Info

- contains general information about the index/column that statistics was collected on

Bias Values and Frequencies

- a list of highly skewed values along with the frequencies for each value

Detail intervals

- the remaining non-skewed values are spread equally across a default of 250 intervals, with each interval carrying summarized demographics

Previous Summary Info Record(s)

- SummaryInfo data from previous collections is saved at the end of the histogram (when statistics are recollected)

In addition to the Summary option in Teradata 14.0, collecting statistics has been enhanced to capture more data demographic information so that the Optimizer can generate more accurate plans. You can specify up to 500 intervals with a specific collect statistics statement. With a default of 250 intervals, each interval can characterize 0.4 percent of the data.

The DBSControl utility has an internal parameter (#127 - MaxStatsInterval) – default is 250.

The increase in the number of statistics intervals:

- Improves single table cardinality estimates that are crucial for join planning. Having more intervals gives a more granular view of the demographics.
- Increases the accuracy of skew adjustment because of the higher number of modal frequencies that can be stored in a histogram.
- Does not change the procedure for collecting or dropping statistics, although it affects the statistics collected.

Remember that a statistics are maintained within a data row that has a maximum length of approximately 64K. This 64K limit can impact the number of biased values, intervals, etc. depending on the data type of the columns/index and the MAXVALUELENGTH specified.

If MAXVALUELENGTH is not specified, the system uses a default maximum value length as 25. For single-column statistics of non-LATIN type, the default is used as 25 Unicode characters which translate to 50 bytes. In all other cases, it is used as 25 bytes.

Statistics Collection – Table Level

Table level summary statistics are also automatically collected when you collect statistics for a column(s) or index.

- Provides row count, average row size, and other information to the optimizer for better extrapolations

However, it is possible to just collect summary statistics for a table.

COLLECT SUMMARY STATISTICS ON tablename;

- If only SUMMARY statistics are collected, no histogram is built
- This might be useful for large tables with a UPI as this is a very fast collection

When you recollect statistics, a table summary record is saved automatically.

- To view the history information:

SHOW SUMMARY STATISTICS VALUES ON tablename;

- More trend-based extrapolation decisions can be performed because the optimizer see the pattern in growth for the table
 - Some statistics recollections may be skipped if history is adequate

Starting with Teradata 14.0, table level summary statistics are also automatically collected when you collect statistics for a column(s) or index.

This is often referred to as Table Summary statistics.

Collecting at a summary level can be very helpful after a large load job when you don't have time for full recollections of the table's statistics.

Table Level Summary Statistics

```
SHOW SUMMARY STATISTICS VALUES On Claim_RP;
```

```

/** TableLevelSummary **/
/* Version */6,
/* NumOfRecords */7,
/* Reserved */0.000000,
/* Reserved */0.000000,
/* SummaryRecord[1] */
/* Temperature */0,
/* TimeStamp */TIMESTAMP '2019-12-19 02:45:45-00:00',
/* NumOfAMPs */16,
/* OneAMPsSampleEst */30103376,
/* AllAMPsSampleEst */30129858,
/* RowCount */30000000,
/* DelRowCount */0,
/* PhyRowCount */30129858,
/* AvgRowsPerBlock */980.221619,
/* AvgBlockSize (bytes) */32768.000000,
/* BLCpctCompressed */100.00,
/* BLCBlkUcpuCost (ms) */0.005413,
/* BLCCompRatio */88.000000,
/* AvgRowSize */130.000000,
:
/* SummaryRecord[2] */
/* Temperature */0,
/* TimeStamp */TIMESTAMP '2019-12-19 02:33:17-00:00'
/* NumOfAMPs */16,

```

Only if table has multiple temperatures

Actual number of rows

Columnar tables only

Block Compression Ratio

30M Claims were stored for 10 years and the claim dates were between 2010 and 2019.

```

CREATE SET TABLE TFACT.Claim_RP, FALLBACK,
  NO BEFORE JOURNAL,
  NO AFTER JOURNAL,
  CHECKSUM = DEFAULT,
  DEFAULT MERGEBLOCKRATIO
(
  claimid INTEGER NOT NULL,
  custid INTEGER NOT NULL,
  claimstatus CHAR(1) DEFAULT '0 ',
  claimcost DECIMAL(9,2),
  claimdate DATE FORMAT 'YYYY-MM-DD' NOT NULL,
  claimpriority SMALLINT DEFAULT 0 ,
  agentname CHAR(16) CHARACTER SET LATIN NOT CASESPECIFIC,
  claimlocation SMALLINT DEFAULT 0,
  paymentcode SMALLINT DEFAULT 0,
  claimcomment CHAR(79)
    DEFAULT 'Medical - Optional Comment completed by Agent')
PRIMARY INDEX (claimid)
PARTITION BY RANGE_N
  (claimdate BETWEEN DATE '2011-01-01' AND DATE '2020-12-31'
    EACH INTERVAL '1' DAY, NO RANGE);

```

Statistics Example (cont.)

```
COLLECT STATISTICS COLUMN claimdate ON CLAIM_RP;
SHOW STATISTICS VALUES COLUMN claimdate ON Claim_RP;
```

SummaryInfo	Bias Value	Bias Value	Detail Interval #1	Detail Interval #2		248 more Intervals
			2011-01-02	2011-01-25	2011-01-30	
Summary Section	Bias Value	Bias Value	Detail Interval #1	Detail Interval #2		
	Value 1 2011-01-31	Value 2 2011-02-28	Max Value – 2011-01-25 Modal Value – 2011-01-02 Modal # of Rows – 4296 Other Values – 23 Other Rows – 50,064	Max Value – 2011-01-30 Modal Value – 2011-01-28 Modal # of Rows – 2215 Other Values – 4 Other Rows – 8686		
	Frequency 7,242	Frequency 13,752				

SQL Statement

```
SELECT * FROM Claim_RP WHERE claimdate = DATE '2011-01-02';
SELECT * FROM Claim_RP WHERE claimdate = DATE '2011-02-28';
SELECT * FROM Claim_RP WHERE claimdate = DATE '2011-01-05';
SELECT * FROM Claim_RP WHERE claimdate
    BETWEEN DATE '2011-01-05' AND DATE '2011-01-08';
```

Note: The Optimizer estimates that 4 days @ 2176.7 values is 8707.

Optimizer assumes

4296

8707

Statistics were collected on the claimdate column of a Claim table. Using the information shown with the SHOW STATISTICS VALUES on the previous page, the following SQL statements will generate the following estimates for the optimizer.

If the user executes:

```
SELECT * FROM Claim_RP WHERE claimdate = DATE '2011-01-02';
```

The optimizer will assume there are 4296 claims on this date since '2011-01-02' is the modal value for the first statistics interval.

If the user executes:

```
SELECT * FROM Claim_RP WHERE claimdate = DATE '2011-02-28';
```

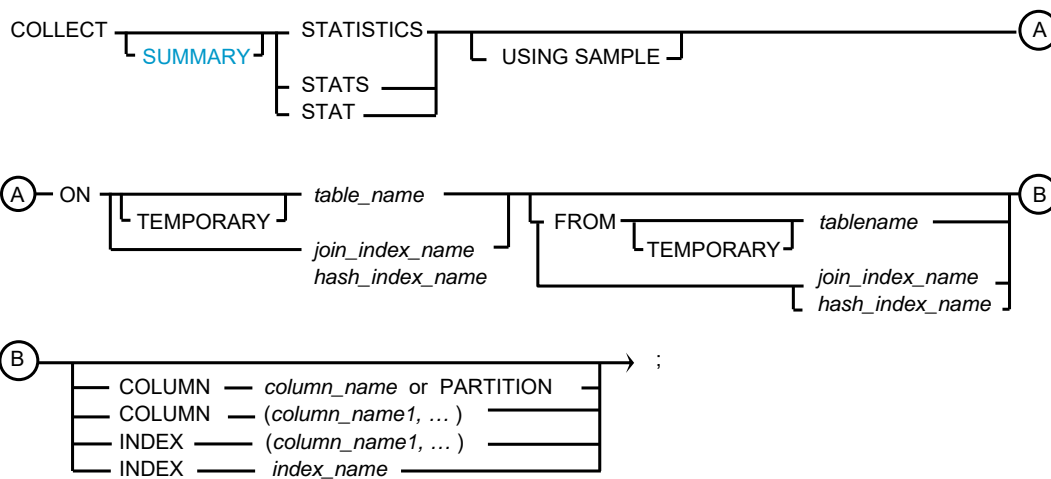
The optimizer will assume there are 13,752 claims for this date since '2011-02-28' is a high bias value.

If the user executes:

```
SELECT * FROM Claim_RP WHERE claimdate = DATE '2011-01-05';
```

The optimizer will assume there are 2177 claims for this date since '2011-01-05' is not a high bias or modal value. How is 2177 determined? Since there are 50,064 other claims in the interval and 23 other claim dates, the optimizer simply divides 50,064 by 23 which equals 2176.7 which is rounded up to 2178.

COLLECT STATISTICS Command



SUMMARY – collects table-level summary statistics – cardinality (number of rows), average # of rows per AMP, etc.

If collecting on a column or index, summary statistics are also collected automatically.

With this format, multiple COLUMN and/or INDEX specifications are not allowed.

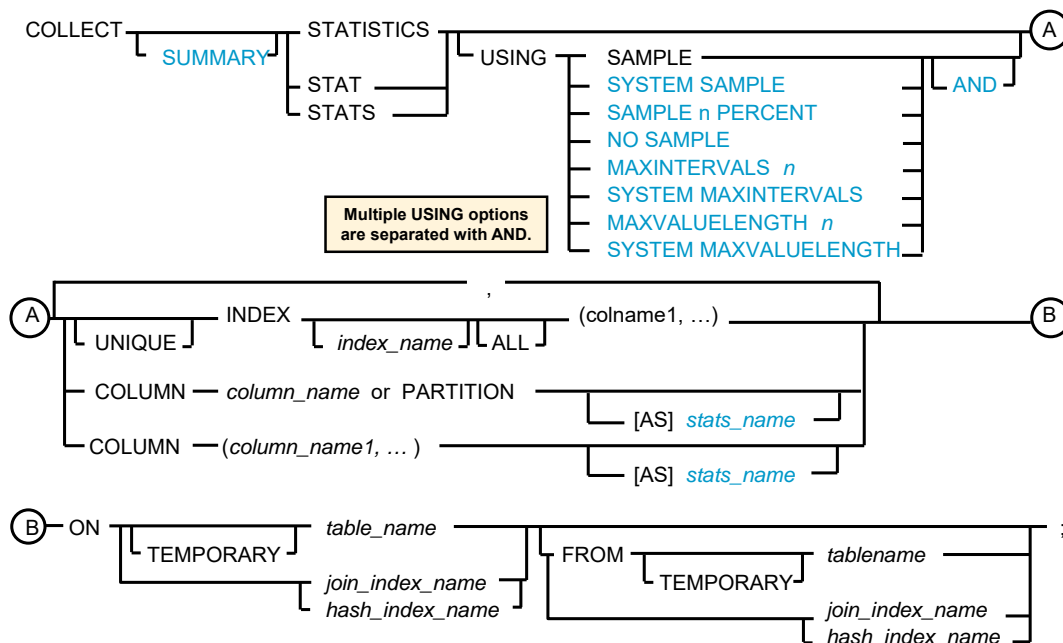
Miscellaneous Considerations

The maximum number of columns on which you may collect statistics has is 512 for a table. You can collect or recollect statistics on a combined maximum of 512 implicitly specified columns and indexes. An error results if statistics are requested for more than 512 columns and indexes in a table. This error condition could result during re-collection, when all statistics that have been previously collected are re-generated. For example, if you collected statistics six different times, each for a set of 100 columns, each of those collect statements would work, because they would each be under the 512 limit. However, if you re-collected the statistics the system would attempt to generate all 600 and would fail when it hit the 512 limit.

Increasing this limit allows users to issue larger re-collections involving more spool files. While this may increase system resource usage, it also provides better, more accurate information to the optimizer. Improved optimization results in more efficient execution plans, which significantly reduce system resource usage during query processing. Efficient system usage reduces overhead cost and time.

The bottom line on effective use of COLLECT STATISTICS is: Collect **all** and **only** what is needed to help the Optimizer make the best possible execution plans. Collecting more statistics than are needed or used wastes resources by putting extra processing and storage burdens on the system.

COLLECT STATISTICS Command (Index Format)



The rule of thumb with statistics, when you ask yourself whether to collect them or not on a specific column or set of columns is whether they improve the query plans or not. If they do, collect them, if they don't drop them.

This is no longer an issue starting with Teradata 14.0. You can override the maximum value length (based on column lengths by using the `MAXVALUELENGTH n` option. Teradata never truncates numeric values for single-column statistics. The system increases the interval size automatically if the specification is not sufficient to accommodate the full value for single-column statistics on numeric columns.

For multicolumn statistics, if the maximum interval size truncates numeric statistical data, Teradata automatically increases the maximum interval size to accommodate the numeric column on the maximum size boundary. A larger maximum value size causes Teradata to retain the value until the specified maximum is reached, which can enable better single-table and join selectivity estimates for skewed columns. However, you should be selective when increasing the size for the required columns because increasing the maximum value size also increases the size of the histogram, which can increase query optimization time. You can only specify this option if you also specify an explicit column or index.

NO SAMPLE

`NO SAMPLE` specifies to use a full-table scan to collect the specified statistics. You can only specify this option if you also specify an explicit index or column set.

Collecting Statistics Example

Initial Definition and Collection

EXPLAIN

```
COLLECT STATISTICS COLUMN custid, COLUMN claimdate, COLUMN PARTITION ON Claim_RP;
/* Example using the Index format */
```

Explanation

-
- 1) First, we lock TFACT.Claim_RP in TD_MAP1 for access.
 - 2) Next, we do an all-AMPs SUM step in TD_MAP1 to aggregate from TFACT.Claim_RP by way of an all-rows scan with no residual conditions, grouping by field1 (TFACT.Claim_RP.custid). Aggregate Intermediate Results are computed globally, then placed in Spool 3 in TD_Map1. The size of Spool 3 is estimated with no confidence to be 1,048,576 rows (30,408,704 bytes). The estimated time for this step is 1 minute and 56 seconds.
 - 3) Then we save the UPDATED STATISTICS for ('custid ') from Spool 3 (Last Use) into spool 5, which is built on a single AMP derived from the hash of the table id.
 - 4) We SKIP collecting STATISTICS for ('claimdate '), because no data change detected from the last statistics collection.
 - :
 - 7) We compute the table-level summary statistics from spool 15 and save them into spool 16, which is built on a single AMP derived from the hash of the table id.
 - 8) We lock DBC.StatsTbl in TD_DATADictionaryMAP for write on a RowHash.
 - :
 - 14) We spoil the statistics cache for the table, view or query.
 - 15) We spoil the parser's dictionary cache for the table.
 - 16) 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.

Initial Definition and Collection

Each of the statements applies to a separate column of the table. The COLUMN or INDEX parameters must be specified with COLLECT STATISTICS.

An alternative to the Index format is to issue separate collect stats commands.

```
COLLECT STATISTICS ON Claim_RP COLUMN custid;
COLLECT STATISTICS ON Claim_RP COLUMN claimdate;
COLLECT STATISTICS ON Claim_RP COLUMN PARTITION;
```

Locks and Concurrency

When you perform a COLLECT STATISTICS statement, the system places an ACCESS lock on the table from which the demographic data is being collected. The system places row-hash-level WRITE locks on the DBC.StatsTbl while collecting statistics at the column and index levels.

In general, COLLECT STATISTICS statements can run concurrently with DML statements, other COLLECT STATISTICS statements, DROP STATISTICS statements, and HELP STATISTICS statements against the same table. COLLECT STATISTICS can also run concurrently with a CREATE INDEX statement against the same table as long as they are not for the same index.

Refresh or Re-Collect Statistics

To refresh or recollect statistics for all of the previously collected columns and/or indexes, you can collect at the table level.

```
COLLECT STATISTICS ON Claim_RP;
```

Refresh statistics whenever 5 – 10% of the rows have changed:

- As part of batch maintenance jobs
- After significant periods of OLTP updating
- After new low and/or high values have extended the range of values

Notes:

- **Statistics collection as a process Industry-wide is CPU intensive** – since there is a lot of aggregation and computation of statistical data. A best practice is to drop statistics on columns not used

Statistics should not be “collected” once and then forgotten. They tend to become stale as the tables change, and can cause performance problems by misleading the Optimizer into making poor decisions. Always keep your statistics current and valid by refreshing them on a regular, production-schedule basis. To refresh or recollect statistics for all of the previously collected columns, simply execute the COLLECT STATISTICS command for the table.

Rollup Aggregations

Rollup Aggregations is a Collect Statistics feature enhancement starting with Teradata 14.0 and beyond.

If you collect stats on a column and also on a set of multi-columns and one of multi-columns includes the same single column, rollup aggregation is used. Basically, statistics is collected for the multi-column and these statistics are reused (rolled up) to produce the single column statistics. Teradata doesn't need to scan the table for the single column statistics. This only occurs if you specify the single column and multicolumn in the same COLLECT STATS statement OR if you recollect at the table level. This is done automatically when you collect at the table level.

Why are Collect Statistics done serially?

If you collect statistics on individual columns that are not part of a multicolumn set, Teradata will single thread the collections – they will be done one at a time. The Collect Statistics code has to pull the data independently for each stats definition. It basically has to do a large aggregation grouping on the stats columns.

Viewing Statistics

HELP STATISTICS Claim_RP;

Displays information about current statistics.

Date	Time	Unique Values	Column Names
19/12/19	02:15:48	30,000,000	*
19/12/19	02:00:06	3,620	claimdate
19/12/19	02:00:20	2,886	custid
19/12/19	02:33:17	3,620	PARTITION

(Effectively table summary or row count)

Note that the DATE and TIME show when statistics were last collected or refreshed.

4,000,000 rows were inserted into the Claim_RP table.

HELP CURRENT STATISTICS Claim_RP;

Date	Time	Unique Values	Column Names
19/12/19	02:42:34	32,866,916	*
19/12/19	02:42:34	3,967	claimdate
19/12/19	02:42:34	2,886	custid
19/12/19	02:42:34	3,620	PARTITION

(This is the new estimated row count)

(This is an extrapolated number of dates)

Without Collecting
Statistics to see
estimated changes

COLLECT STATISTICS ON Claim_RP;

HELP STATISTICS on Claim_RP; or HELP CURRENT STATISTICS ON Claim_RP; (Same result)

Date	Time	Unique Values	Column Names
19/12/19	02:48:07	34,000,000	*
19/12/19	02:48:07	3,620	claimdate

(This is the actual row count)

(This is the actual number of dates)

On Collecting
Statistics to see
actual changes

Help Statistics

HELP STATISTICS returns the following information about each column or index for which statistics have been COLLECTED in a single table:

Use Date and Time to help you determine if your statistics need to be refreshed or dropped. The example on this slide illustrates the HELP STATISTICS output for the Orders table.

To view details about the statistics for an index/column, use the **SHOW STATISTICS VALUES** option.

Help Index (not shown)

HELP INDEX is an SQL statement which returns information for every index in the specified table.

HELP INDEX returns the following information:

- Whether or not the index is unique
- Whether the index is a PI or an SI
- The name(s) of the column(s) which the index is based on
- The Index ID Number
- The approximate number of distinct index values
- Is the index hash-ordered, valued-ordered, or partitioned (H, V, P)

This information can be very useful in reading EXPLAIN output. Since the EXPLAIN statement only returns the Index ID number, you can use the HELP INDEX statement to determine the structure of the index with that ID.

Collect Statistics Using Sample

Sampling Statistics

Why: - The COLLECT STATISTICS operation can be very time consuming because it performs a full table scan and it sorts the data to compute the number of occurrences of each distinct value, moreover sometimes the tables can be in terabytes or even petabytes.

When: Sampling is useful for very large tables with reasonable distribution of data (not advisable for skewed tables). Its more appropriate for indexed columns. However, on PARTITION columns, full statistics are always collected.

How: Using the following command: -
COLLECT STATISTICS USING SAMPLE n PERCENT ...

Note: A single column cannot have a mix of sample and well as full statistics collected on it.

The SAMPLING option significantly reduces the resources consumed by COLLECT STATISTICS by collecting on only a percentage of the data. While it should not be used to replace all existing statistics collection, it can be used as an effective query tuning option where statistics are not being collected because of the required overhead.

COLLECT STATISTICS can be very time consuming because it performs a full table scan and it sorts the data to compute the number of occurrences of each distinct value. Most users accept this performance because it can be run infrequently and it benefits query optimization. Without statistics, query performance often suffers. The drawback to sampled statistics is that they may not be as accurate, which in turn may affect the quality of Optimizer plans. In most cases sampled statistics are better than no statistics.

Consider using sampling when:

- Collecting statistics on very large tables.
- Resource consumption from the collection process is a serious performance or cost concern.

Do not use sampling:

- On small tables.
- To replace all existing full scan collections.

Sampling Considerations

The system automatically determines the appropriate sample size to generate accurate statistics for good query plans and performance.

Collecting Statistics (Additional Options)

Given: Claim_RP NUPI on claimid, partitioned on claimdate
 NUSI on custid

To collect sample statistics using the system default sample:

```
COLLECT STATISTICS USING SYSTEM SAMPLE COLUMN (claimdate) ON Claim_RP;
```

To collect sample statistics by scanning 10 percent of the rows and use 100 intervals:

```
COLLECT STATISTICS USING SAMPLE 10 PERCENT AND MAXINTERVALS 100  
COLUMN (custid) AS custid_stats ON Claim_RP;
```

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

```
COLLECT STATISTICS USING SAMPLE 20 PERCENT AND MAXINTERVALS 250  
COLUMN (custid) AS custid_stats ON Claim_RP;
```

To display the COLLECT STATISTICS statements for a table:

```
SHOW STATISTICS ON Claim_RP;
```

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

```
SHOW STATISTICS VALUES COLUMN claimdate ON Claim_RP;
```

SAMPLE or SYSTEM SAMPLE

This option specifies to scan a system-determined percentage of table rows to collect the specified statistics. This option is not valid for single-table views. SAMPLE has the same meaning as SYSTEM SAMPLE and is provided for backward compatibility.

The SYSTEM SAMPLE option specifies to scan a system-determined percentage of table rows to capture the statistical information. Teradata might decide to sample 100% for the first few times before downgrading the sample percent to a lower level.

MAXINTERVALS n

Valid range is 0 – 500.

SYSTEM MAXINTERVALS

The system default is typically 250, but can be adjusted using DBSControl.

MAXVALUELENGTH n

This option specifies the maximum size to use for histogram values such as MinValue, ModeValue, MaxValue, etc. This option is valid for both tables and single-table views. The value for n must be an integer number.

For multicolumn statistics, Teradata concatenates the values and truncates them if necessary to fit into the specified maximum size.

Collecting Statistics on PARTITION

You can (and should) collect statistics on the system-derived PARTITION for row partitioned tables.

- Statistics on the PARTITION column provide information about partitions and allow the Optimizer to generate a more aggressive plan with respect to row partitioned tables
- Specifically, the Optimizer can use PARTITION statistics to estimate the cost of various operations more accurately, including:
 - Static partition elimination
 - Dynamic partition elimination
- The Optimizer can use this information to better estimate the query cost when there are a significant number of empty partitions

Example:

```
COLLECT STATISTICS ON TFACT.Claim_RP COLUMN PARTITION;
```

This feature allows you to collect statistics on the system-derived PARTITION column of a row partitioned table.

Statistics on the PARTITION column allow the Optimizer to generate an aggressive plan with respect to row partitioned tables. Specifically, the Optimizer can use PARTITION statistics to estimate the cost of various operations very accurately, including:

- Static partition elimination
- Dynamic partition elimination

When the Optimizer can use partition elimination more frequently, query efficiency and performance improves. In addition, the collection process itself for single-column PARTITION statistics is highly optimized, which significantly reduces their collection time and overhead.

The Optimizer can use this information to better estimate the query cost when there are a significant number of empty partitions. This means that you can have empty partitions for years into the future, and not be required to ADD and DROP partitions. If PARTITION statistics are not collected, empty partitions may cause the Optimizer to underestimate the number of rows in a partition.

When the Optimizer uses PARTITION statistics in creating its plan, the EXPLAIN will show the estimate with high confidence.

Copying STATISTICS

The COLLECT STATISTICS command includes a **FROM** option to copy statistics from one table to an identical target table.

```
COLLECT STATISTICS ON Claim_new FROM Claim;
```

The CREATE TABLE AS command includes an option to copy statistics from one table to another by including the "AND STATISTICS" option.

- The **CREATE TABLE ... AS ... WITH DATA AND STATISTICS;**

- In this case, a new table is created via a copy definition (DDL) of an existing table, data is copied from the source table to the target table, and the applicable statistics are replaced with the same statistics (histograms) from the target table

```
CREATE TABLE Customer_Test1 AS CUSTOMER WITH DATA AND STATISTICS;
```

- The **CREATE TABLE ... AS ... WITH NO DATA AND STATISTICS;**

- In this case, a new table is created via a copy definition (DDL) of an existing table, no data is copied from the source table to the target table, and the applicable statistics are replaced with zeroed statistics (histograms)

```
CREATE TABLE Customer_Test2 AS CUSTOMER WITH NO DATA AND STATISTICS;
```

General Rules For CREATE TABLE AS ... WITH DATA AND STATISTICS. The following list of rules applies only to an AS ... WITH DATA AND STATISTICS clause.

- If there are no columns or indexes in the target table for which statistics are eligible to be copied, the system returns a warning message to the requestor.
- If you specify an explicit index definition for the target table, then the system does not copy PARTITION statistics from the source table to the target table.
- This is true for both single-column PARTITION statistics and for composite statistics on a column set that includes the system-derived PARTITION column.
- If no statistics have been collected on the specified source table column or index sets, the system ignores the AND STATISTICS option and returns a warning message to the requestor.
- If only a subset of the statistics from the source table are eligible to be copied to the columns and indexes of the target table, the system returns a warning message to the requestor.
- If the number of multicolumn statistics you specify to be copied to the target table exceeds the maximum number of multicolumn statistics allowed, then the system copies multicolumn statistics only up to the limit, does not copy the remainder of the multicolumn statistics to the target table, and reports a warning message to the requestor.
- If all columns in a MULTiset source table are non-unique, and if the target table is a SET table, then the system does not copy statistics to the target table. This is because of the possible violation of the rule of equal cardinalities in the source and target tables: if there are duplicate rows in the source table, the system eliminates them before copying to the target table, resulting in unequal cardinalities between the two tables.

Teradata AutoStats Features

Teradata Automated Statistics Management (**AutoStats**) features include:

- Implemented as a **portlet** in **Viewpoint**
- **Key Goal of AutoStats**: - Automate and provide intelligence to DBA tasks related to Optimizer Statistics Collections:
 - View all statistics on a system
 - Identify and collect **missing** (new) statistics needed for accurate query optimization
 - Detect **stale** statistics and promptly refresh them
 - Identify and remove **unused** statistics from ongoing maintenance
 - Prioritize the list of pending collections such that important and stale statistics are given precedence
- **Repository** – A system supplied database named TDSTATS that stores metadata for all stats collections Created by DIPSTATS
- **Open API** – in the form of SQL external stored procedures (XSPs) that perform important stats management operations
- **DBQL Enhancements** – Log Optimizer statistics recommendations and usage with dedicated DBQL logging option

Overview of Teradata AutoStats DBS Features (14.10)

- Managing statistics is labor intensive and mistake prone.
Customers complain of spending too much time and machine resources managing stats:
 - Which columns and indexes should we collect stats on?
 - How often do we need to refresh them?
 - How do we know if the stats we've been collecting are still being used?
- **Repository** – A system supplied database named TDSTATS that stores metadata for all stats collections. Created by DIPSTATS.
- **Open APIs** in the form of SQL external stored procedures (XSPs) that perform important stats management operations.
- **DBQL Enhancements** – Log Optimizer statistics recommendations and usage with dedicated DBQL logging option. Works with or without related option XMLPLAN enabled.

Statistics Highlight

- Recommendations for collect statistics include:
 - All non-unique indexes
 - For small tables (less than 10,000 rows per AMP) with a UPI, collect full statistics
 - For large tables with a UPI or nearly unique values, consider collecting SUMMARY statistics
 - Collect statistics on the key word PARTITION for row partitioned tables
 - Any non-indexed column used for set selection or join constraints
- Collected statistics are not automatically updated by the system
 - The user is responsible for refreshing collected statistics
 - Refresh statistics when 5% to 10% of the table's rows have changed
- The optimizer is more aggressive when it has COLLECTed STATISTICS
 - It is more conservative when it must rely on random AMP sampling
- Limitations
 - Maximum # of columns in a table on which you may implicitly recollect statistics is 512
 - Maximum number of column names within a given COLUMN list is 64
 - Maximum number of multiple column COLLECT STATS statements per table is 32

Remember to refresh statistics on a regular basis.

Dropping a secondary index for a single column deletes the index's definition from DBC.Indexes. However, the column definition and single-column statistics stored in DBC.StatsTbl will still exist.

General Recommendations

- Enable DBQL USECOUNT logging for all important databases whose table row counts may change over time. This provides information about updates, deletes and inserts performed on each table within the logged databases and contributes to better extrapolations.
- Benefit from the default system threshold option, which may allow some submitted statistics to be skipped, by turning on DBQL USECOUNT logging and building up statistic history records. Skipping can potentially reduce the resources required by statistics recollections.
- Expect to perform several full collections before statistic skipping or automatic downgrade to sampling is considered.
- Use the collection statement-level THRESHOLD option only for cases where there is a specific need to override the global threshold default.
- Consider collecting statistics (and providing a name) on SQL expressions if they are frequently used in queries and if they reference columns from a single table.

Summary

Now that you have completed this course, you will be able to:

- Explain how the Optimizer acquires statistics
- Describe random AMP sampling
- State a method for viewing statistics
- Learn how to utilize both the standard and index formats to collect statistics

Remember to refresh statistics on a regular basis.

Dropping a secondary index for a single column deletes the index's definition from DBC.Indexes. However, the column definition and single-column statistics stored in DBC.StatsTbl will still exist.

General Recommendations

- Enable DBQL USECOUNT logging for all important databases whose table row counts may change over time. This provides information about updates, deletes and inserts performed on each table within the logged databases and contributes to better extrapolations.
- Benefit from the default system threshold option, which may allow some submitted statistics to be skipped, by turning on DBQL USECOUNT logging and building up statistic history records. Skipping can potentially reduce the resources required by statistics recollections.
- Expect to perform several full collections before statistic skipping or automatic downgrade to sampling is considered.
- Use the collection statement-level THRESHOLD option only for cases where there is a specific need to override the global threshold default.
- Consider collecting statistics (and providing a name) on SQL expressions if they are frequently used in queries and if they reference columns from a single table.



Module 5: Optimizer and Statistics Collection Bring Up JupyterHub

teradata.

Let's now do the lab together



Thank you.

teradata.

©2023 Teradata



Module 6: Additional Index Options

Teradata Vantage MasterClass

Copyright © 2007–2023 by Teradata. All Rights Reserved.

Objectives

After completing this module, you will be able to:

- List Join Indexes
- Describe the situations where a Join Index can improve performance
- Describe how to create a “sparse join index”



Why Join Index

- **Minimize Base Table Access**
 - Join indexes are designed to permit queries to be resolved by accessing the index instead of accessing their underlying base tables
- **Query Optimization**
 - If a join index does not completely cover a query, the Optimizer can use it to provide better query optimization than scanning the base tables
- **Virtual Data Models**
 - The Join Index provides the flexibility of normalization while at the same time offering the opportunity to create alternative, denormalized virtual data models
- **Summary Tables**
 - Join indexes are useful for queries that aggregate columns from tables with large cardinalities
 - Aggregate Join Indexes help since it does not require denormalizing the logical design of the database

This slide provides the basic reasoning for why Join Indexes are even needed.

Some vendors refer to Join Indexes as **Materialized Views**.

Additional Index Choices

As part of the physical design process, the designer may choose to implement join and/or hash indexes for performance reasons.

- **Join Indexes**

- Can be created to pre-join multiple tables or on a single table to redistribute on a different column – effectively creating an alternative primary index on a foreign key column
- Can be used as a summary table to aggregate one or more columns from a table or tables
- Provides the optimizer with additional options and may use the join index if it “covers” the query
- For known queries, this typically will result in much better performance

- **Hash Indexes**

- Can only be placed on a single table and automatically includes the Primary Index value
- The syntax is similar to secondary index syntax, thus simpler SQL to code
- The Hash Index can be ordered by value or hash

Join indexes provide the performance benefits of prejoin tables without incurring update anomalies and without denormalizing your logical or physical database schemas. Although join indexes create and manage prejoins and, optionally, aggregates, they do not denormalize the physical implementation of your normalized logical model because they are not a component of the fully normalized physical model.

Join indexes are defined to reduce the number of rows processed in generating result sets from certain types of queries, especially joins. Like secondary indexes, users may not directly access join indexes. They are an option available to the optimizer in query planning. The following are properties of join indexes:

- Are used to replicate and “pre-join” information from several tables into a single structure.
- Are designed to cover queries, reducing or eliminating the need for access to the base table rows.
- Usually do not contain pointers to base table rows (unless user defined to do so).
- Are distributed based on the user choice of a Primary Index on the Join Index.
- Permit Secondary Indexes to be defined on the Join Index (except for Single Table Join Indexes), with either “hash” or “value” ordering.

Unlike traditional indexes, join indexes do not store “pointers” to their associated base table rows. Instead, they are a fast path *final* access point that eliminates the need to access and join the base tables they represent. They substitute *for* rather than point *to* base table rows.

Hash Indexes

- Contains properties of both secondary indexes and single table join indexes.
- Can only be placed on a single table and automatically includes the Primary Index value.
- The syntax is similar to secondary index syntax, thus simpler SQL to code.
- The Hash Index can be ordered by value or hash.

- This course will not cover hash indexes in detail.

Join Indexes (1 of 3)

- A Join Index is an optional index that may be created by the user
- **Single-table Join Index**
 - Distribute the rows of a single table on the hash value of a foreign key value
 - Facilitates the ability to join the foreign key table with the primary key table **without redistributing the data**
- **Multi-table Join Index**
 - Pre-join multiple tables; stores and maintains the result from joining two or more tables
 - Facilitates join operations by possibly **eliminating join processing** or by **reducing/eliminating join data redistribution**

There are multiple ways in which a join index can be used. Three common ways include:

- Single table Join Index — Distribute the rows of a single table on the hash value of a foreign key value
- Multi-Table Join Index — Pre-join multiple tables; stores and maintains the result from joining two or more tables.
- Aggregate Join Index — Aggregate one or more columns of a single table or multiple tables into a summary table

A join index is a system-maintained index table that stores and maintains the joined rows of two or more tables (**multiple table join index**) and, optionally, aggregates selected columns, referred to as an **aggregate join index**.

Join indexes are defined in a way that allows join queries to be resolved without accessing or joining their underlying base tables. A join index is useful for queries where the index structure contains all the columns referenced by one or more joins, thereby allowing the index to cover all or part of the query. For obvious reasons, such an index is often referred to as a covering index. Join indexes are also useful for queries that aggregate columns from tables with large cardinalities. These indexes play the role of pre-join and summary tables without denormalizing the logical design of the database and without incurring the update anomalies presented by denormalized tables.

Another form of join index, referred to as a **single table join index**, is very useful for resolving joins on large tables without having to redistribute the joined rows across the AMPs.

Join Indexes (2 of 3)

- Aggregate Join Index (AJI)
 - Aggregate (SUM or COUNT) one or more columns of a single table or multiple tables into a summary table and supports MIN/MAX aggregate operators
 - Facilitates aggregation queries by **eliminating aggregation processing**. The pre-aggregated values are contained in the AJI instead of relying on base table calculations
- Sparse Join Indexes
 - When creating any of the join indexes, you can include a WHERE clause to limit the rows created in the join index to a subset of the rows in the base table or tables

This slide highlights two options that are available with join indexes.

A Sparse Join Index is simply a term that is used when a join index is created with a WHERE condition. You can use a WHERE clause in the CREATE JOIN INDEX statement to limit the rows that are created in the join index. This effectively reduces the size (PERM space) of the join index. The rows included in the join index are a subset of the rows in the base table or tables based on an SQL query result.

A Global Index is simply a term that is used to define a join index that contains the Row IDs of the base table rows. This means that the user includes the ROWID as a user-defined column within the join index. Row IDs that are included within a join index always include the partition number (0 for non-partitioned tables), regardless if the base table is partitioned or not.

Join Indexes (3 of 3)

- Global Join Indexes
 - You can include the Row ID of the table(s) within the join index to allow an AMP to join back to the data row for columns not referenced (covered) in the join index
- Miscellaneous notes:
 - [Materialized Views](#) are implemented as [Join Indexes](#). Join Indexes improve query performance at the expense of slower updates and increased storage
 - When you create a Join Index, there is no duplicate row check – you don't have to worry about a high number of NUPI duplicates in the join index

This slide highlights two options that are available with join indexes.

A Sparse Join Index is simply a term that is used when a join index is created with a WHERE condition. You can use a WHERE clause in the CREATE JOIN INDEX statement to limit the rows that are created in the join index. This effectively reduces the size (PERM space) of the join index. The rows included in the join index are a subset of the rows in the base table or tables based on an SQL query result.

A Global Index is simply a term that is used to define a join index that contains the Row IDs of the base table rows. This means that the user includes the ROWID as a user-defined column within the join index. Row IDs that are included within a join index always include the partition number (0 for non-partitioned tables), regardless if the base table is partitioned or not.

Join Index Considerations (1 of 2)

Join Index considerations include:

- Join indexes are updated automatically as base tables are updated (additional I/O)
- Take up PERM space and will also be Fallback protected
- You can specify no more than 64 columns per referenced base table per join index
- BLOB and CLOB data types **cannot** be defined within a Join Index
- If a table has an associated join index
 - Cannot use TPT Load (FastLoad) and TPT Update (MultiLoad)
 - Use TPT Stream (TPump)
- With a multi-table join index, you can specify INNER, LEFT OUTER, and RIGHT OUTER joins, but not a FULL OUTER join

Join Index considerations include:

- You cannot specify a column with a data type of either BLOB or CLOB in the definition of a join index (nor for any other kind of index).
- Be aware that when you define a join index using an outer join, you must reference all the columns of the outer table in the select list of the join index definition. If any of the outer table columns are not referenced in the select list for the join index definition, the system returns an error to the requestor.

Because join indexes generated from inner joins do not preserve unmatched rows, you should always consider using outer joins to define simple join indexes, noting the following restrictions.

- Inequality conditions are not supported under any circumstances for ON clauses in join index definitions.
- Outer joins are not supported under any circumstances for aggregate join indexes.

Load Utilities — TPT Load cannot be used to load data into base tables that have an associated join index defined on them because join indexes are not maintained during the execution of TPT Load. The TPump utility, which perform standard SQL row inserts and updates, can be used because join indexes are properly maintained during the execution of such utilities.

Collecting Statistics — statistics should be collected on the primary index and secondary indexes of the Join Index to provide the Optimizer with baseline statistics, including the total number of rows in the Join Index.

Join Index Considerations (2 of 2)

In many respects, a Join Index is similar to a base table.

- You may create non-unique secondary indexes on its columns
- Perform COLLECT/DROP STATISTICS, DROP, HELP, and SHOW statements

Unlike base tables, you cannot do the following:

- Directly query or update join index rows
- Create unique secondary indexes on its columns
- Store and maintain arbitrary query results such as expressions

Join Index considerations include:

- You cannot specify a column with a data type of either BLOB or CLOB in the definition of a join index (nor for any other kind of index).
- Be aware that when you define a join index using an outer join, you must reference all the columns of the outer table in the select list of the join index definition. If any of the outer table columns are not referenced in the select list for the join index definition, the system returns an error to the requestor.

Because join indexes generated from inner joins do not preserve unmatched rows, you should always consider using outer joins to define simple join indexes, noting the following restrictions.

- Inequality conditions are not supported under any circumstances for ON clauses in join index definitions.
- Outer joins are not supported under any circumstances for aggregate join indexes.

Load Utilities — TPT Load cannot be used to load data into base tables that have an associated join index defined on them because join indexes are not maintained during the execution of TPT Load. The TPump utility, which perform standard SQL row inserts and updates, can be used because join indexes are properly maintained during the execution of such utilities.

Collecting Statistics — statistics should be collected on the primary index and secondary indexes of the Join Index to provide the Optimizer with baseline statistics, including the total number of rows in the Join Index.

Compressed and Non-Compressed Join Indexes

- A compressed join index gives:
 - Flexibility to split the columns into repeating and non-repeating values
 - All repeating values would be stored only once thereby compressing the data records being stored
- A non-compressed join index provides:
 - Additional flexibility than a compressed join index by allowing partitioning
 - Feature of MVC on a base table being carried into a non-compressed join index thereby reducing storage space
- **Recommendation: Normally use non-compressed join indexes** except when creating a Global Join Index

This slide illustrates the difference between a compressed and a non-compressed join index.

PERM Space Required

The amount of PERM space used by the compressed multi-table join index (previous example) and the non-compressed join index can vary.

In this example, the sizes (in bytes) of the two join indexes are:

<u>Join Index Name</u>	<u>Permanent Space</u>	<u>Type of Join Index</u>
Cust_Ord_JI1	20,848,640	non-compressed multi-table
Cust_Ord_JI2	16,949,248	compressed multi-table
Cust_Ord_JI3	26,558,464	non-compressed partitioned multi-table

Compressed Multi-Table Join Index

```
CREATE JOIN INDEX Cust_Ord_JI2 AS
  SELECT (C.custid AS c_custid, lastname, firstname),
         (orderid, orderstatus, ordertotal, orderdate)
  FROM    Customer C
  INNER JOIN Orders O
  ON      C.custid = O.custid
  PRIMARY INDEX (c_custid);
```

Fixed portion of Join Index
(max # columns is 64)

Repeating portion of Join Index
(max # of columns is 64)

Maximum # of columns for compressed join index is 128.

Fixed Portion

Variable Portion

c_custid	lastname	firstname	orderid	orderstatus	ordertotal	orderdate
111443	Federer	Roger	142292	C	500.25	01/10/2021
			145893	C	1393.95	01/12/2021
			157093	O	1025.45	01/31/2021
			135993	C	1025.37	12/14/2020
114000	Djokovic	Novak	142000	C	745.76	12/20/2020
			149600	C	945.40	01/05/2021
			154798	C	1540.50	01/07/2021
115809	Gauff	Cori	149698	C	2081.43	12/31/2020
			156599	C	680.91	01/05/2021
			101199	C	845.33	12/24/2020
			158999	O	1002.50	01/23/2021
			152399	C	945.20	01/07/2021

Within the join index, this is one row of data.

One row in Join Index.

One row in Join Index.

This slide includes an example of creating a Multiple Table Join Index using the repeating group option. The storage organization for join indexes supports a compressed format to reduce storage space.

If you know that a join index contains groups of rows with repeating information, then its definition DDL can specify repeating groups, indicating the repeating columns in parentheses. The column list is specified as two groups of columns, with each group stipulated within parentheses. The first group contains the fixed columns and the second group contains the repeating columns.

As another option, you can elect to store join indexes in value order, ordered by the values of a 4-byte column. Value-ordered storage provides better performance for queries that specify selection constraints on the value ordering column. For example, suppose a common task is to look up sales information by order date. You can create a join index on the Orders table and order it by order date. The benefit is that queries that request orders by order date only need to access those data blocks that contain the value or range of values that the queries specify.

Limitations

For a compressed multi-table join index, the maximum number of columns defined in the fixed portion is 64 and the maximum number of columns defined in the repeating portion is also 64. The total maximum number of columns in this type of join index is 128.

With a LEFT or RIGHT OUTER join, at least 1 column from each inner table must be NOT NULL.

FULL OUTER joins are not allowed with a compressed multi-table join index.

Non-Compressed Multi-Table Join Index

```
CREATE JOIN INDEX Cust_Ord_JI1 AS
  SELECT      C.custid as c_custid, lastname, firstname,
            orderid, orderstatus, ordertotal, orderdate
  FROM        Customer C
  INNER JOIN  Orders O
  ON          C.custid = O.custid
  PRIMARY INDEX (c_custid);
```

Max # columns per
referenced table is 64

Maximum # of columns for non-compressed join index is 2048.

Fixed Portion

Variable Portion

c_custid	lastname	firstname	orderid	orderstatus	ordertotal	orderdate
111443	Federer	Roger	142292	C	500.25	01/10/2021
111443	Federer	Roger	145893	C	1393.95	01/12/2021
111443	Federer	Roger	157093	O	1025.45	01/31/2021
111443	Federer	Roger	135993	C	1025.37	12/14/2020
114000	Djokovic	Novak	142000	C	745.76	12/20/2020
114000	Djokovic	Novak	149600	C	945.40	01/05/2021
114000	Djokovic	Novak	154798	C	1540.50	01/07/2021
115809	Gauff	Cori	149698	C	2081.43	12/31/2020
115809	Gauff	Cori	156599	C	680.91	01/05/2021
115809	Gauff	Cori	101199	C	845.33	12/24/2020
115809	Gauff	Cori	158999	O	1002.50	01/23/2021
115809	Gauff	Cori	152399	C	945.20	01/07/2021

Within the join index, each order is represented by a separate join index row.

Options that are available with a non-compressed JI.

- Partitioning
- MVC on base tables is carried into a non-compressed JI.
- Unique Primary Index (13.10 feature)

This slide includes an example of creating a Multiple Table Join Index without using the repeating group option.

The storage space in this example for the non-compressed multi-table join index will be higher.

For a non-compressed multi-table join index, the maximum number of columns defined per referenced table is 64. The total maximum number of columns in this type of join index is 2048.

Example 1 – Does a Join Index Help?

List the valid customers who have open orders?

```
SELECT      C.custid, C.lastname, C.firstname,
            O.orderdate
FROM        Customer C
INNER JOIN  Orders O
ON          C.custid = O.custid
WHERE       orderstatus = 'O'
ORDER BY    1;
```

[SQL Query](#)

Without Join Index

With Join Index

[Time](#)

14.63 seconds

0.15 seconds

<u>custid</u>	<u>lastname</u>	<u>firstname</u>	<u>orderdate</u>
1460114	Cobb	Finn	02/28/2021
1514800	Mosley	Delilah	01/28/2021
1516554	Harper	Meredith	01/16/2021
2571123	Moon	Bently	12/28/2020
4423625	Simon	Dixie	12/19/2020
5772681	Paul	Damarion	12/31/2020
5915196	Brewer	Lucca	01/16/2021
9461517	Barnett	Tate	03/16/2021
9602794	Thompson	Jared	02/17/2021

All referenced columns are part of the Join Index.

Optimizer picks Join Index rather than doing a join.

[Join Index covers query and helps this query.](#)

The non-compressed Join Index was used for this example.

Note: Statistics were collected on the custid and orderdate columns of the Orders table and on the city, lastname, and zipcode columns of the Customer table.

This partial EXPLAIN is **without** a Join Index.

- 4) We do an all-AMPs RETRIEVE step in TD_MAP1 from TFACT.O by way of an all-rows scan with a condition of ("TFACT.O.orderstatus = 'O'") into Spool 2 in TD_MAP1 (all_amps), which is redistributed by the hash code of (TFACT.O.custid) to all AMPs in TD_MAP1. Then we do a SORT to order Spool 2 by row hash. The size of Spool 2 is estimated with high confidence ...
 - 5) We do an all-AMPs JOIN step in TD_MAP1 from TFACT.C by way of a RowHash match scan with no residual conditions, which is joined to Spool 2 (Last Use) by way of a RowHash match scan. TFACT.C and Spool 2 are joined using a merge join, with a join condition of ("TFACT.C.custid = custid"). The result goes into Spool 1 (group_amps), which is built locally on the AMPs. Then we do a SORT to order Spool 1 by the sort key in spool field1 (TFACT.C.custid). The size of Spool 1 is estimated with low confidence ...
- > The contents of Spool 1 are sent back to the user as the result of statement 1. The total estimated time is 14.63 seconds.

This partial EXPLAIN is **with** a non-compressed Join Index.

- 3) We do an all-AMPs RETRIEVE step in TD_MAP1 from TFACT.CUST_ORD_JI1 by way of an all-rows scan with a condition of ("TFACT.CUST_ORD_JI1.orderstatus = 'O'") into Spool 1 in TD_MAP1 (group_amps), which is built locally on the AMPs. Then we do a SORT to order Spool 1 by the sort key in spool field1 (TFACT.CUST_ORD_JI1.c_custid). The size of Spool 1 is estimated with no confidence to be 50,205 rows (2,359,635 bytes). The estimated time for this step is 0.15 seconds.
- > The contents of Spool 1 are sent back to the user as the result of statement 1. The total estimated time is 0.15 seconds.

Example 2 – Does a Join Index Help?

List the valid customers and their addresses who have open orders?

```
SELECT  C.custid, C.lastname, C.firstname,
        C.address, C.city, C.statecode,
        O.orderdate
FROM    Customer C
INNER JOIN Orders O
ON      C.custid = O.custid
WHERE   O.orderstatus = 'O'
ORDER BY 1;
```

SQL Query
Without Join Index
With Join Index

Time
14.80 seconds
3.45 seconds

- Some of the referenced columns are **NOT** part of the Join Index. The Join Index does not cover the query but is used in this example
- A Join Index is used in this query and is merge joined with the Customer table

<u>custid</u>	<u>lastname</u>	<u>firstname</u>	<u>address</u>	<u>city</u>	<u>statecode</u>	<u>orderdate</u>
1460114	Cobb	Finn	2300 Madrona Ave	Carson City	NV	02/28/2021
1514800	Mosley	Delilah	903 La Pierre Ave	Jackson	MS	01/28/2021
1516554	Harper	Meredith	36 Main Street	New York	NY	01/16/2021
2571123	Moon	Bentley	4564 Long Beach Bl	Trenton	NJ	12/28/2020
4423625	Simon	Dixie	1083 Beryl Ave	Los Angeles	CA	12/19/2020
5772681	Paul	Damarion	4021 Eleana Way	St. Paul	MN	12/31/2020
5915196	Brewer	Lucca	5603 Main Street	Pierre	SD	01/16/2021
9461517	Barnett	Tate	917 Carnation Dr	Beavercreek	OH	12/16/2020
9602794	Thompson	Jared	203 Stratford Lane	Wapakoneta	OH	02/17/2021

This partial EXPLAIN is **with** a non-compressed Join Index.

- 4) We do an all-AMPs RETRIEVE step in TD_MAP1 from TFACT.CUST_ORD_JI1 by way of an all-rows scan with a condition of ("TFACT.CUST_ORD_JI1.orderstatus = 'O'") into Spool 2 (all_amps), which is built locally on the AMPs. The size of Spool 2 is estimated with no confidence to be 49,407 rows (1,630,431 bytes). The estimated time for this step is 0.13 seconds.
 - 5) We do an all-AMPs JOIN step in TD_MAP1 from TFACT.C by way of a RowHash match scan with no residual conditions, which is joined to Spool 2 (Last Use) by way of a RowHash match scan. TFACT.C and Spool 2 are joined using a merge join, with a join condition of ("c_custid = TFACT.C.custid"). The result goes into Spool 1 (group_amps), which is built locally on the AMPs. Then we do a SORT to order Spool 1 by the sort key in spool field1 (TFACT.CUST_ORD_JI1.custid). The size of Spool 1 is estimated with no confidence to be 50,205 rows (4,618,860 bytes). The estimated time for this step is 3.32 seconds.
 - 6) 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 3.45 seconds.

Example 3 – Partitioning a Join Index

```
CREATE JOIN INDEX Cust_Ord_JI3 AS
SELECT
  C.custid AS c_custid, C.lastname, C.firstname,
  O.orderid, O.orderstatus, O.Ordertotal, O.orderdate
FROM
  Customer C
INNER JOIN
  Orders O
ON
  C.custid = O.custid
PRIMARY INDEX
  (c_custid)
PARTITION BY RANGE_N (orderdate BETWEEN
  DATE '2012-01-01' AND DATE '2021-12-31' EACH INTERVAL '1' DAY);
```

```
COLLECT STATISTICS COLUMN c_custid, COLUMN orderdate, COLUMN PARTITION
ON Cust_Ord_JI3;
```

List the valid customers who have open Orders for January 31, 2021?

```
SELECT
  C.custid, C.lastname, C.firstname, O.orderdate
FROM
  Customer C
INNER JOIN
  Orders O
ON
  C.custid = O.custid
WHERE
  O.orderstatus = 'O'
AND
  O.orderdate = DATE '2021-01-31'
ORDER BY
  1;
```

[SQL Query](#)
Without Join Index
With Join Index

[Time](#)
13.79 seconds
0.01 seconds

The join index is used in this query and partition elimination will occur on the join index.

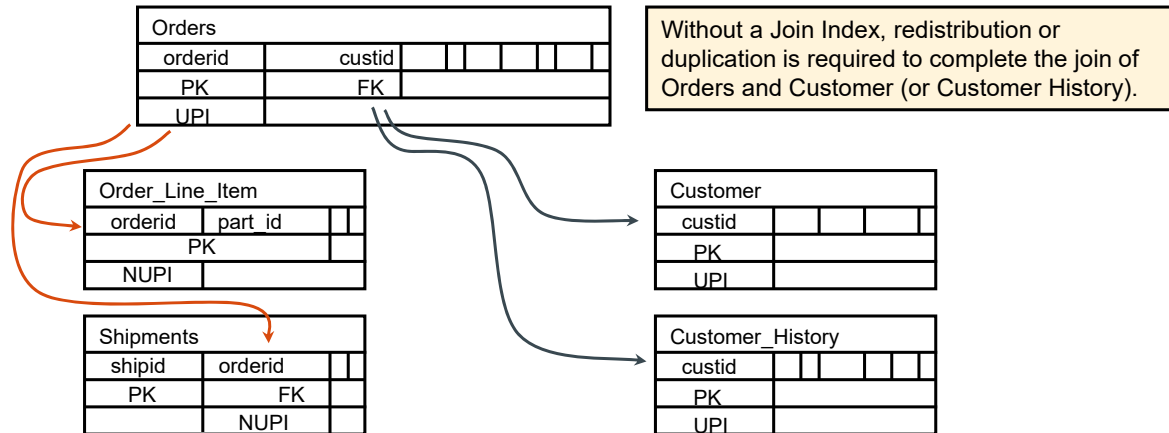
This slide includes an example of partitioning a non-compressed join index.

This EXPLAIN is **with** a non-compressed partitioned Join Index.

- 1) First, we lock TFACT.CUST_ORD_JI3 in TD_MAP1 for read on a reserved RowHash in a single partition to prevent global deadlock.
 - 2) Next, we lock TFACT.CUST_ORD_JI3 in TD_MAP1 for read on a single partition.
 - 3) We do an all-AMPs RETRIEVE step in TD_MAP1 from a single partition of TFACT.CUST_ORD_JI3 with a condition of ("TFACT.CUST_ORD_JI3.orderdate = DATE 2021-01-31") with a residual condition of ("(TFACT.CUST_ORD_JI3.orderdate = DATE '2021-01-31') AND (TFACT.CUST_ORD_JI3.orderstatus = 'O')") into Spool 1 (group_amps), which is built locally on the AMPs. Then we do a SORT to order Spool 1 by the sort key in spool field1 (TFACT.CUST_ORD_JI3.c_custid). The size of Spool 1 is estimated with no confidence to be 677 rows (31,819 bytes). The estimated time for this step is 0.01 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 0.01 seconds.

Join Index – Single Table (1 of 2)

- The [Single Table Join Index](#) is useful for resolving joins on large tables without having to redistribute the joined rows across the AMPs
- In some cases, this may perform better than building a multi-table join index on the same columns



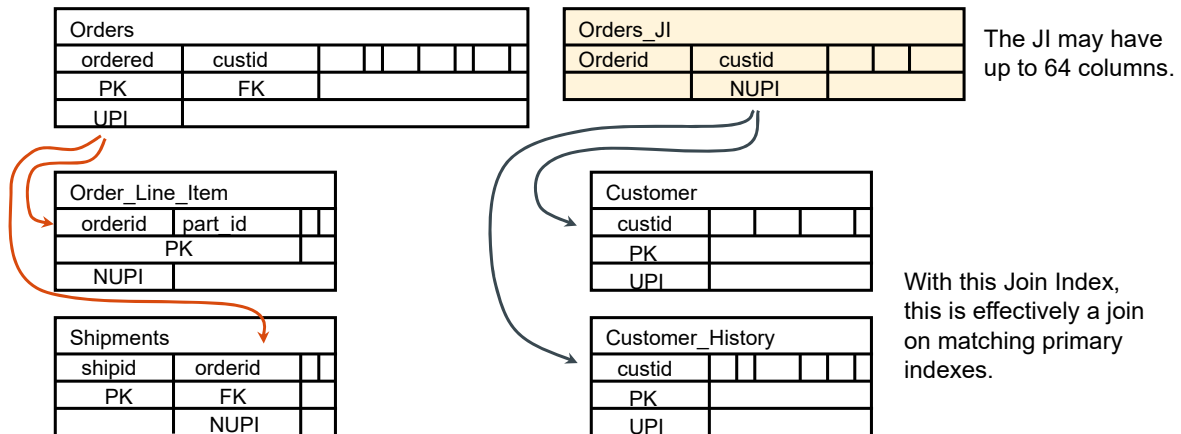
A denormalization technique is to replicate a column in a table to avoid joins. If an SQL query would benefit from replicating some or all of its columns in another table that is hashed on the join field (usually the primary index of the table to which it is to be joined) rather than the primary index of the original base table, then you should consider creating one or more single table join indexes on that table.

For example, you might want to create a single table join index to avoid redistributing a large base table or to avoid the possibly prohibitive storage requirements of a multi-table join index. For example, a single table join index might be useful for commonly made joins having low predicate selectivity but high join selectivity.

Join Index – Single Table (2 of 2)

Possible advantages include:

- Less update maintenance on the single table Join Index than a multi-table Join Index
- Maybe less storage space for a single table Join Index than for a multi-table Join Index



You can also define a simple join index on a single table. Only testing can determine what the best design choice is for a given set of tables, applications, and hardware configuration.

This example shows a technique where the join index is effectively substituted for the underlying base table. The join index has a primary index that ensures that rows are hashed to the same AMPs as rows in tables being joined. This eliminates the need for row redistribution when the join is made.

An example of creating a compressed single table join index is:

```
CREATE JOIN INDEX Orders_JI4compressed AS
  SELECT      (custid),
              (orderid, orderstatus, ordertotal, orderdate)
  FROM        Orders
  PRIMARY INDEX (custid);
```

PERM Space Required

The amount of PERM space used by the compressed multi-table join index (previous example) and single table join indexes is listed below. Remember that these tables are small and note that a compressed join index (with repeating data) usually requires less storage space.

However, if the base tables have MVC, then MVC will carry into the non-compressed join indexes and would be smaller and maybe even smaller than the compressed versions.

Creating a Join Index – Single Table

Non-Compressed

```
CREATE JOIN INDEX Orders_JI4 AS
  SELECT     orderid,
              custid,
              orderstatus,
              ordertotal,
              orderdate
  FROM        Orders
  PRIMARY INDEX (custid);
```

Non-Compressed and Partitioned

```
CREATE JOIN INDEX Orders_JI4A AS
  SELECT     orderid,
              custid,
              orderstatus,
              ordertotal,
              orderdate
  FROM        Orders
  PRIMARY INDEX (custid)
  PARTITION BY RANGE_N (orderdate BETWEEN
    DATE '2012-01-01' AND DATE '2021-12-31'
    EACH INTERVAL '1' DAY);
```

- The **Orders** base table is distributed across the AMPs based on the hash value of the **orderid** column (primary index of the base table)
- These Join Indexes effectively represent a subset of the **Orders** table (selected columns) and are distributed across the AMPs based on the hash value of the **custid** column
- The optimizer can use this Join Index to improve joins using the "customer id" to join with the Orders table because the Orders table does not have to be redistributed in the spool
- A single table join can also be created as "compressed" join index

Example 4 – Does Single Table Join Index Help?

List the valid customers who have open orders?

```
SELECT      C.custid, C.lastname, C.firstname,
            O.orderdate
FROM        Customer C
INNER JOIN  Orders O
ON          C.custid = O.custid
WHERE       orderstatus = 'O'
ORDER BY 1;
```

[SQL Query](#)

Without Join Index

With Join Index

[Time](#)

14.63 seconds

8.29 seconds

<u>custid</u>	<u>lastname</u>	<u>firstname</u>	<u>orderdate</u>
1460114	Cobb	Finn	02/28/2021
1514800	Mosley	Delilah	01/28/2021
1516554	Harper	Meredith	01/16/2021
2571123	Moon	Bentley	12/28/2020
4423625	Simon	Dixie	12/19/2020
5772681	Paul	Damarion	12/31/2020
5915196	Brewer	Lucca	01/16/2021
9461517	Barnett	Tate	03/16/2021
9602794	Thompson	Jared	02/17/2021

- The rows of the customer table and the Join Index are located on the same AMP
- A single table Join Index will help this query

This EXPLAIN is **with** a non-compressed Single Table Join Index.

- 1) First, we lock TFACT.ORDERS_JI4 in TD_MAP1 for read on a reserved RowHash to prevent global deadlock.
 - 2) Next, we lock TFACT.C in TD_MAP1 for read on a reserved RowHash to prevent global deadlock.
 - 3) We lock TFACT.ORDERS_JI4 in TD_MAP1 for read, and we lock TFACT.C in TD_MAP1 for read.
 - 4) We do an all-AMPs JOIN step in TD_MAP1 from TFACT.C by way of a RowHash match scan with no residual conditions, which is joined to TFACT.ORDERS_JI4 by way of a RowHash match scan with a condition of ("TFACT.ORDERS_JI4.orderstatus = 'O'"). TFACT.C and TFACT.ORDERS_JI4 are joined using a merge join, with a join condition of ("TFACT.C.custid = TFACT.ORDERS_JI4.custid"). The result goes into Spool 1 (group_amps), which is built locally on the AMPs. Then we do a SORT to order Spool 1 by the sort key in spool field1 (TFACT.C.custid). The size of Spool 1 is estimated with low confidence to be 117,995 rows (5,545,765 bytes). The estimated time for this step is 8.29 seconds.
 - 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. The total estimated time is 8.29 seconds.

Why Use Aggregate Join Indexes?

Summary Tables

- Queries involving aggregations over large tables are subject to high compute and I/O overhead. Summary tables often used to expedite their performance

Summary Tables Limitations

- Require the creation of a separate summary table
- Require initial population of the summary table
- Requires refresh of summary table
- Queries must access summary table, not the base table.
- Multiple “versions of the truth”

Aggregate Join Indexes

- Aggregate join indexes enhance the performance of the query while reducing the requirements placed on the user
- An aggregate join index is created similarly to a join index with the difference that sums, counts and date extracts may be used in the definition

Aggregate Join Index Advantages

- Do not require any user maintenance
- Updated automatically when base tables change (requires processing overhead)
- User will have up-to-date information in the index

Summary Tables

If the tables are large, query performance may be affected by the cost of performing the aggregations. Traditionally, when these queries are run frequently, users have built summary tables to expedite their performance. While summary tables do help query performance there are disadvantages associated with them as well. Summary Tables Limitations

- Require the creation of a separate table
- Require initial population of the table
- Require refresh of changing data, either via update or reload
- Require queries to be coded to access summary tables, not the base tables
- Allow for multiple versions of the truth when the summary tables are not up-to-date

Aggregate Indexes

The primary function of an aggregate join index is to provide the Optimizer with a performance, cost-effective means for satisfying any query that specifies a frequently made aggregation operation on one or more columns. The aggregate join index permits you to define a summary table without violating schema normalization. An aggregate index is created similarly to a join index with the difference that sums, counts and date extracts may be used in the definition.

Aggregate indexes do not require any user maintenance. When underlying base table data is updated, the aggregate index totals are adjusted to reflect the changes. While this requires additional processing overhead when a base table is changed, it guarantees that the user will have up-to-date information in the index.

Aggregate Join Index Properties

Aggregate Indexes are similar to other Join Indexes:

- It is automatically kept up to date without user involvement
- Never accessed directly by the user
- Optional and provide an additional choice for the optimizer
- TPT Load (FastLoad) and TPT Update (MultiLoad) can **NOT** be used to load tables for which indexes are defined

Aggregate Indexes differ from other Join Indexes:

- Uses the SUM, COUNT, MIN, or MAX functions
- Join index definitions only permit the extraction of years and months from a date column (e.g., EXTRACT (YEAR... or EXTRACT (MONTH ...))

Privileges required to create any Join Index:

- CREATE TABLE in the database or user which will own the join index, or INDEX privilege on each of the base tables. Additionally, you must have this privilege
- DROP TABLE rights on each of the base tables

Aggregate Indexes are different from other Join Indexes in that they:

- Use the SUM and COUNT functions.

Define an aggregate join index as a join index that specifies SUM or COUNT aggregate operations. No other aggregate functions are permitted in the definition of a join index. To avoid numeric overflow, the COUNT and SUM fields in a join index definition can be typed as FLOAT. If you do not assign a data type to COUNT and SUM, the system types them as FLOAT automatically. If you assign a type other than FLOAT, an error message occurs.

AJIs support MIN/MAX aggregate operators (14.10)

- MIN and/or MAX value of a column expression can be pre-computed and saved in an aggregate join index.
- MIN and MAX values materialized in an AJI are updated when the base table rows are updated and when rows are inserted into or deleted from the base table.

If you attempt to create an aggregate join index with AVG, you will receive the following error.

CREAT JOIN INDEX Failed 5464: Error in Join Index DDL. Only columns and SUM, COUNT, MIN, MAX, or non-aggregate expressions with aliases are allowed in the select list.

Aggregation Without an Aggregate Index

List the minimum, maximum, and total of ordertotals by year and month for every location.

```
SELECT    orderlocation
          ,EXTRACT (YEAR FROM orderdate) AS "Year"
          ,EXTRACT (MONTH FROM orderdate) AS "Month"
          ,MIN (ordertotal), MAX(ordertotal), SUM (ordertotal)
FROM      Orders
GROUP BY  1, 2, 3
ORDER BY  1, 2, 3;
```

An all-rows scan of base table is required.

The base table has 30,000,000 rows.

EXPLAIN without an Aggregate Index (Partial Listing)

- :
- 3) We do an all-AMPS SUM step in TD_MAP1 to aggregate from TFACT.Orders by way of an all-rows scan with no residual conditions, grouping by field1 (TFACT.Orders.orderlocation , EXTRACT(YEAR FROM (TFACT.Orders.orderdate)) ,EXTRACT(MONTH FROM (TFACT.Orders.orderdate))). Aggregate Intermediate Results are computed globally, then placed in Spool 3 in TD_MAP1. The size of Spool 3 is estimated with low confidence to be 3,000 rows (141,000 bytes). The estimated time for this step is 21.88 seconds.
 - 4) We do an all-AMPS RETRIEVE step in TD_MAP1 from Spool 3 (Last Use) by way of an all-rows scan into Spool 1 (group_amps), which is built locally on the AMPS. Then we do a SORT to order Spool 1 by the sort key in spool field1 (TFACT.Orders.orderlocation, EXTRACT(YEAR FROM (TFACT.Orders.orderdate)), EXTRACT(MONTH FROM (TFACT.Orders.orderdate))) . The size of Spool 1 is estimated with low confidence to be 3,000 rows (213,000 bytes). The estimated time for this step is 0.03 seconds.
 - 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. The total estimated time is 21.91 seconds.

```
SELECT    orderlocation
          ,EXTRACT (YEAR FROM orderdate) AS "Year"
          ,EXTRACT (MONTH FROM orderdate) AS "Month"
          ,MIN (ordertotal), MAX(ordertotal), SUM (ordertotal)
FROM      Orders
GROUP BY  1, 2, 3
ORDER BY  1, 2, 3;
```

- 3) We do an all-AMPS SUM step in TD_MAP1 to aggregate from TFACT.Orders by way of an all-rows scan with no residual conditions, grouping by field1 (TFACT.Orders.orderlocation ,EXTRACT(YEAR FROM (TFACT.Orders.orderdate)) ,EXTRACT(MONTH FROM (TFACT.Orders.orderdate))). Aggregate Intermediate Results are computed globally, then placed in Spool 3 in TD_MAP1. The size of Spool 3 is estimated with low confidence to be 3,000 rows (141,000 bytes). The estimated time for this step is 21.88 seconds.
 - 4) We do an all-AMPS RETRIEVE step from Spool 3 (Last Use) by way of an all-rows scan into Spool 1 (group_amps), which is built locally on the AMPS. Then we do a SORT to order Spool 1 by the sort key in spool field1 (TFACT.Orders.orderlocation, EXTRACT(YEAR FROM (TFACT.Orders.orderdate)), EXTRACT(MONTH FROM (TFACT.Orders.orderdate))). The size of Spool 1 is estimated with low confidence to be 3,000 rows (213,000 bytes). The estimated time for this step is 0.03 seconds.
- > The contents of Spool 1 are sent back to the user as the result of statement 1. The total estimated time is 21.91 seconds.

Creating an Aggregate Join Index

```
CREATE JOIN INDEX Monthly_Totals_AJI AS
SELECT  orderlocation
        ,EXTRACT (YEAR FROM orderdate) AS "Year"
        ,EXTRACT (MONTH FROM orderdate) AS "Month"
        ,MIN (ordertotal) AS "Min Ordertotal"
        ,MAX (ordertotal) AS "Max Ordertotal"
        ,SUM (ordertotal) AS "Order_Totals"
FROM    Orders
GROUP BY 1, 2, 3;
```

COLLECT STATISTICS		COLUMN orderlocation, COLUMN "Year", COLUMN "Month"			
ON		Monthly_Totals_AJI;			
HELP STATISTICS		Monthly_Totals_AJI;			
Date	Time	Unique Values	Column Names	Column Dictionary Names	Column SQL Names
20/12/26	09:17:36	1200	*	*	""
20/12/26	09:17:36	10	orderlocation	orderlocation	orderlocation
20/12/26	09:17:36	10	"Year"	Year	"Year"
20/12/26	09:17:36	12	"Month"	Month	"Month"

Example 5: Aggregation With an AJI

List the minimum, maximum, and total of ordertotals by year and month for every location.

```
SELECT    orderlocation
          ,EXTRACT (YEAR FROM orderdate) AS "Year"
          ,EXTRACT (MONTH FROM orderdate) AS "Month"
          ,MIN (ordertotal), MAX(ordertotal), SUM (ordertotal)
FROM      Orders
GROUP BY  1, 2, 3
ORDER BY  1, 2, 3;
```

An all-rows scan of aggregate join index is required.

EXPLAIN with an Aggregate Index

- 1) First, we lock TFACT.MONTHLY_TOTALS_AJI in TD_MAP1 for read on a reserved RowHash to prevent global deadlock.
 - 2) Next, we lock TFACT.MONTHLY_TOTALS_AJI in TD_MAP1 for read.
 - 3) We do an **all-AMPs RETRIEVE step in TD_MAP1 from TFACT.MONTHLY_TOTALS_AJI** by way of an **all-rows scan** with no residual conditions into Spool 1 (group_amps), which is built locally on the AMPs. Then we do a SORT to order Spool 1 by the sort key in spool field1 (TFACT.MONTHLY_TOTALS_AJI.orderlocation, TFACT.MONTHLY_TOTALS_AJI."Year", TFACT.MONTHLY_TOTALS_AJI."Month"). The size of Spool 1 is estimated with high confidence to be 1,200 rows (85,200 bytes). The estimated time for this step is 0.03 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 0.03 seconds.**

This slide illustrates execution of the same query with an aggregate join index

Sparse Join Indexes

Sparse Join Indexes

- Allows you to index a portion of the table using the WHERE clause in the CREATE JOIN INDEX statement to limit the rows indexed
- Any join index, whether simple or aggregate, multi-table or single-table, can be created as a sparse index
 - Ignore rows that are NULL or are most common
 - Index a time segment of the table – rows that relate to this quarter

Benefits

- A sparse index can focus on the portion of the table(s) that are most frequently used
 - Reduces the storage requirements for a join index
 - Faster to create or build
 - Makes access faster since the size of the Join Index is smaller
 - Better update performance on the base table when its indexes do not contain the most common value(s)

Another capability of the join index allows you to index a portion of the table using the WHERE clause in the CREATE JOIN INDEX statement to limit the rows indexed. You can limit the rows that are included in the join index to a subset of the rows in the table based on an SQL query result. This is also referred to as a “Partial Covering” join index.

Any join index, whether simple or aggregate, multi-table or single-table, can be sparse.

Customer Benefit

A sparse index can focus on the portion of the table(s) that is most frequently used.

- Reduces the storage requirements for a join index
- Makes access faster since the size of the JI is smaller

Like other index choices, a sparse JI should be chosen to support high frequency queries that require short response times. A sparse JI allows the user to:

- Use only a portion of the columns in the base table.
- Index only the values you want to index.
- Ignore some columns, e.g., nulls, to keep access smaller and faster than before.
- Avoid maintenance costs for updates

When the index size is smaller there is less work to maintain and updates are faster since there are fewer rows to update. A sparse JI contents can be limited by date, location information, customer attributes, or a wide variety of selection criteria combined with AND and OR conditions.

Creating a Sparse Join Index

```
CREATE JOIN INDEX Cust_Ord_JI5 AS
  SELECT      C.custid as c_custid, C.lastname, C.firstname,
              0.orderid, 0.orderstatus, 0.ordertotal, 0.orderdate
  FROM        Customer C
  INNER JOIN  Orders O
  ON          C.custid = 0.custid
  WHERE       EXTRACT (YEAR FROM 0.orderdate) =
              EXTRACT (YEAR FROM Current_Date)
  PRIMARY INDEX (c_custid);
```

Assume the current year is 2022.

```
SELECT      C.custid, C.lastname, 0.orderdate
FROM        Customer C
INNER JOIN  Orders O
ON          C.custid = 0.custid
WHERE       0.orderdate = '2022-01-18'
AND         0.orderstatus = '0';
```

The join index will be used for this SQL and the EXPLAIN estimated cost is 0.05 seconds.

This slide contains an example of creating a “Sparse Join Index”. The following EXPLAIN shows that the Sparse Join Index is used.

- ... (Locking steps)
- 3) We do an all-AMPs RETRIEVE step in TD_MAP1 from TFACT.CUST_ORD_JI5 by way of an all-rows scan with a condition of ("(TFACT.CUST_ORD_JI5.orderstatus = 'O') AND (((EXTRACT(DAY FROM (TFACT.CUST_ORD_JI5.orderdate)))= 18) AND ((EXTRACT(MONTH FROM (TFACT.CUST_ORD_JI5.orderdate)))= 1)))" into Spool 1 (group_amps), which is built locally on the AMPs. The size of Spool 1 is estimated with no confidence to be 6,547 rows (274,974 bytes). The estimated time for this step is 0.05 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 0.05 seconds.

PERM Space Required

The amount of PERM space used by this sparse join index as compared to the full join index is listed below.

<u>Join Index Name</u>	<u>Permanent Space</u>	<u>Type of Join Index</u>
Cust_Ord_JI1	20,848,640	non-compressed multi-table
Cust_Ord_JI5	5,799,936	sparse non-compressed multi-table

Global (Join) Indexes

A Global Index is a term used to define a join index

- Which contains the Row IDs of all of the tables in the join index

Example:

- Assume that you are joining 2 tables (Table_A and Table_B) and each has 100 columns
- The join index can include (at most) 64 columns from each base table
- You can include the ROWID as part of the 64 columns for each table (ex., A.ROWID and B.ROWID)
- Each join index subtable row will include the Row IDs of the corresponding base table rows for Table_A and Table_B
- The optimizer can build a plan that uses the join index to get most of the data and [can join back to either or both of the tables for the rest of the data](#)

A Global Index is a term used to define a join index that contains the Row IDs of the base table rows. Some queries are satisfied by examining only the join index when all referenced columns are stored in the index. Such queries are said to be covered by the join index.

A partial-covering join index takes less space than a covering join index, but in general may not improve query performance by as much. Not all columns that are involved in a query selection condition must be stored in a partial-covering join index. The benefits are:

- Disk storage space for the JI decreases when fewer columns are stored in the JI.
- Performance increases when the number of selection conditions that can be evaluated on the join index increases.

When a Row ID is included in a Join Index, 10 or 16 bytes are used for the Row ID (Part # + Row Hash + Uniqueness Value). This is true even if the base table is not partitioned.

Another use for a Global Join Index for a single table is that of a Hashed NUSI. This capability is captured in the Appendix.

Another option – use a single table Global Index as a “Hashed NUSI” – the join index contains the “secondary index column” and the Row IDs in the join index.

- Useful when a column is used as a secondary index, but the typical number of rows for a value is much less than the number of AMPs in the system.
- For queries with an equality condition on a fairly unique column, it changes:
 - Table-level locks to row hash locks
 - All-AMP steps to group-AMP steps

Customer Benefit

- Improved performance for certain class of queries.

- Some of the query improvement benefits that join indexes offer without having to replicate all the columns required to cover the queries resulting in better performance.
- Improved scalability for certain class of queries

Global Index – Multiple Tables

```
CREATE SET TABLE Customer
(custid      INTEGER NOT NULL,
 lastname    VARCHAR(15),
 firstname   VARCHAR(10),
 address     VARCHAR(100),
 city        VARCHAR(20),
 statecode   CHAR(2),
 zipcode     CHAR(5))
UNIQUE PRIMARY INDEX (custid);
```

- The Global Index contains those columns most frequently used in queries (covering join index)
- The [Global Index may be used to join back \(via the Row ID\) to the tables](#) when columns are referenced that are not part of the join index

```
CREATE SET TABLE Orders
(orderid      INTEGER NOT NULL,
 custid       INTEGER NOT NULL,
 orderstatus  CHAR(1),
 ordertotal   DECIMAL(9,2) NOT NULL,
 orderdate    DATE
              FORMAT 'YYYY-MM-DD' NOT NULL,
 orderpriority SMALLINT,
 clerkid      CHAR(16),
 shippriority SMALLINT,
 orderlocation SMALLINT,
 ordercomment VARCHAR(79))
UNIQUE PRIMARY INDEX (orderid);
```

```
CREATE JOIN INDEX Cust_Ord_JI6 AS
SELECT      C.custid AS c_custid, C.lastname, C.ROWID AS crid,
           orderid, orderstatus, orderdate, O.ROWID AS orid
FROM        Customer C
JOIN        Orders O
ON          C.custid = O.custid
PRIMARY INDEX (c_custid);
```

Join back simply means that the ROWID is carried as part of the join index. This permits the index to “join back” to the base row, much like a NUSI does. It is one of the features of Partial-Covering Join Indexes. An additional example is provided below.

```
EXPLAIN  SELECT  C.custid, C.lastname,
                C.address, C.city, C.statecode,
                O.orderdate
FROM      Customer C
INNER JOIN Orders O
ON        C.custid = O.custid
WHERE     O.orderstatus = '0'
ORDER BY  1;
```

This EXPLAIN shows that a Global Join Index is used.

- 5) We do an all-AMPs JOIN step in TD_Map1 from Spool 2 (Last Use) by way of an all-rows scan, which is joined to TFACT.C by way of an all-rows scan with no residual conditions. Spool 2 and TFACT.C are joined using a row id join, with a join condition of ("Field_1 = TFACT.C.ROWID"). The result goes into Spool 1 (group_amps), which is built locally on the AMPs. Then we do a SORT to order Spool 1 by the sort key in spool field1 (TFACT.CUST_ORD_JI6.custid). The size of Spool 1 is estimated with no confidence to be 50,205 rows (4,367,835 bytes). The estimated time for this step is 0.59 seconds.

Highlights

Teradata provides additional index choices that can be used to improve performance for known queries.

Reasons to use a Join Index:

- May be used to pre-join multiple tables
- May be used as an aggregate index
- The WHERE clause can be used to limit the number of rows in the join index (“Sparse Index”)
- Row ID(s) of a table (or tables) can be included to create a “Global Index”
- Secondary indexes can be created on a join index. Secondary indexes can be ordered by value or hash

Summary

Now that you have completed this course, you will be able to:

- List Join Indexes
- Describe the situations where a Join Index can improve performance
- Describe how to create a “sparse join index”



Module 6: Additional Indexes Bring Up JupyterHub

teradata.

Let's now do the lab together



Thank you.

teradata.

©2023 Teradata



Module 07: Compression

Teradata Vantage MasterClass

Copyright © 2007–2023 by Teradata. All Rights Reserved.

Objectives

After completing this course, you will be able to:

- Describe what is compression and what benefit it brings to Vantage
- Identify different types of Compression mechanisms in Vantage

What is Compression!

- It reduces the physical size of stored information
- Compression is used for the following reasons
 - Reduces storage costs
 - By storing more logical data per unit of physical capacity
 - Compression produces smaller rows, resulting in more rows stored per data block and fewer data blocks
 - Enhance system performance
 - There are fewer physical data to retrieve per row for queries
 - Compressed data remains compressed while in memory, thereby FSG cache can hold more rows, reducing the size of disk I/O

What is Compression

Compression is a technique of squeezing data in lesser space than actual and hence reduces the physical size of stored information. The goal of compression is to represent information accurately using the fewest number of bits. Compression methods are either logical or physical. Physical data compression re-encodes information independently of its meaning, while logical data compression substitutes one set of data with another, more compact set.

Compression is used for the following reasons.

- To reduce storage costs
- To enhance system performance

Compression reduces storage costs by storing more logical data per unit of physical capacity. Compression produces smaller rows, resulting in more rows stored per data block and fewer data blocks.

Compression enhances system performance because there is fewer physical data to retrieve per row for queries. Also, because compressed data remains compressed while in memory, the FSG cache can hold more rows, reducing the size of disk I/O.

Most forms of compression are transparent to applications, ETL utilities, and queries. However, this can be less true of algorithmic compression (ALC), because a poorly performing decompression algorithm can have a negative effect on system performance, and in some cases a poorly written decompression algorithm can even corrupt data.

Experience with real world customer production databases with very large tables indicates that compression produces performance benefits for a table even when more than 100 of its columns have

been compressed.

Compression in Vantage



Algorithmic Compression (ALC)

- Column Level compression
- User-defined compression and decompression algorithms using UDFs
- UDFs need to be mentioned during table creation/alteration



Multi-Value Compression (MVC)

- Column Level compression
- Upto 255 distinct values and NULL values can be compressed
- Also called Value List Compression (VLC)



Block Level Compression (BLC)

- Data Block at the file system level

The 3 types of compression are listed here.

MVC Considerations

- Compresses value(s) and NULL to zero space
- Compression is case-sensitive
- Compress columns where at least 10% of the rows participate
- Multi-value Compression and VARCHAR are both carried into intermediate spool files

Cannot be compressed

- | | |
|-------------------------------|-----------------------------------|
| × Primary Index | × Geospatial column |
| × Partition column | × BLOB/CLOB |
| × Standard RI (PK-FK) columns | × UDT (User Defined Types) |
| × Identity columns | × Derived, Volatile table columns |

Multi-value Compression

Multi-value Compression (MVC) is sometimes referred to as Value List Compression (VLC). MVC allows specification of more than one compressed value on a column. This feature allows multiple values (up to 255 distinct values plus NULL to be compressed on a column. This compression capability reduces storage cost by storing more logical data per unit of physical capacity. Performance is improved because there is less physical data to retrieve during scan-oriented queries.

Therefore, Multi-value Compression is a feature that reduces the effective price of logical data storage capacity, and usually improves query performance because of reduced I/O. With MVC, the additional CPU time required to compress/decompress is minimal.

Compression does not require extra computer resources to uncompress the block or row. Performance is enhanced since data remains compressed in memory so that the cache can hold more logical rows. The compressed values are stored in the table header row.

The maximum size of a compress value is 510 bytes (starting with TD 13.10)

The default for all columns is nullable and not compressible which means that, unless otherwise specified, NULL values are allowed and Teradata will not automatically compress columns no matter what values are in them. You can override the default by using the **COMPRESS** clause at the column level when creating (or altering) a table.

The COMPRESS clause works in two different ways:

- When issued by itself (without a value or values), COMPRESS only causes all NULL values for that column to be compressed to zero space.
- When issued with an argument (e.g., COMPRESS value), the COMPRESS clause will compress every occurrence of the value in that column to zero space as well as cause every

NULL value to be compressed.

You **CANNOT** compress the following:

- Components of the primary index
- Partitioning columns
- Identity columns
- Volatile table columns
- Derived table columns
- Referencing and referenced columns with Standard RI cannot be compressed. Batch and Soft RI is allowed with compressed columns.
- Columns defined with a UDT, Period, Geospatial, BLOB, or CLOB data type.

Example 1 – MVC List of Values

Compress 5 popular last names and the top 5 most frequent countries.

```
CREATE TABLE People
(
  IdNumber BIGINT NOT NULL
  ,LastName CHAR(30) NOT NULL COMPRESS ('Smith', 'Johnson', 'Williams', 'Brown', 'Jones')
  ,FirstName CHAR(20) COMPRESS
  ,Address VARCHAR(100)
  ,Country VARCHAR(40) COMPRESS ('Australia', 'Bangladesh', 'Brazil', 'China', 'England')
  ,Gender CHAR(1) NOT NULL
);
```

MVC - Example

Example 2 – MVC Alter Table

ALTER TABLE allows you to add a new column with compressed values.

- Add an “Education” column
- UPPERCASE is specified, hence all values are compressed regardless of case

```
ALTER TABLE People
  ADD Education CHAR(12)
      UPPERCASE COMPRESS ('ELEMENTARY', 'MIDDLE', 'HIGH SCHOOL', 'COLLEGE GRAD', 'POST GRAD');
```

ALTER TABLE allows you to add additional compressed value to a list of values

- All of the compressed values must be listed
- SHOW table will always show compressed values in alphabetical sequence regardless of how they are entered

```
ALTER TABLE People
  ADD Country COMPRESS ('Australia', 'Bangladesh', 'Brazil', 'China', 'England', 'Czech Republic' );
```

MVC – Altering a Table - 2

Additional compressed values can be added to a column (ALTER). Include the complete list of compressed values with each ALTER command.

Algorithmic Compression Considerations (ALC)

- This is a Column-level physical compression mechanism
- User-defined compression and decompression algorithms using UDFs (User Defined Function)
 - UDFs need to be mentioned during table creation/alteration
- Suitable for columns
 - With the most unique values
 - Not good candidates for MVC
- ALC and MVC can be applied to the same column
 - ALC is invoked for non-null values that are not specified in the value compression list of the MVC specification

Algorithmic Compression (ALC)

In some cases, such as when column values are mostly unique, Algorithmic Compression (ALC) can provide better compression results than Multi-value Compression. Algorithmic Compression allows you to define your own compression and decompression algorithms and apply them to data at the column level.

A user can create their own compression algorithms as external C/C++ scalar UDFs, and then specify them in the column definition of a CREATE TABLE/ALTER TABLE statement. Teradata invokes these algorithms to compress and decompress the column data when the data is moved into the tables or when data is retrieved from the tables. ALC allows you to implement the compression scheme that is most suitable for data in particular column. The cost of compression and decompression depends on the algorithm chosen. You can specify ALC alone, or both MVC and ALC on the same column. If you define both on the same column, ALC is applied only to those non-null values that are not specified in the value compression list of the MVC specification.

Example 1 – ALC

Compress Unicode character data to UTF8 format

- Unicode character set uses 2 bytes
- UTF8 character set uses 1 byte (Latin characters)
- When data is predominantly Latin characters
 - Utilize Teradata-supplied function to compress space to 1 byte
 - The TransUnicodeToUTF8 function compresses all non-null values in the Description column

```
CREATE TABLE Products
( ProductID      INTEGER      NOT NULL
  ,Category      CHAR(10)     NOT NULL
  ,Description    VARCHAR(500) CHARACTER SET UNICODE
                                COMPRESS USING TD_SYSFNLIB.TransUnicodeToUTF8
                                DECOMPRESS USING TD_SYSFNLIB.TransUTF8ToUnicode
);
```

ALC - Example

Teradata provides domain-specific functions for compressing and decompressing data. These functions are stored in the TD_SYSFNLIB system database. Two of these algorithms are:

- Compress TransUnicodeToUTF8 - takes Unicode character input and stores it in UTF8 format. This is useful when the input data is predominantly Latin because UTF8 uses one byte to represent Latin characters and Unicode uses two bytes.
- Decompress TransUTF8ToUnicode - takes the data previously compressed using the TransUnicodeToUTF8 function and converts it back to Unicode.

Block Level Compression Considerations (BLC)

- Capability to perform compression on entire data blocks at the file system level
- BLC most applicable for large tables, i.e., in uncompressed form, are more than 5 times as large as system memory
- BLC can reduce the I/O demand

Compression Objects	BLC Applicable
Primary & Fallback subtables	Either both are compressed or both are uncompressed
Secondary Indexes	Cannot be compressed
Join Index	Base Tables compressed values can be carried into Join Index
Spool Data Blocks Volatile Table Global Temporary Table Permanent Journal	Compressed via system-wide tunable

BLC will compress/decompress only data blocks but will not be used on any file system structures such as Master/Cylinder indexes, the WAL log and table headers. Only the compressed form of the data block will be cached, each block access must perform the data block decompression.

- Only primary data subtable and fallback copies can be compressed. Both objects are either compressed or neither is compressed.
- Secondary Indexes (USI/NUSI) cannot be compressed but Join Indexes can be compressed since they are effectively user tables.
- Spool data blocks can be compressed via a system-wide tunable, as well as Temporary (Volatile and Global Temporary), WORK (sort Work space) & permanent journal.
- Once BLC is enabled on a table, reversion back to an earlier release for compressed tables is not allowed.
- There is a CPU cost to compress/decompress on whole data blocks and is generally considered a good trade since CPU cost is decreasing while I/O cost remains high.
- BLC most applicable for large tables, for example, those tables which, in uncompressed form, are more than 5 times as large as system memory.

Example – BLC

BLC can be applied as below:

- At System-wide using DBSControl settings
- At Table-level using Ferret utility

```
COMPRESS/UNCOMPRESS table
```

- While loading data, mentioning via Queryband

```
SET QUERY_BAND = 'BLOCKCOMPRESSION=YES/NO;' FOR SESSION;
```

BLC - Example

Teradata Compression Comparison

	Multi-value Compression (MVC)	Algorithmic Compression (ALC)	Block Level Compression (BLC)
Ease of Use	Easy to apply to well-understood data columns and values.	Easy to apply at column with CREATE TABLE.	Set once and forget.
Analysis Required	Need to analyze data for common values.	Use Teradata algorithms or user-defined compression algorithms to match unique data patterns.	Analyze CPU overhead trade-off. Turn on for all data on system <u>OR</u> apply on a per-table basis.
Flexibility	Works for a wide variety of data and situations.	Automatically invoked for values not replaced by MVC.	Automatically combined with other compression mechanisms.
Performance Impact	Reduced I/O NO / Minimal CPU usage.	Depends on compression algorithm used.	Reduced I/O due to compressed data blocks. CPU cycles are used to compress/decompress.
Applicability	Replaces common values	Industry data, UNICODE, Latin data	All Data

A customer can choose any combination or all three on a column/table.

The chart on this slide su**Considerations for Compression**

Multi-value Compression (value list) is extended to intermediate spool files. Thus, when compressed columns are selected, compression is propagated to resulting spool files. Without compression for spool files, the intermediate join results for compressed tables can be very large, causing the need for additional spool space. Uncompressed VARCHAR data is also carried into spool.

Try to avoid compressing columns whose NULL values will be changing. When the value changes, the column will expand and you may get block growth and/or split operations. In practice, there may be exceptions to this rule. For example, it might be good to compress the NULL values in columns related to shipping an order until the order is actually shipped.

Additional sizing considerations:

- Adding a column that is not compressible expands all rows.
- Adding a column that is compressible and there are no spare presence bits expands all rows.
- Adding an additional value (or values) to a compressed set of values may cause additional presence bits to be allocated. If there are no spare presence bits, then additional presence bytes are allocated and all the rows (without that compressed value) are expanded.
- Dropping a column changes all row sizes where data is present.
- The maximum number of characters that can be listed in a COMPRESS clause is 8,188.

mmarizes the three types of compression.

Key Highlights

- Teradata Vantage provides different types of compression techniques, viz., MVC, BLC, ALC
- MVC and ALC can be applied at the column level whereas BLC can be applied at the system/database/table level
- ALC is a pure user-defined compression technique that can be implemented by UDFs
- Columnar Table can have automatic compression techniques where Vantage decides if the candidate column is suitable for compression or not and if yes, then it decides which compression technique would be best

Summary

Now that you have completed this course, you will be able to:

- Describe what is compression and what benefit it brings to Vantage
- Identify different types of Compression mechanisms in Vantage



Module 7: Compression Bring Up JupyterHub

teradata.

Let's now do the lab together



Thank you.

teradata.

©2023 Teradata



Module 10: 4D Analytics

Teradata Vantage MasterClass

Copyright © 2007–2023 by Teradata. All Rights Reserved.

Objectives

After completing this module, you will be able to:

- Discuss Customer 360
- Describe how Vantage provides 4D Analytics
- Discuss Geospatial functions
- Describe Temporal Access
- Discuss Teradata Time Aware functions



Customer 360 View

1. Who are they?
2. Where are they?
3. How much is their Disposable Income?
4. When were they here?



A Business is Multi-Dimensional

- **WHAT**

- Understand what transactions occurred that are relevant

- **WHERE**

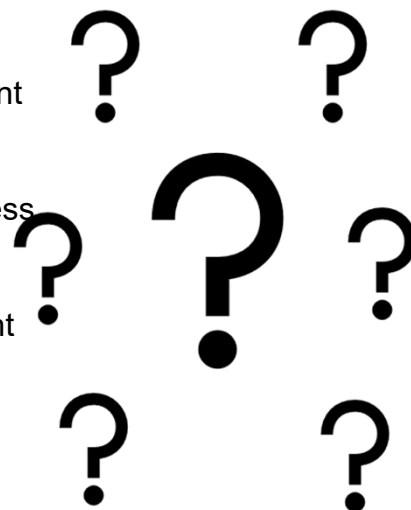
- Pinpoint where the locations are that impact the business

- **WHEN**

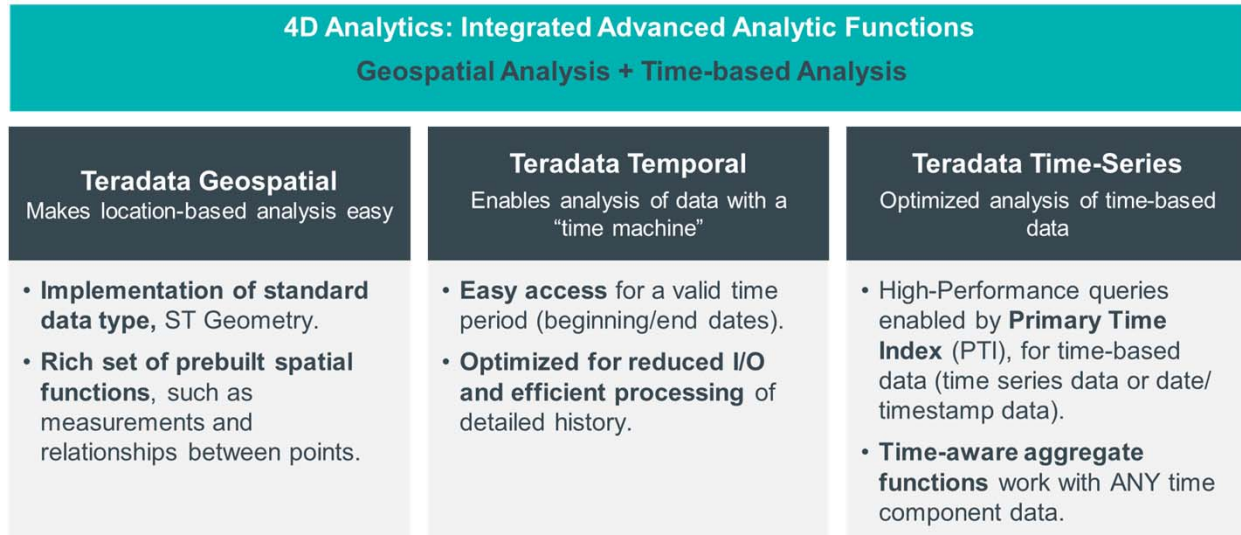
- Determine the period which provides the needed insight

- **WHY**

- Analyze the causes or events that benefit the business



Vantage – 4D Analytics



4D analytics is applying 3D geospatial location data and adding the 4th dimension of time.

4D Analytics primarily consist of the following 3 Teradata advanced analytic functions:

1. First is Teradata Geospatial, which has been around a while. Teradata Geospatial implements a standard data type, ST_Geometry natively within Teradata and makes location-based analysis easy and scalable. With Teradata Geospatial, you can use a rich set of prebuilt spatial functions, like measurements, relationship between two objects, spatial operators, and attributes.
2. Second, Teradata Time-Series is now available with Teradata Database 16.20 and has two key features. One is the primary time index, or PTI, which enables hash bucketing of time-based data (any time-based data: either time series data or date/ timestamp data), facilitating high-performance queries. Second is an introduction of new time-aware aggregate functions, GROUP BY TIME, which allows fast analysis of any time-based data.
3. Third is Teradata Temporal, which also has been around a while and enables analysis of data with a time machine, so to speak. It automatically handles update complexities related to temporal data and lets users easily specify full bi-temporal queries with no knowledge of effective date fields and complex condition clauses. It includes optimizations for reduced I/O and efficient processing of detailed history.

The 4D Analytics capability is an industry first because the PTI is a unique Teradata technology and we integrate all three analytics functions, including the time series, natively within a single analytics platform.

Importance of Geospatial Analytics

- Geospatial technology enables us to acquire data that is referenced to the earth and use it for analysis, modeling, simulations, and visualization
- Geospatial technology allows us to make informed decisions based on the importance and priority of resources most of which are limited in nature
- The 4 main ideas of Geospatial Information:
 - Create geographic data
 - Manage it in a database
 - Analyze and find patterns
 - Visualize it on a map



A geographic coordinate system is a method for describing the position of a geographic location on the earth's surface using spherical measures of latitude and longitude. These are measures of the angles (in degrees) from the center of the earth to a point on the earth's surface when the earth is modeled as a sphere. When using a spheroid (ellipsoid), latitude is measured extending a line perpendicular to the earth's surface to the equatorial plane. Except at the equator or a pole, this line will not intersect the center of the earth.

In each coordinate system, geodists use mathematics to give each position on Earth a unique coordinate. A geographic coordinate system defines two-dimensional coordinates based on the Earth's surface. It has an angular unit of measure, prime meridian and datum (which contains the spheroid).

Use Cases of Geospatial Analytics

- Every day, Geospatial Data powers millions of decisions around the world. It makes a big impact on our life, and we might not even realize it
- For example, we use Geospatial Data for:
 - Pinpointing new store locations
 - Reporting power outages
 - Analyzing crime patterns
 - Routing in-car navigation
 - Forecasting and predicting weather



Some particular use cases for Geospatial

Questions Spatial Analysis Can Answer

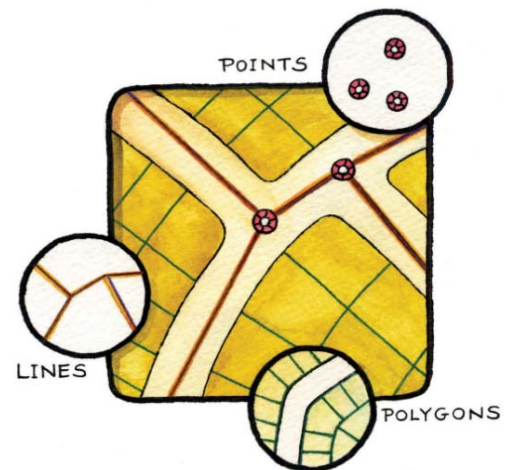
- Show me all the clubs in this neighborhood?
- How many people pass by this billboard per day?
- What is the commute trend on daily basis in some cities?
- Which ads should I place for people living area?
- In which areas do my mobile subscribers have network problems?
- How much time will I need to get to location A?



These are unique queries only systems which capture geospatial data can answer

Geospatial Data – The Library

- **ST_Geometry**, used for supporting geospatial types of data such as:
 - *Point* (x y (z)): Single location points such as GPS location
 - *Line or curve* (xy, xy, xy): Lines and curves to represent roads, tracks, or rivers
 - *Polygon* (xy, xy, xy, xy..): Represents area objects (e.g., sales regions, neighborhoods, etc.)



ST_GEOMETRY

Teradata provides the ST_Geometry data type for creating and manipulating geometric shapes in the database. ST_Geometry is implemented as a user-defined type (UDT). ST_Geometry is an instantiable type within Teradata Database. (Teradata Database UDTs do not support inheritance or subtyping.) You can use ST_Geometry as the data type of a table column to represent most of the geospatial types specified in the standard.

A column of type ST_Geometry can contain any of these geospatial types:

- GeometryCollection
- GeoSequence
- LineString
- MultiLineString
- MultiPoint
- MultiPolygon
- Point
- Polygon

Geospatial – Where are the Profitable Customers Located?

teradata.

Driving Radius

- 5-mile radius
- 15-mile radius

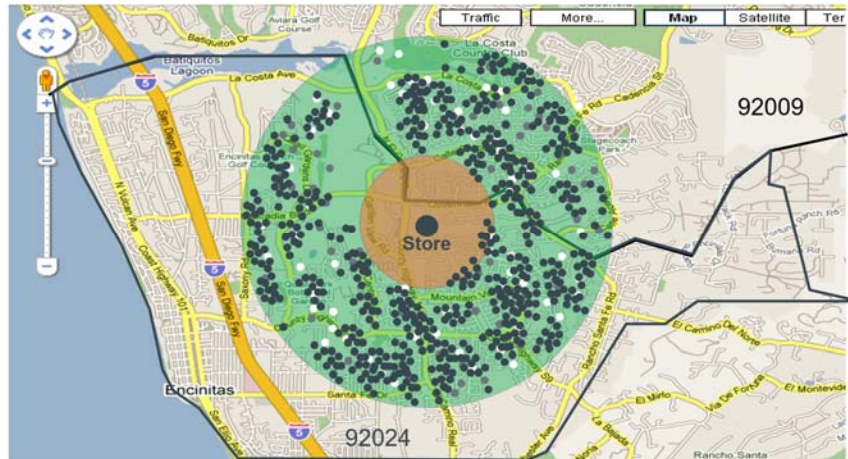
Customers

Customer segments

- All customers

Filter customers:

- $\geq \$50$ basket
Profitability < 75
- $\geq \$50$ basket
Profitability ≥ 75



The term geospatial is a term that has only recently been gaining in popularity and is used to define the collective data and associated technology has a geographic or locational component.

Geospatial Example

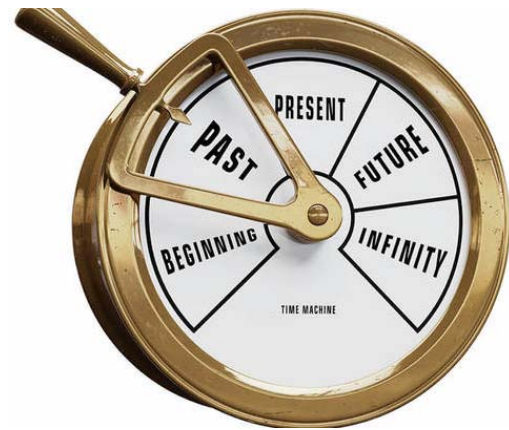
**“Where are my most profitable customers located?”
(Customers whose total sales exceeds \$50)**



Customer ID	Geo Location	Sales Basket (\$)
101	POINT (9.050511820962159 45.375713253642203)	100
102	POINT (9.041495445455103 45.358794354939448)	45
103	POINT (9.191545478314627 45.36710255521163)	160
104	POINT (9.212559416800952 45.369180698327554)	20
105	POINT (9.026501148757708 45.375721348144687)	10
106	POINT (9.272578590367074 45.369053236102772)	200
107	POINT (9.161536817755321 45.367148937051248)	75
108	POINT (9.236602412981291 45.377594681657342)	53

Teradata Temporal

- Maintains data with respect to time and allows time-based interpretation of information
- Provides temporal data types and stores information relating to time periods:
 - The past, present, and future
- For example:
 - The history of billing charge rates and when they were implemented



A temporal database management system (DBMS) is a DBMS that provides built-in support for the time dimension, including special facilities for storing, querying, and updating data with respect to time. A temporal DBMS can distinguish between historical data, current data, and data that will be in effect in the future.

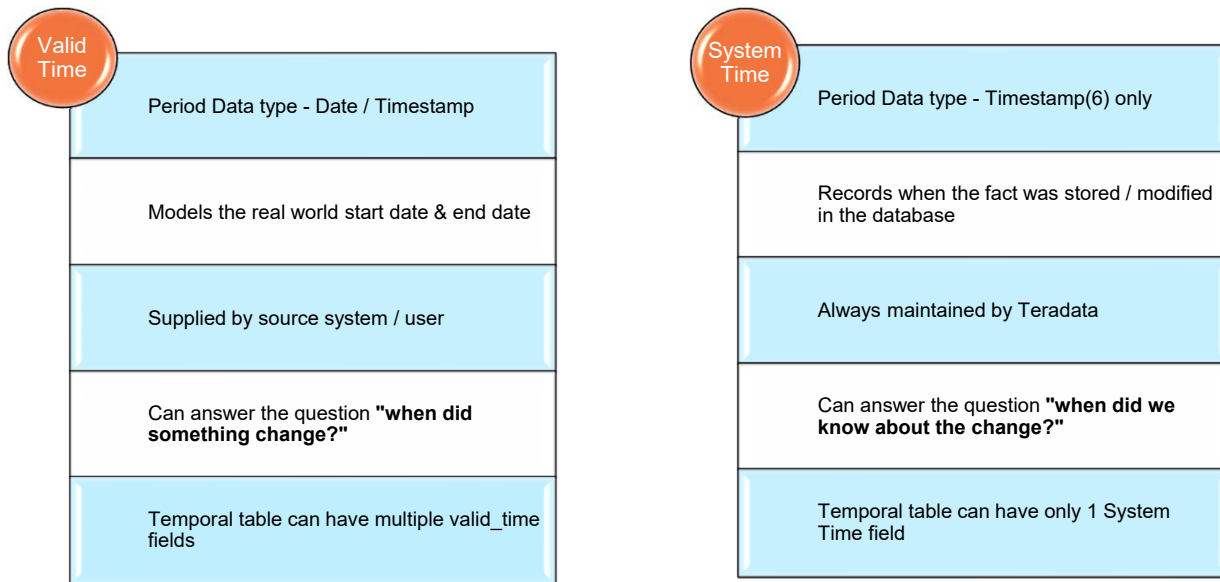
The intent of a temporal database management system is to reason with time.

A temporal DBMS provides a temporal version of SQL, including enhancements to the data definition language (DDL), constraint specifications and their enforcements, data types, data manipulation language (DML), and query language for temporal tables.

A temporal database stores data that relates to time periods and time instances. It provides temporal data types and stores information relating to the past, present, and future. For example, it stores the history of a stock or the movement of employees within an organization. The difference between a temporal database and a conventional database is that a temporal database maintains data with respect to time and allows time-based reasoning, whereas a conventional database captures only a current snapshot of reality.

For example, a conventional database cannot directly support historical queries about past status and cannot represent inherently retroactive or proactive changes. Without built-in temporal table support from the DBMS, applications are forced to use complex and often manual methods to manage and maintain temporal information.

Temporal Time



Temporal tables include 1 or 2 special columns, which store time information.

- **Valid Time** – defined as a **derived Period data type**, either as DATE or TIMESTAMP
A valid time column models the real world and represents a start date and an end date. It is used to determine when the fact was valid.
 - Can answer the question "when did something change?"
 - Typically supplied by the source system or user.

Examples: the time an insurance policy or product warranty is valid, the length of employment of an employee.

- **System Time** – defined as a **derived Period data type**, TIMESTAMP(6)
The system time column records when the fact was stored or changed in the database.
 - Can answer the question "when did we know about the change?"
 - Almost always maintained by Teradata, not by the user or source system.

A temporal table can have a valid time or a system time or both. If a table has both, it is called bi-temporal.

A temporal table can also include other date/time columns which are supplied, maintained and queried by the user/application, not the system.

A temporal table can have multiple valid time periods, but only one system time.

Bi-Temporal Table DDL (ANSI Temporal)

This DDL creates a [ANSI bi-temporal table](#) with valid time and system time columns.

```
CREATE MULTISET TABLE Employee
  (EmpNum          INTEGER          NOT NULL
  ,DeptNum         INTEGER
  ,LastName        CHAR(20)
  ,Salary          DECIMAL(10,2)
  ,StartDate       DATE NOT NULL
  ,EndDate         DATE NOT NULL
  ,PERIOD FOR JobVT(StartDate,EndDate) AS VALIDTIME
  ,SysStart        TIMESTAMP WITH TIME ZONE NOT NULL GENERATED ALWAYS AS ROW
  START
  ,SysEnd          TIMESTAMP WITH TIME ZONE NOT NULL GENERATED ALWAYS AS ROW
  END
  ,PERIOD FOR SYSTEM_TIME(SysStart,SysEnd)
  )
PRIMARY INDEX (EmpNum) WITH SYSTEM VERSIONING;
```

- A Temporal table must be a MULTISET table. If you specify SET, you will get error #9383.
- If a table includes either VALIDTIME or SYSTEM_TIME, then it is a "temporal table".
- If a table includes both VALIDTIME and SYSTEM_TIME, then it is a "bi-temporal table".
- VALIDTIME and SYSTEM_TIME are keywords and require a derived PERIOD data type.
- SYSTEM_TIME requires TIMESTAMP(6) WITH TIME ZONE. If just TIMESTAMP WITH TIME ZONE is specified, it will default to TIMESTAMP(6) WITH TIME ZONE.
- The columns that makeup SYSTEM_TIME and VALIDTIME require the NOT NULL attribute.

A temporal table has a valid -time column, a system-time column, or both. Temporal databases may have several types of tables.

Valid Time table	a table that has a Valid Time column (but not a System Time column)
System Time table	a table that has a System Time column (but not a Valid Time column)
Bi-temporal table	a table that has both types of columns
Temporal table	a table that has one or both
Nontemporal table	a table that has neither

CREATE TABLE AS

Temporal To/From Non-Temporal

- The CREATE TABLE AS can be used to create a temporal table from a source temporal table or to create a non-temporal table from a source temporal table.

With or Without Data

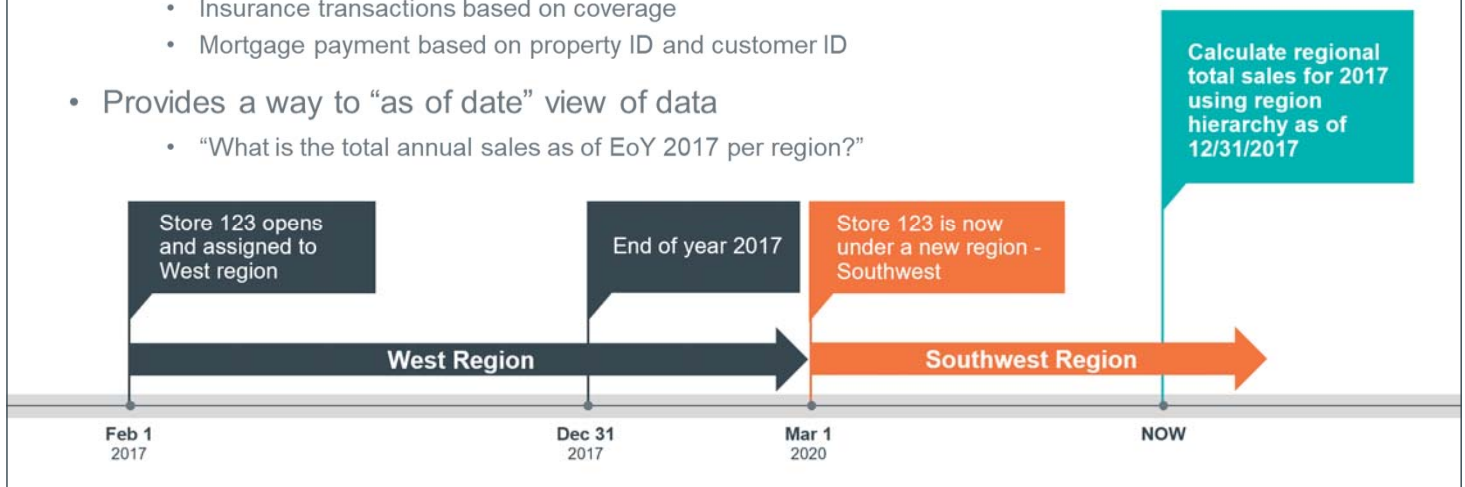
- CREATE TABLE AS WITH DATA can be to recreate a bi-temporal table.
- CREATE TABLE AS WITH NO DATA will recreate the bi-temporal table, which can then be populated with INSERT SELECT.
- To completely replicate the source table data use NONTEMPORAL INSERT/SELECT.

Create Table with Subquery

- CREATE TABLE AS with subquery can also be used to populate bi-temp table.

Temporal Data and Analysis

- Track history and understand data changes over time
- Analysis of time-referenced data
 - Regional sales data based on sales organization hierarchy
 - Insurance transactions based on coverage
 - Mortgage payment based on property ID and customer ID
- Provides a way to “as of date” view of data
 - “What is the total annual sales as of EoY 2017 per region?”



Temporal is really about understanding periods of time and providing historical time relevance in your data. For example, for a customer who moves around, you want to know when they lived where so their buying patterns can be placed in context. A common example is territory assignments that tend to shift as businesses expand and re-organize. It is critical to understand what was, what is, and what possibly may be.

Temporal Example

“What is a store’s active region assignment in 2017?”

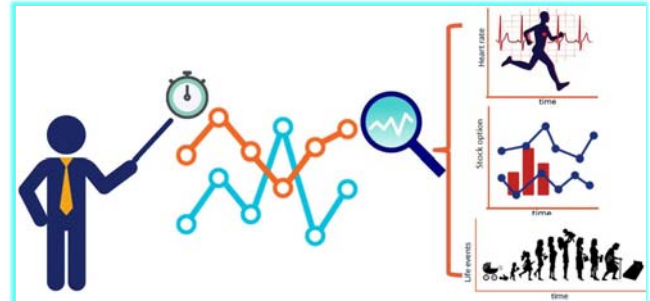


Store ID	Region Assignment	Assignment Date
10	<u>South West</u> Region	2015-11-10, 2016-12-31
10	West Region	2016-12-31, 9999-12-31
12	<u>South East</u> Region	2017-11-10, 9999-12-31
13	<u>North East</u> Region	2017-11-10, 9999-12-31
14	<u>North West</u> Region	2014-11-10, 9999-12-31
15	<u>South East</u> Region	2019-11-10, 9999-12-31

Is Assignment Date a Valid Time or System Time field?

Characteristics of Time Series Data

- The data that arrives is almost always recorded as a new entry
- Possibility of data being duplicated as there may not be any changes
- Data could be discontinuous
- The data typically arrives in time order
- Should be able to handle high reads & writes
- Time is a primary axis (time intervals can be either regular or irregular)
- Deletes & Updates are Rare



One classification of time-series is sometimes called regular time-series also known as discrete series. This is the case where the different sample points that are collected are sampled at equal intervals of time. For example, we start at zero seconds and then every second we would take a sample and no missing data values are permitted. A second classification is called irregular time series. This is more prevalent in Sensor data since data may be collected at unequal intervals. For example, a Buoy may collect a sample of its sensors every 30 minutes; however, if it detects a surge in water current then it might start collecting every 5 or 10 minutes.

The following are the characteristics of Time Series Data:

- **Regular** Time Series, also sometimes called “Discrete”
 - Recorded observations sampled at equal intervals of time.
 - Time = 0 sec, 1 sec, 2 secs, 3 secs, 4 secs, ..., N secs
 - NO missing values permitted.

Note: As far as Time Series feature concerns, Teradata does NOT enforce “No missing observation data” for each time interval; it is up to the application to enforce it.

- **Irregular** Time Series
 - Recorded observations are sampled at unequally intervals of time.
 - Time = 0 sec, 3 secs, 7 secs, 9 secs, 14 secs, 21 secs, ..., N secs
 - Bulk of SENSOR data can be characterized as being an Irregular Series.

Note: Irregular time series data is also discrete. The granularity of the irregular data is based on the TD_TIMECODE data type – DATE or TIMESTAMP. An Irregular series is also discrete in contrast of continues in the context of math.

What Constitutes Time Series Data?

- **Device Identifier/Series Identifier**
 - Uniquely Identify the series
 - Example: - (StockExchange=NYSE, StockSym=TDC)
- **Time Code**
 - A timestamp or date indicating when the data was collected
 - Example: '2016-06-06 10:32:12.122200'
- **Observation/M Measurement**
 - A payload or measurement collected at the time point
 - Example:- Engine temperature, Stock value, etc.



Infoworld

Time series data is unstructured machine-generated sensor data that is continuously produced and collected by a wide range of applications and devices that make up the Internet of Things.

Time series data is typically composed of:

- A **unique identifier**: One or more columns that identify the data source, for example, BuoyID or Vehicle Identification Number (VIN).
- A **timecode**: A timestamp or date indicating when the data was collected, for example: '2017-07-07 10:32:12.122200'.
- A **sequence number**: Data is stored first in the order in which it was collected. If different data arrives at exactly the same time, the sequence number differentiates between the data.
- **Measurements**: One or more measurements collected at a point in time, for example:
 - Temperature, salinity, wave height, wave speed, and wave direction
 - Engine temperature, oil temperature, oil pressure, tire inflation ratio

What was Measured and When It was Logged

- Data can be regular intervals or sporadic
- It can contain one or multiple observation values
- It can be bounded by an event or infinite

Timecode	Beacon_ID	Value
2017-08-11 8:41:00	22	3
2017-08-11 8:41:10	22	5
2017-08-11 8:41:20	23	2
2017-08-11 8:41:30	23	7
2017-08-11 8:41:20	22	1
2017-08-11 8:41:50	23	4
2017-08-11 8:41:40	22	3

Teradata Time Aware Capabilities

Agile Analysis enabled by Time-Aware Aggregate Functions

- Time period aware aggregations
- Work with ANY time component data
- Handle missing values
 - Ignore or fill with a value

High-Performance enabled by Primary Time Index (PTI)

- Enables a new way of storing and ordering time-based data (time series as well as date/timestamp data)
- Supports time sensitive decisions
- Fast access through:
 - Hash distribute by time bucket
 - AMP-local processing
 - Sequenced data

Time Aware Functions

- **\$TD_GROUP_BY_TIME**
 - Use on ANY time component data (*field with a timestamp attached – Time / Timestamp*)
 - Works with some existing aggregation functions
 - Average, Count, Describe, Kurtosis, Maximum, Minimum, Percentile, Rank, Skew, Sum, Std. population deviation, Std. sample deviation, Population variance, Sample variance
 - Works with all new aggregation functions
 - Bottom, Top, First, Last, Delta_T, Median, Mode, Mean absolute deviation
- **\$TD_TIMECODE_RANGE**
 - Defines time range
- **FILL ()**
 - Imputes missing values : - NULLS, <Constant>, PREV/PREVIOUS, NEXT

Time-Aware Aggregate Functions – GROUP BY TIME

teradata.

Existing Aggregate Functions

Average	Count
Describe	Kurtosis
Maximum	Minimum
Percentile	Rank
Skew	Sum
Std. population deviation	Std. sample deviation
Population variance	Sample variance

If not in the list above, then function is not time aware and cannot be used with the GROUP BY TIME clause

New Aggregate Functions

Bottom	Delta_T
First	Last
Median	Mode
Top	Mean absolute deviation

These new aggregate functions are only invocable with the GROUP BY TIME clause

Time Aware Example – How Does it Look?

“For each beacon sensor location, show me the average foot traffic in a ½ hour increment, over 2 hours”



Timecode-Range	Beacon ID	Location	People
'2017-08-11 08:00:00', '2017-08-11 08:30:00'	22	Frozen Foods	50
'2017-08-11 08:30:00', '2017-08-11 09:00:00'	22	Frozen Foods	95
'2017-08-11 09:00:00', '2017-08-11 09:30:00'	22	Frozen Foods	114
'2017-08-11 09:30:00', '2017-08-11 10:00:00'	22	Frozen Foods	37
'2017-08-11 08:00:00', '2017-08-11 08:30:00'	23	Cereals	80
'2017-08-11 08:30:00', '2017-08-11 09:00:00'	23	Cereals	65
'2017-08-11 09:00:00', '2017-08-11 09:30:00'	23	Cereals	93
'2017-08-11 09:30:00', '2017-08-11 10:00:00'	23	Cereals	40

Time Aware Example – How Does it Work?

```
SELECT $TD_TIMECODE_RANGE, $TD_GROUP_BY_TIME, SENSORID, AVG(TEMPERATURE) FROM BUOYS
WHERE TIMECODE BETWEEN TIMESTAMP '2017-08-11 01:00:00' AND TIMESTAMP '2017-08-11 03:00:00'
GROUP BY TIME( MINUTES(30) AND SENSORID) USING TIMECODE(TD_TIMECODE)
ORDER BY SENSORID, $TD_GROUP_BY_TIME;
```

Timecode-Range	Group by 30 minutes	Sensor ID	Temperature
'2017-08-11 01:00:00', '2017-08-11 01:30:00'	1	22	63.5
'2017-08-11 01:30:00', '2017-08-11 02:00:00'	2	22	64.6
'2017-08-11 02:00:00', '2017-08-11 02:30:00'	3	22	65.0
'2017-08-11 02:30:00', '2017-08-11 03:00:00'	4	22	65.1
'2017-08-11 01:00:00', '2017-08-11 01:30:00'	1	23	66.4
'2017-08-11 01:30:00', '2017-08-11 02:00:00'	2	23	65.1
'2017-08-11 02:00:00', '2017-08-11 02:30:00'	3	23	64.9
'2017-08-11 02:30:00', '2017-08-11 03:00:00'	4	23	65.1

7

Time Aware – Fill Clause

FILL SCHEME	Aggregate result for the missing time bucket will be
NULLS	Null
<Constant>	A constant value.
PREV/PREVIOUS	Same as the previous time bucket's result
NEXT	Same as the next time bucket's result

```
SELECT $TD_TIMECODE_RANGE, $TD_GROUP_BY_TIME,  
BUOYID, AVG(TEMPERATURE) FROM OCEAN_BUOYS  
WHERE TD_TIMECODE BETWEEN  
TIMESTAMP '2017-05-02 09:45:00' AND TIMESTAMP '2017-05-02 11:45:00'  
AND SENSOR_ID=44  
GROUP BY TIME (MINUTES(15) AND BUOYID) FILL(PREV)  
ORDER BY 2,3;
```

Primary Time Index

- Supports time sensitive decisions
- Either Primary Index or Primary Time Index
- Fast access through:
 - Hash distribute by time bucket
 - AMP-local processing
 - Sequenced data



Time series data is stored in a specialized table, starting in Teradata Database 16.20; this type of table is called a Primary Time Index (PTI) table. PTI tables are created using the time series forms of CREATE TABLE or CREATE TABLE AS, which have a PRIMARY TIME INDEX clause. PTI tables also contain system-generated columns of time information. The Teradata Database uses the time information to organize the data across the system which is a facilitator for efficient time-range queries.

Why PTI?

Time series data is continuously produced and collected by a wide, growing range of applications & technologies which intersect both Teradata's existing customer base, as well as touching many future customer candidates.

Time series constitutes one of the major media types that are driving the IoT revolution. A time series database, especially a time series database with proven scalability and reliability properties, can not only offer a means of storing the data, but also provide a strong analytical engine for extracting value from that data in a near real-time manner.

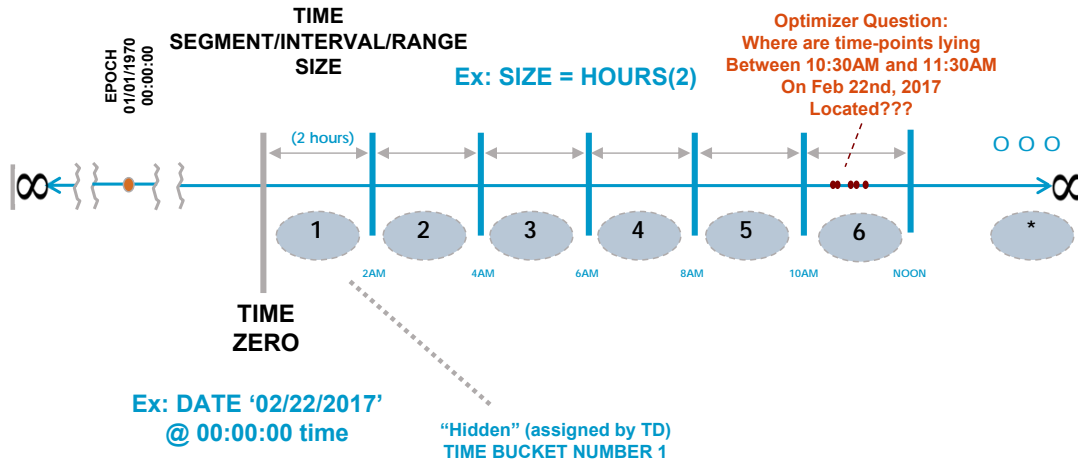
Teradata everywhere, means that the power of Teradata time series analytics will also be everywhere.

- On-Premise Teradata offering time series analytics with unparalleled scalability and reliability.
- In-the-Cloud Teradata offering time series analytics that can be exercised against in-flight IoT data transport streams.

Time-series Data – Time Bucket

How might the Time Continuum be organized to facilitate storage distribution by time interval, ΔT ?

Terminology Introduction: TIME BUCKET



So how might the Time Continuum be organized to facilitate storage distribution by time interval, ΔT ?
Let's introduce the terminology called: TIME BUCKET

This will allow rows in a same bucket to reside on the same AMP. Within an AMP, the rows are ordered by TIMECODE.

Let's take a look at Organizing the Time Continuum with Ad hoc Time Zero of February 22nd and a Time Bucket interval of 2 hours. Teradata behind the scenes assigns this bucket number.

Key Aspects of PTI

The following are some key aspects of PTI:

- Store time series data and is [Optimized for time-range queries](#) can:
 - Store either [Regular/Discrete](#) or [Irregular](#) time series data
 - Store Multiple time series data (i.e., 1,000 Buoys or 1 million Cars) within the same PTI table
 - Contain any number of columns of any supported Teradata data type
 - Be a permanent, global temporary, or volatile table
- Created with a CREATE TABLE statement containing a [PRIMARY TIME INDEX](#) clause
 - The CREATE TABLE statement is used to specify how the rows are stored (distribution strategies) and how the rows are ordered logically (sorting)
 - There are 3 distribution strategies and two ordering strategies that the user can choose from
- Suite of Time Series Aggregate Functions featuring
 - A [GROUP BY TIME](#) clause that is used to perform time-aware aggregate operations against time series data residing within a Teradata or Teradata-accessible (QueryGrid) table

The change-in-#-measurements is the reason that sensor data is often stored/encapsulated within some sort of semi-structured data type – JSON, AVRO, CSV, binary, etc.

PTI provides a repository to store one or many time series. It provides three storage distribution strategies, devised to facilitate storage of three different classes of time series data. It also provides two logical in-table ordering strategies, devised to facilitate time-based positioning and filtering operations.

In addition, it provides a GROUP BY TIME clause, which can be used to perform time-aware aggregate operations. This SQL clause will enable group-by-time operations to be performed against many existing aggregate functions. Also, some additional aggregate functions have been added, which can be used in conjunction with the group-by-time clause. We will cover these later in the module.

Distribution and Ordering Strategies

User chooses from three distribution strategies and two ordering strategies as shown below:

Distribution strategy across AMPs

- A – by Time Interval only
- B – by Time Interval and Column List
- C – by Column List only

```
CREATE TABLE TS_Dist_a_Sort_1
( DeviceID      INTEGER,
  Temperature    FLOAT,
  CO_Level       FLOAT
)
PRIMARY TIME INDEX
( TIMESTAMP(0), HOURS(4));
```

Distribution: ($\Delta T=4$ hours) – strategy A
 Ordering: non-sequenced logical ordering
 based on TIMECODE alone – Strategy 1

Ordering of data within the AMPs

- 1 – by Timecode only
- 2 – by Timecode and Sequence Number

```
CREATE TABLE TS_Dist_b_Sort_2
( DeviceID      INTEGER,
  Temperature    FLOAT,
  Rotation_Vel   FLOAT
)
PRIMARY TIME INDEX
( TIMESTAMP(2), HOURS(1),
  COLUMNS(DeviceID),
  SEQUENCED(1024));
```

Distribution: ($\Delta T=1$ hours, DeviceID) – strategy B
 Ordering: sequenced logical ordering based on
 TIMECODE and SEQNO – Strategy 2

Concepts

PTI introduces a way to store time series data. The requirements to such a storage being:

1. Ability to store data in logical order
2. Ability to distribute data based off time or time and columns or columns only.

A PTI table provides two choices with respect to how the time series rows can be logically ordered within the table:

1. By Time: Series data is stored in the table in time order
2. By Time and Sequence Number: Series data is stored in the table in time, then sequence number order. When two sets of data or more are arriving exactly at the same time, the sequence number provides a way to differentiate them.

Three different storage options are available to the user:

1. Hash By or distribute by time interval only
2. Distribute by time interval and column list
3. Distribute by column list only.

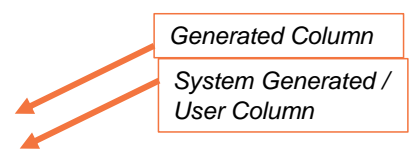
Note: It is important to remember irrespective of the storage option chosen, the logical ordering of the rows will be as per the time or time and sequence number.

Storage Distribution Related PRIMARY TIME INDEX Parameters

There are multiple parameters which make up a PTI, but the `<timebucket>` and `<columns_clause>` influence how time series rows are distributed between AMPs.

PTI Table Definition – How It Looks

```
CREATE MULTISET TABLE trng_retaildse.WEB      Fallback
NO BEFORE JOURNAL
NO AFTER JOURNAL
CHECKSUM = DEFAULT
DEFAULT MERGEBLOCKRATIO
MAP = TD_MAP1
(
  TD_TIMEBUCKET BIGINT NOT NULL GENERATED SYSTEM TIMECOLUMN
  TD_TIMECODE  TIMESTAMP(6) NOT NULL GENERATED TIMECOLUMN
  customer_id  DECIMAL(18  0) NOT NULL
  SERVER_ID   VARCHAR(5) CHARACTER SET LATIN CASESPECIFIC NOT NULL
  PAGE        VARCHAR(50) CHARACTER SET LATIN CASESPECIFIC
  BROWSE_ID   VARCHAR(20) CHARACTER SET LATIN CASESPECIFIC)
PRIMARY TIME INDEX (TIMESTAMP(6), DATE '2016-01-01', MINUTES(1), COLUMNS(SERVER_ID), NONSEQUENCED);
```



Generated Column

System Generated / User Column

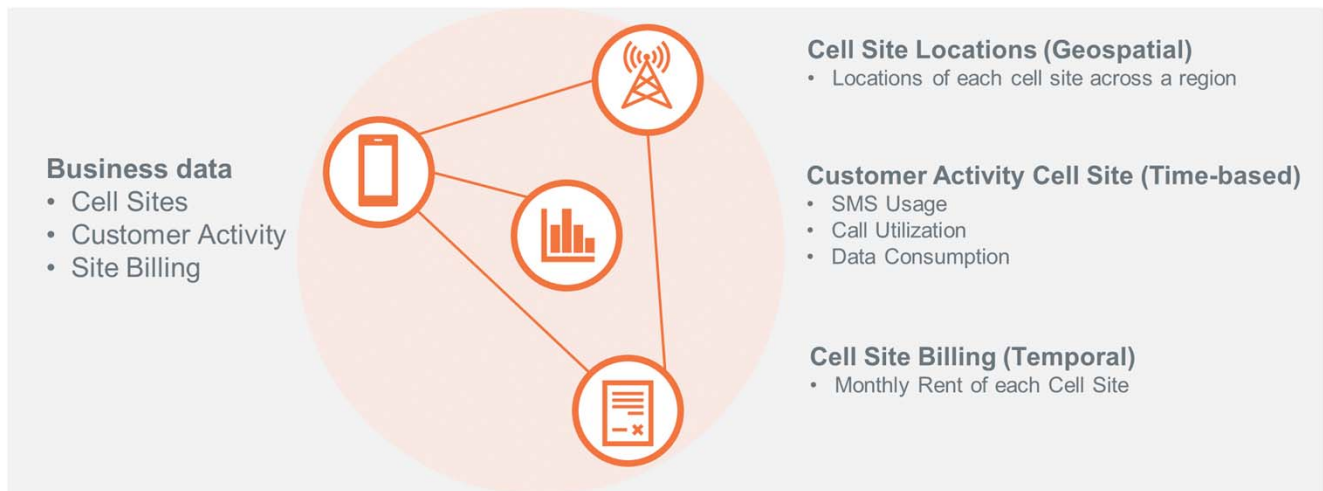
GROUP BY TIME Rules and Restrictions

- GROUP BY TIME and GROUP BY cannot be used together in the same query (this restriction includes GROUP BY ROLLUP, GROUP BY CUBE, etc..)
- If GROUP BY TIME is used on a non-PTI table, the USING TIMECODE clause must be included; *otherwise, an error is reported*
- A supported Time Series function must be used in conjunction with a GROUP BY TIME clause; *otherwise, an error is reported*
- The time bucket is computed based on time zero. Time zero defaults to DATE '1970-01-01'



4D Analytics in Action

Use Case: How do we know if a cell site is still viable to be active?



Summary

Now that you have completed this course, you will be able to:

- Describe how Vantage provides 4D Analytics
- Discuss Geospatial functions
- Describe Temporal Access
- Discuss Teradata Time Aware functions

Thank you.

teradata.

©2023 Teradata



Module 9: Teradata Columnar

Teradata Vantage MasterClass

Copyright © 2007–2023 by Teradata. All Rights Reserved.

Objectives

After completing this module, you will be able to:

- Compare Row versus Columnar
- Describe why Columnar is used
- Describe the components that comprise a column-partitioned table
- Identify two advantages of creating a column-partitioned table
- Identify two disadvantages of creating a column-partitioned table
- Identify the recommended way to populate a column-partitioned table
- Specify how rows are deleted in a column partitioned table



Why Columnar?

Benefits of using the Teradata Columnar feature include:

- **Improved query performance**
 - This is achieved via column partition elimination. Column partition elimination reduces the need to access all the columns in a row while row partition elimination reduces the need to access all the rows
- **Reduced disk space**
 - The feature also allows for the possibility of using an auto-compression capability which allows data to be compressed automatically (as applicable)
- **Increased flexibility**
 - Provides a physical database design option to improve performance for suitable classes of workloads
- **Reduced I/O**
 - Allows fast and efficient access to selected data from column partitions, thus reducing query I/O
- **Ease of use**
 - Provides simple default syntax for the CREATE TABLE and CREATE JOIN INDEX statements. No change is needed to queries

Columnar Join Indexes

A join index can also be created as column-partitioned for either a columnar table or a non-columnar table. Conversely, a join index can be created as non-columnar for either type of table as well.

Sometime within a mixed workload, some queries might perform better if data is not column partitioned and some where it is column partitioned. Or, perhaps some queries perform better with one type of partitioning on a table, whereas other queries do better with another type of partitioning. Join indexes allow creation of alternate physical layouts for the data with the optimizer automatically choosing whether to access the base table and/or one of its join indexes.

A column-partitioned join index must be a single-table, non-aggregate, non-compressed, join index with no primary index, and no value-ordering, and must include RowID of the base table. A column-partitioned join index may optionally be row partitioned. It may also be a sparse join index.

This module will only describe and include examples of base tables that utilize column partitioning.

Row vs. Columnar

• Row Storage

- Organize data by record
- Row Partitioning is a CPU reduction feature
- Physical Design choice when all the columns are selected
- Compression needs to be defined
- When individual rows are deleted, they are physically deleted

Winner	Loser	Game_Date	Game_Score
Dallas Cowboys	Denver Broncos	01-15-1978	27-10
Chicago Bears	New England Patriots	01-26-1986	46-10

• Column Store

- Organize data by column
- Column Partitioning is I/O reduction feature
- Auto Compression is enabled by default
- Physical Design choice when few columns are being selected
- When individual rows are deleted, they are not physically deleted but are marked as deleted

Winner	Loser	Game_Date	Game_Score
Part 1-HBN-Row #1	Part 2-HBN-Row #1	Part 3-HBN-Row #1	Part 4-HBN-Row #1
1's & 0's	1's & 0's	1's & 0's	1's & 0's
Dallas Cowboys	Denver Broncos	01-15-1978	27-10
Chicago Bears	New England Patriots	01-26-1986	46-10

This slide provides a contrast between row storage and column storage to clarify understanding.

Teradata Columnar

- Description

- Columnar (or Column Partitioning) is a physical database design implementation option that allows sets of columns (including just a single column) of a table or join index to be stored in separate partitions
- This is effectively an I/O reduction feature to improve performance for suitable classes of workloads

- Considerations

- This physical design choice is especially suitable if both a small number of rows are selected, and a few columns are projected
- When individual rows are deleted, they are not physically deleted but are marked as deleted

Teradata Columnar (Industry term) or Column Partitioning (CP – Teradata implementation) is a physical database design implementation option that allows single columns or sets of columns of a NoPI table to be stored in separate partitions. Column partitioning can also be applied to join indexes.

Teradata Columnar offers the ability to partition a table or join index by column. Teradata Columnar can be used alone or in combination with row partitioning in multilevel partitioning definitions.

The key benefit in defining row-partitioned (PPI) tables is when queries access a subset of rows based on constraints on one or more partitioning columns. The major advantage of using column partitioning is to improve the performance of queries that access a subset of the columns from a table, either for predicates (e.g., WHERE clause) or projections (i.e., SELECTed columns).

Because sets of one or more columns can be stored in separate column partitions, only the column partitions that contain the columns needed by the query need to be accessed. Just as row-partitioning can eliminate rows that need not be read, column partitioning eliminates columns that are not needed.

The advantages of both can be combined, meaning even less data moved and thus reduced I/O. Fewer data blocks need to be read since more data of interest is packed together into fewer data blocks.

Columnar makes more sense in CPU-rich environments because CPU cycles are needed to “glue” columns back together into rows, for compression and for different join strategies (mainly hash joins).

No Primary Index Table DDL

```
CREATE TABLE Super_Bowl_NoPI
(Winner      CHAR(25)    NOT NULL
,Loser       CHAR(25)    NOT NULL
,Game_Date   DATE        NOT NULL
,Game_Score  CHAR(7)     NOT NULL
,Attendance  INTEGER)
NO PRIMARY INDEX;
```

In this module, we will use an example of Super Bowl history information to simply demonstrate column partitioning.



This slide simply illustrates the DDL to create a NoPI table. This example will be as a basis for multiple examples of creating tables with various column partitioning options.

NoPI Table

What is a No Primary Index (NoPI) Table?

- It is simply a table without a primary index
- As rows are inserted into a NoPI table, **rows are always appended at the end of the table and never inserted in a middle of a hash sequence.**
 - Organizing/sorting rows based on row hash is therefore avoided.

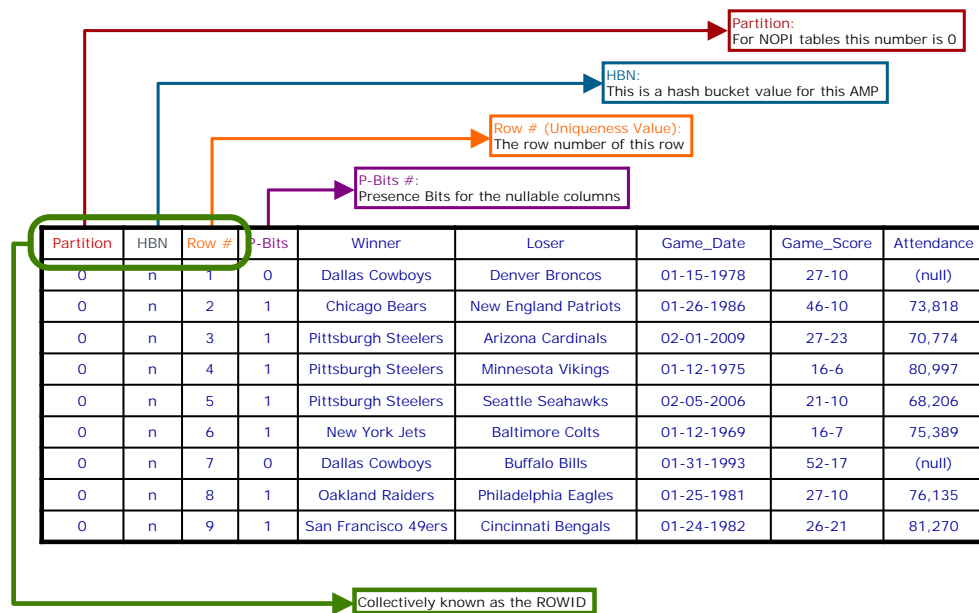
Basic Concepts

- Rows will still be distributed between AMPs. NoPI uses Random Generator code for SQL inserts or round robin logic for data blocks (e.g., TPT LOAD) to determine which AMP will receive rows or blocks of rows.
- Within an AMP, rows are simply appended to the end of the table. Rows will have a unique RowID – the Uniqueness Value is incremented.

Benefits

- A NoPI table will reduce skew in intermediate ETL tables which have no natural Primary Index.
- Loads using TPT Load (FastLoad) or TPT Stream (TPump) Array Insert into a NoPI staging table are faster.

No Primary Index Table



The No Primary Index table is shown on this slide.

Column Partition Table DDL (without Auto-Compression)

```
CREATE TABLE Super_Bowl_CP_noAC
(Winner      CHAR(25) NOT NULL
,Loser       CHAR(25) NOT NULL
,Game_Date   DATE      NOT NULL
,Game_Score  CHAR(7)   NOT NULL
,Attendance  INTEGER
)
PARTITION BY COLUMN NO AUTO COMPRESS;
```

Defaults for a column partitioned table.

- Assumes NO PRIMARY INDEX
- Single-column partitions; options include multicolumn partitions
- Auto compression is on; NO AUTO COMPRESS turns off auto-compression for the column
- System-determined column-store for above column partitions; options include column-store (COLUMN) or row-store (ROW)

With column partitioning, each column or specified group of columns in the table can become a partition containing the column partition values of that column partition. This is the simplest partitioning approach since there is no need to define partitioning expressions, as seen in the example on this slide.

The clause PARTITION BY COLUMN specifies that the table has column partitioning.

Note that a primary index is not specified since this is NoPI table. A primary index may not be specified if the table is column partitioned.

Characteristics of a Columnar Table

- Each column partition can be composed of single or multiple columns.
- Each column partition usually consists of multiple physical rows.
- A physical row format COLUMN may be utilized for a column partition. Such a physical row is called a 'container' and it is used to implement columnar-storage for a column partition.
- Alternatively, a column partition may also have traditional physical rows with ROW format. Such a physical row for columnar partitions is called a subrow. This is used to implement row-storage for a column partition.
- Note that in subsequent discussions, when 'row storage' or 'row format' is stated, it is referring to columnar partitioning with the ROW storage option selected. This is not to be confused with row-partitioning which we associate with a PPI table.

In a table with multiple levels of partitioning, only one level may be column partitioned. All other levels must be row-partitioned.

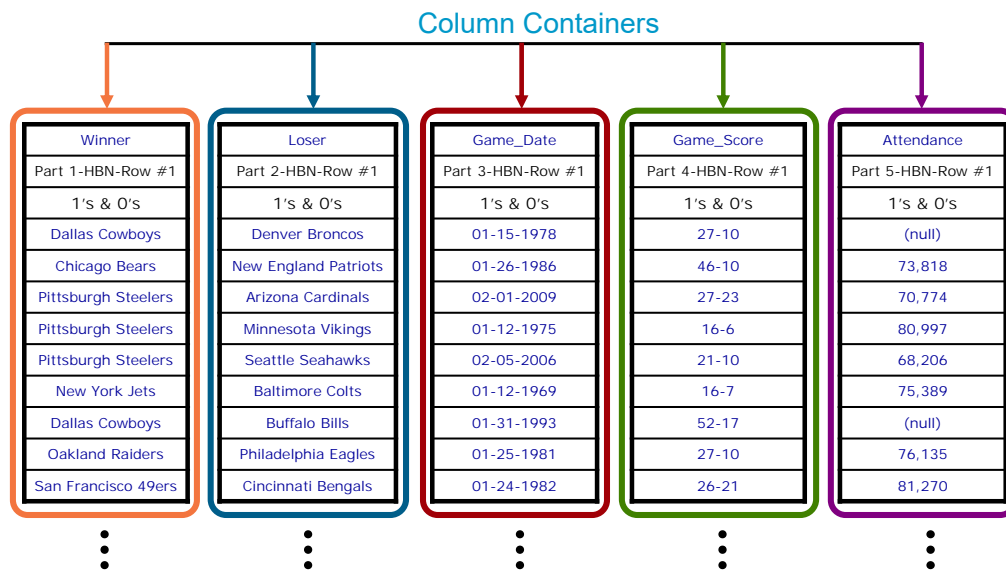
Column Partition Container (No Automatic Compression) (1 of 3)

teradata.

Partition	HBN	Row #	P-Bits	Winner	Loser	Game_Date	Game_Score	Attendance
0	n	1	0	Dallas Cowboys	Denver Broncos	01-15-1978	27-10	(null)
0	n	2	1	Chicago Bears	New England Patriots	01-26-1986	46-10	73,818
0	n	3	1	Pittsburgh Steelers	Arizona Cardinals	02-01-2009	27-23	70,774
0	n	4	1	Pittsburgh Steelers	Minnesota Vikings	01-12-1975	16-6	80,997
0	n	5	1	Pittsburgh Steelers	Seattle Seahawks	02-05-2006	21-10	68,206
0	n	6	1	New York Jets	Baltimore Colts	01-12-1969	16-7	75,389
0	n	7	0	Dallas Cowboys	Buffalo Bills	01-31-1993	52-17	(null)
0	n	8	1	Oakland Raiders	Philadelphia Eagles	01-25-1981	27-10	76,135
0	n	9	1	San Francisco 49ers	Cincinnati Bengals	01-24-1982	26-21	81,270

The No Primary Index table is shown on this slide.

Column Partition Container (No Automatic Compression) (2 of 3)



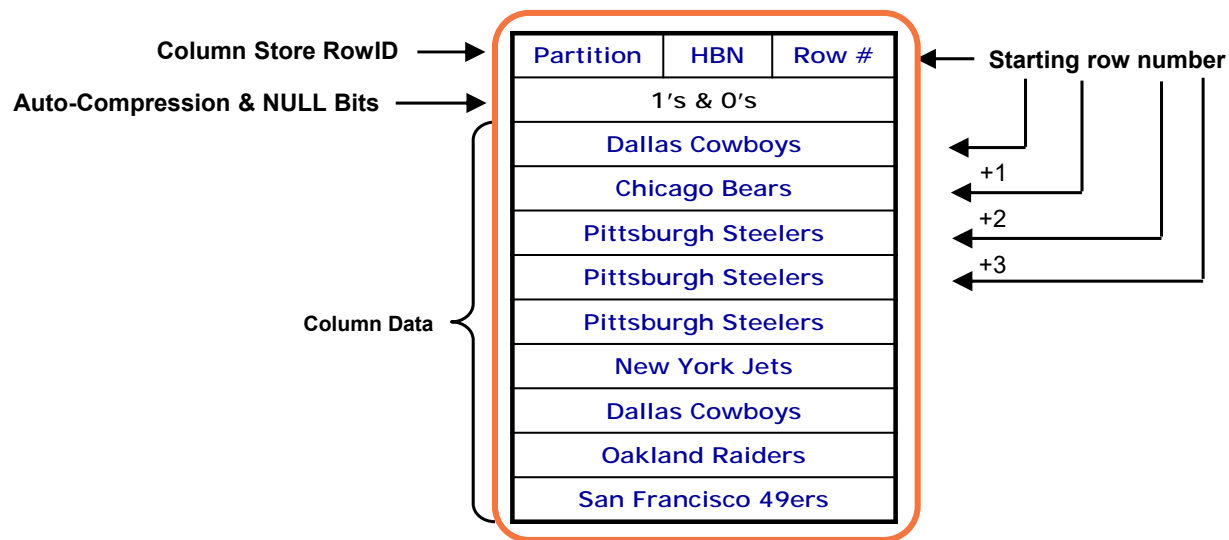
The result of creating a column partitioned table (as shown previously) is shown on this slide with some sample data.

The clause `PARTITION BY COLUMN` specifies that the table has column partitioning. Each column of this table will have its own partition and will be (by default) in column storage since no explicit column grouping is specified.

The default of auto-compression is overridden for each of the columns.

Column Partition Container (No Automatic Compression) (2 of 3)

teradata.



Column Container is effectively a row in the partition.

In order to support columnar-storage for a column partition, a format, referred to as a COLUMN format in the syntax, is available for a physical row. A physical row with this format is referred to as a **container** and each container holds a series of column partition values for a column partition.

Each container is assigned a specific partition number which identifies the column or group of columns whose column partition values are held in the container. When a column partition is stored on disk, one or more containers may be needed to hold all the column partition values of the column partition. Since a container is a physical row, the size of a container is limited by the maximum physical row size.

The example on this slide assumes that NO AUTO COMPRESS has been specified for the column.

Containers hold multiple values for the same column (or columns) of a table. For purposes of this explanation, the assumption is being made that each partition contains only a single column so a column partition value is the same as a column value. Recall that each column value belongs to a specific row and that each row is identified by a RowID consisting of a row-hash and uniqueness value. Since all of the rows on a single AMP of a NoPI table share the same row hash, the uniqueness value becomes the real differentiator. So the connection between a specific column value for a particular row on a given AMP and its uniqueness value is the key in locating the corresponding column value.

Assume that a given container holds 1000 values. The RowID of each container carries a hash bucket and uniqueness which represents the first column value entry in the container. The first value's hash bucket and uniqueness is explicit while the other values' hash bucket and uniqueness are implicit and are understood based on their position in their container.

CP Table Query #1 (without Auto-Compression) (1 of 2)

teradata.

Which teams have lost to the "Dallas Cowboys" in the Super Bowl?

Winner	Loser	Game_Date	Game_Score	Attendance
Part 1-HBN-Row #1	Part 2-HBN-Row #1	Part 3-HBN-Row #1	Part 4-HBN-Row #1	Part 5-HBN-Row #1
1's & 0's	1's & 0's	1's & 0's	1's & 0's	1's & 0's
Dallas Cowboys	Denver Broncos	01-15-1978	27-10	(Null)
Chicago Bears	New England Patriots	01-26-1986	46-10	73,818
Pittsburgh Steelers	Arizona Cardinals	02-01-2009	27-23	70,774
Pittsburgh Steelers	Minnesota Vikings	01-12-1975	16-6	80,997
Pittsburgh Steelers	Seattle Seahawks	02-05-2006	21-10	68,206
New York Jets	Baltimore Colts	01-12-1969	16-7	75,389
Dallas Cowboys	Buffalo Bills	01-31-1993	52-17	(Null)
Oakland Raiders	Philadelphia Eagles	01-25-1981	27-10	76,135
San Francisco 49ers	Cincinnati Bengals	01-24-1982	26-21	81,270
⋮	⋮	⋮	⋮	⋮

Only the accessed columns are needed.

One of the key advantages of column partitioning is opportunity for reduced I/O. This can be realized only if a subset of the columns in a table are read and if those column values are held in separate column partitions. Data is stored on disk by partition, so when partition elimination takes place, data blocks in the eliminated partitions are simply not read.

There are three ways to initiate read access to data within a column-partitioned table:

- A full column partition scan
- Indexed access (using a secondary, join index, or hash index),
- A RowID join.

Both unique and non-unique secondary indexes are allowed on column-partitioned tables, as are join indexes and hash indexes.

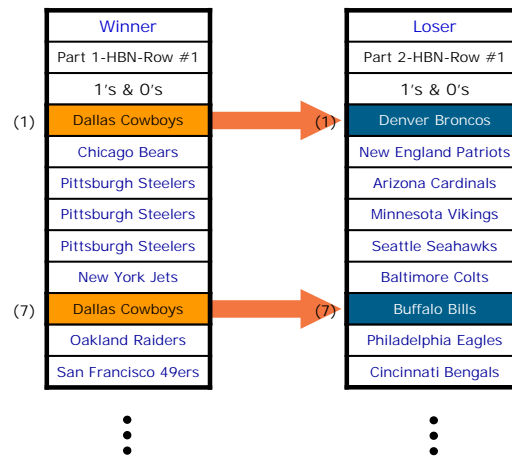
Queries best suited for scanning a column-partitioned table are queries that:

- Contain one or a few predicates that are very selective in combination.
- Require a small enough number of columns to be accessed that the caches required to support their consolidation can be held in memory.

CP Table Query #1 (without Auto-Compression) (2 of 2)

teradata.

Which teams have lost to the "Dallas Cowboys" in the Super Bowl?



The relative row number in each container is used to access the data.

If indexing is not available, Teradata can access data in a CP table by scanning a column partition on all the AMPs in parallel.

In the example on this slide, the "Winner" column containers are scanned for "Dallas Cowboys".

The following describes the scanning of CP data:

1. Columns within the table definition that aren't referenced in the query are ignored due to partition elimination.
2. If there is a predicate column in the query, its column partition is read.
3. Values within the predicate column partition are examined and compared against the value passed in the query WHERE clause.
4. Each time a qualifying value is located, the next step is building up a row for the output spool.
5. All the column partition values for a logical row have the same RowID except for the column partition number.

If there is more than one predicate column in the query that can be used to disqualify rows, the column for one of these predicates is chosen and its column partition is scanned.

If there are no useful predicate columns in the query (for instance, OR'ed predicates), one column partition is chosen to be scanned and for each of its column partition values additional corresponding column partition values are accessed until either predicate evaluation disqualifies the logical row or all the projected column values have been retrieved and brought together to form rows for the output spool.

Column Partitioning – 2nd Example

CDR_NoPI

➤ Find 3rd record

Part #, HBN, Uniq, PB	Orig#	Term#	Call_TS	Duration	Col_5	Col_100
0, HBN, 00000001, PB	DEC(15,0)	DEC(15,0)	Timestamp	INT	BIGINT	VARCHAR(80)
0, HBN, 00000001, PB	18584850001	18584856001	TS000001	100	1001	A
0, HBN, 00000002, PB	18584850002	18584856002	TS000002	20	1001	B
0, HBN, 00000003, PB	18584850003	18584856003	TS000003	33	1002	C
:	:	:	:	:	:	:
0, HBN, 00001999, PB	18584851999	18584857999	TS001999	403	2000	E
0, HBN, 00002000, PB	18584852000	18584858000	TS002000	75	2001	F

CDR_CP

2 Containers

2 Containers

3 Containers

Part 1, HBN, Uniq, PB	1000 Values in each row from Orig# (each value uses 8 bytes)					
1, HBN, 00000001, PB	18584850001	18584850002	18584850003	:	18584850999	18584851000
1, HBN, 00001001, PB	18584851001	18584851002	18584851003	:	18584851999	18584852000
Part 2, HBN, Uniq, PB	1000 Values in each row from Term# (each value uses 8 bytes)					
2, HBN, 00000001, PB	18584856001	18584856002	18584856003	:	18584856999	18584857000
2, HBN, 00001001, PB	18584857001	18584857002	18584857003	:	18584857999	18584858000
Part 3, HBN, Uniq, PB	667 Values in each row from Call_Timestamp (each value uses 12 bytes)					
3, HBN, 00000001, PB	TS000001	TS000002	TS000003	:	TS000666	TS000667
3, HBN, 00000668, PB	TS000668	TS000669	TS000670	:	TS001333	TS001334
3, HBN, 00001335, PB	TS001335	TS001336	TS001337	:	TS001999	TS002000

For each container, add 2 to Start Row#

This slide illustrates a second example of a table with column partitioning and assumes the automatic compression is not being used. The top table is a NoPI table. The data in this table is used to populate the partitions of a column partitioned table.

A row in a CP table is effectively known as a "container". The typical size of a container (or row) is approximately 8K.

The first column has a data type of DECIMAL (15, 0), therefore 8 bytes of storage is needed for the each phone number stored in the container. Approximately 1000 values (e.g., phone numbers) can be stored in each container for the first partition (e.g., first column or originating #).

The second column also has a data type of DECIMAL (15, 0), therefore 8 bytes of storage is needed for the each phone number stored in the container. Approximately 1000 values (e.g., phone numbers) can be stored in each container for the second partition (e.g., second column or terminating #).

The third column has a data type of TIMESTAMP WITH TIMEZONE, there 12 bytes of storage is needed for the each time stamp stored in the container. Approximately 667 values (e.g., time stamps) can be stored in each container for the third partition (e.g., third column or Call TS).

As always, a group of rows (containers) in a CP table are group into blocks, blocks are grouped in a cylinder, and the cylinder will have a cylinder index with entries for each of the data blocks within the cylinder.

Column Partition Table DDL (with Auto-Compression)

teradata.

```
CREATE TABLE Super_Bowl_CP  
(Winner      CHAR(25)      NOT NULL  
,Loser       CHAR(25)      NOT NULL  
,Game_Date   DATE          NOT NULL  
,Game_Score  CHAR(7)       NOT NULL  
,Attendance  INTEGER)  
PARTITION BY COLUMN;
```

Note: Auto Compression is on by Default.

The DDL to create a column partitioned table with auto-compression is shown on this slide. Each column will be maintained in a separate partition.

The clause PARTITION BY COLUMN specifies that the table has column partitioning. Each column of this table will have its own partition and will be (by default) in column storage since no explicit column grouping is specified.

Auto-Compression for CP Tables

Auto Compression

- When a column partition is defined to have auto-compression (i.e., the NO AUTO COMPRESS option is not specified), data is compressed by the system
- For some values, there is no applicable compression technique, and the system will determine not to compress the values for that physical row
- The system decompresses any compressed column-partition values when they are retrieved
- Auto-compression is most effective for a column partition with a single column and COLUMN format

Auto-compression is a completely transparent compression option for column partitions. It is applied to a container when a container is full after appending some number of column partition values without auto-compression by an INSERT or UPDATE statement. Each container is assessed separately to see how, and if, it can be compressed.

Several available compression techniques are considered for compressing a container but, unless there is some size reduction, no compression is performed. If a container is compressed, the needed data is automatically uncompressed as it is read.

Auto-compression happens automatically and is most effective when the column partition is based on a single column only, and less effectively as more columns are included in the column partition.

User-defined compression, such as multi-value or algorithmic compression that is already defined by the user is honored and carried forward if it helps compress the container. If block-level compression is specified, it applies to data blocks holding the physical rows of the table independent of whether auto-compression is applied or not.

There is overhead in determining whether or not a physical row is to be compressed and, if it is to be compressed, what compression techniques are to be used. This overhead can be eliminated by specifying the NO AUTO COMPRESS option for the column partition.

Auto-Compression Techniques for CP Tables (1 of 2)

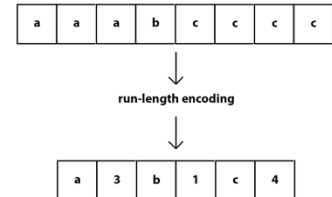
teradata.

- Run-Length Encoding

- Each series of one or more column-partition values that are the same are compressed by having the column-partition value occur once with an associated count of the number of occurrences in the series

- Local Dictionary Compression

- This is similar to a Multi-value compression for a column. Often occurring column-partition values within a physical row are placed in a value-list dictionary local to the physical row



- Trim Compression

- Trim high-order zero bytes of numeric values and trailing pad bytes of character and byte values with bits to indicate how many bytes were trimmed or what the length is after trimming

This slide lists and briefly describes each of the auto-compression techniques that Teradata may utilize.

Value List Compression (VLC) aware Bulk Evaluation

This Teradata 16.0 internal performance feature (VLC/RL aware bulk evaluation) was implemented to improve CPU performance by evaluating on value list and, value list plus run length (RL) compressed columns, and run length compressed columns in a columnar table. Increased CPU performance is achieved by reducing CPU consumption because column data does not need to be decompressed. This form of predicate evaluation is possible when the columns in the table are compressed using either of the following:

- Container VLC or Container VLC plus run length compression or Container run length compression.
- Table header VLC with container VLC compression or VLC plus run length compression or container run length compression.

This feature supports VLC aware compressed bulk evaluation (EVL) for the entire relational search based scans.

Note: After predicate evaluation, other operations such as projection may decompress the data.

If the container row is not compressed, the predicate evaluation fallbacks to regular EVL in which actual comparison of uncompressed values occur.

Auto-Compression Techniques for CP Tables (2 of 2)

- Null Compression

- A single-column or multicolumn-partition value is a candidate for null compression if all the column values in the column-partition value are null

- Delta on Mean Compression

- Delta on Mean compression computes the mean/average of all the values in the column container which is saved and stored in the container
- After Delta on Mean compression, the value stored for a row is the difference from the mean

- Unicode to UTF8 Compression

- This is applicable for a column defined with a UNICODE character set and the value consists of ASCII characters
- Compress the Unicode representation (2 bytes per character) to UTF8 (1 byte per character)

Birth Date container 1 Original = 07-04-1970	
	+73
	-198
	+38

User-Defined Compression Techniques

All the current compression techniques are available and can be leveraged and used for column-partitioned tables.



Dictionary-based compression (MVC)

Allows end-users to identify and target specific values that would be compressed in each column.

Algorithmic compression

Allows users to define custom compression/decompression algorithms that would be implemented as UDFs and applied to data at the column level in a row.

Block-Level compression

The feature provides the capability to perform compression on whole data blocks at the file system level before actually being written to the storage.

User-defined compression, such as multi-value or algorithmic compression that is already defined by the user is honored and carried forward if it helps compress the container.

If block level compression is specified, it applies for data blocks holding the physical rows of the table independent of whether auto-compression is applied or not.

Note that auto-compression is applied locally to a container based on column partition values (which may be multicolumn) while user-specified MVC and ALC are applied globally for a column and are applicable to both containers and sub-rows.

Auto-compression is differentiated from block level compression in several key ways:

- Auto-compression requires no parameter setting, but rather is completely transparent to the user while block level compression is a result of the appropriate settings of parameters.
- Auto-compression acts on a container (a physical row) while block level compression acts on a data block (which consists of one or more physical rows).
- Decompressing a column partition value in a container has little overhead while software-based block level compression incurs noticeable decompression overhead.
- Only column partition values that are needed by the query are decompressed. BLC has to decompress the entire data block even if only one or a few values are needed from the data block.
- Determining the auto-compression to use for a container, compressing a container, and compressing additional values to be inserted into the container can cause an increase in the CPU needed for appending values to column partitions.

Column Partition Container (Automatic Compression)

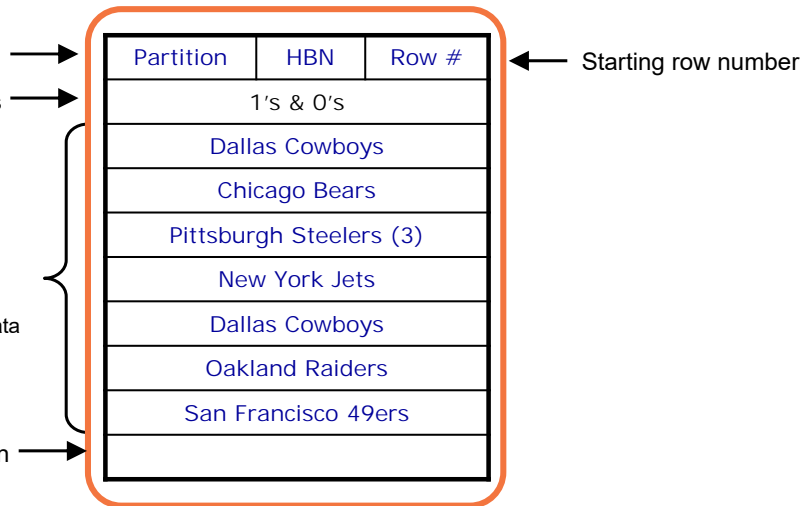
teradata.

Column Store RowID

Auto-Compression & NULL Bits

Column Data

Column (Local) Compression
Dictionary



Column Container is effectively a row in the partition.

In order to support columnar-storage for a column partition, a format, referred to as a COLUMN format in the syntax, is available for a physical row.

The example on this slide assumes that automatic compression is on for the column.

Column Partition Table (with Auto-Compression)

Winner	Loser	Game_Date	Game_Score	Attendance
Part 1-HBN-Row #1	Part 2-HBN-Row #1	Part 3-HBN-Row #1	Part 4-HBN-Row #1	Part 5-HBN-Row #1
1's & 0's	1's & 0's	1's & 0's	1's & 0's	1's & 0's
Dallas Cowboys	Denver Broncos	01-15-1978	27-10	(Null)
Chicago Bears	New England Patriots	01-26-1986	46-10	73,818
Pittsburgh Steelers	Arizona Cardinals	02-01-2009	27-23	70,774
Pittsburgh Steelers	Minnesota Vikings	01-12-1975	16-6	80,997
Pittsburgh Steelers	Seattle Seahawks	02-05-2006	21-10	68,206
New York Jets	Baltimore Colts	01-12-1969	16-7	75,389
Dallas Cowboys	Buffalo Bills	01-31-1993	52-17	(Null)
Oakland Raiders	Philadelphia Eagles	01-25-1981	27-10	76,135
San Francisco 49ers	Cincinnati Bengals	01-24-1982	26-21	81,270

⋮
Run-Length
Encoding

⋮
Trim Trailing
Spaces

⋮
No Compression

⋮
Local Dictionary
Compression
(27-10 is
compressed)

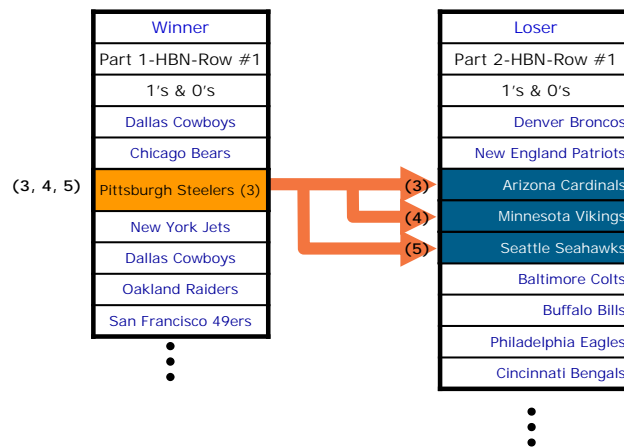
⋮
Null
Compression

Columnar compression is based on each Container. Therefore, each Container may have different compression characteristics and even different compression methods.

The result of creating a column partitioned table with auto-compression is shown on this slide.

CP Table Query #2 (with Auto-Compression)

Which teams have lost to the "Pittsburgh Steelers" in the Super Bowl?



The Pittsburgh Steelers team was compressed, but effectively represented 3 values in the container. These 3 values correspond to 3, 4, and 5 in the other container (Loser column).

CP Table with Row Partitioning DDL

```
CREATE TABLE Super_Bowl_CP_RP
(Winner      CHAR(25)    NOT NULL
,Loser       CHAR(25)    NOT NULL
,Game_Date   DATE        NOT NULL
,Game_Score  CHAR(7)     NOT NULL
,City        CHAR(40))
PARTITION BY
(COLUMN
,RANGE_N(Game_Date BETWEEN
          DATE '1960-01-01' and DATE '2059-12-31'
          EACH INTERVAL '10' YEAR));
```

Note: Auto Compression is on by Default.

Row partitioning can be combined with column partitioning on the same table. This allows queries to read only non-eliminated combined partitions. Such partitions are defined by the intersection of the columns referenced in the query and any partitioning column selection criteria.

There is usually an advantage to putting the column partitioning at level one of the combined partitioning scheme.

The DDL to create a column partitioned table with auto-compression and Row partitioning is shown on this slide.

If the table or join index is defined with multiple partitioning levels, you can (and possibly should consider) explicitly specify the number of partitions per partitioning level using the ADD option. If you do not, Teradata reserves any excess partitions for partitioning level 1 which is usually the COLUMN partitioning level, and these partitions cannot be added to any other partitioning level once the table or join index is created.

The maximum number of partitions that are available for the level 1 partition from excess partitions depends on whether you have defined the object with 2-byte partitioning or with 8-byte partitioning.

Column Partition Table (with Row Partitioning)

In the 1970s, which teams won Super Bowls, who were the losing teams, and when the game was played?

Winner	Loser	Game_Date	Game_Score	City
Part 1-HBN-Row #1	Part 2-HBN-Row #1	Part 3-HBN-Row #1	Part 4-HBN-Row #1	Part 5-HBN-Row #1
1's & 0's	1's & 0's	1's & 0's	1's & 0's	1's & 0's
New York Jets	Baltimore Colts	01-12-1969	16-7	Miami, FL

Winner	Loser	Game_Date	Game_Score	City
Part 11-HBN-Row #1	Part 12-HBN-Row #1	Part 13-HBN-Row #1	Part 14-HBN-Row #1	Part 15-HBN-Row #1
1's & 0's	1's & 0's	1's & 0's	1's & 0's	1's & 0's
Dallas Cowboys	Denver Broncos	01-15-1978	27-10	New Orleans, LA (2)
Pittsburgh Steelers	Minnesota Vikings	01-12-1975	16-6	

⋮

⋮

⋮

⋮

⋮

Winner	Loser	Game_Date	Game_Score	City
Part 41-HBN-Row #1	Part 42-HBN-Row #1	Part 43-HBN-Row #1	Part 44-HBN-Row #1	Part 45-HBN-Row #1
1's & 0's	1's & 0's	1's & 0's	1's & 0's	1's & 0's
Pittsburgh Steelers	Seattle Seahawks	02-05-2006	21-10	Detroit, MI

The result of creating a column partitioned table with auto-compression and row partitioning is shown on this slide.

CP Table with Multi-Column Container DDL

```
CREATE TABLE Super_Bowl_CP_MC_noAC
( Winner          CHAR(25) NOT NULL
, Loser           CHAR(25)   NOT NULL
, Game_Date       DATE       NOT NULL
, Game_Score      CHAR(7)    NOT NULL
, Attendance      INTEGER)
PARTITION BY COLUMN NO AUTO COMPRESS
( Winner
, Loser
, (Game_Date
, Game_Score
, Attendance))
;
```

This example is created without Auto-Compression for demonstration purposes.

Recommendation:
The group of multiple columns should be less than 256 bytes.

Watch the difference between 'Projection' and 'Predicate'

- If you are always projecting three columns, it may make sense to group these columns into one Container; however, if one of these columns is used in a WHERE Predicate, then it may be better to place this column into its own Container

When a table is defined with column partitioning, by default each column becomes its own column partition. However, it is possible to group multiple columns into a single partition.

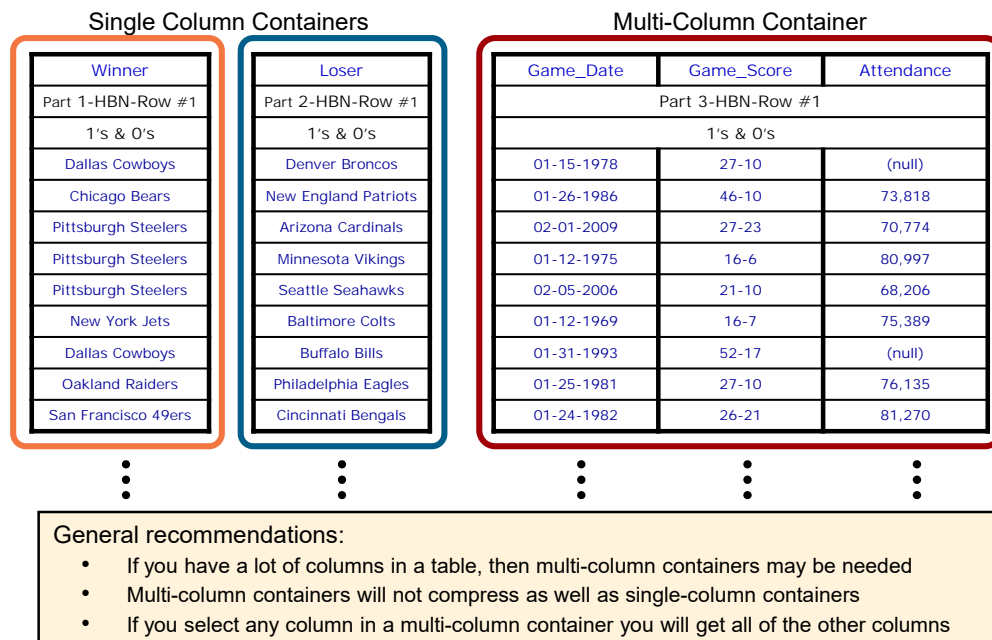
This has the result of fewer column partitions with more data held within each column partition.

Grouping columns into fewer column partitions may be appropriate in these situations:

- When the table has a large number of columns (having fewer column partitions may improve the performance of INSERT-SELECT and UPDATE statements).
- When access to the table often involves a large percentage of the columns and the access is not very selective.
- When a common subset of columns are frequently accessed together.
- When a multicolumn NUSI is created on a group of columns.
- There are too few available column partition contexts to access all the needed column partitions for queries.

Note that auto-compression will probably be less effective if columns are grouped together instead of being in their own column partitions.

CP Table with Multi-Column Container



The example on this slide illustrates a CP table that has a multi-column container.

CP Table Hybrid Row and Column Store DDL

```
CREATE TABLE Super_Bowl_CP_RF_noAC
(Winner      CHAR(25)      NOT NULL
,Loser       CHAR(25)      NOT NULL
,Game_Date   DATE          NOT NULL
,Game_Score  CHAR(7)       NOT NULL
,Attendance  INTEGER
,City        CHAR(40))
PARTITION BY COLUMN NO AUTO COMPRESS
(Winner
,Loser
,ROW ( Game_Date
      ,Game_Score
      ,Attendance
      ,City)
);
```

This example is created without Auto-Compression for demonstration purposes.

This example illustrates the syntax to create a row store, but you would only define the row format if the set of columns was greater than 256 bytes.

General recommendations:

- A column (or set of columns) should be at least 256 bytes wide before considering ROW format
- Row stores will take up more space, but act like a row in terms of retrieving data
- Each row will have a row header and require more space

The example on this slide illustrates the DDL to create a column partitioned table that has a combination of row and column storage.

COLUMN Format Considerations

The COLUMN format packs column partition values into a physical row, referred to as a container, up to a system-determined limit. Whether or not to change a column partition to use ROW format depends on the whether the benefit of row header compression and auto-compression can be realized or not.

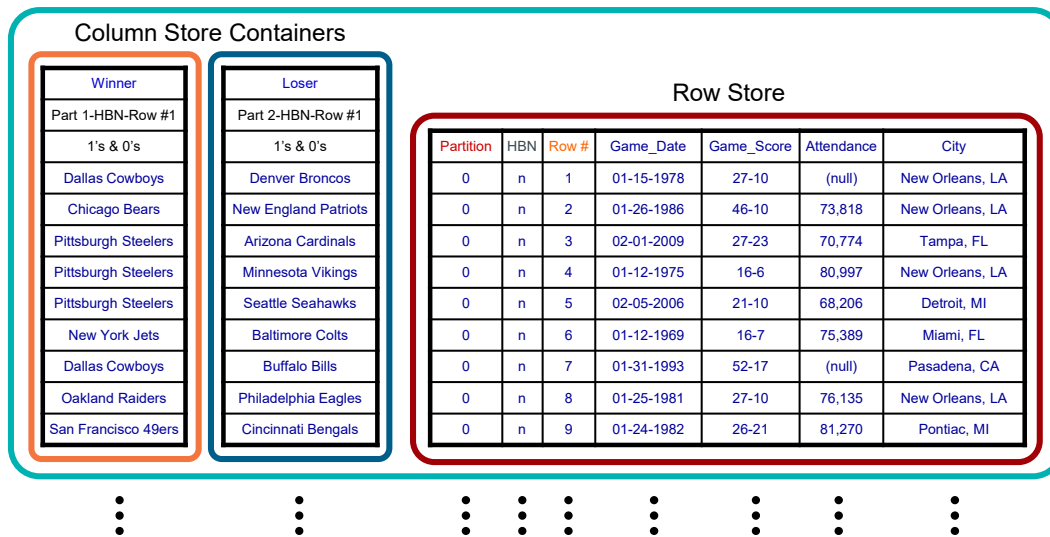
A row header occurs once per container, with the RowID of the first column partition value becoming the RowID of the container itself. In a column-partitioned table, each column partition value is assigned its own RowID, but in a container these RowIDs are implicit except for the first one specified in the header. The uniqueness value can be determined from the position of a column partition value relative to the first column partition value. Thus the row id for each value in the container is implicitly available and an explicit RowID does not need be carried for each individual column value in the container.

ROW Format Considerations

A subrow, on the other hand, has a format that is the same as traditional row (except it only has the values of a subset of the columns). Subrows are appropriate when column partition values are very wide and you expect only one or a few values to fit in a columnar container.

CP Table (with Hybrid Row and Column Store)

Column and Row Store in "one" table.



As an alternative to COLUMN format, column partition values may be held in a physical row using what is known in Teradata syntax as ROW format. The type of physical row supports row-storage for a column partition and is referred to as a subrow. Each subrow holds one column partition value for a column partition. A subrow has the same format as regular row except that it is generally a subset of the columns for a table row instead of all the columns. Just like a container, each subrow is assigned to a specific partition. One or more subrows may be needed to hold the entire column partition. Since a subrow is a physical row, the size of a subrow is limited by the maximum physical row size.

A column partition may have COLUMN format or ROW format but not a mix of both. However, different column partitions in column-partitioned table may have different formats.

Populating a CP Table

```
CREATE TABLE Super_Bowl_Staging
(Winner      CHAR(25) NOT NULL
,Loser       CHAR(25) NOT NULL
,Game_Date   DATE      NOT NULL
,Game_Score  CHAR(7)   NOT NULL
,Attendance  INTEGER)
NO PRIMARY INDEX;
```

```
CREATE TABLE Super_Bowl_CP
(Winner      CHAR(25) NOT NULL
,Loser       CHAR(25) NOT NULL
,Game_Date   DATE      NOT NULL
,Game_Score  CHAR(7)   NOT NULL
,Attendance  INTEGER)
PARTITION BY COLUMN;
```

1. Load data into staging table
2. `INSERT INTO Super_Bowl_CP ... SELECT * FROM Super_Bowl_Staging ...`

INSERT-SELECT

INSERT-SELECT is the expected and most efficient method of loading data into a column-partitioned table. If the data originates from an external source, FastLoad can be used to load it into a staging table from which the INSERT-SELECT can take place. If the source was a SELECT that included several joins and as a result skewed data was produced, options can be added to the INSERT-SELECT statement to avoid a skewed column-partitioned table and improve the effectiveness of auto-compression:

Options

HASH BY (RANDOM or hash_spec_list):

The selected rows are redistributed by the hash value of the expressions in the hash_spec_list. Alternatively, HASH BY RANDOM can be specified to have data blocks redistributed randomly. It is important that a column or columns be selected that distributes well if the HASH BY option is used.

LOCAL ORDER BY

A local sort is done on each AMP before physically storing the rows. This could help auto-compression to be more effective by ensuring that like values of the sorting columns appear together.

During an INSERT-SELECT process, each source row is read, and its columns individually appended to the column partitions to which they belong. As many column partition values as can fit are built up simultaneously in memory, and written out to disk when the buffer is filled.

DELETE Considerations

- DELETE ALL uses the standard fast-path delete as would be done on a primary-indexed table
 - If a CP table also happens to include row partitioning, the same fast-path delete can be applied to one or more row partitions
 - Space is immediately reclaimed
- The selective DELETE, in which only one or a few rows of the table are deleted, requires a scan of a column partition or indexed access to the column-partitioned table
 - In this case, the row being deleted is not physically removed, but only flagged as having been deleted
 - This form of delete should only be used to delete a small percentage of rows
- Delete Column Partition – each column-partitioned table has one delete column partition, in addition to the user-specified column partitions. It holds information about deleted rows so they do not get included in an answer set
 - One bit in the delete column partition is set as an indication that the hash bucket and uniqueness associated with the table row have been deleted

Rows can be deleted from a column-partitioned table using the DELETE ALL, or selectively using DELETE. DELETE ALL uses the standard fast-path delete as would be done on a primary-indexed table. If a column-partitioned table also happens to include row partitioning, the same fast-path delete can be applied to one or more row partitions. Space is immediately reclaimed.

The selective DELETE, in which only one or a few rows of the table are deleted, requires a scan of a column partition or indexed access to the column-partitioned table. In this case, the row being deleted is not physically removed, but only flagged as having been deleted. The space taken by a row being deleted is scattered across multiple column partitions and is not reclaimed at the time of the deletion. This form of delete should only be used to delete a small percentage of rows.

During a delete operation, all large objects are immediately deleted, as are entries in secondary indexes. Join indexes are updated to reflect the change as it happens. Starting with Teradata 15.10 data from column partitions with ROW format are also immediately deleted. In addition, if all the column partitions have ROW format, the delete column partition is not used and is empty.

The Delete Column Partition

Each column-partitioned table has one delete column partition, in addition to the user-specified column partitions. It holds information about deleted rows so they do not get included in an answer set.

This delete column partition is accessed any time a query is made against a column-partitioned table without indexed access.

UPDATE and USI/NUSI Considerations

UPDATE Considerations

- Updating rows in a column partitioned table requires a delete and an insert operation.
- It involves marking the appropriate bit in the delete column partition and then re-inserting columns for the new updated version of the table row
- An UPDATE statement should only be used to update a small percentage of rows

Updating rows in CP table requires a delete and an insert operation. It involves marking the appropriate bit in the delete column partition, and then re-inserting columns for the new updated version of the table row. The cost of this update is less severe than a Primary Index update (also a delete plus insert) because in the column-partitioned table update, the deletion and re-insertion takes place on the same AMP. An UPDATE statement should only be used to update a small percentage of rows.

The part of the update that re-inserts a new table row is essentially a re-append. The highest uniqueness value on that AMP is incremented by one, and all the column values for that updated row are appended to their corresponding column partitions. Because multiple I/Os are performed in doing this re-append, row-at-a-time updates on column-partitioned tables should be approached with caution. The space that is being used by the old row is not reclaimed, but a delete bit is turned on in the delete column partition, indicating that the old version of the row is obsolete.

USI Access

For example, consider a unique secondary index (USI) access. The USI subtable provides the specific RowID of the base table row. In the columnar case, the base table row is located on a specific AMP which can be stored in multiple containers.

NUSI Access

With NUSI access, a row-id list is retrieved from the NUSI subtable. In the case of a column-partitioned table, the table row has been decomposed into columns that are located in different column partitions on disk. Several different internal partition numbers come into play in reconstructing the table row.

USI/NUSI Considerations

- For a USI on a CP table, the base table row is located on a specific AMP which can be stored in multiple containers. The hash bucket in the RowID carried in the USI is used to locate the AMP that contains the base table row.

- For a NUSI on a CP table, the table row has been decomposed into columns that are located in different column partitions on disk. Several different internal partition numbers come into play in reconstructing the table row.
- Rather than relying on the column partition number, it is only the hash bucket and uniqueness that is of importance in the NUSI subtable RowID list.

CP Table Restrictions

- Column Partitioning is predicated on the NoPI table structure and as such the following restrictions apply:
 - Set Tables | Queue tables | Global Temporary Tables | Volatile Tables | Derived Tables
 - Multi-table or Aggregate Join Index | Compressed Join Index | Hash Index
 - Secondary Indexes are not column partitioned
- Column Partitioned tables cannot be loaded with
 - TPT Load (FastLoad) or TPT Update (MultiLoad)
- Merge-Into and UPSERT statements are not supported
- Population of Column Partition tables will require an Insert-Select process after data has been loaded into a staging table
- No synchronized scanning with Columnar Tables

The following limitations apply:

- Column partitioning for join indexes is restricted to single-table, non-aggregate, non-compressed join indexes with no PI and no ORDER BY clause
 - ROWID of base table must be included in a CP join index
- Column partitioning is not allowed for the following:
 - Global temporary, volatile, and queue tables
 - Secondary indexes
- Column partitioning is not applicable for the following:
 - Global temporary trace tables
 - Error tables
 - Compressed join indexes
- NoPI table with only row partitioning is not allowed
- A column cannot be specified to be in more than one column partition
- Column grouping cannot be specified in both the column definition list of CREATE TABLE statement and in the COLUMN clause.
- Column grouping cannot be specified in both the select list of a CREATE JOIN INDEX statement and in the COLUMN clause.

Key Highlights

When is column partitioning useful?

- Queries access varying subsets of the columns of a table
or
Queries of the table are selective (Best if both occur for queries)
- Data can be loaded with large INSERT-SELECTs
- There is no or little update/delete maintenance between refreshes or appends of the data

Do not use this feature when:

- Queries need to be run on current data that is changing (deletes and updates)
- Performing tactical queries or OLTP queries
- Workload is CPU bound such that a trade-off of reduced I/O with increased CPU does not improve the throughput
- Column partitioning is *not* intended to be a CPU savings feature

This slide contains a summary of the key concepts associated with column partitioning and a list of scenarios where column partitioning is not useful.

Summary

Now that you have completed this course, you will be able to:

- Compare Row versus Columnar
- Describe why Columnar is used
- Describe the components that comprise a column-partitioned table
- Identify two advantages of creating a column-partitioned table
- Identify two disadvantages of creating a column-partitioned table
- Identify the recommended way to populate a column-partitioned table
- Specify how rows are deleted in a column partitioned table



Module 9: Columnar Bring Up JupyterHub

teradata.

Let's now do the lab together



Thank you.

teradata.

©2023 Teradata



Module 10: Native Object Store

Teradata Vantage MasterClass

Copyright © 2007–2023 by Teradata. All Rights Reserved.

Objectives

After completing this module, you will be able to:

- Define and describe the Native Object Store (NOS) feature and the capabilities it offers
- List and describe some use cases for NOS
- Describe READ_NOS, WRITE_NOS



Why Object Storage? (1 of 2)

Benefits of object stores:

- Pay-as-you-go
- Low-cost storage tier
- Scalable
- Highly available
- Improved data management

Large numbers of companies are now storing vast amounts of data in object stores



Why Object Storage? (2 of 2)

Customer Use Cases

HYBRID CLOUD

On-premises Customer must accelerate the pay-as-you-go cloud model for lines of business

DATA EXPLORATION

Customer struggles to understand what relevant data might exist within their numerous data lakes

QUERY ARCHIVE

Customers can reduce storage costs while extending longevity access to their data for compliance

DATA SHARING

Customers can share data with several 3rd party vendors via public cloud object stores

BATCH & STREAM ANALYTICS

Customer looking to predict equipment failure to avoid customer disruptions while reducing maintenance costs

What is Object Storage?

- Data Stored as distinct atomic units, called objects
- Objects are kept in a single storehouse and are not ingrained in files inside other folders
- Combines the pieces of data that make up a file, adds all its relevant metadata to that file, and attaches a custom identifier
- Provides comprehensive metadata, eliminating tiered file structure
- Places everything into a flat address space called Storage Pool



Object storage, also known as object-based storage, is a mechanism that manages data storage as distinct units, called objects. These objects are kept in a single storehouse and are not ingrained in files inside other folders. Instead, object storage combines the pieces of data that make up a file, adds all its relevant metadata to that file, and attaches a custom identifier.

Object storage adds comprehensive metadata to the file, eliminating the tiered file structure used in file storage, and places everything into a flat address space, called a storage pool. This metadata is key to the success of object storage.

Pros and Cons of Object Stores

Advantages

- ✓ Infinite scalability and Durable
- ✓ Faster data retrieval for Serial I/O
- ✓ Reduction in cost
- ✓ Optimization of resources

Disadvantages

- ✗ Object Stores are immutable
- ✗ Not suitable for random I/O
- ✗ Not suitable for low-latency

Object stores vs. objects: "Objects" are the discrete units that compose an "object store". Objects are stored in buckets and can be organized with shared names called prefixes. An "object store" is a collection of related objects, with all participating objects located in the same bucket and organized in a hierarchy.

Files vs. objects: "Objects" and "files" can be used interchangeably to describe the components of an object store. Each file or object contains records in which the detailed data itself is held.

Advantages

Infinite scalability and Durable.

Keep adding data, forever. There's no limit.

Faster data retrieval for Serial I/O.

Due to the categorization structure of object storage, and the lack of folder hierarchy, you can retrieve your data much faster for serial I/O.

Reduction in cost.

Due to the scale-out nature of object storage, it's less costly to store all your data.

Optimization of resources.

Because object storage does not have a filing hierarchy, and the metadata is completely customizable, there are far fewer limitations than with file or block storage.

Disadvantages

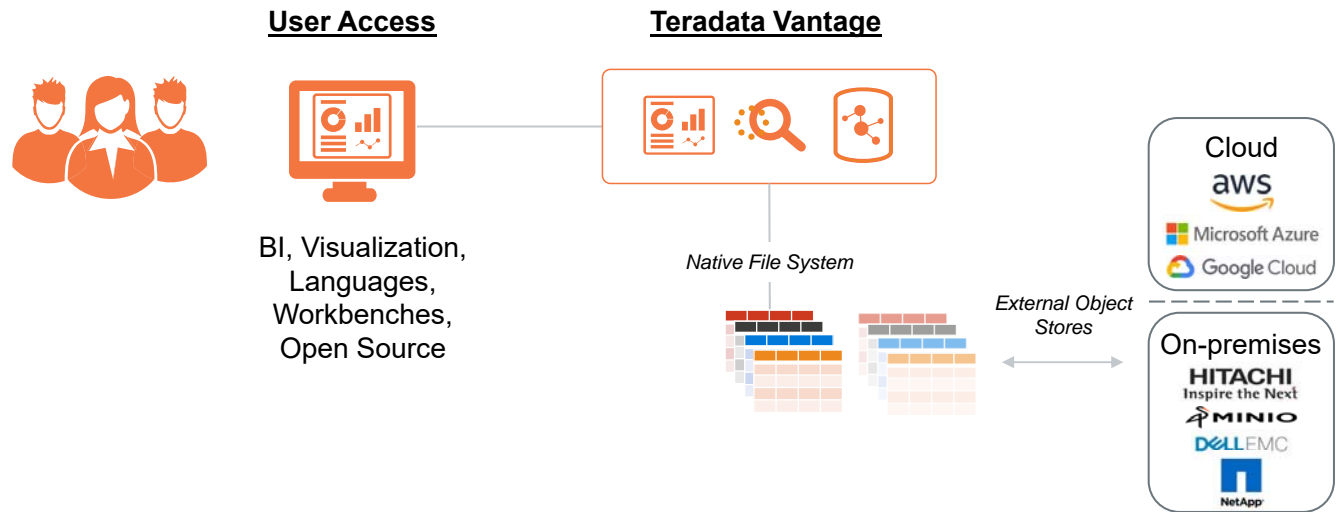
Object Stores are immutable which means implementing updates/deletes via an Application is very expensive. Object storage doesn't allow you to alter just a piece of a data blob, you must read and write an entire object at once

Not suitable for random I/O

Not suitable for low latency applications

What is Native Object Store?

- Seamless Access To External Object Stores



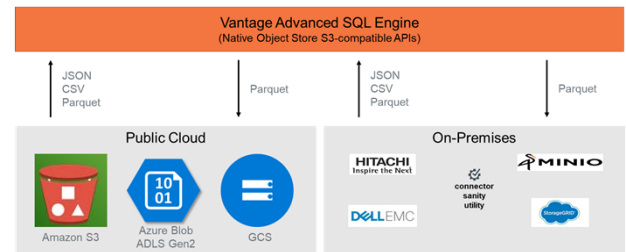
Note: Teradata Vantage supports NOS on-premises as well as in AWS, Azure and GCP

NOS provides seamless access to external object stores. Users submit their queries in Teradata Vantage which can go to AWS SE, Azure or Google Cloud. And we can also access object stores on-premise, which are third-party objects stores that you can setup within a customer's datacenter. This means your data doesn't have to be in the cloud to take advantage of the NOS capabilities. You can still offload data in an internal object store, and access it via NOS. This on-premise feature came along with Teradata Vantage 2.2.

Native Object Store Capabilities

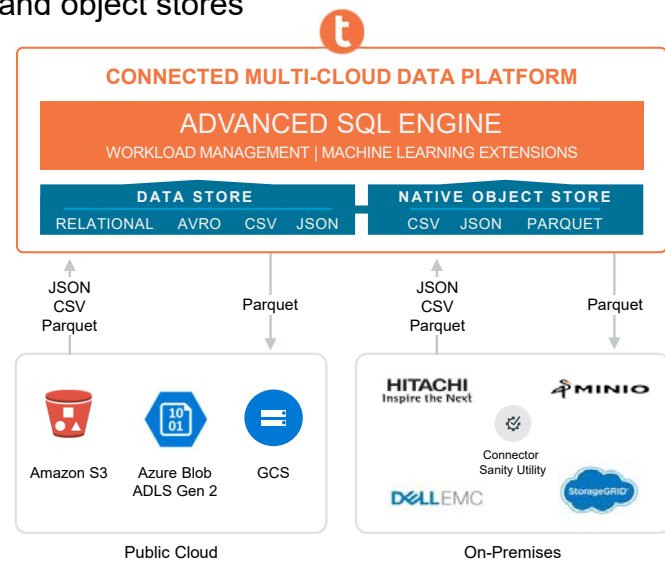
Native Hybrid Multi-Cloud Integration With External Object Storage

- Facilitates ad hoc **data exploration** of unknown data sets
- Enables users to **join relational and semi-structured data** sets
- Enables searchable archives by **offloading relational data**, yet query-able using the same SQL as when the data was in Vantage
- **Simplifies loading** semi-structured data into relational tables with a single SQL command
- Accelerates **data sharing**



How NOS is Modernizing Data Management

- Break data silos between data warehouses and object stores
- Read and write open-format data sets on object storage
- Query data at a scale where it lives
- Join relational and semi-structured data in a single query
- Transparent to end users and applications – works with all SQL + functions (extends to R and Python)
- Leverages proven workload management, parallelism, optimizer, and security controls
- Ideal for self-service exploration, searchable archives, data sharing, and ad hoc data ingest use cases



Data gravity in object stores is accelerating and is projected to reach about 7.5 zettabytes of data by 2025.

With Teradata Vantage's Native Object Store Capabilities, business users can effectively join relational and curated data that lies within the data warehouse with the semi-structured data that lands on the object stores in place with a single query effectively.

What this means is that you don't have to do any preprocessing of the data to load this data into the warehouse. this breaks down the data silos between the enterprise data warehouse and data lakes

Typically, customers can leverage Native Object Stores to

Facilitate ad hoc data exploration of unknown data sets

Enable query-able archives by offloading relational data, yet query-able using the same SQL as when the data was in Vantage

Simplify loading semi-structured data into relational table with a single SQL command

NOS Feature Description

- NOS supports open data file formats: JSON, CSV, and Parquet data
 - Compression options
 - GZIP for JSON and CSV (suffix .gz)
 - SNAPPY for Parquet
- NOS provides direct read and write access to the external data
 - HTTPS / TLS (recommend TLS 1.2 though will support earlier, configured on object store)
 - AMP parallelism for reading and writing
 - Optimizer aware
- Currently supported object storage platforms
 - Cloud: Amazon S3, Azure Blob storage, Azure Data Lake Storage Gen2, Google Cloud Storage
 - On-prem: Hitachi HCP, MinIO, Dell EMC/ECS, NetApp StorageGrid, IBM COS

JSON and CSV support Latin-1 or UTF8 encoding, not supported: UTF16, UTF32

Vantage NOS On-Premises

- **On-Premises** is comprised of the Advanced SQL Engine and Tools and Utilities (Vantage 2.2/17.05 and greater)
- SLES 12 is required to leverage the NOS capability
- NOS is certified with **On-Premises** external object stores including the following:
 - Hitachi Content Platform | MinIO
 - NetApp StorageGRID | Dell EMC/ECS
- Users can read and write to both on-premises and cloud external object stores with NOS starting

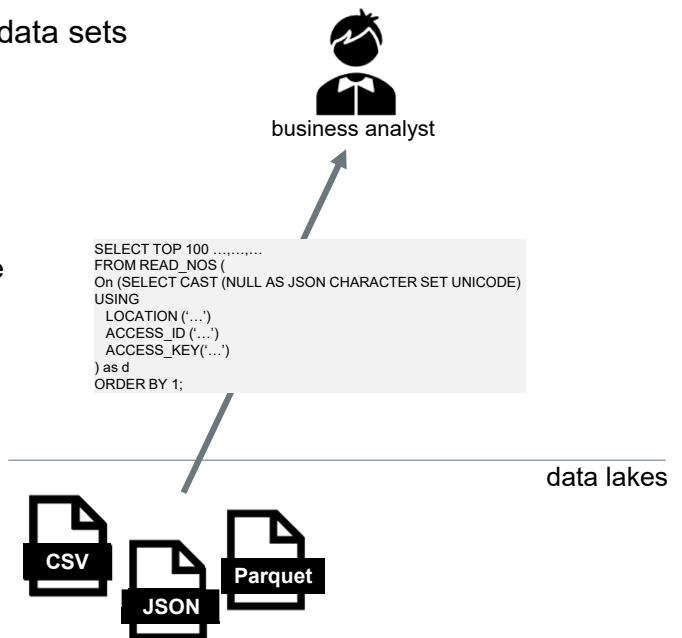


Supports most existing hardware,

Exploring External Data in Place – Use Case 1

Facilitates ad hoc **data exploration** of unknown data sets

- Problem
 - Manual data prep costs for ad hoc queries are too high
- Solution
 - Automate the data prep activities at query time
- Benefits
 - Removes data replication costs
 - Free up DBA time for other business priorities
 - Get answers in minutes versus days or even weeks
 - Empower users to do it themselves

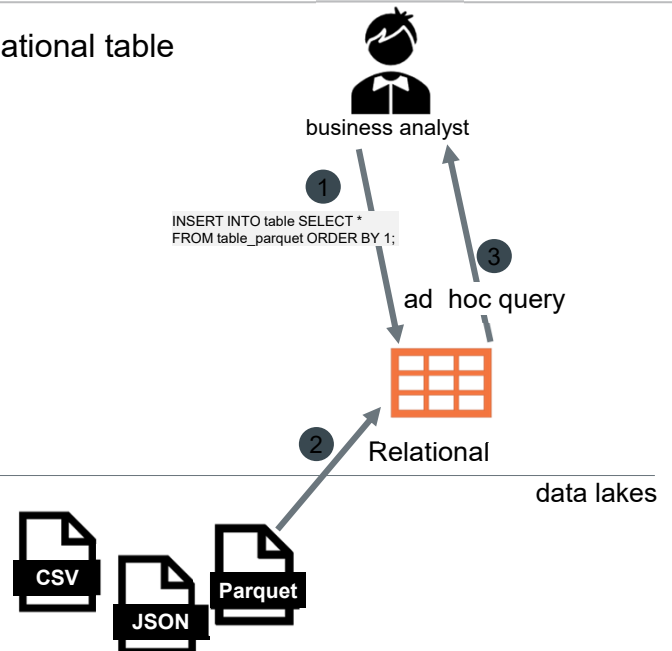


In this first use case that maps to our next module, module 1, addresses how can we do data exploration of unknown data sets. So, part of the problem is there is a lot of manual data prep involved with wrangling the data, making the costs for ad hoc queries too expensive. The solution to this problem is to automate the data prep at query time. We can use the READ_NOS operator to query these unknown data sets. This allows us to remove any kind of data replication costs by simply leaving the data out in the object store. And even more importantly it frees up DBA time, allowing DBAs to work on other business priorities. And for the end user, data scientist or business analyst, the key thing is they can get answers in minutes instead of days or weeks. They are no longer dependent on the DBA to get the data transformed and loaded for them. So, NOS empowers the users to do it themselves.

Simplified Data Load – Use Case 2

Simplifies loading semi-structured data into a relational table

- Problem
 - Manual data load is complicated and time-consuming
- Solution
 - Load data in minutes from an object store with standard SQL INSERT SELECT statement
 - Read in parallel and at scale
- Benefits
 - Free up DBA time for other business priorities
 - Get to answers in minutes versus days or even weeks
 - Empower users to do it themselves

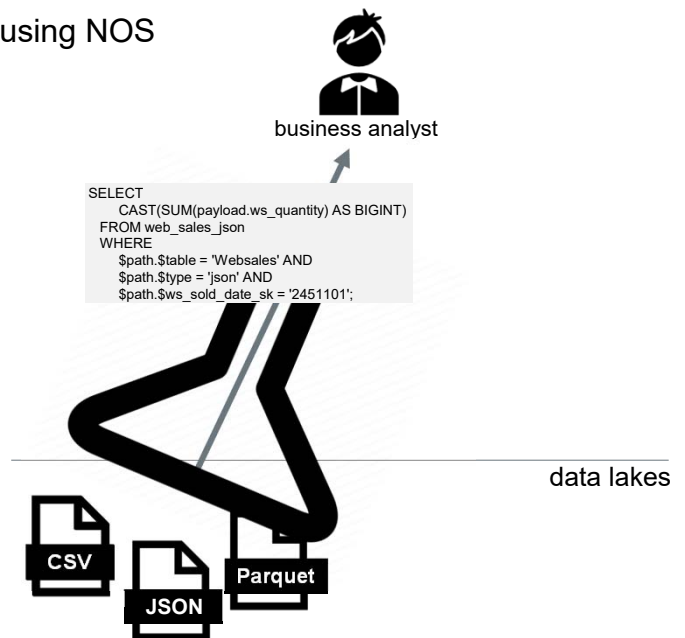


This use case addresses how to simplify the loading of semi-structured data into relational tables, and it maps to the second module of this course. The problem is that manual data loading is complicated and can be very time consuming. So, if we can have a solution that in minutes we can load data from an object store using SQL INSERT/SELECT statements, we can alleviate the problems related to manual loading of data. And NOS can read data from the object store in parallel and at scale. We can see that the benefits are very similar to the first use case. It frees up DBA time, allowing DBAs to work on other business priorities. And for the end user, data scientist or business analyst, the key thing is they can get answers in minutes instead of days or weeks. They are no longer dependent on the DBA to get the data transformed and loaded for them. Again, NOS empowers the users to do it themselves.

Performance Considerations – Use Case 3

Understand **Performance considerations** when using NOS

- Problem
 - Object data can be very large causing performance issues
- Solution
 - Use Path/Payload filtering to reduce the amount of data being read
- Benefits
 - Improved query performance



In this third use case, that maps to our module 3 of our course, addresses the performance considerations when using NOS. The problem is that the amount of data in object storage can get very large, and that massive amount of data can cause some performance issues. The solution is to use path/payload filtering. We will get into the details of path/payload filtering in module 3, but this filtering will reduce the amount of data that needs to be read. We will also look at this notion of a manifest file, which we can use to specify the precise data objects to include in a NOS foreign table, which can also reduce how much data we are accessing. These are ways that allow us to manage the amount of data that we want to read from these massive data stores, which will improve query performance.

READ_NOS Table Operator Overview

- A **fast path table operator** to read external data without having to create a foreign table
 - Can be used stand-alone or more complicated SQL e.g., Subqueries, joins, insert/select
- It can be used to do the following:
 - Read data from a given location
 - Recursively list all the files in a given location
 - Sample a percentage of the data
 - and many more
- Only supports JSON and CSV for querying data
 - It has a PARQUET display schema option
- Security Clauses
 - ACCESS_ID → Same as USER in CREATE AUTHORIZATION
 - ACCESS_KEY → Same as PASSWORD in CREATE AUTHORIZATION

Let's talk about this READ_NOS table operator. READ_NOS is a fast path operator that allows us to read data from a data object store. You can even use it to do more complicated SQL like subqueries, joins, and insert/selects. We'll be using this READ_NOS operator and you will pass along the access_id and access_key to it that I just showed you. (I didn't show you the password in AWS, but you generate a password.) And you will use that to connect. Now, in this version of NOS in Vantage 2.2 you can use READ_NOS to read JSON and CSV data. But PARQUET only has a display schema option. That is, you can't read PARQUET data with READNOS. We'll cover READNOS in much more detail in the next module. But we wanted to give you a brief overview of READNOS because we are going to introduce it to you in this first lab so you can see basically how it works.

READ_NOS Functions

Purpose	Key Input Parameters	Return Type	Output
Read external data from a given location string	LOCATION	NOSREAD_RECORD	Record along with metadata like location string, version-id, timestamp, offset in the file, record length
List all the files in a given location string	LOCATION	NOSREAD_KEYS	Recursively list of files along with file metadata like location string, version-id, timestamp and length
Read raw external data	LOCATION	NOSREAD_RAW	Read the raw external data and return it as a CLOB for every file. The maximum CLOB size is 2GB – so if a file has more than 2GB, it gets truncated after that.
Sample external data	LOCATION SAMPLE_PERC	N/A	Sample the data at the specified location. Can sample between 0.0 and 1.0
Display Schema	LOCATION FULLSCAN	NOSREAD_SCHEMA NOSREAD_PARQUET_SCHEMA	Displays schema information (columns and types) For Parquet there is also the ability to do a full scan of complex objects.

Note: In Vantage 2.0 can use TPT to offload data. In Vantage 2.2, the NOS feature can write data in Parquet format.

The READ_NOS table operator has a variety of uses related to accessing both JSON or CSV data. READ_NOS accesses data through Native Object Store but without going through a foreign table. Using READ_NOS eliminates the need for creating a foreign table as a first step. As a result, the granting of CREATE TABLE privileges can be bypassed when accessing data through READ_NOS.

CREATE FOREIGN TABLE – JSON (1 of 2)

```
CREATE FOREIGN TABLE riverflowjson
,EXTERNAL SECURITY PublicAuth
USING ( LOCATION('/S3/s3.amazonaws.com/td-usgs-public/JSONDATA/') );

show table riverflowjson;

CREATE MULTiset FOREIGN TABLE USERNOS.riverflowjson ,FALLBACK ,
EXTERNAL SECURITY USERNOS.PUBLICAUTH ,
MAP = TD_MAP1
(
  Location VARCHAR(2048) CHARACTER SET UNICODE CASESPECIFIC,
  Payload JSON(16776192) INLINE LENGTH 64000 CHARACTER SET LATIN)
USING
(
  LOCATION  ('/S3/s3.amazonaws.com/td-usgs-public/JSONDATA/')
  MANIFEST  ('FALSE')
  PATHPATTERN  ('$Var1/$Var2/$Var3/$Var4/$Var5')
  ROWFORMAT  ('{"record_delimiter":"\n", "character_set":"LATIN"}')
  STOREDAS  ('TEXTFILE')
) NO PRIMARY INDEX ;
```

Note that the SHOW TABLE output contains many more clauses than the original CREATE FOREIGN TABLE statement illustrated. This is because the defaults are filled in automatically and are visible with the SHOW statement.

CREATE FOREIGN TABLE – JSON (2 of 2)

- If using IAM or the same account → **EXTERNAL SECURITY** clause can be omitted
- **MULTISET, NOPI** table only (defaults)
- **USING** → Defines connection and other characteristics. Only Location is required
- **LOCATION**
 - S3 – the type of storage
 - S3.amazonaws.com – Endpoint
 - td-usgs-public – S3 bucket name
 - Data – key prefix

Explanations for some of the important defaults that are part of the SHOW TABLE output include:

- **LOCATION** – Uniform Resource Identifier (URI) pointing to the external file system in the format /connector/end point/bucket/prefix. Above, the connector is "s3" and the end point is "s3.amazonaws.com".
- **MANIFEST** - TRUE/FALSE. If 'TRUE', Location points to a manifest file, otherwise Location points to an object name or object name prefix (full name or partial name). Default value is 'FALSE'.
- **PATHPATTERN** - Access information for key filtering. The default value of '\$Var1/\$Var2.../\$Var20' can be replaced with more meaningful path names, as will be discussed later.
- **ROWFORMAT** - Indicates the encoding format of the row. Record_delimiter can only be "\n", and character_set can be 'UTF8' or 'Latin'.
- **PAYLOAD** – Represents the data contained in the JSON object and can be either of these:
 - PAYLOAD JSON(16776192) INLINE LENGTH 64000 CHARACTER SET LATIN
 - PAYLOAD JSON(8388096) INLINE LENGTH 32000 CHARACTER SET UNICODE

Note that foreign tables are always defined as No Primary Index (NoPI) tables.

Bad Data in Numeric Fields

```
CREATE FOREIGN TABLE bad_numeric_data
,External security PublicAuth
USING (
  LOCATION ('/s3/s3.amazonaws.com/td-usgs-public/DATA-BAD/bad_numeric_data')
);
```

```
SELECT CAST(payload AS VARCHAR(200)) FROM bad_numeric_data;
```

Payload

```
-----
{ "site_no":"09396100", "datetime":"2018-07-16 00:00", "Flow":"447", "GageHeight":"1.50", "Precipitation":"0.00",
{ "site_no":"09396100", "datetime":"2018-07-14 00:00", "Flow":"232", "GageHeight":"2.16", "Precipitation":"0.00",
{ "site_no":"09396100", "datetime":"2018-07-14 00:14", "Flow":"186", "GageHeight":"2.05", "Precipitation":"0.00",
{ "site_no":"09400812", "datetime":"2018-07-12 00:09", "Flow":"", "GageHeight":"jjjj", "Precipitation":"bcde",
{ "site_no":"09400815", "datetime":"2018-07-12 00:00", "Flow":"0.00", "GageHeight":"-0.01", "
```

```
SELECT payload.site_no, payload.GageHeight(int) FROM bad_numeric_data;
```

```
*** Failure 2620 The format or data contains a bad character.
      Statement# 1, Info =0
*** Total elapsed time was 1 second.
```

Now another thing you can run into is that you could have bad data. Here, we created some bad data within our bucket. Now, when we created this foreign table with a location pointed to our bad data, and selected from its payload, you can see that there's a record with Gageheight and Precipitation, which should be numeric, but they have some character data.

So, when you run the select shown below here, you get a failure message, which says there's some bad data in there.

Data Dictionary – Foreign Table Views

- **DBC.ForeignTablesV** – shows the foreign tables

```
SELECT databasename(char(10)), tablename(char(20))
FROM dbc.foreigntablesv
WHERE databasename='usernos';
```

DataBaseName	TableName
-----	-----
usernos	skip_records
usernos	rivers_tsv_oops
usernos	riverflow_csv

- **DBC.ForeignTablesInfoV** - Shows the USING clause information

```
SELECT databasename(char(20)), tablename(char(20)), optioninfo(char(20)), valueinfo(char(80))
FROM dbc.foreigntablesinfov
WHERE tablename='riverflow' and databasename= DATABASE;
```

DatabaseName	TableName	OptionInfo	ValueInfo
-----	-----	-----	-----
-			
usernos	RIVERFLOW	MANIFEST	'FALSE'
usernos	RIVERFLOW	STOREDAS	'TEXTFILE'
usernos	RIVERFLOW	ROWFORMAT	'{"record_delimiter":"\n","character_set":"LATIN"}'
usernos	RIVERFLOW	PATHPATTERN	'\$Data/\$siteno/\$year/\$month/\$day'
usernos	RIVERFLOW	LOCATION	'/s3/s3.amazonaws.com/td-usgs/DATA/'

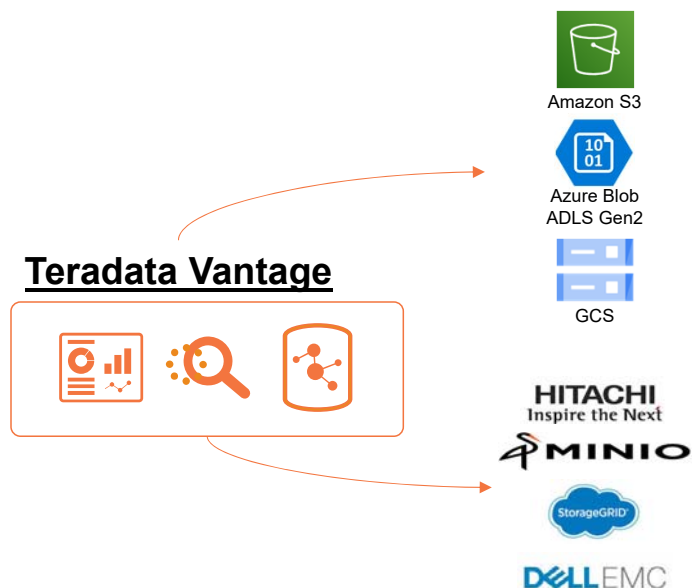
WRITE_NOS Table Operator Overview (1 of 2)

- A **fast-path table operator** and resides in TD_SYSFNLIB
- It writes selected/all columns in a table or derived results to an external file system (i.e. AWS-S3, MS-AZURE, Google Cloud..) in the desired format.
 - WRITE_NOS creates objects in the object store, it is not appropriate to run it from utilities like TPT or Data Mover
- Current platforms: AWS, Azure, GC, IFX, and existing Teradata hardware
- Current external file format → PARQUET
- Available from Release 17.10 / Vantage 2.3

WRITE_NOS is a fast-path table operator used to copy select columns and rows from a source relational table or a derived data set to an external object store target.

WRITE_NOS Table Operator Overview (2 of 2)

- Optimized **Parquet** data format
- WRITE_NOS fast-path table operator using a standard SQL statement
- CSV and JSON data format.
 - Create Teradata Parallel Transporter (TPT) scripts



WRITE_NOS makes it easy to organize data in an object store in a way that is optimized for subsequent read access via Native Object Store. Now we can write data from the Vantage sql engine to the object store using a table operator called WRITE_NOS. This is a fast-path table operator that uses a standard sql statement and writes the data out in the optimized Parquet data format. It can write to the three main cloud providers Amazon, Azure, and Google, as well as to on-premise data stores. Now if you needed to write the data out in CSV or JSON format, you could use a TPT type script to do that. There are alternate approaches that include using existing cloud utilities to write back to object stores, and third-party utilities with JDBC connectivity. So, there's other ways in which you can write the data out, but in this module we are going to focus in on WRITE_NOS table operator.

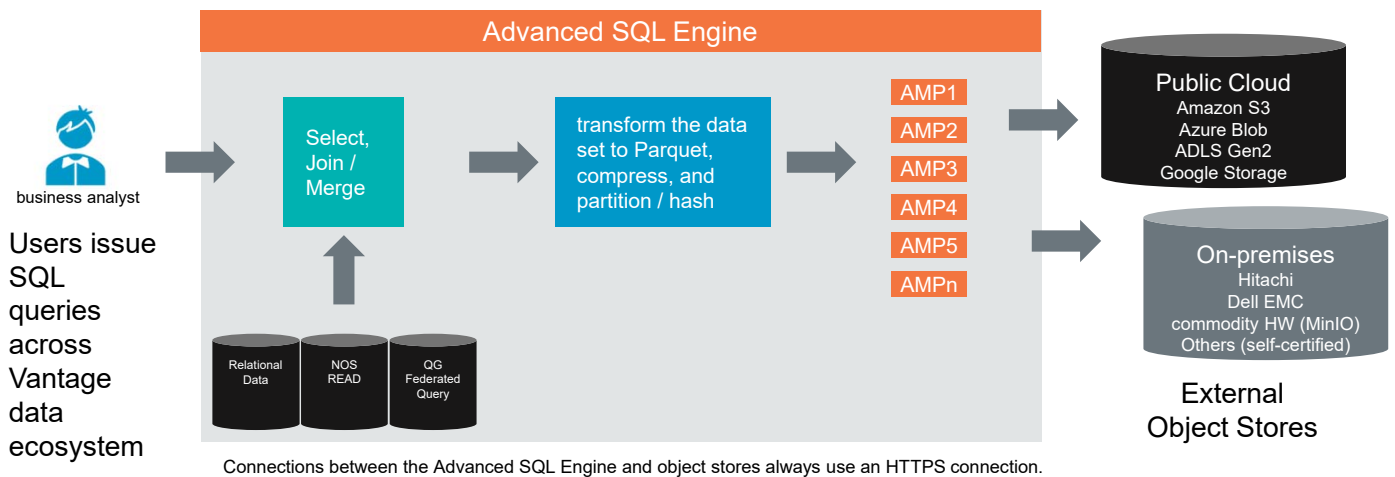
Alternative approaches:

Can use existing cloud utilities to write back to object stores

Can use third party utilities with JDBC connectivity to write to object stores

WRITE_NOS High Level Overview

• How Does it Work?



WRITE_NOS does this by:

- Automatically distributing data across multiple objects subject to a maximum object size of 16 MB, enabling AMP parallelism on both write and read operations
- Automatically converting the data to uncompressed Parquet format unless Snappy compression is specified by the user
- Enabling the user to specify one or more columns in the source table as partition attributes in the target object store, thereby facilitating more relevant path pattern filtering opportunities
- Enabling automatic creation and facilitate updating of manifest files with all objects created during the copy process
- Enabling lossless data conversions—any data copied to the object store can be read back using Native Object Store and will look exactly as it originally was

Let's look at how WRITE_NOS works from a high-level perspective. Users can issue SQL requests across the whole Vantage eco system. I could, for example, select data from an external system via query grid, join that to some other data that's internal in the system, and write that data out to the public cloud or on-premises, object stores. I could also issue a READ_NOS and write out the data, say in a different format to the public cloud. Or I just select some relation data and write that out. So, there's a lot of different options for selecting the data, and then it's transformed to Parquet, and, if specified, compression is applied. The data can be hashed distributed, and optionally sorted within partitions. We'll talk about the partition and hash options later in this module. And it will determine how to put that data out to the object store. We have a lot of flexibility here on how we select the data. In this release the only option we have for the data type is Parquet. But if you recall from our previous module Parquet is very efficient.

WRITE_NOS Table Operator

Copying Data To An External Object Store

- Copy data to an external object store by invoking the **WRITE_NOS** table operator

```
SELECT * FROM WRITE_NOS (  
  ON ( <subquery> )  
  USING  
    LOCATION ('/s3/s3.amazonaws.com/bucket/folder/')  
    AUTHORIZATION  
    ({ "access_id": "AKIAZRTV4BMS6EXAMPLE", "access_key": "hIZ9  
o5uqUNQ30vb33hJ7rX+8mNmIB0mXQEXAMPLE" })  
    STOREDAS ('PARQUET')  
  ) AS d;
```

The AUTHORIZATION attribute supports two methods for providing credentials to the external object store target: it can be passed using the access key id and secret password using this JSON format: AUTHORIZATION ({ "ACCESS_ID": "YOUR-ACCESS-KEY-ID", "ACCESS_KEY": "YOUR-SECRET-ACCESS-KEY" })

(as shown in the example above); or if authorization object have been created, it can be passed the authorization object name using

AUTHORIZATION (AUTH-OBJECT-NAME)

Let's look at the minimum that's required to issue a write_nos request. To copy data to an external object store we use this write_nos table operator. The first thing I do in my query is I say: SELECT * FROM WRITE_NOS.

Now what gets returned is the nodeid, ampid, sequence, object name, object size, and record count. So, this information is returned to show you what data was written out to the object store.

Summary

Now that you have completed this course, you will be able to:

- Define and describe the Native Object Store (NOS) feature and the capabilities it offers
- List and describe some use cases for NOS
- Describe READ_NOS, WRITE_NOS



Module 10: Native Object Store Bring Up JupyterHub

teradata.

Let's now do the lab together



Thank you.

teradata.

©2023 Teradata