Teradata Query Performance Tuning

The DWHPro Guide

This work is subject to copyright. All rights are reserved by the authors, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

Trademarked names, logos, and images may appear in this book. Rather than use a trademark symbol with every occurrence of the trademarked name, logo, or image, we use the names, logos, and images only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark. The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether they are subject to proprietary rights.

While the advice and information in this book are believed to be true and accurate at the publication date, the authors cannot accept any legal responsibility for any errors or omissions that may be made. The authors make no warranty, express or implied, regarding the material contained herein.

Copyright © 2020 Roland Wenzlofsky & Artemios Vogiatzis
All rights reserved
ISBN-13: **979-8-6748-5395-4**

To my wife Marianela,
my children Christian, Melany and Leah,
and my parents.
Roland.

To Vasiliki, for all and even more!
Artemios.

Chapter 1: Introduction

"There is nothing so useless as doing efficiently something that should not have been done at all."
– P. Drucker

Data processing is performed for millennia. *Manual processing* of large data volumes has repeatedly proven a repetitive, boring, slow, and error-prone task for humans.

The 1890 United States Census addressed twice as many questions as the one of 1880 and to a finer granularity of household members rather than per household. Yet, it took four years less to complete and saved some five million US dollars in processing costs at that time. Thanks to the inventions of Herman Hollerith, such as the punch card depicted in Figure 1, this was the first successful demonstration of *automatic data processing*.

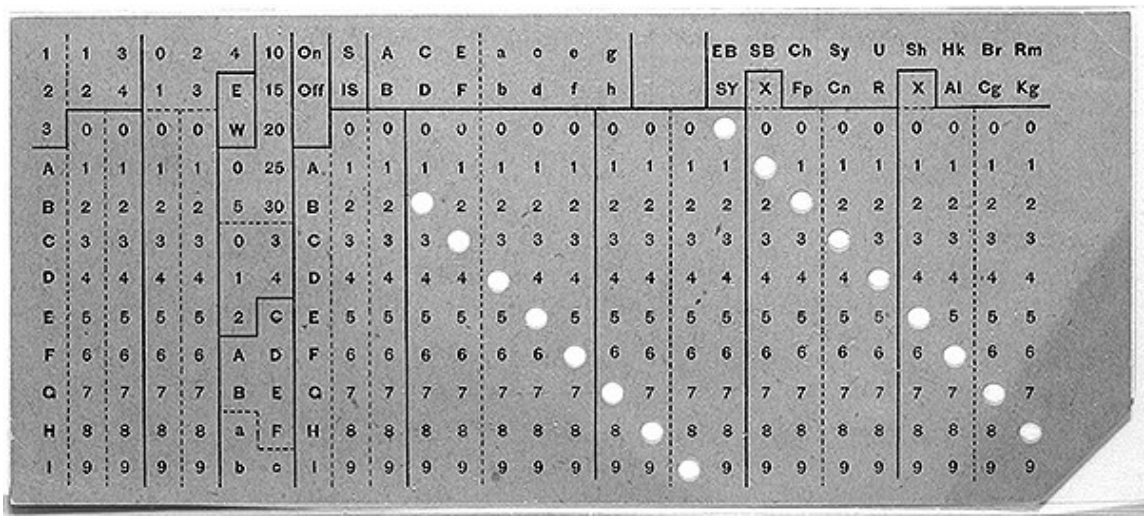

Figure 1 A paper-based Hollerith punch card

The computer technology evolution in the 1900's allowed not only to perform accurate calculations at unpreceded speeds but also to store and retrieve data in digital form. Advancements in permanent computer storage made the punch cards obsolete, entering the era of *computerized (*or *electronic) data processing*.

The more the available data, the more the questions we would like to answer and in more detail. The increase in volume pushes the data down the computer memory hierarchy, away from the processor, to allow for larger storage capacity at the expense of slower access time, as depicted in Figure 2.

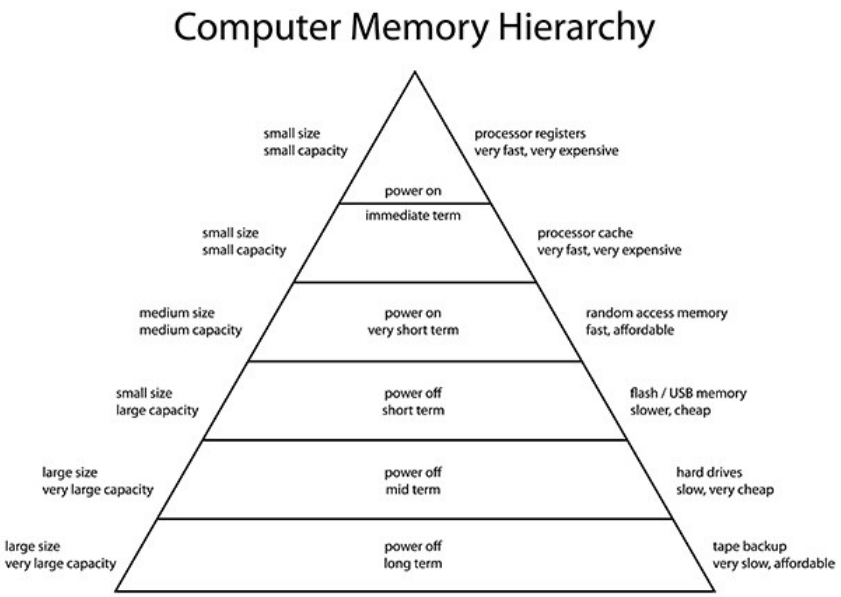

Figure 2 Computer memory hierarchy with the processor on the top

The increase in data velocity (i.e., how often new data arrive in the processing system) and data variety (i.e., what different pieces of information are recorded) drive the need to organize data storage to reduce processing complexity. Databases, as logical data storages, and database management systems (DBMS) emerged in the 1970's. SQL became the standard language to express complex questions (i.e., queries) towards a DBMS and retrieve answers compiled from the stored data.

SQL query performance tuning refers to adjusting the parameters, at SQL and DBMS level, towards optimizing identified metrics. Common performance tuning (or *performance optimization*) targets are query run time, number of input/output (I/O) operations, memory consumption, and utilization of processing resources.

Modern enterprises make informed and timely decisions about their operation and strategy based on data-supported evidence. Data warehouses have evolved as the central repository of enterprise-wide data from dispersed sources, capable of storing current and historical data. They are the single authoritative source for creating analytical reports across the whole enterprise. In this environment, query performance tuning is not a nice-to-have luxury but a critical success factor in keeping up with the global competition.

In this book, we share over 25 years of knowledge and experience in SQL query performance tuning with a focus on the Teradata DBMS and data warehouses. Our ambition is to describe in a simple and comprehensible way all the information that must be considered when one is called to improve anything from a single badly-performing query up to the whole data warehouse workload of overnight batch processes failing to complete on time.

The book comprises six more chapters, which follow the lifecycle of a data warehouse:

- Chapter 2 serves as a refresher for Teradata DBMS fundamentals, including those architectural elements that make it an ideal choice for a performance-sensitive environment.
- Chapter 3 focuses on tuning the design of a data warehouse; the design decisions are the hardest to tune once the system becomes operational and full of data.
- Chapter 4 deals with tuning the import of data into the data warehouse. These operations are typically bulk in nature and require special treatment to execute in the most time- and resource-efficient way.
- Chapter 5 dives into the operational status of the data warehouse, starting with tuning the retrieval of data from the storage area for a query. The performance impact of data distribution skew, partitions, and indices are discussed.
- Chapter 6 continues with tuning the execution plans for multiple queries that concurrently retrieve massive volumes of data. The query execution plan steps are explained along with their performance impact and optimization strategies for distributing work to different processors. Additionally, we discuss how the use of specific SQL language constructs can affect the performance of a query.
- Chapter 7 departs from query tuning towards business processes and human bias. Often, these are the hardest to tune. We expand the discussion on how the performance tuning techniques can be re-used in cloud-based Teradata and beyond.

Performance tuning is a challenging and complex task, both art and science. One needs to have a deep understanding of computer architecture, database design, data modeling, parallel systems, set theory, and business informatics.

We are far from being experts in all these, and there is always something new to learn. We will be grateful to receive your feedback and suggestions for content improvement at tuning.book@dwhpro.com!

Chapter 2: The approach of Teradata

"If you know the system well enough, you can do things that aren't supposed to be possible."
– Linus Torvalds

The purest abstraction of a database is a file stored on a computer disk. Then, a (relational) database management system is a piece of software that allows one to read portions of the file using the SQL language. This over-simplification serves as a starting point of how (relational) databases and DBMS are built.

The Teradata Corporation defines Teradata Database (renamed to Vantage) as "*an information repository supported by tools and utilities that make it a complete and active relational database management system.*" The Teradata Database follows a parallel processing architecture.

Parallel database architectures

Parallel database architectures have been designed for over 40 years now. They have evolved from *shared memory architecture* to *shared disk architecture* to *shared-nothing architecture*.

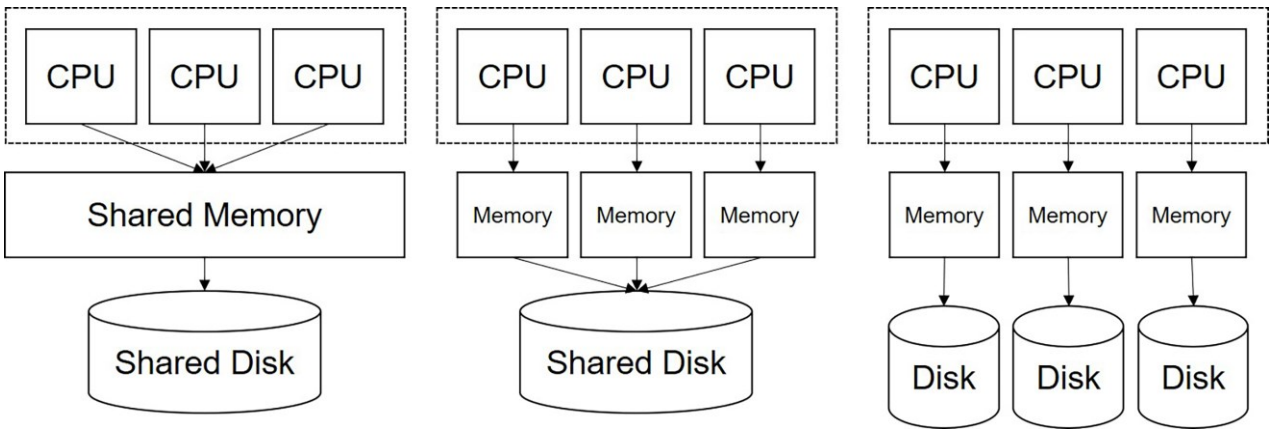

Figure 3 Evolution of parallel database architecture

In a *shared memory architecture*, the operating system allows multiple processors to access a common, shared main memory and storage space. This architecture favors processing-intensive applications. It nicely scales with the number of processors up to where access to main memory becomes a bottleneck.

The *shared disk architecture* allows a dedicated memory for each processor and shares only the storage space. Then, two communication channels are preferred to avoid delay

processing. One channel is dedicated to *inter-processor communication* and the second one to access the shared storage space.

A *shared-nothing architecture* offers a dedicated memory and storage for each processor. Then, only one communication channel is necessary for inter-processor communication to coordinate the work. As depicted in Figure 3, shared-nothing architecture reduces the total storage available per processor. Still, it allows for faster access to the stored information since no processors are competing for access. A *shared-nothing architecture* is an excellent choice for basing a DBMS. It offers tremendous performance gains, provided that the DBMS software balances the work assignment and space consumption among the processors. This is why the Teradata Database is designed honoring the *shared-nothing architecture* to its extreme.

Teradata Database infrastructure components

Hardware

There are many possibilities to deploy a Teradata Database: on-premises commodity or Teradata hardware, public or Teradata cloud, and hybrid clouds (combinations of the previous). In the case of Teradata hardware (on-premises or cloud), *Teradata cabinets* host one or more cliques of *nodes*, *disk arrays* for storage, and other IT components. One of the clique nodes can be assigned as a *hot standby node* (HSN). This is an idle node that takes over immediately the work if a node crash occurs, avoiding any performance degradation. The crashed node is assigned as an HSN as soon as it is recovered.

Teradata software

A Teradata Database *node* is a collection of database components that runs on top of an operating system (e.g., Linux). The latter controls all the underlying hardware and network resources.

The *Teradata Parallel Database Extension* (PDE) is a software interface layer of a node that lies between the operating system and the Teradata Database. PDE provides the parallel capabilities to the Teradata Database. The Teradata Database management system provides the Teradata Database File System, the Parsing Engine, the Messaging Layer, and the Access Module Processor components.

The *Teradata Database File System* (TDFS) is a software layer that implements a special-purpose file system. TDFS isolates the Teradata Database from the underlying operating system file system abstractions and hardware storage dependencies. The TDFS elementary unit of storage is a *physical row*, a data structure comprising two parts: *row header* and *stored data*. The data structure is generic enough to hold the contents of several kinds of data objects used by the Teradata Database.

The *Parsing Engine* (PE) receives an SQL query, devises a cost-efficient execution plan, and splits the work to the available AMP's. Chapters 5 and 6 are devoted to how one can tune the PE in selecting the best execution plan.

The *Messaging Layer* (BYNET, an acronym for "Banyan network") lies between the PE's and the AMP's. The BYNET is a combination of hardware and software that enables high-speed communication inside and between the nodes. When only one node is available, we have a Symmetric Multiprocessing (SMP) database system. The name "boardless BYNET" is used in these cases to describe that no BYNET hardware component is installed. When two or more nodes are available, we have a Massively Parallel Processing (MPP) database system. Then, the software component is the BYNET driver to interface the PDE software with the BYNET hardware.

There are typically two BYNET available per system, BYNET 0 and BYNET 1, for performance and fault-tolerance reasons. As long as both BYNET operate without errors, they are used simultaneously to increase performance. If one BYNET fails, the second one allows the system to continue its operation. We note that the historical name "BYNET" is retained, although InfiniBand replaced the proprietary switching fabric in modern Teradata hardware. In public cloud implementations of Teradata (e.g., Microsoft Azure and Amazon Web Services), BYNET is virtualized. The available public cloud's network fabric is used instead to realize the Messaging Layer.

The *Access Module Processor* (AMP) is the database execution engine. A node has multiple AMP's. PE's and AMP's are examples of a node's virtual processes (VProc).

Each AMP has one virtual disk (VDisk), which is *exclusive storage* and works *independently* on its own set of data. An AMP VDisk is the collection of PDisk's assigned to a specific AMP. An AMP PDisk is nothing more than a slice of Logical Unit Numbers (LUN) built atop a group of disks (moving-head hard disk or solid-state drives) organized in RAID arrays or farms.

An AMP assigns the PE-dispatched work to one or more of its available *Worker Tasks*. The AMP Worker Tasks (AWT) are organized in groups, each devoted to one task type. This allows multiple task types to execute within an AMP at any time instance.

The *AMP Message Queue* buffers the PE requests (i.e., dispatched work tasks) whenever all the AWT's that can handle the specific task type are occupied. A *flow control mode* is activated when the Message Queue becomes full. In this case, the AMP notifies the PE via the BYNET that it does not accept new requests. In turn, the PE notifies via the BYNET all other AMP's that received the other dispatched work parts to hold on for a while; the PE then retries to re-assign the work to all the AMP's, including the overloaded one, hoping the latter can now accept it. This cycle repeats until the request is accepted by all AMP's and the PE doubles its waiting (back off) time after each failed attempt.

At this point, we have revisited the infrastructure components of a Teradata Database system and how they are brought together to realize a shared-nothing architecture. Figure 4 depicts a simplified view of the Teradata Database infrastructure components hierarchy.

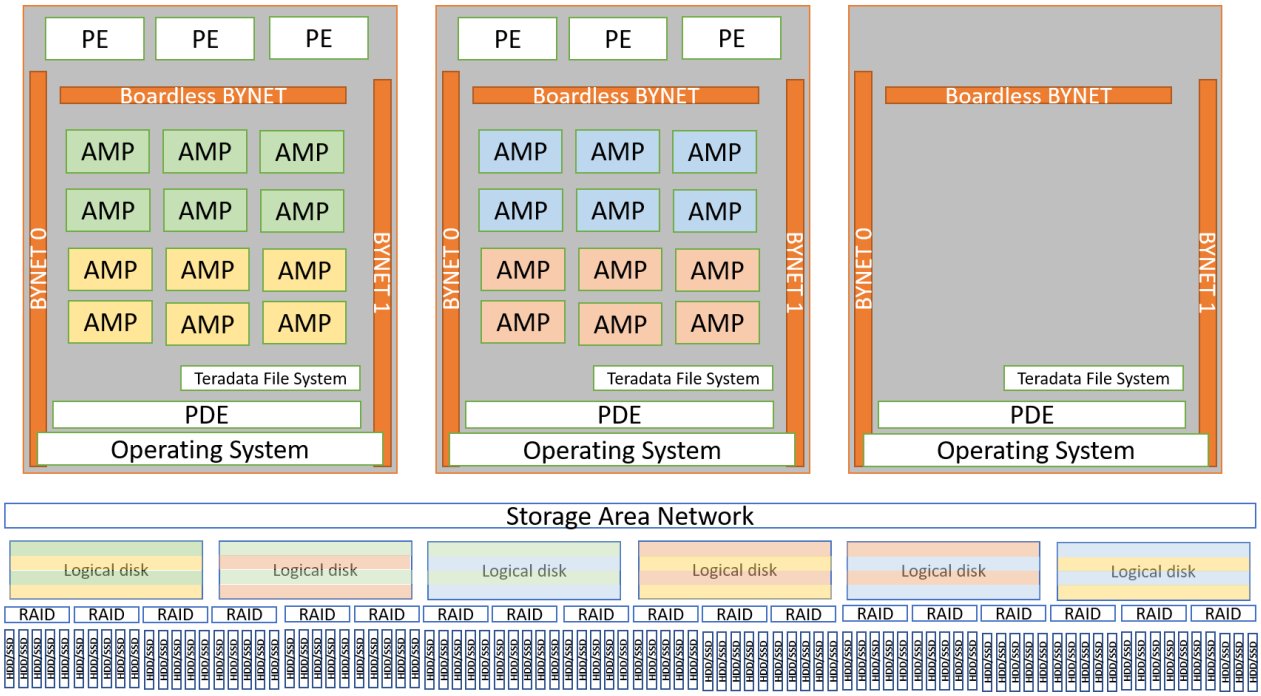

Figure 4 A Teradata Database MPP example with a clique of two nodes and one HSN interconnected via BYNET and accessing a disk farm. The VDisk of each group of AMP's spans a set of stripes (PDisk) of multiple logical disks

Teradata objects

A database management system (DBMS) is to a database of what an operating system is to a computer. Teradata DBMS supports a hierarchy of objects having as its root the DBC system user. DBC stands for "Database Computer" and stems from the first Teradata system named "DBC 1012" dating back to 1988. The International System of Units (SI) prefix "tera" denotes 10 to the power of 12.

A Teradata user is nothing more than a database with a password. The DBC user can then create other databases, tables (row- and column-based data); views on subsets of data; macros (predefined, stored SQL statements); triggers associated with tables; stored procedures; user-defined functions (provide functionality additional to SQL), join and hash indices, and permanent journals (tables storing recovery snapshots).

Data storage and retrieval

Every database name and object is assigned a globally-unique numeric identifier (ID), consuming numbers from the `DBC.Next` dictionary table. An object-to-number association is called the Unique Value of the object.

The database objects are stored in one or more TDFS physical rows. The physical row header includes the Unique Value of the object and the stored data part, which holds the contents of the object. When the size of the object contents exceeds a physical row's maximum size, TDFS creates as many additional physical rows as needed to accommodate the object contents in multiple physical rows. The row header of all these physical rows includes the same object Unique Value.

Teradata hashing and AMP selection

The Teradata DBMS employs a proprietary, parallel hashing algorithm to distribute the storage (and subsequently, retrieval) of the physical rows across the available AMP VDisk's. This hashing algorithm creates a fixed-length output of 32 bits (the hash) for any input length. The first 16 bits (20 in newer Teradata versions) are the Destination Selection Word (DSW) or bucket. The bucket value is then fed to a *hash map* function of 2^{16} (or 2^{20}) entries, which return a number between 0 and the total number of AMP's in the system. This is the AMP that will be responsible for the specific physical row.

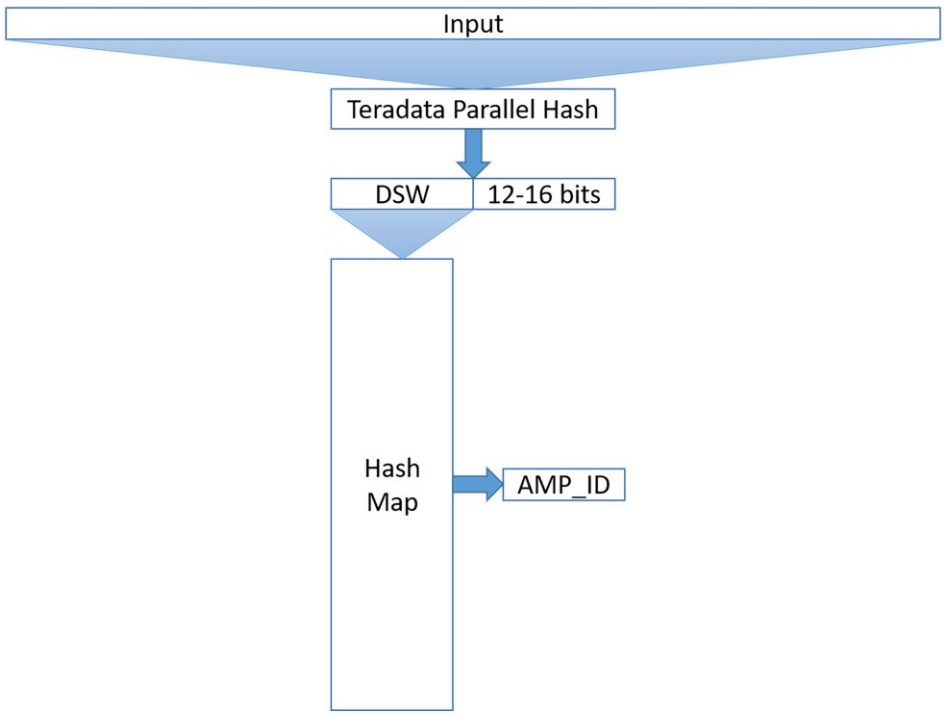

Figure 5 AMP selection via Teradata hashing

The Teradata parallel hashing approach provides critical properties for a high-performance system:

- The hashing algorithm has excellent diffusion characteristics: inputs exhibiting spatial locality (e.g., when a sequential transaction number is used to record transactions) will be spread across all available AMP's rather than concentrating in only a couple of them.
- The hash map allows reusing the same good hashing algorithm in systems with a different number of AMP's, without a need for further configuration or any performance reduction. This is also true if a system upgrade occurs, where the number of available AMP's increases. No change to the hashing algorithm is required: it suffices to define a new hash map that the database objects can utilize.
- The hashing algorithm also ensures that the same input (i.e., data types and values) will always produce the same hash. This is independent of their order, as long as the data types match (e.g., the hash of the (5,3) pair of integers is the same as the one of (3,5)).
- The hashing approach ensures that all physical rows with the same hash will always end up in the same AMP and that the probability of two different inputs producing the same hash (i.e., a hash collision) is small.
- Finally, the hashing approach identifies on-the-fly the AMP that should be queried using only inputs already available in memory; the system needs not to perform expensive input-output operations to record and recall the storage locations of existing physical rows and their AMP.

Data residence in an AMP VDisk

TDFS divides an AMP VDisk into (logical) *cylinders* and each cylinder into (logical) *sectors*. Both terms originate in the era of moving-head (mechanical) disks. Each sector contains *data blocks*, each storing a set of *physical rows*.

For simplicity, we use the example of an AMP storing the logical row of a database table into a VDisk physical row in the following paragraphs. The table will be identified by the database object Unique Value TABLE_ID and its set of physical rows by their ROW_ID (or the ROWHASH thereof). Chapters 3, 4, and 5 will explore how a ROW_ID is defined for different types of tables. Other database objects are similarly mapped to physical rows, using the respective object Unique Value and physical row identifiers.

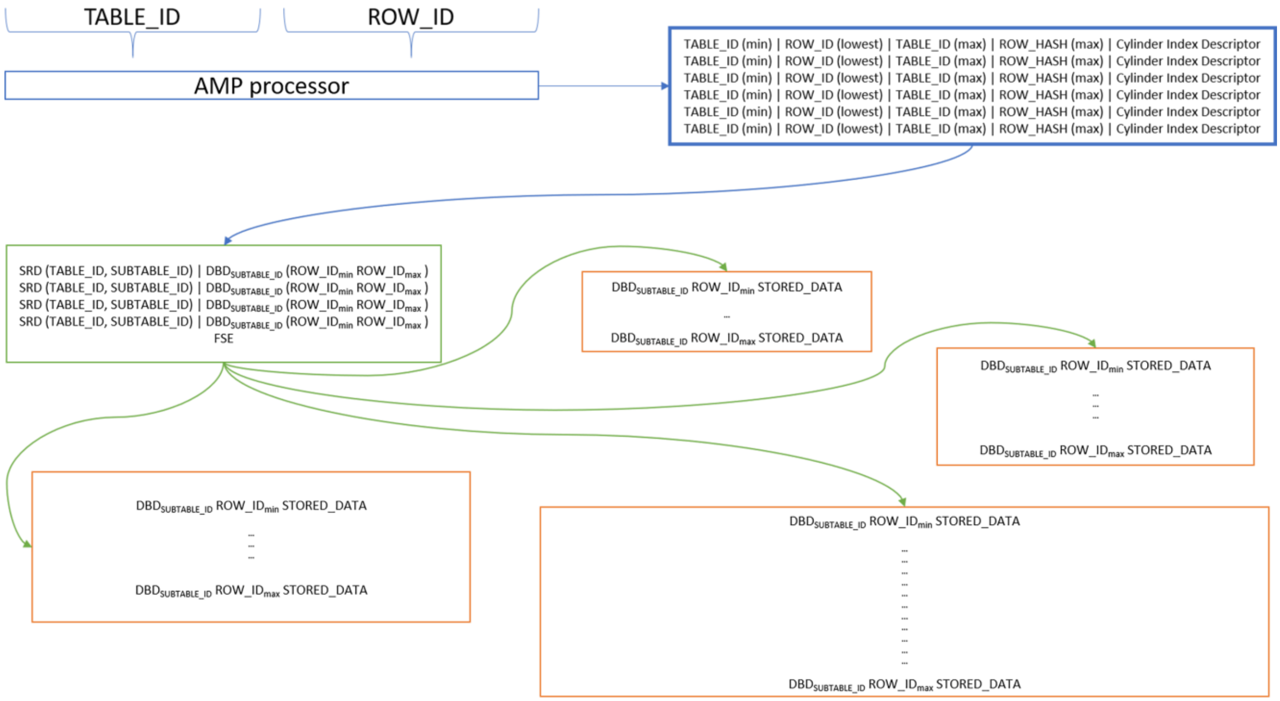

Figure 6 Accessing a physical row (identified by ROW_ID) of a database table (TABLE_ID) through the memory-resident Master Index (top right), the disk-resident Cylinder Index (middle left), and the disk-resident Data Blocks (bottom right)

Master Index

The AMP maintains a *Master Index*, which is a memory-resident TDFS data structure. The Master Index contains *Cylinder Index Descriptor* (CID) entries that point to the *Cylinder Index* area of a VDisk cylinder. The CID entries are sorted by the lowest TABLE_ID, lowest ROW_ID of the lowest TABLE_ID, highest TABLE_ID, and highest ROWHASH. This allows the AMP to quickly search in memory for a given TABLE_ID and ROW_ID or ROWHASH and reduce the disk access to only those cylinders containing the data blocks to fulfill the request.

Cylinder Index

The *Cylinder Index* is a disk-resident TDFS data structure at the beginning of each cylinder. It provides information about the location of the cylinder's data blocks and the list of any free sectors (Free Segment Entry, FSE). The Cylinder Index entries are sorted by TABLE_ID and ROW_ID, as every data block contains information for one table; still, a table can span multiple data blocks. Each entry provides information about the table (TABLE_ID) stored in the data block, the lowest ROW_ID, and highest ROWHASH, and the starting sector and the number of sectors in the data block.

The Cylinder Index logically groups all data blocks of a table under one SRD (Sub-table Reference Descriptor) entry for fast reference. Each SRD contains one more DBD (Data Block Descriptor) for the specific part of the stored table (i.e., a sub-table). Sometimes, it is possible to hold the Cylinder Index in the cache memory of a node. Then, an AMP again reduces the disk access to only those data blocks containing the physical rows to fulfill the request. The data block pointers in the DBD are sorted by ROW_ID to allow a fast binary search for identifying the one that will store the physical row.

Data Block

A *data block* is a disk-resident file system data structure holding physical rows of a (sub-) table. Three important properties of a data block are that the size of a data block is always aligned to a specific boundary (e.g., 4 KB or 512 bytes); the block size within a table is not constant, and TDFS can dynamically adjust the size as required. A physical row is always fully contained within a single data block. TDFS splits the data block when a new or changed physical row does not entirely fit within the data block. A *data block header* describes the physical rows in the block, the table these rows belong to, and the space available in the block.

The data block size parameter `DATABLOCKSIZE` determines the count of sectors per data block. On the one hand, a smaller data block size is beneficial when a query is selective enough to retrieve only one or a few physical rows per data block. However, if large physical rows are to be stored, Teradata must split the data blocks more often to accommodate the larger rows. On the other hand, a larger data block size allows transferring many physical rows to the AMP memory at once. This is beneficial when a query is not very selective and, thus, many rows of the data block will be used for further processing. To put this into perspective, Teradata 13.10 increased the cylinder size from 1.9 MB to 11.3 MB, and Teradata 14.10 increased the data block size from 127.5 KB to 1 MB. Teradata 16.00 increased the physical row size from 64 KB to 1 MB.

Physical row contents

Relational databases have evolved over the concept of organizing information records as logical rows of tables, where each piece of information is represented in one table column. All information related to one record could be stored next to each other, speeding up the retrieval from the disk storage. As more detailed information records are created, the width of the table rows (i.e., the number of different columns) increases. Often, the queries need to work with a small set of columns per table row. Yet, the whole table row must be transferred from the AMP disk to AMP memory with all the defined columns. This is suboptimal from a time and AMP memory consumption perspective.

The *column-oriented databases* address this challenge by organizing physical row storage around columns rather than rows. The assumption is that only a few columns will be needed and, thus, transferred from disk storage, contributing to higher performance. Teradata offers the columnar feature since version 14.0. When a column-oriented database object is

used, the physical rows structures inside the data blocks store sets of object columns rather than object rows. Teradata transparently handles the conversion from column- to row-orientation before further processing a query. The latter always occurs using a row-orientation representation.

Teradata Database Management System

This section is devoted to a selection of Teradata DBMS services that play an essential role in performance and performance tuning.

Fallback AMP hash maps

Every physical row is handled by one AMP and stored in its exclusive VDisk. In the unforeseen case of an AMP crash, the rows become inaccessible. This is until an HSN AMP takes over or the original AMP is recovered.

Teradata implements a *fallback protection strategy* to perform better during the outage of such improbable cases. This is realized using a *fallback hash map* for AMP selection besides the main (primary) one. The fallback hash map function returns an AMP number different from the one of the primary hash map function. Also, the strategy dictates that AMP's are organized into clusters. Here, the system becomes inaccessible only if two AMP's *within* the same cluster crash.

Last but not least, the primary and fallback AMP's should lie on different nodes. Then, it would take two AMP crashes on two different nodes before the system becomes inaccessible. Hence, at the expense of *doubling the storage* to place each physical row on two different VDisk's, one can retain the system performance even in events of complete nodes failures. The fallback protection strategy was optional in the past due to space considerations. It is mandatory starting with Teradata 16.10.

Teradata Intelligent Memory technology

Moving data blocks between the disk storage and the memory hierarchy is a costly operation. Any reduction (or elimination) of disk access time overhead can positively affect performance. In-memory databases offer such a promise, but the amount of data they can hold is limited. The advent of SSD in favor of moving-head disks significantly reduces but does not eliminate this overhead.

The Teradata Intelligent Memory technology was introduced in version 14.10. It continually measures the relative data block use frequency and classifies data into four categories: very hot, hot, warm, and cold. Further, it devotes a portion of AMP's shared memory to store the very-hot temperature data. It aims to keep the most used data blocks in memory for as much time as possible.

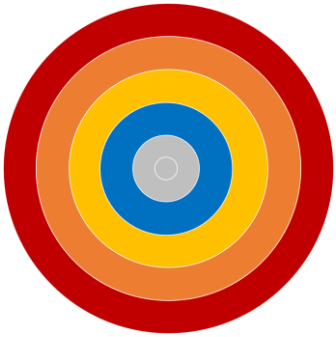

Figure 7 Storing data blocks in sectors from outer (very hot data) to inner (cold data) moving-head disk cylinders based on data temperature

In systems with moving-head disks, the Teradata Intelligent Memory further organizes data block storage in cylinders based on the data temperature, as depicted in Figure 7: higher-temperature data blocks are placed in outer cylinders for faster access. Last but not least, Teradata Intelligent Memory technology transparently handles, whenever necessary, the conversion between in-memory row orientation and on-disk column orientation of physical rows.

Teradata transactions and journals

A transaction is a single unit of work that must be done consistently. Proper support for transactions is one of the cornerstone services a DBMS offers. The DBMS must ensure that a database transaction is ACID: Atomic (completes in full or does not affect), Consistent (respects defined database constraints), Isolated (does not impact other transactions), and Durable (written in permanent storage). Teradata supports different types of transactions; Chapter 6 discusses how they can be utilized in performance tuning. For now, it suffices to consider a *Teradata transaction* as the elementary unit of processing that interfaces with the database contents.

A *Teradata Permanent Journal* is a table that can be specified at a database- or database-table level to capture the contents of the whole database or the specific table, respectively, before and after a Teradata transaction. The journal entries can then be utilized to roll back (before) or roll forward (after) the effects of a transaction (e.g., recover deleted table rows or undo a table structural change). A permanent journal offers an additional layer of protection against data loss. It is utilized to shift in time or skip the full table backup operations at the expense of keeping two or three copies per affected physical row. Hence, the benefits of a permanent journal must be carefully evaluated on a case by case basis.

A *Teradata Transient Journal* is a dictionary table maintained across the whole system used to protect *transactions* against system failures and *deadlocks* (cf. next section). Each AMP manages its local Transient Journal, where the table rows are copied before changed by a transaction. Should one transaction fail to complete (i.e., *aborts*), the Transient Journal is used to roll back and restore the contents of the affected table rows. When the

transaction completes, the related rows are removed from the Transient Journal. Despite being short-lived compared to a Permanent Journal, the Transient Journal can significantly affect performance: a transaction causing an update on billions of rows requires an equal space for journal entries. The Transient Journal cannot be manually disabled, or else it would not be possible to ensure the ACID properties of a transaction. Chapter 6 discusses tuning techniques to smoothen its performance impact.

The *Teradata Write-Ahead Logic (WAL)* protocol is one more layer of protection against data loss due to system failures. The protocol dictates that when a data block is changed in memory, it is not stored (written) directly back to permanent storage. Instead, the changed rows are written in a *WAL Log* file. The log file, having a much simpler structure than of a Transient Journal, is periodically written to the permanent storage, e.g., when a transaction finishes execution. The changed data block remains available in memory for the next steps of the transaction or future transactions. The latter can introduce additional changes in the data block contents. If the data block size also changes, it is temporarily written to the *WAL Depot*, a dedicated area of an AMP's VDisk occupying a fixed number of cylinders. If possible, multiple data blocks are written at once to the WAL Depot for performance reasons. At a later, more convenient time, the data block containing all the changes is written back to the permanent storage, and the WAL Log is cleaned from the respective entries. If the data block size did not change, it is written back to its original VDisk location. In the case of system failure, the WAL protocol combines the information from the WAL Log and the WAL Depot to recover the changed data block.

The WAL protocol can positively affect the performance of the transactions. This is because the data block remains in memory, allowing for multiple changes by multiple transactions. At the expense of occasional WAL Depot writes, a series of expensive operations to write the data block to the permanent storage and subsequently read it back to AMP memory is avoided.

Teradata locks and deadlock handling

The *Teradata Lock Manager* is the component that realizes the database transaction isolation (the "I" in ACID) through *database object locks*. The objects are locked when a transaction starts and unlocked once the transaction finishes. The less strict the lock, the fewer database objects must be isolated from other transactions. Then, the more opportunities for the database to execute multiple transactions simultaneously (i.e., *concurrently*) rather than one at a time (i.e., *sequentially).*

The Teradata Lock Manager supports four granularities and four levels, namely:

- Database: all database objects are locked.
- Table: all table contents are locked.
- View: all tables involved in the view are locked.
- Row hash: all table rows with the specific row hash are locked.

- Exclusive: no other transaction may access by any means the locked objects.
- Write: no other transaction may write to the locked object. Other transactions may read from the locked object but might not read the newly-written information.
- Read: other transactions may also read from the locked object, but they cannot modify it.
- Access: other transactions may also read from the locked object. They may also modify it (i.e., an access lock is granted even when another transaction was already granted a write lock).

In contrast with many other DBMS, Teradata takes care automatically and transparently of all necessary locks needed by a transaction. It is sophisticated enough to obtain the most fine-grained lock possible and upgrade to a stricter one only when deemed necessary. Yet, the Teradata Lock manager has a limited amount of resources to lock on row hash granularity. An attempt to lock a huge number of row hashes may cause the transaction to abort and roll back any changes up to the point of abort using the information recorded in the Transient Journal.

The Teradata Lock Manager is not a silver bullet for increasing concurrency. Little it can offer when a poorly-designed transaction takes longer to execute and, thus, locks the objects for long periods, unnecessarily blocking access to other transactions. Chapters 4-6 extensively discuss how to design high-performant transactions.

The concurrent execution of multiple transactions significantly increases performance compared to sequential execution but can also result in a *deadlock*. This is a system state where the execution of two or more transactions cannot proceed further because the set of obtained and necessary locks by each transaction conflicts with the other's ones. Among the available approaches to handle deadlocks, Teradata chooses *detection*, i.e., deadlocks may occur in the first place and then resolved. Deadlocks in Teradata can occur within an AMP (i.e., a *local deadlock*) or across multiple ones (i.e., a *global deadlock*) and even with row-hash locks.

Teradata uses a queueing strategy to serialize locking requests and reduce the deadlock occurrence probability. Up to Teradata version 14.10, this strategy is called "*pseudo table locks*". Starting with version 15.10, two additional locks are available: *table partition lock*, where all table partition rows are locked, and *table partition range lock*, where all rows of a subset (range) of the table's partitions are locked. Table and table partition locking are then called "*proxy locking*" while "*pseudo table locking*" is only retained for row-hash level locking. Despite the name changes, the deadlock handling strategy remains the same.

The queueing strategy works as follows. First, the TABLE_ID is passed through the Teradata-proprietary parallel hashing algorithm, as in the case of a physical row. The AMP selection process described earlier derives then an *AMP gatekeeper*, i.e., an AMP that acts as the gatekeeper for the locks on this specific table despite how the rows of the table are spread among the available AMP's. A transaction must first obtain the pseudo table or

proxy lock from the AMP gatekeeper. Once this lock is obtained, the transaction may proceed and obtain the actual table locks. This serialization enforces an order-in-time of the transactions competing for accessing a table.

The queueing strategy does not eliminate deadlock occurrences. For example, consider the case where one transaction obtained a write lock on table X and queues at another AMP gatekeeper for a write lock on table Y. Concurrently, a second transaction obtained the write lock for table Y and queues at the AMP gatekeeper for table X. Then, neither can proceed until it obtains the missing lock from the respective AMP gatekeeper, and neither can return (release) the obtained lock until it finishes execution.

To cope with the improbable-but-possible case of a deadlock, Teradata searches for local deadlocks every 30 seconds and global deadlocks every four minutes. These periods are configurable. One must carefully assess the optimal values for each environment: more frequent searches allow to resolve faster deadlocks and continue processing but put additional stress on the system.

Teradata sessions

The communication with a Teradata Database is performed over logical connections called *sessions*. During a session's lifetime, more than one transaction may be requested. However, they are only sequentially executed, i.e., at any time instance, only one of these transactions is outstanding.

Sessions have a significant impact on performance. On the one hand, the more the transactions grouped in a session, the less the system resources and processing overhead per transaction for maintaining the logical connection with the Teradata Database (i.e., the connection setup and shutdown costs are amortized across all the transactions of the session). On the other hand, the more the transactions in a session, the less the opportunities to concurrently execute more transactions.

Teradata workloads

A Teradata workload is a group of requests with common characteristics. Request grouping allows better classification and monitoring of how the system resources are used. Then, resource allocation can be prioritized at the workload level, enforcing the desired limits on how many, how long, and how often specific resources can be devoted to a specific group of requests.

The workload classification considers five groups of request characteristics:

1. Request source, including among others, the username and the client IP address.
2. Request target, i.e., the database object(s) included in the request (e.g., database, table, and macro). Starting with Teradata version 14, sub-criteria can be considered as well, such as if a full-table scan (FTS) is involved and the number of involved rows.

3. Query characteristics, e.g., the type of statement and the number of involved AMP's.
4. Utility usage, i.e., if a Teradata load utility is involved (e.g., FastLoad or Backup).
5. A query band, i.e., a set of name-value pairs defined as metadata wrapped around the request to identify the originating source of a query.

The exact definition of workloads is system-dependent. Typically, workloads are defined for single-AMP requests, load utilities, ad hoc queries, tactical queries, and strategic queries. More fine-grained groups can be defined as necessary. Starting with Teradata version 14, up to 250 workloads can be defined, including one "WD-Default" default for those requests that match no defined workload. Starting with Teradata version 13.10, a request must match *all* the criteria of a group (AND logic) to be assigned as a member of the workload. In earlier versions, it suffices one criterion to match (OR logic).

Workload management

Modern Teradata systems provide some form of *workload management* to help prioritize system resource allocation among the defined workloads. The *Teradata Integrated Workload Management (TIWM)* is available with every system; an additional license may be necessary for the *Teradata Active System Management (TASM)*, which offers advanced functionality.

Teradata Priority Scheduler and Tiers

The *Teradata Priority Scheduler* is a software component responsible for assigning system resources to active requests. It is part of both TIWM and TASM. In Teradata version 14, the Teradata Priority Scheduler is tightly coupled with the Completely Fair Scheduler (CFS) used in the Linux kernel for process scheduling at the operating system level.

The Teradata Priority Scheduler implements priorities using hierarchies, similarly to CFS. The hierarchy consists of six tiers, as depicted in Figure 8:

1. TDAT
2. User, default, and system
3. One (TIWM) or multiple (TASM) Virtual Partitions
4. Tactical workload
5. Zero or more Service Level Goal (SLG) workload (TASM-only)
6. Timeshare workload

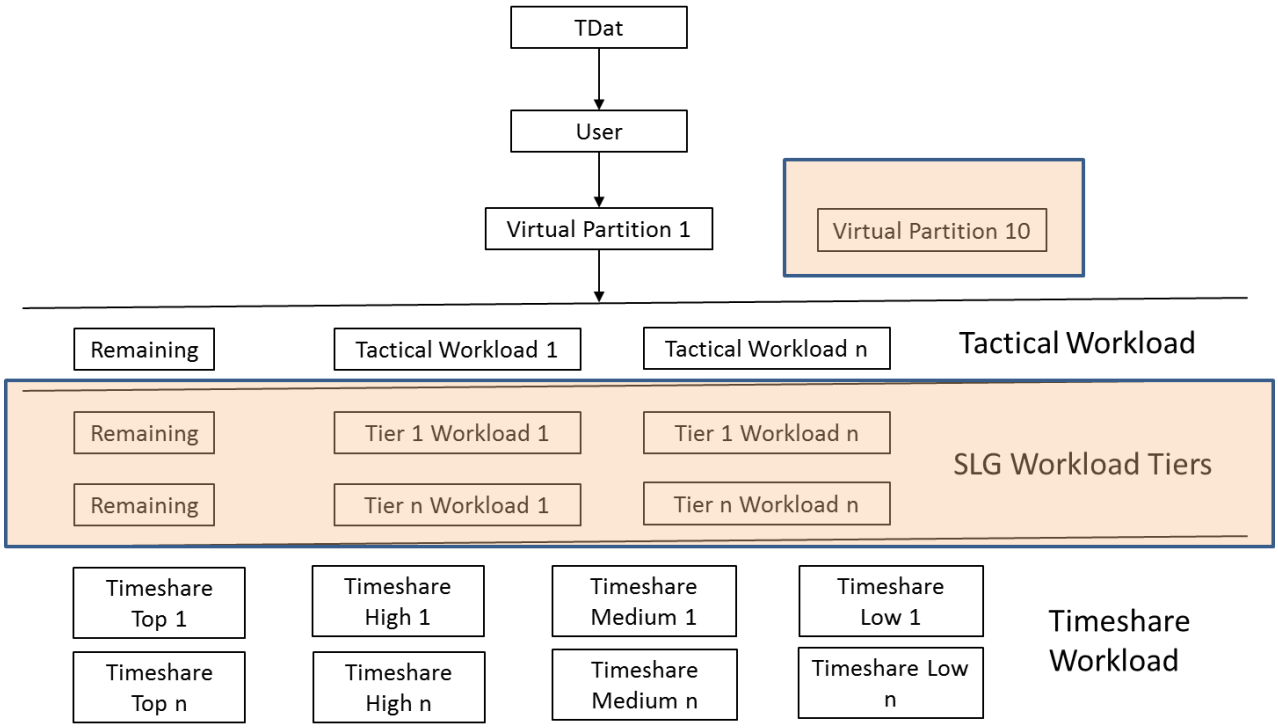

Figure 8 Teradata Priority Scheduler hierarchies

The system resource allocation policy is straightforward: the higher the tier of the workload, the more critical it is considered and allocated the needed resources. Any left resources are then provided to the next (lower) tier. To avoid starvation of the less-critical workloads, a safety margin is applied: at least 5% of the available resources are reserved for the timeshare tier.

The first two tiers include critical workloads for system operation, where the resource availability is guaranteed. Virtual Partitions are available since Teradata version 14 and allow a Teradata system to be shared among different business owners. Then, each business owner gets only a share of the available system resources. The resources are reused by other owners if not consumed. System partitioning is available only with a TASM license.

The *tactical tier* can consume as many resources as needed (except the 5% margin). Tactical workload requests must be carefully assigned. They can get immediate access to system resources (e.g., the AMP processor), interrupt and suspend any other request, even if executing already. They can also use reserved AWT's. In principle, a poorly defined tactical workload request can put the rest of the requests on starvation. To protect against misuse, TIWM and TASM implement *tactical workload exceptions*. Long-running tactical workload requests are automatically moved to a lower-priority tier.

The *Service Level Goal (SLG) tier* is available only in TASM. A relative percentage is defined for each SLG tier workload. This percentage defines the share of the system resources

reaching this tier that the specific workload needs to fulfill its service goal. The share is an *upper limit*; the workload might need less than this; what remains flows down the next tier.

Figure 9 depicts an n-by-two hierarchy of priorities within an SLG tier. Tier 1 defines a 50% share of the resources left after the tactical tier and splits this share into 20% for its workload 1 and 30% for its workload 2. The rest 50% (plus any unused share of the Tier 1 workloads) flows down to Tier 2, and the definition repeats in a cascading fashion. Then, Tier n gets any remaining share and evenly splits it to its workloads, as there is no specific share defined for each.

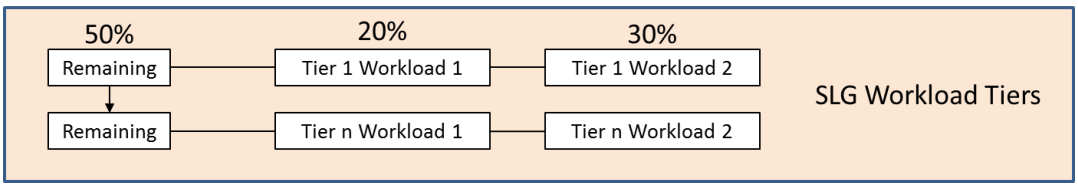

Figure 9 An n-tier SLG, each with two workloads (only tier 1 and n shown)

The more the number of SLG tiers and workloads within, the less stable the resource allocation becomes. The actual share of each workload becomes dependent on more and more workloads that may exhibit different behavior across different executions.

The *timeshare tier* is the lowest in the priority hierarchy and contains four internal tiers: top, high, medium, and low. Each workload is assigned to precisely one of these tiers. Each tier has a fixed relative share of the overall resources available in the timeshare tier: a request in a top workload can consume eight times the resources of a request in a low workload; the ratios for high and low are respectively four and two. In contrast to SLG, all timeshare workloads compete for the same pool of resources without a prioritization logic. Should all timeshare workloads get their fair share of resources, and there is still excess system resource capacity, the system allocates the remaining on an as-need basis.

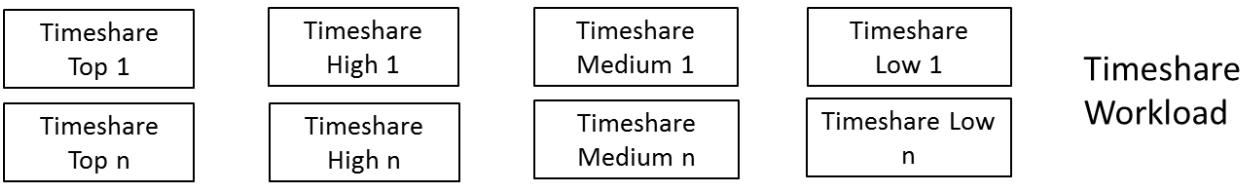

Figure 10 An n-by-four timeshare tier with top, high, medium, and low workloads

Filters and throttles

TIWM and TASM support *filters* and *throttles*, on top of workload classification. *Filters* act as an access control mechanism: they prevent (or at least issue a warning) when specific queries try to execute. For example, a filter might be defined to deny any reporting user

logins during a period that batch processing occurs; the respective queries will then get an error message but will execute as normal outside the defined period.

Throttles are used for rate limiting. Throttle targets are the number of concurrent sessions, requests, or utilities. Throttles aim to reduce resource contention (e.g., memory, AMP processor time, and AWT) on systems with a high level of concurrency, thus, contributing to improved system performance.

Session throttling controls the number of concurrent sessions. When the limit is exceeded, no more sessions can start, i.e., session logins will be rejected. Session limits can be defined at the system level or for specific classification targets (e.g., the user or client IP). Additional classification characteristics are not considered.

Request throttling controls the number of concurrent requests (queries). TIWM and TASM maintain a simple counter to keep track of executing requests. Once the threshold is reached, new requests are entered in a *delay queue* or get immediately rejected. When the counter falls under the threshold, the delay queue is emptied on a First-In-First-Out (FIFO) basis. Request limits can be defined at the system and workload level. System-level throttling is always applied to a request. Workload-specific throttling is applied only if the request is classified under the specific workload based on its characteristics, as described earlier.

Typically, it is easier to define workload throttling limits rather than system ones. Still, system-level throttling is useful for coping with overall system conditions, as discussed in the next section. Throttling of load utilities is often defined as they aggressively consume system resources. Chapter 4 discusses further the topic of load utilities. Throttling of tactical workloads must never be defined, as it defies the reason for defining a tactical request in the first place: retrieve information the fastest possible.

Dynamic workload management

TIWM relies on static workload definitions. TASM allows to dynamically adjust the workload management based on the actual system conditions. The adjustment can be triggered by time-based (e.g., a specific time of the day is reached) or system-specific events (e.g., a predefined number of AWT-in-use is exceeded). In TASM terminology, they are respectively named "*planned environments*" and "*health conditions*" and their combination form a "*state matrix*". Each matrix cell is a state with a defined set of workload priorities and management rules (e.g., filters and throttles).

The state matrix is system-dependent. In batch processing environments, a workload designer typically defines two planned environments (e.g., "night" and "day", or "business hours" and "out of business hours", or "ETL" and "report") and two health conditions (e.g., "1" and "2" or "Normal" and "Heavy"), as depicted in Figure 11.

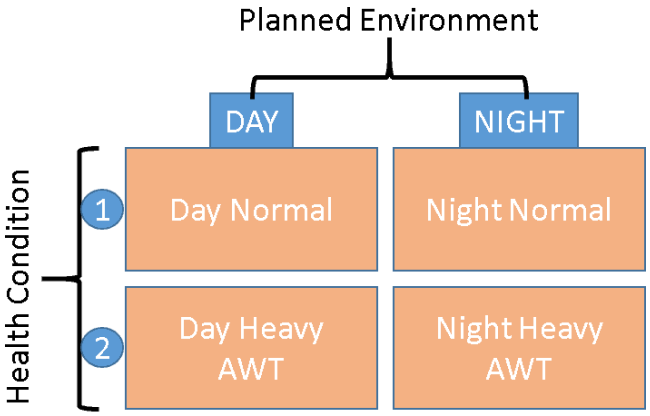

Figure 11 A two-by-two state matrix for dynamic workload management

Once the state matrix is defined, TASM periodically checks whether specific events have occurred that would cause a state switch. TASM first checks for changes in the health conditions and then in the planned environment. If more than one health condition is met, the one closer to the bottom of the matrix is taken. If more than one planned environment conditions are met, the rightmost is taken. TASM applies from that point on the workload management rules of the new state for system resource allocation, filters, and throttles.

Chapter summary

In this chapter, we revisited all the building blocks we will need to explore performance tuning in the next chapters. These were the topics of parallel database architectures; the shared-nothing architecture of Teradata; hardware and software components of Teradata, including cabinets, cliques, clusters, nodes, hot-standby nodes (HSN), Parallel Database Extensions, Parsing Engine (PE), BYNET, Access Module Processor (AMP), VProc, and Database File System; database objects; storing physical rows in an AMP VDisk, including the Teradata parallel hash algorithm, AMP hash maps, row hashes, physical rows, data blocks, VDisk logical sectors and cylinders, and the Master and Cylinder Index; logical row- and columnar orientation; and DBMS services, including the fallback AMP hash maps, Teradata Intelligent Memory, transactions, permanent and transient journals, WAL protocol, locks and deadlock handling, sessions, and workload management.

From this point on, a node will be the lowest level of abstraction used to explain any Teradata Database concept. Any database and query performance tuning discussion will revolve around three elementary units:

- A physical row, the elementary unit of storage in an AMP VDisk.
- A data block, the elementary unit of transfer between an AMP VDisk and AMP memory.
- A transaction, the elementary unit of processing among AMP's.

Chapter 3: Tuning the data warehouse design

"The most important motivation for the research work that resulted in the relational model was the objective of providing a sharp and clear boundary between the logical and physical aspects of database management." – E.F. Codd

A generally-accepted definition of an *(enterprise) data warehouse (EDWH or DWH)* is the area where enterprise-wide data are centrally integrated, stored for the long term, and are available across the whole enterprise as the single authoritative source of truth. The DWH definition implies an agreed way to describe all the data across the whole enterprise and as the enterprise shapes itself over time. Additionally, an agreement on how to store and access these data. These are the topics respectively described by the *logical* and *physical data model* (LDM and PDM). In its purest form, a *data model* is a set of symbols and text for representing information in an easy-to-understand way.

The choice of a data model is a core concept of the data warehouse and, often intensely debated. Nonetheless, once a choice is made, the enterprise lives with it. The more the enterprise information is described under a chosen data model and subsequently implemented, the harder it becomes to change or tune the model.

A data warehouse is typically implemented using one or more databases. A *database model* is an application area of data models and describes how a database is structured and used. The seminal work of E.F. Codd in the 1970's laid the foundations of the *relational model* for database management grounded on predicate logic and set theory of mathematics. One example of a relational database management system (RDBMS) is the Teradata Database.

A *relation* is the basic data structure of the relational model, and it is represented as a two-dimensional *table*. The data are inserted in the table as *tuples* of information, each as one *table row*. Each relation has an agreed number of *attributes*, represented as *table columns*. A database relation (e.g., a database table) meets the *third normal form* (3NF) when all its attributes (e.g., the columns of a table) depend functionally on solely the *primary key*, i.e., a specific choice of a minimal set of attributes (e.g., the table columns) that uniquely specify a tuple (e.g., a table row).

Design approaches

A good *data warehouse design* withstands the test of time and caters to evolving data requirements. There are two major schools of thought on how to approach the data warehouse design. On the one hand, the *top-down design* favors a normalized data model (e.g., 3NF) and *Entity-Relationship* modeling. It was promoted by Bill Inmon, whom many consider as the father of the data warehouse. On the other hand, the *bottom-up design* favors a denormalized data model and *Dimensional* modeling. It was promoted by Ralph Kimball, who introduced the enterprise data warehouse bus architecture.

A normalized data model achieves less data redundancy and consumes less storage space, but it is harder for business-oriented people to navigate and access information. In contrast, a flattened (denormalized) data model exhibits more redundancy, consumes more storage space, and can be harder to change or update the same piece of information across the whole data warehouse. However, it is way easier for business-oriented people to use.

The Teradata Database is data-model agnostic; being an RDBMS, it can support any well-defined data model. Based on the extensive data warehousing experience, the preference of the Teradata company and professionals, including us, is a top-down design favoring the third normal form (3NF) to address the enterprise needs. The design can then be augmented with views based on dimensional modeling techniques and Teradata join indices (discussed in Chapter 6) to address the needs of specific business groups and functions, whenever deemed necessary. This approach combines the best of both approaches and contributes to data warehouse longevity and sustainability.

System architecture

A data warehouse system typically comprises three data layers, namely *staging*, *integration*, and *access*. There are two main approaches to build a data warehouse system. Their difference lies in how the data are brought from dispersed sources into the data warehouse core. In an *Extract-Transform-Load* (ETL) approach, the source data are first extracted in the data stage layer. Then the software tools of the integration layer perform all the necessary transformations to harmonize the data format and store them in a database. The harmonized data are then moved to the data warehouse core (another database). The access layer is then used to assist the retrieval of the stored data.

An Extract-Load-Transform (ELT) data warehouse bypasses the need for specialized software tools. Instead, the data are extracted and directly loaded in a staging area of the data warehouse database. Then, the data are internally transformed into their final form stored in the data warehouse core. The access layer remains unaffected. For convenience, it can be structured in so-called *data marts*, i.e., simplified views of the whole data warehouse with data stored in forms serving the needs of specific groups or functions of the enterprise.

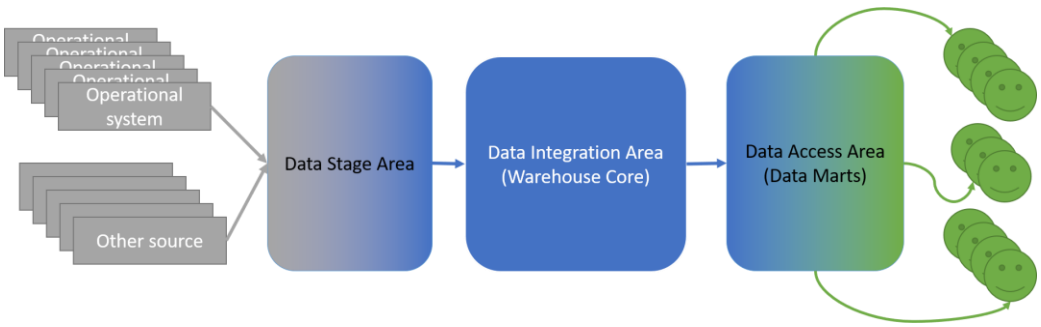

Figure 12 A simplified architecture of an ELT-based data warehouse

Tables: the DWH building blocks

The building block of a data warehouse is the table. Different data layers have different table needs and utilize tables differently. Hence, they exhibit different performance tuning targets. This section focuses on those tables comprising the core of an ELT-based data warehouse following a 3NF data model. The next chapters will explore the table design for the data stage and data marts.

The definition of the table structures and the relationships among the tables are the output of the logical and physical data modeling process. A poor table design can cause severe performance degradation once the table is loaded with data and becomes part of the regular data processing chain. Additional costs are then introduced to redesign it, with a potential for unavailability periods (i.e., data warehouse downtime) to restructure its contents.

LDM and primary keys

At the LDM level, an entity (table) consists of attributes (columns). A table has a primary key (PK) and may have *foreign keys* (FK), i.e., one or more columns referencing another table's primary key.

Table and column names

An LDM that covers the needs of the whole enterprise mandates that table columns are consistently defined across the DWH. Lack of consistency is more often than not an indication of compromised modeling, which can provide early warnings for upcoming performance issues.

There are many categories of inconsistencies, including:

- *Naming conventions* for tables and columns, such as mixed language (e.g., the customers stored in table "`CUSTOMERS`" but the products in "`PRODUKTE`"), mixed-use of plural and singular number (e.g., the customer table is "`CUSTOMERS`" but the product table is "`PRODUCT`"), mixed-use of name prefixes and suffixes (e.g., the same column named "`TOT_AMOUNT`" in one table, "`AMOUNT_TOT`" in another one, and "`AMOUNT_SUM`" in yet another), changing abbreviations of common words (e.g., "`TTL`" vs. "`TOT`" for a total), and choosing names matching the respective interface fields of a source system (this might change in time) rather than the actual content they carry (e.g., a customer name or a product description).
- *Semantic mismatches* of names and content (e.g., storing a multi-value status in a column named "FLAG" or allowing all "Y/N" (English), "J/N" (German), and "N/O" (Greek) pairs as contents of a flag column because of different sources. Or, storing values like "INT1234" and "EXT2345" in a column named "EMPLOYEE_NUM").
- *Data type mismatches* (e.g., a flag stored as a number in one table, as a string in another one, and as a character in yet another one).

- *Data type, character set, and length mismatches* (e.g., defining a float column to store an always-integer-in-source value or defining a fixed-length string of 2,000 Unicode characters to store a 100-Latin-character-long source value with a justification "*just in case the source system starts to send data in another format*." If such changes do occur, the impact across the enterprise, including its DWH, should be explicitly considered and assessed before the change is introduced.
- *Decomposable columns* containing combined pieces of information as one value but processed as two or more (e.g., one column storing the concatenation of country of employment and country-specific employee's number, for example, "EN1234" and "DE5678", but later processed independently per country with country-specific rules. Such decomposed columns violate the 3NF and can hurt query performance. Chapters 5 and 6 further explore this topic.

Primary keys

The primary key of a table uniquely identifies each table row. The primary key consists of one or more columns carrying actual, real-world observations within the enterprise's scope. This is an example of a *natural primary key*. A second option is to enrich the table definition by adding an artificial column to carry an artificial identification key used only within the scope of the database. This is an example of a *surrogate primary key*.

The decision to use a natural or surrogate key should be part of the data model design considerations. On the one hand, defining and properly maintaining an additional column requires more human effort and storage space. Also, the data marts may need as a final step to convert back from surrogate keys to natural keys, as business users are only familiar with the latter.

On the other hand, a proper surrogate key allows smooth data integration from different source systems, for example, in scenarios when different source systems deliver the same natural key but with a different meaning. Then, without a distinct surrogate key, one cannot distinguish the different rows. Or, in scenarios when different source systems deliver a different natural key but for the same piece of information (e.g., the same customer being identified with a different number in each functional group of the enterprise); a surrogate key ensures there is a single (and source-system-neutral) identifier of the customer across the whole data warehouse.

A surrogate key also increases the sustainability and flexibility of the data warehouse when source systems change, and when business mergers and acquisitions happen. In such scenarios, often new natural keys will be delivered. If surrogate keys are not used, then the old natural keys must be updated to the new ones throughout the whole data warehouse. This might not even be allowed in highly-regulated environments, causing a costly re-design of the LDM to accommodate multiple equivalent primary keys. If surrogate keys are used, it will suffice to update the natural-to-surrogate-key mapping function and smoothly integrate the new source of data.

From a performance point of view, a surrogate key substituting a natural key comprising of more than one column may lead to more performant and readable SQL queries that need to join two or more tables on the same natural key. Our clear preference is always to use surrogate keys. Nonetheless, the data modeler has the final say on which key is better in their environment.

PDM and primary indices

The PDM can differ from the LDM. The transition from LDM to PDM is an area where many opinions take shape and can be a source of confusion for data modeling. The primary aim of a PDM is to achieve the best performance for the specific database technology.

The Teradata Database distinguishes between LDM and PDM more evident than any other RDBMS:

- The LDM defines a primary *key* that ensures the uniqueness of table rows.
- The PDM defines a primary *index* (PI) that distributes the table rows into data block physical rows across all the available AMP's.

The sole purpose of the PI is to exploit the Teradata shared-nothing architecture and achieve the highest performance possible. For the rest of this chapter, we will consider tables with a primary index. Subsequent chapters will explore in context those few cases where the performance improves by *purposefully* skipping the PI definition or defining a PI that skews the table row distribution.

AMP distribution of PI table rows

The minimal SQL example of a Teradata table definition is:

```
CREATE SET TABLE database_name.table_name (
   primary_index_column   data_type_1,
   column_name_1   data_type_2,
   column_name_2   data_type_3
) UNIQUE PRIMARY INDEX (primary_index_column);
```

The `UNIQUE PRIMARY INDEX` hints Teradata how to distribute the rows of the table to the available AMP's. The `primary_index_column` of every row is fed as input to the process described earlier in Figure 5 to derive an `AMP_ID`. This is the AMP that will store the specific table row.

The assigned AMP instructs the TDFS to generate a physical row and populates the stored data part with the table row contents. The physical row header includes the hash value (i.e., the ROW_HASH) and a row *Uniqueness Value* that counts from one. Subsequent table rows that their primary index hashes on the same ROW_HASH, use an increased-by-one *Uniqueness Value* to be distinct from the previous ones. The combination of ROW_HASH and Uniqueness Value form the ROW_ID, the unique identifier of the physical row, as depicted in Figure 13. As the last step, TDFS permanently stores the physical row in the

appropriate AMP VDisk data block and updates all related storage structures described in Section Data storage and retrieval of Chapter 2.

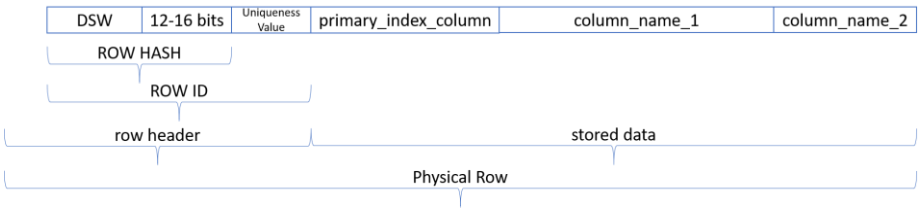

Figure 13 Mapping a PI table row to a physical row

Non-primary-key primary index

The primary key columns are the obvious candidates for primary indices. However, this is not always the optimal choice, as the following example demonstrates.

Consider the case of a table consisting of a few columns and having a row count at the same order of magnitude as the number of available AMP's (e.g., a table holding the country two-letter codes and names in the English language). A primary key (e.g., the two-letter code) ensures row uniqueness. When used as a primary index, Teradata distributes the rows across all VDisk's.

For the scenario above, the primary key is a suboptimal choice for a primary index. The whole storage hierarchy described in Figure 6 will be maintained by each AMP to store just one or a couple of physical rows. The data block of every VDisk for this table is utilized to store only one or a few physical rows. These combined cause disk access operations across all AMP's. Also, if one AMP is under flow control, the whole processing will be delayed. Hence, from a performance point of view, it is better to define a primary index column that will cause all rows to concentrate on one or a couple of AMP's (e.g., a technical column with a constant value). Chapter 6 further discusses design techniques to handle such cases efficiently.

Primary index data types

Teradata supports many different column data types. A non-exhaustive list includes:

- Logical bitstream types (e.g., `BYTE`, `VARBYTE`, and `BLOB`).
- Character types (e.g., fixed-length `CHAR`, variable-length `VARCHAR`, and `CLOB`).
- Date- and time-oriented types (e.g., `DATE`, `TIME`, `TIMESTAMP`, and various `INTERVAL` and `PERIOD` types).
- Numeric types (e.g., the whole-number-oriented BYTEINT, SMALLINT, INTEGER, BIGINT, NUMBER, and the fractional-number-oriented `DECIMAL` (or `NUMERIC`) and `FLOAT`).
- Structured data format types (e.g., AVRO, XML, and JSON).

Not all data types are allowed for a primary index; for example, a BLOB, CLOB, XML, or JSON column cannot be part of the primary index definition. The primary index column data types can have a significant impact on how uniformly the table rows are distributed to AMP's. The wrong data type selection can have a negative performance impact. The root cause is that the Teradata hashing algorithm may hash the same logical value into a different hash (and subsequently, AMP_ID) when the underlying data type differs. We can easily confirm this using the Teradata HASHROW() function:

```
SELECT '3' AS VAL_CHR, HASHROW(VAL_CHR), 3 AS VAL_INT, HASHROW(VAL_INT);

VAL_CHR       HASHROW(VAL_CHR)      VAL_INT             HASHROW(VAL_INT)
-------       ----------------      -------       ----------------
3              9A0F8DF8              3             6D27DAA6
```

The hashes are different, so, most probably, a different AMP and fallback AMP will be assigned to store the respective physical rows. We can easily confirm this using the Teradata HASHAMP() function:

```
SELECT '3' AS CVAL, HASHBUCKET(HASHROW(CVAL)) AS CBUCKET, HASHAMP(CBUCKET) AS CAMP;
SELECT 3 AS CVAL, HASHBUCKET(HASHROW(CVAL)) AS CBUCKET, HASHAMP(CBUCKET) AS CAMP;

CVAL      CBUCKET         CAMP
----  -----------  -----------
   3       631032            1
   3       447101            0
```

The negative performance impact surfaces when one SQL query will join the contents of the two tables having the same primary index but defined with a different data type. Then, the join cannot occur locally on one AMP; instead, all AMP's must send to all other AMP's the related table segments they need to perform the join locally.

Teradata offers by design *hash-compatible* data types, i.e., different data types that generate the same hash value for the same logical input value. This is because the hashing algorithm processes the internal, binary representation of an input value. All whole-number data types (i.e., BYTEINT, SMALLINT, INTEGER, and BIGINT columns) and fractional-number data types without a fractional part (e.g., a DECIMAL(n,0) column) generate the same hash value. The same holds for the DATE data type since it is encoded as an integer value in Teradata.

```
SELECT 'BYTEINT ' AS DATA_TYPE, Cast(101 AS BYTEINT) AS DATA_VALUE,
HASHBUCKET(HASHROW(DATA_VALUE)) AS DATA_BUCKET FROM (SELECT NULL AS X) AS X
UNION ALL
SELECT 'SMALLINT' AS DATA_TYPE, Cast(101 AS SMALLINT) AS DATA_VALUE,
HASHBUCKET(HASHROW(DATA_VALUE)) AS DATA_BUCKET FROM (SELECT NULL AS X) AS X
UNION ALL
SELECT 'INTEGER' AS DATA_TYPE, Cast(101 AS INTEGER) AS DATA_VALUE,
HASHBUCKET(HASHROW(DATA_VALUE)) AS DATA_BUCKET FROM (SELECT NULL AS X) AS X
UNION ALL
SELECT 'BIGINT' AS DATA_TYPE, Cast(101 AS BIGINT) AS DATA_VALUE, HASHBUCKET(HASHROW(DATA_VALUE))
AS DATA_BUCKET FROM (SELECT NULL AS X) AS X
```

```
UNION ALL
SELECT 'DECIMAL' AS DATA_TYPE, Cast(101 AS DECIMAL(38,0)) AS DATA_VALUE,
HASHBUCKET(HASHROW(DATA_VALUE)) AS DATA_BUCKET FROM (SELECT NULL AS X) AS X
UNION ALL
SELECT 'DATE' AS DATA_TYPE, DATE '1900-01-01' AS DATA_VALUE, HASHBUCKET(HASHROW(DATA_VALUE)) AS
DATA_BUCKET FROM (SELECT NULL AS X) AS X

DATA_TYPE  DATA_VALUE  DATA_BUCKET
---------  ----------  -----------
BYTEINT           101        24646
SMALLINT          101        24646
INTEGER           101        24646
BIGINT            101        24646
DECIMAL           101        24646
DATE              101        24646
```

The NULL value (no matter the data type), number zero, and zero-length string (i.e., an empty string we often represent as ‘’) hash to the same value:

```
SELECT 'Number' AS DATA_TYPE, HASHBUCKET(HASHROW(0)) AS DATA_BUCKET FROM (SELECT NULL AS X) AS X
UNION ALL
SELECT 'Blank' AS DATA_TYPE, HASHBUCKET(HASHROW('')) AS DATA_BUCKET FROM (SELECT NULL AS X) AS X
UNION ALL
SELECT 'NULL' AS DATA_TYPE, HASHBUCKET(HASHROW(NULL)) AS DATA_BUCKET FROM (SELECT NULL AS X) AS
X

DATA_TYPE  DATA_BUCKET
---------  -----------
Number               0
Blank                0
NULL                 0
```

The character types CHAR and VARCHAR are hash-compatible for the same length of characters. However, space and non-displayable characters (often called “blanks”) do affect the generation of the hash value, even for the fixed-length CHAR data type:

```
SELECT 'CHAR008' AS DATA_TYPE, Cast('       3' AS CHAR(8)) AS DATA_VALUE,
HASHBUCKET(HASHROW(DATA_VALUE)) AS DATA_BUCKET FROM (SELECT NULL AS X) AS X
UNION ALL
SELECT 'CHAR001' AS DATA_TYPE, Cast('3' AS CHAR(8)) AS DATA_VALUE,
HASHBUCKET(HASHROW(DATA_VALUE)) AS DATA_BUCKET FROM (SELECT NULL AS X) AS X
UNION ALL
SELECT 'VARCHAR' AS DATA_TYPE, Cast('3' AS VARCHAR(8)) AS DATA_VALUE,
HASHBUCKET(HASHROW(DATA_VALUE)) AS DATA_BUCKET FROM (SELECT NULL AS X) AS X

DATA_TYPE  DATA_VALUE  DATA_BUCKET
---------  ----------  -----------
CHAR008           3         716794
CHAR001    3                631032
VARCHAR    3                631032
```

A more subtle negative performance impact may surface when a primary index is based on character data types. This is harder to detect and rectify, as the root cause lies deep into how Teradata handles hash collisions. The example below demonstrates such a case.

Consider a table that will store phone call information and have as its primary key a string column comprising eight-digit phone numbers (e.g., using a `CHAR(8)` data type). Let's further assume that the Teradata system will have 20-bit hash buckets and 160 AMP's. As there will be one hundred million possible phone numbers and 1,048,576 hash buckets, we can expect that in an ideal case:

- Every hash bucket (DSW) will be generated 95 times.
- Every AMP will be assigned 625,000 table rows.
- Very few or even no hash collisions, as there are over four billion possible hashes to choose from.

We calculated for all possible input values the respective DSW, AMP, and number of collisions. Figure 14 depicts the number of different hash buckets (vertical axis) as a function of frequency generation for the string-based primary index column. The ideal distribution is an impulse function centered at value 95, with 1,048,576 counts. The observed distribution in our test is still centered at value 95, but it looks more like a bell curve with a maximum at about 43,000 counts, i.e., 25 times less than expected ones. One bucket is generated only 50 times, another one only 51 times, and many more in the range of 120 to 150.

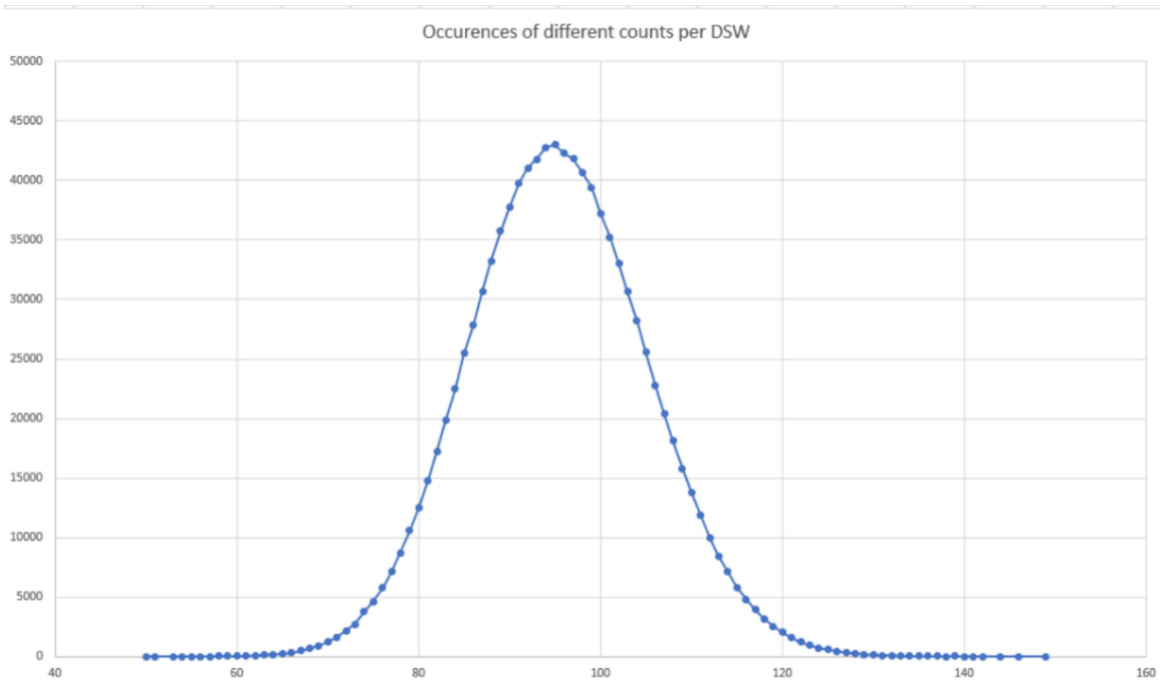

Figure 14 Distribution of DSW occurrences per frequency (CHAR)

The hash bucket distribution is not a problem by itself: each of the 160 AMP's gets assigned, as expected, between 623,000 and 627,000 rows. A closer look at the hash value demographics reveals the actual problem: the bucket distribution consistently generates duplicate hash values across all AMP's, as depicted in Figure 15.

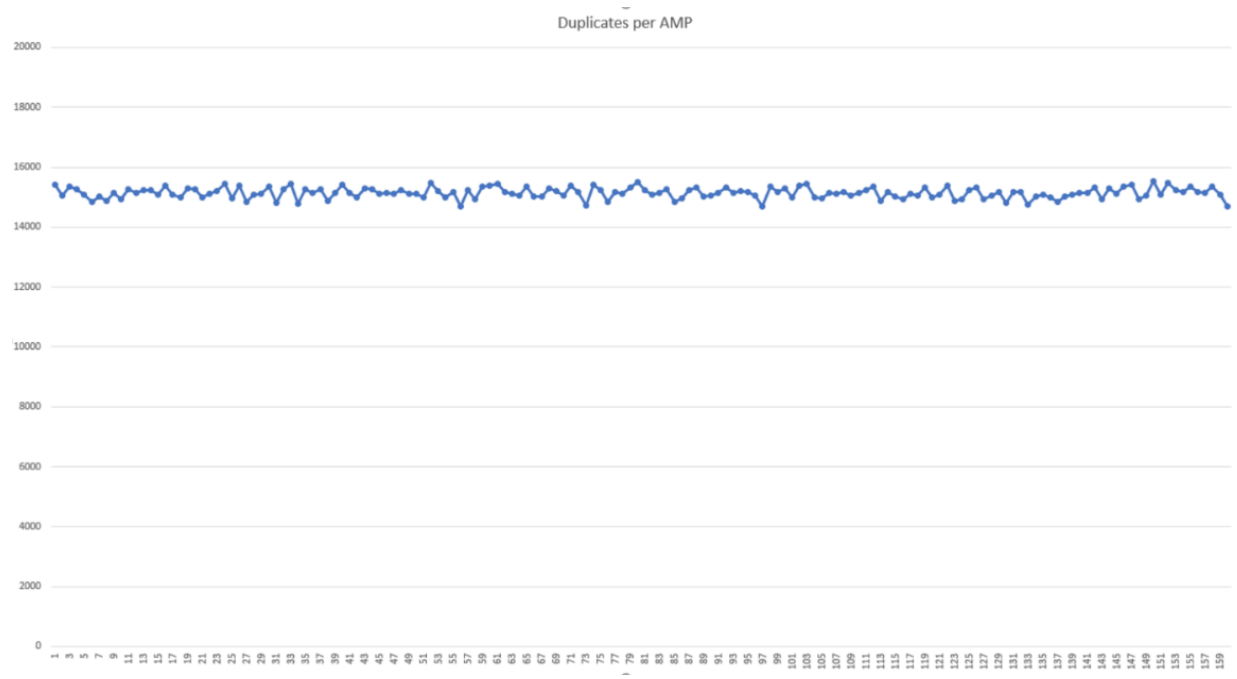

Figure 15 Distribution of duplicate hash values per AMP (CHAR)

There are about 15,000 duplicate hashes per AMP, i.e., 2.4% of all the rows to be stored. It is even more impressive that there are overall over 10,000 hash values repeated not only twice but three or even four times. At retrieval time, all the physical rows with the same hash are sequentially scanned to find the searched one. This results in suboptimal performance for our case.

Let's now revisit the primary index and use a whole-number data type (e.g., an `INTEGER` column). This small change has a considerable impact, as depicted in Figure 16: the curve in Figure 14 becomes a sharp impulse function, with values strictly ranging between 93 and 98. Every AMP is assigned 625,000 rows, with a negligible variation of ±280 rows. Even better, there are no duplicate hashes anymore!

By changing the primary index to a content-equivalent INTEGER column, we helped Teradata distribute the rows more evenly across the AMP's and eliminated all hash collisions that would cause sequential search in the data blocks and decrease performance.

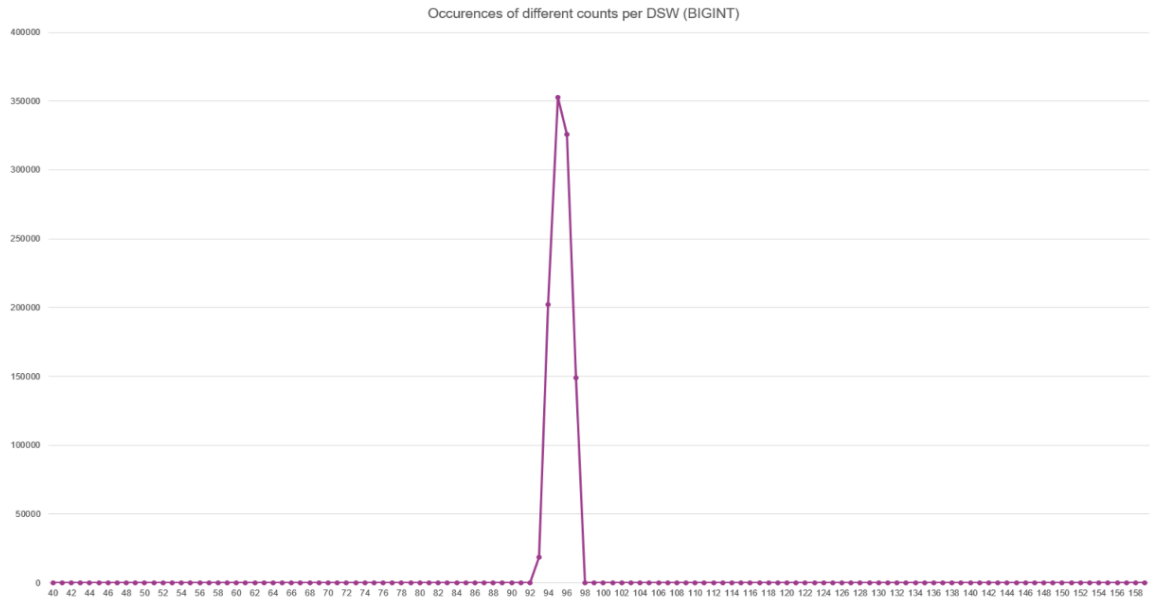

Figure 16 Distribution of DSW occurrences per frequency (BIGINT)

One can decide either to change the primary index column data type to a whole number or to define an additional table column to store the phone number as a whole number and use it as the primary index. All three approaches (reuse primary key column as-is, change its data type, or add a new column) have their advantages and drawbacks, so the decision should be carefully evaluated and aligned with the data modeler.

Surrogate keys as primary indices

Surrogate keys are an excellent option for a primary index, mainly when defined with a whole-number data type (e.g., INTEGER or BIGINT). Such a choice can reduce the number of hash collisions for character-based natural keys, as demonstrated in the previous example.

A surrogate key is typically implemented as an auto-incrementing whole number, allocated to every new row inserted into a data warehouse table. This approach allows us to remove any bias on values in the natural key columns, further assisting the Teradata hashing algorithm in uniformly distributing the rows across available AMP's.

Teradata provides the column identity feature to support the surrogate key generation and remove the administrative burden from the developers. Such an automatically-handled column may be directly used as the primary index of the table, as in the following example:

```
CREATE SET TABLE database_name.table_name (
   unique_row_id BIGINT GENERATED ALWAYS AS IDENTITY (START WITH 1 INCREMENT BY 1 NO CYCLE),
   column_name1   data_type,
) UNIQUE PRIMARY INDEX (unique_row_id);
```

There are a few points we should consider when using an identity column to avoid running into unpleasant surprises, both at the design and regular operation of the data warehouse:

- Each table can have at most one identity column. It is impossible to add such a column once the table is already created, i.e., an ALTER TABLE statement to add an identity column will fail.
- An identity column cannot be part of a composite (multi-column) primary index.
- Uniqueness is guaranteed only when both options `GENERATED ALWAYS` and `NO CYCLE` are present in the definition. If the latter is not present, cycles in numbering are permitted.
- A `GENERATED ALWAYS` column cannot be programmatically updated and cannot be NULL.
- An identity column may be monotonically incrementing (a positive number follows option `INCREMENT BY`) or decrementing (a negative number follows option `INCREMENT BY`).
- There can be gaps in numbering. For performance reasons, each AMP working concurrently picks up a block of numbers from the available pool of numbers rather one by one. Should an AMP do not use the whole block, some numbers are forever skipped.
- The maximum (respectively, minimum) supported number is the DECIMAL(18,0) data type. Thus, even when a BIGINT data type is defined, the column cannot store values outside the range of DECIMAL(18,0). This always holds, even when one configures the maximum decimal type supported by the system to DECIMAL(38,0). As a corollary, a table with `GENERATED ALWAYS` and `NO CYCLE` options can store up to 10^{18} rows provided that no numbering gaps occurred. If deemed necessary, one can specifically restrict the range of an identity column using the `MAXVALUE` (respectively, `MINVALUE`) option.
- If the pool of numbers is exhausted and `NO CYCLE` option is present, an error message "Failure 7545" is reported, and execution is stopped.
- The status of every identity column is available at the system table `DBC.IdCol`.

We consider a good practice to periodically monitor the system table for indications of upcoming exhaustions rather than waiting for an error message to appear. Chapter 4 explores how the column identity feature can be utilized for tuning data loading in the data stage area tables.

Table tuning

The definition of a proper primary index does not exhaust the topic of table tuning. The following sections explore additional topics, structured in a top-down approach, from the LDM and referential integrity down to data block physical rows and compression.

Referential integrity

Referential integrity (RI) is a property of data stating that all its references are valid. In relational databases, if a value of one attribute (column) of a relation (table) references a value of another attribute, then the referenced value must exist. The referencing column is declared as a *foreign key* (FK) and the referenced column as a *primary key* (PK). An FK column can contain either a value that already exists in the PK column or a NULL value.

The maintenance of referential integrity in a database during data inserts, deletes, and updates is tedious and error-prone. Teradata, like many other RDBMS, can *enforce* referential integrity and remove the associated burden from the developers. Teradata supports three kinds of referential integrity: standard, batch, and soft.

Standard RI

One can explicitly define referential integrity in Teradata (i.e., *standard RI*), as in the example below. There are two points to consider in the standard RI definition. The first is that the primary key column of the parent table must hold unique values and cannot be NULL. A `UNIQUE PRIMARY INDEX` column is a good candidate for this. The second point is that the foreign key and the primary key must be of the same data type and, if character-based data types occur, the same case-sensitivity.

```
CREATE SET TABLE database_name.parent_table (
    column_pk_key INTEGER NOT NULL,
    column_name1 data_type,
    column_name2 data_type
) UNIQUE PRIMARY INDEX (column_pk_key);

CREATE SET TABLE database_name.child_table (
    column_primary_index INTEGER NOT NULL,
    column_fk_key INTEGER,
    column_name3 data_type,
    CONSTRAINT f FOREIGN KEY(column_fk_key) REFERENCES parent_table(column_pk_key)
) UNIQUE PRIMARY INDEX (column_primary_index);
```

When standard RI is used, Teradata maintains a *reference index sub-table* for each foreign key defined in a table. The sub-table is distributed to all AMP's by hashing the foreign key value. This ensures that the sub-tables rows (of the child table) and the respective rows of the parent table will be at the same AMP.

Every affected row is immediately checked for an RI violation at every INSERT, DELETE, and UPDATE statement. The check is *locally* performed at each AMP, without exchanging rows with another AMP. If a sub-table row exists but a parent table row does not exist, then there is an RI violation, and the execution of the statement is aborted.

On the one hand, the row-by-row checks of standard RI result in performance degradation. Additionally, Teradata tools for bulk loads, like the ones described in Chapter 4, cannot operate on tables with standard RI (neither with batch RI discussed next). On the other hand, such explicit RI definitions can hint the Teradata to create optimal SQL query execution plans, as further discussed in Chapter 6.

Batch RI

The *batch RI* aims to remedy the performance degradation of standard RI by performing the checks on the entire transaction. The example below demonstrates a batch RI definition.

```
CREATE SET TABLE database_name.parent_table (
    column_pk_key INTEGER NOT NULL,
    column_name1 data_type,
    column_name2 data_type
) UNIQUE PRIMARY INDEX (column_pk_key);

CREATE SET TABLE database_name.child_table (
    column_primary_index INTEGER NOT NULL,
    column_fk_key INTEGER,
    column_name3 data_type,
    CONSTRAINT fk FOREIGN KEY(column_fk_key) REFERENCES WITH CHECK OPTION
        parent_table(column_pk_key)
) UNIQUE PRIMARY INDEX (column_primary_index);
```

Batch RI need not maintain a reference index sub-table. The RI check is instead performed by joining the parent and child tables on the RI column: if a child column cannot be joined, there is an RI violation.

When an RI violation is detected, the whole transaction is rolled back even due to only one row. This can be a time-consuming operation when many rows are involved in one transaction. Also, since the error is on the transaction level, spotting the input row that caused the rollback can resemble an attempt to find a needle in a haystack.

The primary index of the child table may differ from the one of the parent table, as also seen in the above example. This implies that the rows of the two tables will not necessarily reside on the same AMP (in fact, most probably, they will not). Hence, when batch RI is performed, the rows must be redistributed among AMP's for the join to be checked. Thus, the performance gain associated with avoiding the row-by-row checks of the standard RI is lessened.

Soft RI

Teradata allows one to define *soft RI*, using a syntax similar to the example below. The sole purpose of a soft RI definition is to hint Teradata to derive optimized SQL query execution plans, trusting that the developers will maintain themselves the implied referential integrity.

```
CREATE SET TABLE database_name.parent_table (
    column_pk_key INTEGER NOT NULL,
    column_name1 data_type,
    column_name2 data_type
) UNIQUE PRIMARY INDEX (column_pk_key);

CREATE SET TABLE database_name.child_table (
    column_primary_index INTEGER NOT NULL,
    column_fk_key INTEGER,
    column_name3 data_type,
    CONSTRAINT fk FOREIGN KEY(column_fk_key) REFERENCES WITH NO CHECK OPTION
    parent_table(column_pk_key)
) UNIQUE PRIMARY INDEX (column_primary_index);
```

Soft RI causes no performance degradation, like standard and batch RI do. As no column reference checks are performed, Teradata bulk load tools can be used with no restriction. As Teradata does not enforce the referential integrity of a soft RI, an RI violation may occur, resulting in wrong answers to an SQL query.

DWH and RI

In a typical data warehouse environment, it is tolerable to have *occasional* RI violations (as in the case of soft or no RI) rather than a permanent performance degradation accompanying the referential integrity enforcement (as in the case of standard or batch RI). The following list summarizes the key considerations for selecting an RI strategy, purely from a performance tuning perspective:

- The performance degradation of single-row INSERT, UPDATE, and DELETE statements is almost the same in the cases of standard and batch RI.
- The performance degradation of multi-row INSERT, UPDATE, and DELETE statements are typical:
- Higher for standard RI, if RI violations rarely occur.
- Higher for batch RI, if RI violations frequently occur.
- Soft RI causes no performance degradation.

It might seem that soft RI should be the preferred choice for a high-performant DWH. However, what is the benefit of a DWH that returns in lighting speed wrong answers to SQL queries? One should aim to bridge the performance of soft RI with the guarantees of RI enforcement.

Data are typically loaded into a DWH in batches. It is possible to check any PK and FK constraints already at the data stage layer and discard any rows violating the RI constraints. Then, there is no benefit to enforce the same checks again in the DWH core. A DWH may also hold multiple valid versions of the same attribute to track its values over time. A data load or reload can occur for a specific time instance; then, the respective FK column must be checked against that value of the PK column at the same specific time instance. The SQL constructs for defining RI checks are insufficient for this. Additional checks must be developed before the data reach the DWH core.

Last but not least, it is not unheard in a DWH environment that a table must be recreated or reloaded from scratch. If this table has a PK column and RI is enforced, it becomes very challenging to perform this task, considering that all the child table FK columns and their RI definitions must be respected. For all the reasons above, a typical approach is to perform any RI checks already in the data stage area rather than the DWH core. Chapters 4 and 5 revisit referential integrity in the context of data stage and data marts, respectively.

Skewed table tolerance

Every Teradata database has a maximum permanent storage space, defined during creation. The space is *evenly* distributed among the available AMP's. For example, in an eight-AMP system, the SQL statement below allocates 80 GB of permanent storage space. Then, each AMP gets a 10 GB slice of it:

```
CREATE DATABASE database_name AS PERMANENT = 10E9*(HASHAMP()+1) BYTES;
```

The maximum and currently-occupied storage space of a database can be checked by querying the `MaxPermSpace` and `CurrentPermSpace` columns of the `DBC.DatabaseSpace` table. The `CurrentPermSpace` column reports the *cumulative* storage space occupied by all the AMP's.

The tables use the available permanent space of their database to store their rows. Teradata assumes that the table's primary index definition will *evenly* distribute the rows across all the available AMP's. A poor primary index choice or unexpected data demographics can cause more occurrences of some (sets of) primary indices. Then, distributing rows to AMP's becomes skewed. *Skewed tables* can cause database errors even when tries to insert the table rows in an otherwise empty database.

No tolerance strategy

Up to Teradata version 16, the AMP limits are hard. If an insert statement requires more permanent space than the allocated for an AMP, then it fails with an error "2644: No more room in database database_name." In the example above, let's assume that the database is empty and that 12 GB of table space is needed, but the data are so skewed that an AMP will get 11 GB of them. Despite the ample database space, the insert statement will fail. The situation gets even worse once the database is populated with data from many other tables and, thus, less permanent space is available per AMP.

Starting with version 16, Teradata allows one to define a more sustainable strategy and cope more gracefully with skewed tables. The strategy can be defined at the AMP- and global level.

AMP-level strategy

The skew factor controls the AMP-level strategy. In turn, the skew factor is controlled by two means. The first is a SKEW option added in the SQL syntax for creating or modifying a database. When this option is present, as in the example below, an AMP may exceed the storage limit, as long as the cumulative space occupied by all the AMP's does not exceed the maximum allocated for the database:

```
CREATE DATABASE database_name AS PERMANENT = 80E9 BYTES SKEW = 15 PERCENT;
```

Applied in the earlier example, each AMP initially gets a 10 GB slice. However, this limit is tolerable: an AMP slice can grow up to 11.5 GB, if necessary. Then, the table insert statement worth of 11 GB in one AMP and 12 GB in total will succeed without an error. Nonetheless, an event will be recorded in the Software Event Log of the RDBMS to allow the database administrators to analyze the situation and take appropriate actions.

The second skew factor control means is the `DefaultPermSkewLimitPercent` Teradata Database configuration setting (or, "DBS Control utility field" in Teradata terminology). This setting applies when the database SKEW option is defined as "DEFAULT" or is not present. When a non-zero `DefaultPermSkewLimitPercent` is set, different behavior is observed when the AMP limits are exceeded from Teradata 16 onwards compared to earlier versions.

We consider a good practice to explicitly mention the SKEW option in a Teradata 16 or higher environment, like in the example below:

```
CREATE DATABASE database_name AS PERMANENT = 80E9 BYTES SKEW = DEFAULT PERCENT;
```

It can save a lot of performance analysis effort at a later stage. This is especially true when one works in multiple environments with mixed Teradata versions.

Global-level strategy

The `GlobalSpaceSoftLimitPercent` configuration setting allows a temporary increase in the maximum database size to accommodate *transient* processing needs. This space is evenly distributed among all available AMP's. The setting applies independently of any per-AMP setting.

When both per-AMP and global settings are non-zero, their effects are combined. For example, assume a four-AMP system with a database assigned 80 GB of permanent space, a 10% global space soft limit, and a 15% AMP skew limit. The database can then grow up to 88 GB, and an AMP can grow from 20 GB up to 25 GB, i.e., 2 GB of from the global space soft limit plus 3 GB of the AMP skew limit.

We consider that a non-zero setting should be set to benefit performance tuning, always aligned with the database administrators and the Teradata Support Center personnel. The reason is that when the related events are recorded in the Software Event Log, they provide early indications of bad-performing SQL queries. This creates a window of opportunity for remedy actions, ahead of the moment that the processing is halted and the whole DWH operation is affected.

Multiset tables

How many identical table rows can exist with the same primary index? Teradata adheres to the relational model of E.F. Codd, in which a table is a *set* of tuples, and, thus, no duplicate

rows can exist. SQL, as standardized by ANSI and ISO organizations, deviates from this model; it is a *bag* language.

Sets, bags, and SQL

Bags (in database theory) or multisets (in mathematics) are a modification of the concept of a set. A multiset can contain more than one instances of its elements, while a set contains each element only once: $\{a, b, a, c, b, a\}$ is a multiset with three elements, and $\{b, a, c\}$ is a set (and a multiset) with three elements. By definition, a set is also a multiset, but the reverse does always hold.

SQL operates on multisets (bags) and, thus, an SQL query answer can contain duplicate rows (i.e., the answer is a multiset). One must explicitly request unique-row answers (i.e., sets), using, for example, the DISTINCT or GROUP BY keywords of the SQL language.

Performance implications

Teradata supports both set and multiset tables. The semantics of the session (cf. Section Teradata sessions) define the table type when the latter is omitted from the SQL statement. A Teradata session defaults to a SET table while an ANSI session defaults to a MULTISET one.

Teradata guarantees that no duplicate rows are ever inserted in a set table, either by silently discarding the offending rows (e.g., when an `INSERT INTO set_target SELECT * FROM multi_source` statement is executed) or generating an error (e.g., when an `INSERT INTO set_target VALUES(10, 20); INSERT INTO set_target VALUES(10, 20);` is executed). On the one hand, this removes the burden from the developer to perform checks for duplicates. On the other hand, under specific conditions, it can cause severe performance degradation.

Let's consider the following SQL statement:

```
CREATE TABLE database_name.table_name (
    column_name1   data_type,
    column_name2   data_type,
    column_name3   data_type
) PRIMARY INDEX (column_name1);
```

For an ANSI session, this is a perfectly valid *multiset* table definition. Duplicate rows sharing the same primary index can be inserted with no performance penalty. For a Teradata session, the SQL statement above creates a *set* table with a *non-unique* primary index, as the keyword UNIQUE is not included in the definition. Here, Teradata employs DUPLICATE ROW CHECK operations to detect row uniqueness violations every time a row is inserted or updated in this set table. When many rows with the same primary index are inserted or updated, the number of performed checks *exponentially* increases with each new row inserted into the table.

The following example demonstrates the performance difference between otherwise identical SET and MULTISET tables:

```
CREATE SET TABLE TMP_SET (
    PK INTEGER NOT NULL,
    DESCR INTEGER NOT NULL
) PRIMARY INDEX (PK);

CREATE MULTISET TABLE TMP_MUL (
    PK INTEGER NOT NULL,
    DESCR INTEGER NOT NULL
) PRIMARY INDEX (PK);

INSERT INTO TMP_SET
SELECT MOD(10000*B.inp + A.inp, 500) AS PK, 10000*B.inp + A.inp AS DESCR
FROM (
        SELECT TOP 9999 day_of_calendar AS inp
        FROM sys_calendar.calendar
        ORDER BY day_of_calendar ASC
) AS A
CROSS JOIN (
        SELECT TOP 200 day_of_calendar AS inp
        FROM sys_calendar.calendar
        ORDER BY day_of_calendar ASC
) AS B;

INSERT INTO TMP_MUL
SELECT MOD(10000*B.inp + A.inp, 500) AS PK, 10000*B.inp + A.inp AS DESCR
FROM (
        SELECT TOP 9999 day_of_calendar AS inp
        FROM sys_calendar.calendar
        ORDER BY day_of_calendar ASC
) AS A
CROSS JOIN (
        SELECT TOP 1000 day_of_calendar AS inp
        FROM sys_calendar.calendar
        ORDER BY day_of_calendar ASC
) AS B;

/* wall-clock time to complete (not a reliable metric!)
 * *** Insert completed. 1999800 rows added.
 * *** Total elapsed time was 6 minutes and 42 seconds.
 * *** Insert completed. 1999800 rows added.
 * *** Total elapsed time was 4 seconds.
 */

SELECT * FROM DBC.DBQLOGTBL;
/* (a system-independent, absolute, reliable metric!)
 * TMP_SET Total I/O count: 31,620 AMP CPU Seconds: 587.640
 * TMP_MUL Total I/O count:  2,378 AMP CPU Seconds:   3.296
 */
```

The set table with a non-unique primary index caused 13 times more disk input-out (I/O) operations and 178 times more CPU time for the same volume of data!

Once the performance problem becomes evident, one needs to identify the original intention before implementing a change. The options would be to define a unique primary index or another uniqueness construct and deal with the already-inserted duplicate rows (if the intention was to disallow duplicate rows) or change the table type from SET to MULTISET (if the intention was to allow duplicate rows).

There is no performance impact if a unique primary index (UPI) is defined for a set table. The same holds for any index (e.g., the USI discussed in Chapter 5) or column constraint (e.g., IDENTITY, UNIQUE, and PRIMARY KEY) that ensures by design row uniqueness. In such cases, the expensive DUPLICATE ROW CHECK step is skipped. However, Teradata supports a direct change from a non-unique to a unique primary index only when the table is empty. This introduces some additional administrative burden (e.g., reserve, if necessary, additional database space, create a new SET table with a unique primary index, copy the data from the existing table, drop the existing table, rename the new one, collect any additional metadata necessary, and release, the additional database space).

Teradata does not support a change of the table type from SET to MULTISET, even when the table is empty. This is a technical limitation as, in theory, a set is also a multiset. In any case, an administrative burden similar to the one previously described is necessary to realize this change.

We consider a good practice to always define the intended table type rather than relying on session semantics. Also, to carefully evaluate the definition of a set table lacking a unique index or column, the equivalent row uniqueness can be achieved programmatically in SQL language. As the above analysis demonstrated, an overlooked definition of the table type can cause both noticeable performance degradation and significant administrative burden to change it afterward.

Table partitioning

Table partitioning guides the TDFS to store logically-related table rows within the same data blocks. Under the assumption these sets of rows will be mostly queried together, fewer data blocks will be transferred from the AMP VDisk to the AMP memory.

Partition types

Teradata supports *horizontal*, *vertical*, and *multi-level* table partitioning. The difference among them lies in how the table contents are sliced into partitions:

- Horizontal (or row) partitioning): a data block contains sets of physical rows where each row contains the full table row (i.e., all the table columns of the table row).
- Vertical (or column) partitioning: a data block contains sets of physical rows where each row contains a set of table row columns. The partitions then contain a disjoint set of table columns (i.e., the union of the columns in each partition resembles the whole set of table columns while the respective intersection is the empty set).

- Multi-level partitioning allows one to define multiple partitioning levels, either of the same (i.e., multi-level horizontal or multi-level vertical partitioning) or of mixed types (i.e., multi-level hybrid partitioning). In hybrid partitioning, a data block contains sets of physical rows where each row contains a subset of the table columns. The constraints of both horizontal and vertical partitioning apply here.

The following example demonstrates a multi-level hybrid partitioning definition, first by table column (vertical) and then by values of column `item_qty` (horizontal):

```
CREATE SET TABLE database_name.table_name (
    transaction_number INTEGER,
    transaction_date DATE,
    item_number INTEGER,
    item_quantity INTEGER
) UNIQUE PRIMARY INDEX (tx_num)
PARTITION BY (
    COLUMN,
    CASE_N (item_quantity = 1, item_quantity = 5, item_quantity >= 10)
)
```

As the table definition includes four columns and three possible value ranges for the `item_quantity` column, there can be at most twelve different partitions. Teradata can support multiple levels as long as the total number of partitions remains less than $2^{63} - 1$.

AMP distribution of NPPI and PPI table rows

A *Partitioned Primary Index (PPI)* table definition includes both a (unique) primary index and a (two-level) partition. A *Non-Partitioned Primary Index* (NPPI) table definition includes only a (unique) primary index and defines no partitions. Teradata internally handles an NPPI table as a single (zero-level) partition without a slicing criterion defined. In this sense, the terms PI and NPPI can be used interchangeably.

Distributing PPI and NPPI tables rows to AMP's works as follows. First, the primary index column(s) of a table row is fed to the process described in Figure 5 to derive the AMP that will store the specific table row. This is precisely the process described earlier in this chapter for PI tables.

Teradata assigns a unique identifier to each partition and increases the row header size for each physical row by either 2 bytes (when the total number of defined partitions is between 2 and 2^{16}) or 8 bytes (when the number is between 2^{16} and $2^{63} - 1$). This unique identifier serves as a SUBTABLE_ID within the physical row header. In the case of an NPPI table, the SUBTABLE_ID is considered zero, and the row header size is not increased.

The assigned AMP instructs the TDFS to generate a physical row and populates the stored data part with the table row contents. The physical row header now includes the partition (SUBTABLE_ID) and row (ROW_ID) identifiers, as depicted in Figure 17. The ROW_ID, as described earlier already, consists of the ROW_HASH and the Uniqueness Value.

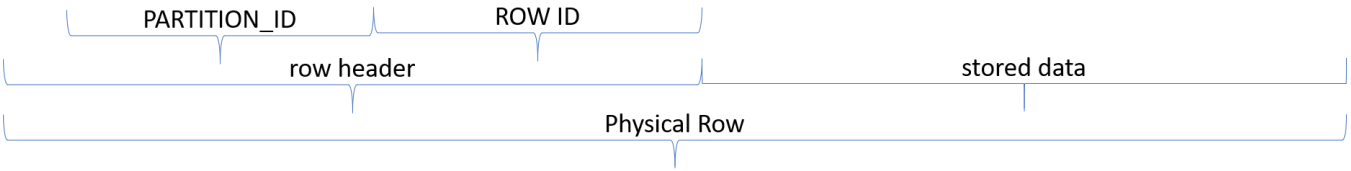

Figure 17 Mapping a PPI table row to a physical row

As the last step, TDFS permanently stores the physical row in the appropriate AMP VDisk data block and updates all related storage structures. The data block header of each data block contains the list of the physical rows sorted first by the SUBTABLE_ID (deriving from the partitions) and then by the ROW_ID (deriving from the primary index), as depicted in Figure 18. In the case of an NPPI table, all the physical rows have a zero SUBTABLE_ID. Then, the data block header list is effectively sorted by the ROW_ID.

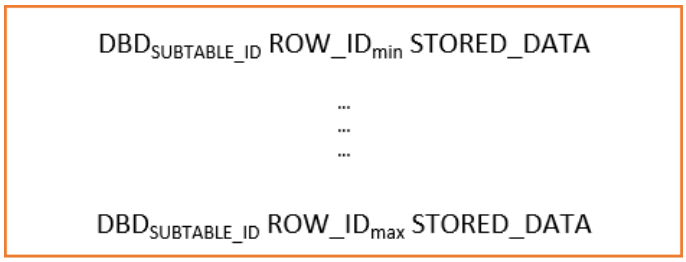

Figure 18 A closer look at the data block organization of the physical rows or a PPI table

Performance considerations

Table partitioning, both in single- and multi-level scenarios, can have significant negative and positive performance implications. Its use must be carefully evaluated during both the design and operational phases of the data warehouse. The focus of this section is on the design phase. Chapter 5 further explores the utilization of table partitions for query tuning during the operational phase of the data warehouse.

As a starting point, by enabling table partitioning, the storage space required per physical row increases by a few bytes per physical row. Then, too many small non-empty partitions can cause performance degradation due to more intensive disk I/O to transfer too many small data blocks instead of a few bigger data blocks. Also, the partition level order impacts the overall partition elimination rate when considering all related SQL queries. The question is then what is more critical from a business perspective to prioritize in the partition level order definition: the more frequent, heavy-duty queries (i.e., reduce the overall resource consumption) or the less frequent but time-critical queries (i.e., achieve faster response times for selected infrequent queries in the expense of increased overall resource consumption). Defining too many partitions (i.e., over-partitioning) can worsen the query performance compared to directly scanning through all the table rows at once (i.e., perform a *full-table scan*) due to the logistics involved in identifying those few relevant data blocks.

It is typically hard to anticipate during the design phase the actual data demographics and the actual queries that the data warehouse will serve. On the one hand, an overdue partitioning or re-partitioning can be a very resource-intensive operation for an already-populated table containing many rows. It can be the case that TDFS has to reconstruct all the data blocks in every AMP to match the new partitions definitions. On the other hand, premature partitioning can be misaligned with actual data demographics and changed specifications. This can cause suboptimal performance, which contradicts the aim of performance tuning. In the provided example definition above, the partitioning implicitly assumes that the queries will be specifically interested in item quantities of one, five, or over ten. But do the actual data demographics and queries support this assumption? Even worse, what if the business specifications change and other item quantities are now permissible (e.g., three or eight items)? An attempt to INSERT the new values in the table will fail, as there is no partition defined to cover these item quantities. So, a performance tuning attempt ends up being an unspecified and unwanted table CHECK constraint.

Data representation overheads

One of the primary performance tuning targets is to minimize the data representation overheads. In the Teradata context, the term "compression" has been historically used to describe such approaches, despite not always a compression algorithm being involved.

PDM overheads

The PDM itself must be the first place to consider for minimizing data overheads. The choice of the correct data types should be reassessed to ensure that no space is wasted. One must not forget that the space savings multiply by the total number of rows in the table. Thus, even a one-byte saving in a table definition that will hold billions of rows translates into GB of saved storage needs.

It can be the case that the column data type was defined as per the source system interface definition (i.e., the information container) rather than the actual business content. This, combined with a tendency to round up sizes "just in case", ends up in situations where a table column data type is defined as DECIMAL(38,2) to store two-fractional-digit percentages. A DECIMAL(38, n) consumes 16 bytes of storage; the more appropriate DECIMAL(5, 2) consumes just four bytes, i.e., a 400% space reduction! Similarly, a column holding small whole numbers could be defined as a BYTEINT, consuming a mere one byte of storage instead of four for an INTEGER or eight for a BIGINT.

Another case is character-oriented columns (CHAR and VARCHAR) defined to use a Unicode character set instead of a Latin one but business content exclusively being 7-bit ASCII strings (e.g., "ACTIVE", "CLOSED", "SUSPENDED", and "CANCELLED"). As Teradata uses UTF-16 rather than UTF-8 encoding for representing Unicode characters, the space consumption for such a column halves just by properly defining to use a Latin character set instead.

As the last case, there is no reason for the PDM to obey the source system interface definition, especially when it involves fixed-length, positional files. A column defined as CHAR(500) consumes 500 bytes (Latin character set) or 1,000 bytes (Unicode). If the business content comprises variable-length strings, the column will be padded with as many space characters as needed, wasting storage space and hampering subsequent data processing. A VARCHAR(500) definition would be more appropriate.

We consider a good practice to perform this activity already in the design phase. While this might appear as causing unnecessary delays in the early stages of a data warehouse project, the effort savings that can be achieved outperform the costs to reach the same result late in the implementation or operation of the data warehouse by redefining the PDM.

Nullable table columns

Teradata includes in the physical row header a "*presence byte*". It encodes the presence of NULL values in the stored data (e.g., the table columns, in the case of a table row). The encoding works as follows. The least-significant bit (LSB) is set to 1 always. If a table column is defined as nullable, the next available bit of the presence byte is reserved for the column. When a non-NULL value is to be stored in the column, the reserved bit is set to 0, and the actual value is stored in the dedicated area of the physical row stored data. When a NULL value is to be stored in the column, the reserved bit of the presence byte is set to 1, and nothing is stored in the column itself (i.e., its data structure is skipped). Depending on the column data type and demographics, this can cause noticeable space savings, especially when lengthy data types are involved.

```
CREATE TABLE TheTable
(
PK INTEGER NOT NULL,
TheCol CHAR(10)
) PRIMARY INDEX (PK);
```

First Presence Byte	TheCol
0 0 0 0 0 0 0 1	‚A‘
0 0 0 0 0 0 1 1	NULL
0 0 0 0 0 0 0 1	‚A‘
0 0 0 0 0 0 0 1	‚B‘
0 0 0 0 0 0 1 1	NULL
0 0 0 0 0 0 0 1	‚C‘
0 0 0 0 0 0 0 1	‚C‘
0 0 0 0 0 0 0 1	‚E‘

Figure 19 A table with a nullable column, the presence byte in the physical row header, and the column in the stored data

At least one presence byte is defined per row header. Then, up to seven nullable columns can be defined without additional overhead. From the eighth nullable column onwards, a

second presence byte must be used since the LSB of the first presence byte is reserved by the system. A third presence byte is needed for the 16th nullable column and so on. All these bytes are added in the row header of every physical row. Again, the space savings per NULL value must be compared against the overhead of the presence bytes for every single physical row. This is also a reason to aim to define columns as NOT NULL whenever possible and reduce the number of presence bytes.

Table multi-value compression

Teradata supports multiple forms of compression: *multi-value compression* was introduced in version 5, while *algorithmic* and data *block-level* compression was introduced in version 13.10. Support for multiple compression options for columnar tables was introduced in version 14.

Multi-value compression (MVC) builds atop the concept of presence bytes in the row header by combining with information stored in the *table header*. Each table in Teradata has an associated sub-table, the table header, that contains table metadata. Each AMP that stores rows of a table permanently holds a copy of the related table header in its memory too. The maximum size of a table header is 1 MB.

MVC is a *dictionary-based compression* approach based on *run-length encoding* and works as follows. First, the set of column values to be included in the dictionary is provided in the table definition. The size of the set defines the number of bits needed to encode the address of the dictionary entries: one bit suffices for one value, two bits for up to three values, and eight bits for up to 255 values. One value is always reserved to indicate the presence of a non-compressed value. The dictionary values are then stored in the table header. The row header is also expanded by one presence (or now, MVC) byte to store the encoded dictionary addresses.

```
CREATE TABLE TheTable
(
PK INTEGER NOT NULL,
TheCol CHAR(10) COMPRESS ('A','B','C') NOT NULL
) PRIMARY INDEX (PK);
```

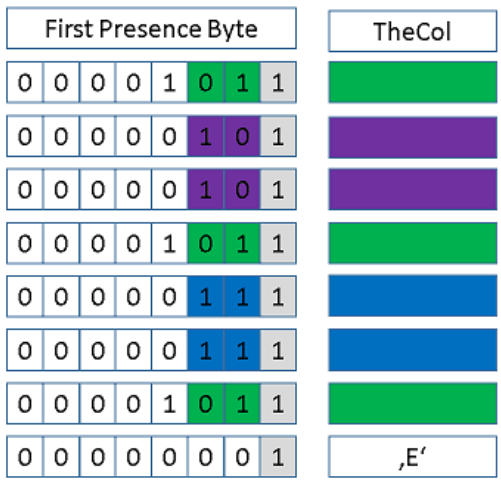

Figure 20 A table definition with one MVC column and three compressed values: ‘ A’ (01), ‘ B’ (11), and ‘ C’ (10) and the three dictionary addresses; byte value 1 indicates a non-dictionary value residing at the stored data

When the column definition allows NULL values to be stored, a combined presence and MVC byte is formed. Here, the NULL presence bit is stored left-most, followed by the MVC encodings. In this case, at most, seven bits of the presence byte can be utilized because the LSB must always be set to 1.

One table definition might contain multiple nullable and not-nullable columns with or without MVC enabled. Teradata efficiently allocates the available address space of the presence byte to multiple columns to reduce the per-row space overhead caused by the presence byte. It can also add more presence bytes once the addressing capacity of the first one is exhausted. For example, the seven bits of the first presence byte can accommodate:

- two nullable columns (two bits in total) and one nullable MVC columns with up to 15 compressed values (five bits in total)
- one nullable column (one bit in total) and one not-nullable MVC columns with up to 31 compressed values (five bits in total)
- one not-nullable MVC column with up to 15 (four bits), one not-nullable, single-MVC column (one more bit), and one nullable, single-MVC column (two bits in total).

```
CREATE TABLE TheTable
(
PK INTEGER NOT NULL,
TheCol CHAR(10) COMPRESS ('A','B','C')
) PRIMARY INDEX (PK);
```

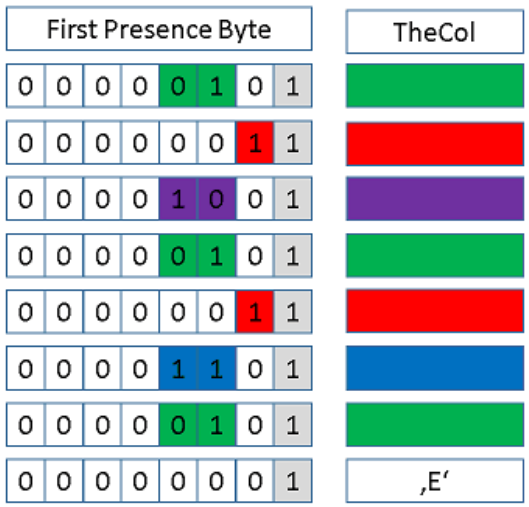

Figure 21 The same table definition but now with a nullable column

Tuning the selection of table columns to be compressed is a multi-objective optimization problem with constraints. One cannot store an indefinite number of dictionary entries in the table header, as the latter has a specific maximum size, including all table metadata. Additional parameters to be considered per each table column are the count of distinct values, the count of NULL values (if any), and the value and data type lengths. These parameters define the per-column storage savings as a function of column nullability and the number of dictionary entries to be created per column.

The savings must then be compared against the presence byte occupancy (i.e., how many presence bytes and bits will this column reserve?) and the row header overhead caused by

the presence bytes in all the table rows. It can be the case that compressing two less popular data values of one column achieves better overall results than compressing the most popular data value of the same or another column. The final output of the calculations will be an optimal subset of columns, the number of dictionary entries thereof, and specific column values.

MVC achieves processing (CPU) as well as storage savings. The less the data blocks a table occupies, the less processing is needed to extract the same amount of information. Furthermore, fewer data blocks result in less caching activities, which are quite CPU-intensive operations, and may also achieve more cache hits. Teradata decompresses MVC tables only when needed; table rows may remain in MVC form through the spool space and up to the very last query execution steps.

Table algorithmic compression

Algorithmic compression (ALC) allows one to define compression and decompression algorithms and apply them to table column data. ALC can perform better than MVC sometimes, especially when the number of unique values is significant. Teradata provides a set of functions for compression and decompression and supports user-defined implementations. The former includes, among others, the `TransUnicodeToUTF8` to store in UTF-8 rather than UTF-16 representation, `TS_COMPRESS` for TIME and TIMESTAMP data types, and `JSON_COMPRESS` for JSON data.

Data block overheads

The physical row size depends on the unavoidable PDM overheads and the MVC and ALC savings (when compression is applied). When the physical row size and the container data block size are not aligned, additional data blocks may be generated for storing just one physical row each time. In the extreme scenario, this can cause both storage and transfers of half-empty data blocks. Assigning a proper `DATABLOCKSIZE` for the data block can improve the situation.

Data block compression

Data block-level compression (BLC) compresses whole data blocks rather than the contents of specific table columns. BLC is an excellent choice to reduce disk space consumption for tables rarely accessed, but it is rather expensive for those tables frequently accessed and changed. The reason is that a BLC data block must be fully uncompressed when transferred to AMP memory for further processing. Moreover, if the same block is needed again in a session, it must be transferred once more from the disk; the in-memory copy cannot be used.

Teradata 14 introduced conditional BLC in the form of *temperature-based BLC*. Data temperature is a metric of data access frequency. The data blocks are classified at cylinder-level granularity (not block) as cold, warm, hot, and very hot. The temperature-based BLC

can be activated for all tables (global setting) or on a table-by-table basis. The DBS control setting `TempBLCTresh` defines which cylinders should be compressed. Its value can be qualitative (i.e., "COLD", "WARM", and "HOT") or quantitative (i.e., an integer between 0 and 100). The first approach defines the highest temperature level to be considered for compression. The 20% least accessed cylinders are considered cold, the 20% most accessed as hot, and anything in between as warm. The second approach defines the upper limit of the percentage of cylinders to be compressed. For example, a value of 30 will cause compressing all the cold cylinders (20%) and 10% of the warm ones.

BLC is a tradeoff between storage and processing; a disk-bound system will benefit from BLC and increased thresholds. A computation-bound system will suffer from additional tasks to compress and decompress all the accessed data blocks. An informed decision is necessary on a system-by-system basis.

Combined compression

Column-value compression (i.e., MVC and ALC) improves query performance and reduces storage space consumption. Date-block compression (i.e., BLC) reduces storage space consumption at the expense of additional processing for compression operations each time the data block is touched. In theory, one can combine column-value and data-block compression. Teradata poses no technical limitation for this. But, from a performance point of view, what would be the benefit of combining these two?

Let's consider a simplified example of inserting ten million rows to each of these four tables:

```
SET QUERY_BAND = 'BLOCKCOMPRESSION=NO;' FOR SESSION;
CREATE SET TABLE Customer_NONE (
    CustomerId BIGINT NOT NULL,
    CategoryCd CHAR(100)
) PRIMARY INDEX ( CustomerId )
;

CREATE SET TABLE Customer_MVC (
    CustomerId BIGINT NOT NULL,
    CategoryCd CHAR(100) COMPRESS ('PLATIN','GOLDEN','SILVER','BRONZE‘)
) PRIMARY INDEX ( CustomerId )
;

SET QUERY_BAND = 'BLOCKCOMPRESSION=YES;' FOR SESSION;
CREATE SET TABLE Customer_BLC (
    CustomerId BIGINT NOT NULL,
    CategoryCd CHAR(100)
) PRIMARY INDEX ( CustomerId )
;

CREATE SET TABLE Customer_BLC_MVC (
    CustomerId BIGINT NOT NULL,
    CategoryCd CHAR(100) COMPRESS ('PLATIN','GOLDEN','SILVER','BRONZE‘)
) PRIMARY INDEX ( CustomerId )
;
```

The four tables are defined with no compression at all (Customer_NONE), column-value compression (Customer_MVC), data-block compression (Customer_BLC), and combined compression (Customer_BLC_MVC). The inserted values for the CategoryCD column are one of “PLATIN”, “GOLDEN”, “SILVER”, and “BRONZE” and MVC fully covers these values (i.e., all are stored in the row header).

Figure 22 depicts the disk I/O operations needed for these insertions. MVC contributes to the more savings, accounting for only 20% when compared to no compression. BLC follows with 50%, while the combined effect (column BLC_MVC) is at the same level as the MVC component, i.e., BLC did not contribute additional savings.

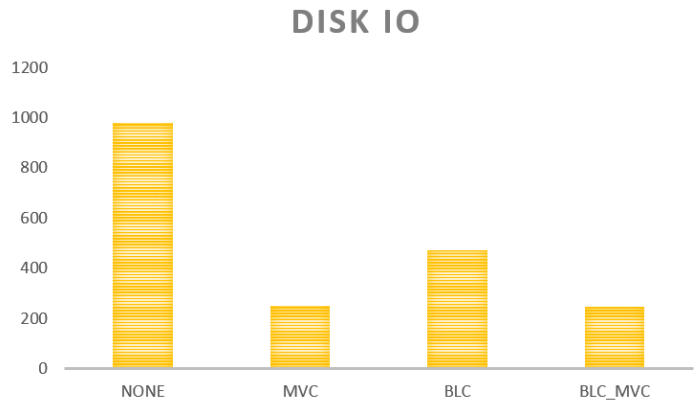

Figure 22 Compression impact on disk I/O

Figure 23 depicts the CPU seconds needed for these insertions. Again, MVC achieves the most savings (about 30%), while both BLC and BLC_MVC need 50% fewer CPU seconds when compared to no compression. The BLC_MVC approach is closer to the BLC one; as these data are still hot, the BLC penalty to compress them for the first time is visible in both approaches.

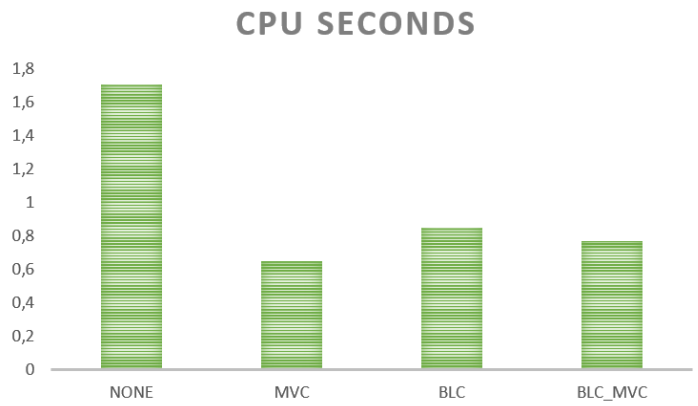

Figure 23 Compression impact on CPU seconds

Figure 24 depicts the disk space occupied by the four tables after the insertions. Not enabling any form of compression has a significant impact on storage space. BLC_MVC achieves most savings (slightly less than 5% the size of the not-compressed table), primarily driven by the BLC component savings (about 5%). MVC alone is less efficient; still, this handpicked selection of values has a tenfold improvement in storage space consumption.

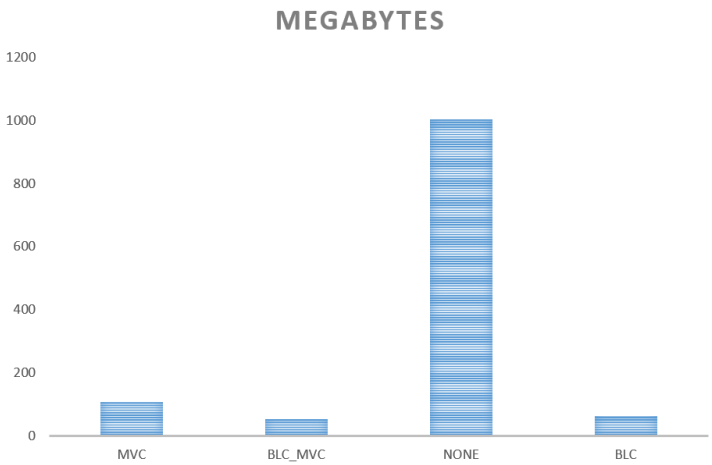

Figure 24 Compression impact (in MB) on disk storage space

The conclusion is that the impact of MVC, BLC, or combined must be carefully assessed for the specific system environment (disk- or CPU-bound) and table usage patterns. BLC is beneficial only for tables rarely accessed; combining with MVC offers little, if any, advantage. Tables regularly accessed can benefit from MVC.

No compression

Typically, little if any information about data demographics is available at the design phase of a data warehouse. A premature selection of tables and columns to apply MVC at this phase can cause an *increase* in storage space requirements during the operational phase. This defeats the sole purpose of utilizing MVC in the first place.

A process to collect necessary metrics and solve the associated multi-objective optimization problem must be defined. The process must be periodically repeated to account for changes in data demographics and ensure optimality. The cost to collect all the necessary metrics is not negligible, both in human effort and system resource consumption; it is a very intensive task.

To reduce the burden of collecting data demographics, one can refrain from manually collecting exact figures and instead utilize the output of a statement like:

```
SHOW STATISTICS VALUES COLUMN <column_name> on <table_name>;
```

Chapter 6 discusses in detail the use of table statistics; what is essential for MVC is that such a statement can be utilized only for those columns already defined to have statistics collected and that the statistics figures must be up-to-date. The statement does not return a full histogram of value frequencies but only the most biased column values. Still, this can be a good starting point for the MVC tuning process. As a minor consideration, one must post-process the text output of the statement as the underlying figures are stored in a binary object (namely, `FieldStatistics`).

We consider a good practice to always perform a cost-benefit analysis for the data warehouse at hand. The output of this analysis will indicate how many and which tables should be considered for MVC and how often the process should be repeated.

Chapter summary

In this chapter, we revisited the enterprise data warehouse design considerations. These include the selection of a data model; logical and physical data model; top-down, normalized vs. bottom-up, dimensional DWH design; data warehouse system architecture and data layers; ETL and ELT approaches; tables, the building blocks of a DWH; relational databases, naming conventions, primary keys, and referential integrity; primary indices and skewness for performant parallel databases; sets and bags; table partitioning; and compression of table contents.

The main takeaway of this chapter is a list of questions to ask before creating a Teradata table:

- Which is the data model to use?
- What are the names and data types (including lengths and characters sets) of the columns?
- Do the table column types match across the database?
- Is there a need to do DBMS-level referential integrity checks?
- Can we define a primary index? Can it be unique? How much skewness does it cause?
- Does the data block size match the table structure?
- What are the envisioned queries and data demographics?
- Are partitions a must-have or a nice-to-have already?
- Is compression a must-have or a nice-to-have already?

Chapter 4: Tuning the data import

"When I use a word," Humpty Dumpty said, in rather a scornful tone, "it means just what I choose it to mean - neither more nor less." "The question is," said Alice, "whether you can make words mean so many different things." "The question is," said Humpty Dumpty, "which is to be master - that's all." – Lewis Carol, Through the Looking-Glass (1935)

In the generally-accepted definition of an *(enterprise) data warehouse (EDWH or DWH)*, data are imported from external (third-party) source systems of the enterprise. In the three-layer data warehouse system architecture, the data stage layer serves as the entry point. From a performance tuning perspective, this is the appropriate time to identify the processing chains as data start to flow into the DWH and ensure that the DWH always serves up-to-date information in the most resource-efficient way.

Data stage approaches

Three generations of data stage approaches have evolved over the years, based on technology capabilities and advancement in RDBMS and SQL. The first generation revolves around the *Extract-Transform-Load* (ETL) approach. There, tools from third-party vendors are used to extract information from disperse sources, connectivity options, and proprietary data formats. As both RDBMS and SQL offered limited functionality to perform complex input calculations and efficiently apply business logic rules, these tools transformed the data into a form convenient for the RDBMS to read and store in a DWH table.

The second-generation revolves around the *Extract-Load-Transform* (ELT or E-LT) approach. There, the use of third-party tools is typically focused on the extraction from source systems. An almost one-to-one load of the source data into a database table is performed using either third-party or RDBMS tools. Then, the transformations occur within the RDBMS environment, utilizing the rich expressiveness of modern SQL variants. This is the contemporary practice, which is also popular in cloud-based MPP systems, further discussed in Chapter 7. Further factors contributing to the shift towards E-LT include the advancements in RDBMS technology (e.g., Teradata natively supports complex formats like JSON, XML, and geospatial data), open data formats, and the evolution of multi-vendor ecosystems within enterprises, often because of mergers and acquisitions.

The third generation relates to *stream processing* of data when they become available to the DWH. Such an approach fits scenarios of (near) real-time delivery of information from source systems. Also, scenarios were data must continuously be integrated into the data warehouse from multiple sources and different time constraints must be respected (e.g., the notion of "out of business hours" spans multiple time zones, and a common time

window for all time zones cannot be defined or is too short to support massive batch processing of all sources at once).

Teradata data import tools

The primary aim of the data stage layer of a Teradata-based DWH is to represent external data in a Teradata database table. Teradata offers both tools (utilities) and application programming interfaces (API's), including Open Database Connectivity (ODBC), to achieve this. Then, third-party tools can utilize either Teradata tools or a Teradata API to interface with a Teradata database table.

The Teradata load utilities can be divided into two groups: the *bulk load* tools import data as fast as possible, bypassing the transient journal. The *transactional load* tools are more failsafe but slower, utilizing the transient journal. The choice depends on the data stage requirements for the specific data warehouse.

Bulk load tools

The bulk load tools comprise `FastLoad` and `MultiLoad`. Both tools import data in chunks of 64 KB. Being bulk in nature, the target table for both utilities cannot have indices, triggers, or referential integrity constraints defined. On the one hand, `FastLoad` is the fastest option to import data. However, it cannot insert data into an already-populated table. The target table must contain no rows.

On the other hand, `MultiLoad` uses an intermediate (temporary) table before inserting the input rows into the target table. Hence, it additionally supports UPDATE, DELETE, and UPSERT (i.e., UPDATE or INSERT) statements and non-empty target tables.

Transactional load tools

The transactional load tools comprise `BTEQ` and `TPump`. `BTEQ` stands for Basic Teradata Query and is the very first Teradata load utility. It is useful for importing a few rows but misses such features as support for concurrent load sessions and deadlock handling. `TPump` was introduced to overcome these limitations. Additionally, it supports rate limiting to control the number of rows imported per second and avoids stressing the tactical operations of the data warehouse.

We use BTEQ to import up to a few thousand rows (always as one transaction, or else it gets prohibitively slow), TPump up to 500,000 rows, and FastLoad and MultiLoad for anything more significant. Depending on the row length, bulk utilities might need to be used already for a smaller number of rows; the number of single-statement transactions and data block operations becomes prohibitively inefficient.

Teradata Parallel Transporter

Each Teradata load utility has its syntax and set of options. Teradata now offers the *Teradata Parallel Transporter (TPT)*, a general-purpose tool that aggregates the functionality of all previous tools under a common scripting language and set of options. Nonetheless, each separate tool remains available and supported. It is only a matter of effort to migrate the existing codebase and personal preference, which of the two options (separate tools or TPT) to follow. We consider a good practice to use the TPT when possible.

Data stage table tuning

Teradata supports all three generations of data staging approaches and provides appropriate table types. The selection of the optimal table type is a performance tuning target. A suboptimal choice can cause severe performance degradation of the DWH processing chains due to unnecessary long load times.

Let's consider the case of bulk load and `FastLoad` in specific. The import consists of two phases. In the *acquisition phase*, any duplicate input is removed before any further processing. The input data are then split into chunks of 64 KB and distributed among the available AMP's in a round-robin fashion. All AMP's in parallel calculate the primary index of the rows contained in their chunks and send the row to the correct AMP. The target AMP stores in a work table all the rows received by the other AMP's.

The second phase is the *application phase*. Here, each AMP sorts the rows of the work table by their row hash and stores them in the data block of the target table. All AMP's must be available during the two phases. Should one or more become unavailable, the whole process must be restarted.

There are some performance implications in such an approach. If the target is a MULTISET table, then duplicate rows will not be provided, as they are removed from the input. If the target is a SET table, then the system will waste a vast amount of resources to detect row uniqueness violations during the final insert, as discussed in Chapter 3, for input already free of duplicate rows. This can be prohibitively expensive when a non-unique primary index is defined for the SET table.

In either case (SET or MULTISET), the input rows are randomly distributed to an AMP, while the target table has a primary index. The latter defines the AMP to store the rows. Hence, it is probable that an AMP-to-AMP transfer must occur for the majority of the input rows during the acquisition phase.

NoPI tables and AMP hash maps

Teradata introduced with version 13.0 the possibility to define tables without a primary index (NoPI). The purpose of this is to improve the performance of FastLoad (and TPump)

operations by reducing the volume of AMP-to-AMP transfers. A minimalistic example of a NoPI table definition is:

```
CREATE TABLE database_name.table_name (
   column_name_1 data_type_1,
   column_name_2 data_type_2
) NO PRIMARY INDEX;
```

A NoPI table is utilized in a FastLoad as follows. During acquisition, Teradata assigns, as before, the first chunk to the first AMP, the second chunk to another AMP, and so on in a round-robin fashion. Then, Teradata calculates the AMP hash bucket (i.e., the 16- or 20-bit DSW in Figure 5) and a 44-bit uniqueness value for the row. When all 44-bit values are exhausted, the next available DSW for the same AMP is used.

This approach keeps the rows in the same AMP as they were initially assigned, avoiding costly AMP-to-AMP transfers. There is also no need to sort the row hashes, as the uniqueness value is, by definition, sorted. The assignment process evenly distributes the rows among the available AMP's and justifies why all AMP's must be available during the whole process. As an additional measure, Teradata maintains a *NoPI hash map* to avoid row redistributions for NoPI tables during reconfiguration and restore operations.

A NoPI table is a perfect target for FastLoad to import bulk data efficiently. Two points must be considered, however. First, one cannot define a NoPI SET table; a SET table always needs a primary index defined. Second, FastLoad will not filter duplicate rows when importing to a NoPI table; if the source contains duplicate rows, the target table will contain them. The aim of a NoPI/FastLoad is to import data as fast as possible. Data quality issues should be addressed at the next step of the processing chain. Chapter 5 discusses usage scenarios beyond FastLoad data staging, where NoPI tables can be used for performance tuning.

Queue tables

Queue tables are similar to other Teradata tables with the additional property they operate like an asynchronous first-in-first-out (FIFO) queue. A minimalistic example of a queue table definition is:

```
CREATE SET TABLE database_name.table_name, QUEUE (
   qits TIMESTAMP(6) NOT NULL DEFAULT CURRENT_TIMESTAMP(6),
   column_name_1 data_type_1,
   column_name_2 data_type_2,
   column_name_3 data_type_3
) NO PRIMARY INDEX;
```

The timestamp column QITS (Queue Insertions Timestamp) indicates when the row was inserted in the queue table unless a specific timestamp is inserted. An INSERT statement adds new rows in the table (i.e., it performs a FIFO push operation). In contrast, a SELECT

statement reads existing rows from the table (i.e., it performs a FIFO peek operation). Rows are dequeued (i.e., a FIFO pop operation) with the statement:

```
SELECT AND CONSUME TOP 1 * FROM database_name.table_name;
```

When no rows exist in the table (i.e., the queue is empty), the transaction enters a wait state until either a row is enqueued or the transaction aborts due to direct (e.g., an ABORT statement is issued) or indirect (e.g., a DROP TABLE statement is issued) intervention.

Queue tables can improve event-driven data staging performance without a need to poll the database for new information periodically. Each Parsing Engine (PE) maintains a FIFO cache of up to 1 MB, 100 queue tables, and 2,000 rows per table. A partial or full cache purge occurs when any of these limits are exceeded. When the FIFO cache cannot be used, the PE resorts to full-table scans. Teradata allows non-FIFO operations on queue tables (i.e., DELETE, MERGE, or UPDATE statements). In this case, the next statement causes a full-table scan to rebuild the FIFO cache in the Parsing Engine (PE).

Load utility tuning

Teradata allows only a few bulk loads to execute, as they consume many resources concurrently. For example, a FastLoad needs three AMP Worker Tasks (AWT) during the acquisition phase and one during the application phase. AWT is a scarce AMP resource and can cause AMP blocking situations. To address this issue, each AMP limits the percentage of AWT that can be assigned to bulk load utilities. The exact number is system-dependent and typically lies in the lower double-digit area.

In this context, despite FastLoad being the fastest option to import data, it can become a bottleneck and cause processing delays. As part of the tuning process, the "loading landscape" must be analyzed to identify which imports are critical to performing via bulk load utilities and which might be possible to offload to transactional ones (i.e., BTEQ or TPump). Such an analysis must be periodically reassessed to validate that the original assumptions still hold. In our experience, any bulk load with consistently less than 100,00 rows is an excellent candidate to switch to a transactional load.

Chapter summary

In this chapter, we explored the design considerations for the data stage processing chain to ensure time- and resource-efficient availability of external data in the DWH environment. These include the different data stage approaches, the Teradata data load utilities (bulk, transactional, and TPT), tuning of data stage tables, including NoPI and queue tables, NoPI AMP hash maps, and how to improve import times by switching from bulk to transactional load utilities.

The main takeaway of this chapter is that NoPI tables contribute to better performance for data staging; the best performance is achieved if they can be combined with FastLoad.

Chapter 5: Tuning the query data retrieval

"There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult." – C. A. R. Hoare

Once the data warehouse is designed and initialized, the tuning attention shifts to the DWH operations. Typically, a Teradata-based data warehouse serves *requests* expressed as SQL statements. During operations, the primary tuning objective is to serve the requests in the *least possible time* and by consuming the *least possible system resources*. Resource consumption becomes a more pressing issue when the requests are grouped into *workloads* that enforce resource usage limits.

Request tuning can be achieved by exploiting the MPP, shared-nothing architecture of Teradata, and its concurrent execution capabilities. For the sake of readability, the example case of an SQL query retrieving data from a table is considered. The same principles equally apply to SQL statements that insert, delete, or update table rows.

Teradata devises an *execution plan* for the SQL statement of a request, i.e., a set of processing steps to reach an answer to the request and the database object locks necessary at each step. A processing step may involve transferring data blocks from an AMP VDisk to its memory or cross-AMP memory transfer of (table) rows. The focus of this chapter is tuning the data block transfers; this is typically the most time-consuming processing step of an execution plan.

Data demographics and static skew

In the DWH design phase, the PDM of a table includes a primary index definition to distribute the table rows across all AMP's evenly. In the operations phase, one must periodically reassess whether the actual data demographics result in a skewed distribution and, if necessary, perform corrective actions. From a performance tuning perspective, there is nothing worse than a *single-AMP sequential table scan*: a whole table stored in just one AMP's VDisk, which is transferred from disk to memory and then sequentially searched for a specific row.

A *static skew* factor check can be performed using a query similar to the one depicted in Figure 25, adjusted for a specific database or set of tables. A factor of zero is hardly ever achievable, and anything less than 10% is typically considered acceptable. On the other hand, a value close to 100% indicates almost single-AMP table storage and typically requires further analysis and corrective actions.

```
SELECT TOP 10
    DATABASENAME
    , TABLENAME
    , SUM(CURRENTPERM) /(1024*1024) AS CURRENTPERM
    , (100 - (AVG(CURRENTPERM)/MAX(CURRENTPERM)*100)) AS SKEW_PERCENT
FROM DBC.TABLESIZEV
GROUP BY DATABASENAME, TABLENAME
ORDER BY SKEW_PERCENT DESC;
```

Teradata Result Set Viewer

Result Set - SQL Editor 1 (1)

	DataBaseName	TableName	CURRE...	SKEW_P...
1	DBC	SysRcvStatJournal	0	10.0
2	DBC	SessionTbl	0	10.0
3	SYSSPATIAL	spatial_ref_sys	0	0.684931...
4	DBC	EventLog	1	0.299401...
5	TD_SYSFNLIB	WeekNumber_Of_Calendar	0	0.0
6	TD_SYSFNLIB	DECODE9	0	0.0
7	TD_SYSFNLIB	XMLSPLIT	0	0.0
8	DBC	TableConstraints_TD15	0	0.0
9	SQLJ	INSTALL_JAR	0	0.0
10	tuning	FTS_STAGE_TABLE	0	0.0

Figure 25 Querying the top 10 tables with static skew

The occupied storage space per AMP of a statically-skewed table can be checked using a query similar to the one depicted in Figure 26.

```
SELECT
    VPROC
    , Cast(CURRENTPERM AS Decimal(15,5)) /(1024*1024) AS CURRENTPERM
FROM DBC.TABLESIZEV
WHERE DATABASENAME= 'DBC'
AND TABLENAME = 'SysRcvStatJournal'
ORDER BY 1;
```

Teradata Result Set Viewer

Result Set - SQL Editor 1 (1)

	Vproc	CURRE...
1	0	0.01953
2	1	0.01562

Figure 26 Example query for static skew storage distribution across AMP's

A *hidden skew* check can be performed using a query similar to the one depicted in Figure 27. This check aims to detect single outlier values of an overall even distribution; such outliers can cause imbalanced work assignments across AMP's. Chapter 6 will further discuss such cases in the context of cross-AMP, in-memory, *dynamic skew*.

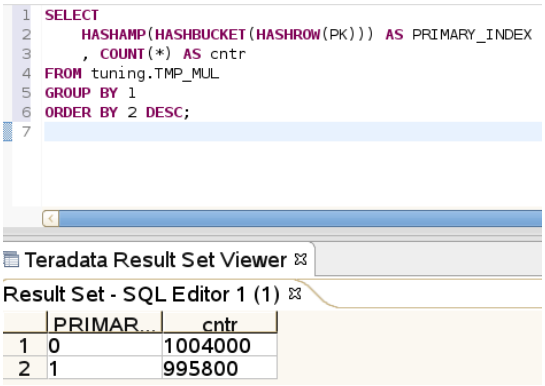

Figure 27 Example query for the primary index hidden skew check

Query demographics and data access paths

Once the data demographics are confirmed and any static skew is addressed, the *query demographics* are analyzed to assess whether the *data access paths* must be tuned. The term "query demographics" collectively refers to the set of SQL queries part of current or planned requests and workloads along with their execution frequency, data volume, and resource consumption metrics (e.g., CPU time, response time, memory footprint, and the number of I/O operations).

The analysis of query demographics must consider any tailor-made views defined over the DWH tables. Access layer views may contain additional data access logic, which might also deviate from the normalized form of the DWH integration area. In our experience, this is an often-neglected area of analysis and the one that raises the most complaints, as the poor performance is directly visible by humans interfacing the DWH.

The term "*data access path*" refers to the approach that Teradata uses to decide which data blocks to transfer from disk to memory. The EXPLAIN statement describes which of the possible data access paths an execution plan will take. A simple example is depicted in Figure 28.

```
EXPLAIN SELECT 1 FROM (SELECT NULL AS X) AS X;

Explanation
---------------------------------------------------------------------
  1) First, we do an INSERT step into Spool 3.
  2) Next, we do a single-AMP RETRIEVE step from Spool 3 (Last Use) by
     way of an all-rows scan into Spool 2 (group_amps), which is built
     locally on that AMP.  The size of Spool 2 is estimated with high
     confidence to be 1 row (22 bytes).  The estimated time for this
     step is 0.00 seconds.
  3) Finally, we send out an END TRANSACTION step to all AMPs involved
     in processing the request.
  -> The contents of Spool 2 are sent back to the user as the result of
     statement 1.
```

Figure 28 An example EXPLAIN of the execution plan for a simple SQL statement

Full-table scan

An *all-AMP sequential search* is a data access path that can always be taken. It equals a *full table scan* (FTS): all AMP's are involved in the search; all their table data blocks are transferred from AMP VDisk to AMP memory (spool), and all data blocks must be sequentially searched for matching rows. From a tuning perspective, *FTS is the least-performing strategy to access data*: it stresses the Teradata system resources, both in terms of I/O operations and memory, to spool all the data blocks.

An SQL query execution plan taking the FTS path should always be analyzed for potential improvement. It could be that FTS is taken because the WHERE clause does not mention the primary index of the table. But it could also be that FTS is indeed the right path to take. One such example is when transforming the rows of a NoPI stage table (see example in Figure 29).

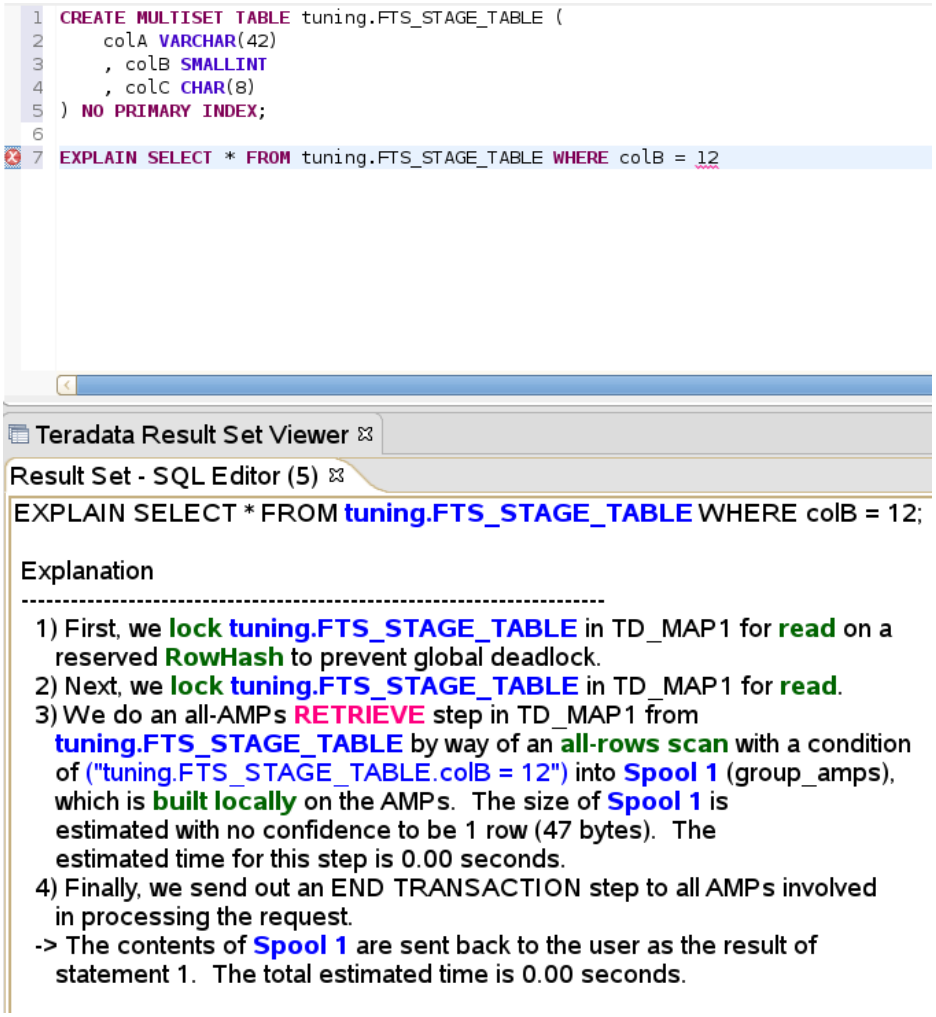

Figure 29 Example of an all-rows scan (FTS) on a NoPI table

Primary index access path

The *single-AMP binary search* lies at the other end of the performance spectrum. It is used when the table has a primary index (PI or NPPI), and the WHERE clause includes all the primary index columns of the table (see example in Figure 30). Then, the *primary index access path* is taken. The Teradata hashing algorithm defines the single AMP that holds the searched table row. The single AMP performs a binary search in its cylinder and master index and transfers from disk to memory only the data block that contains the ROWHASH row. Once in memory, the AMP performs a binary search inside the data block to read the row matching the WHERE condition.

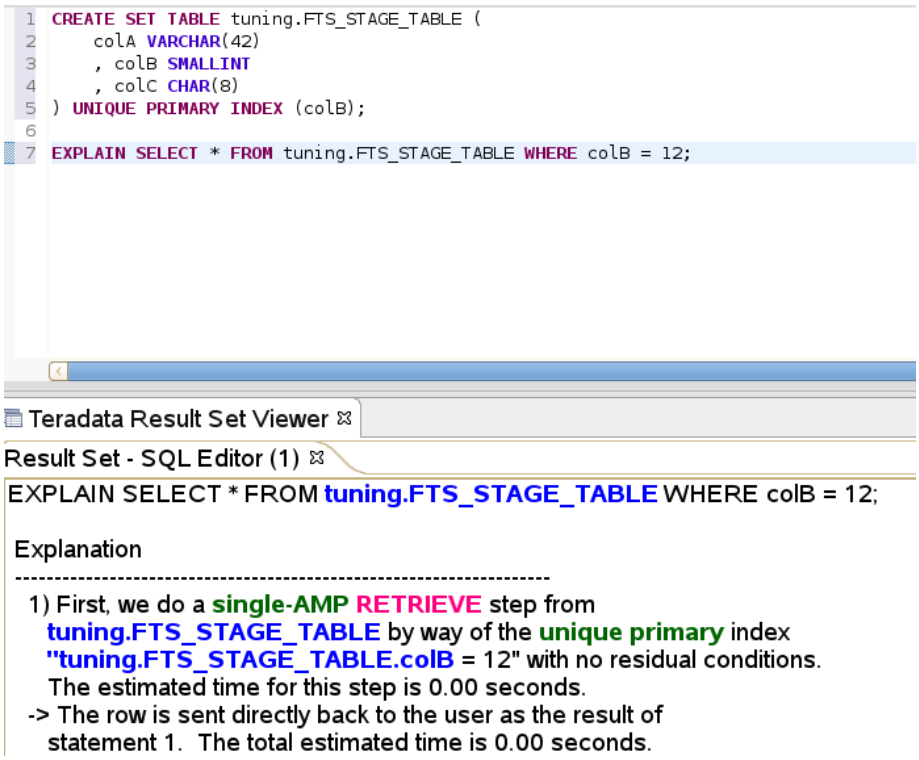

Figure 30 Single-AMP binary search example on a table with a non-partitioned UPI

The primary index access path only needs row-hash-level access locks, allowing more requests for the same table to execute concurrently. When the primary index is unique, no additional AMP memory space must spool the table rows. The primary index path is also taken when the WHERE clause includes additional (residual) conditions to the primary index columns (see example in Figure 31). The residual conditions are then used to filter the rows once the data blocks that contain them are moved to the AMP memory.

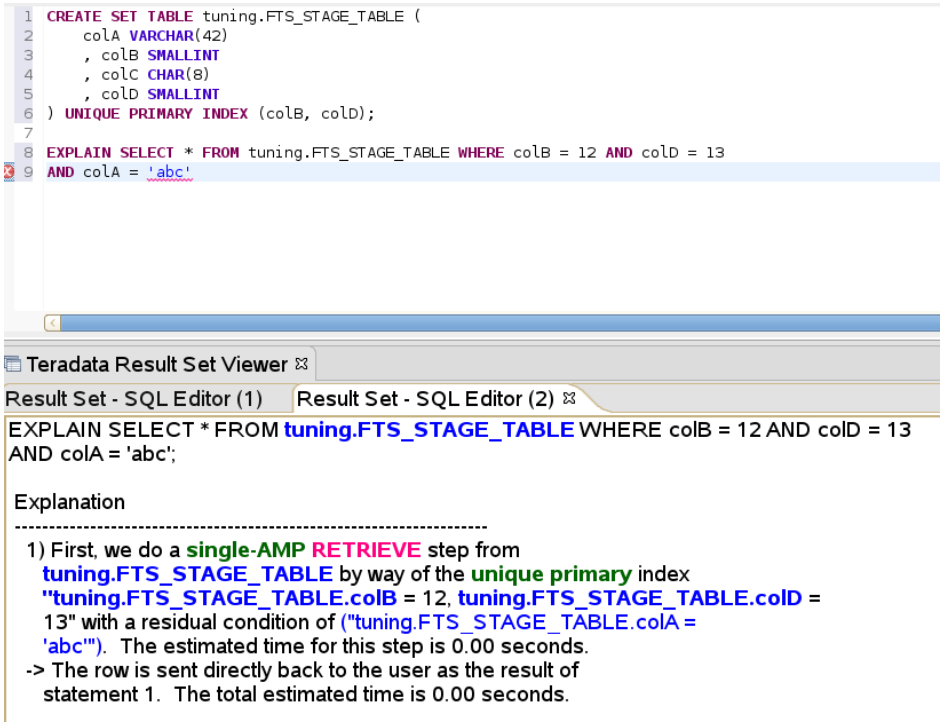

Figure 31 Primary index access path with residual conditions

When one or more columns of the primary index are missing from the WHERE clause, all the AMP's must be involved in the search for the rows (see example in Figure 32). The reason is that any of them may contain rows with a matching ROWHASH as the latter is calculated over all the primary index columns.

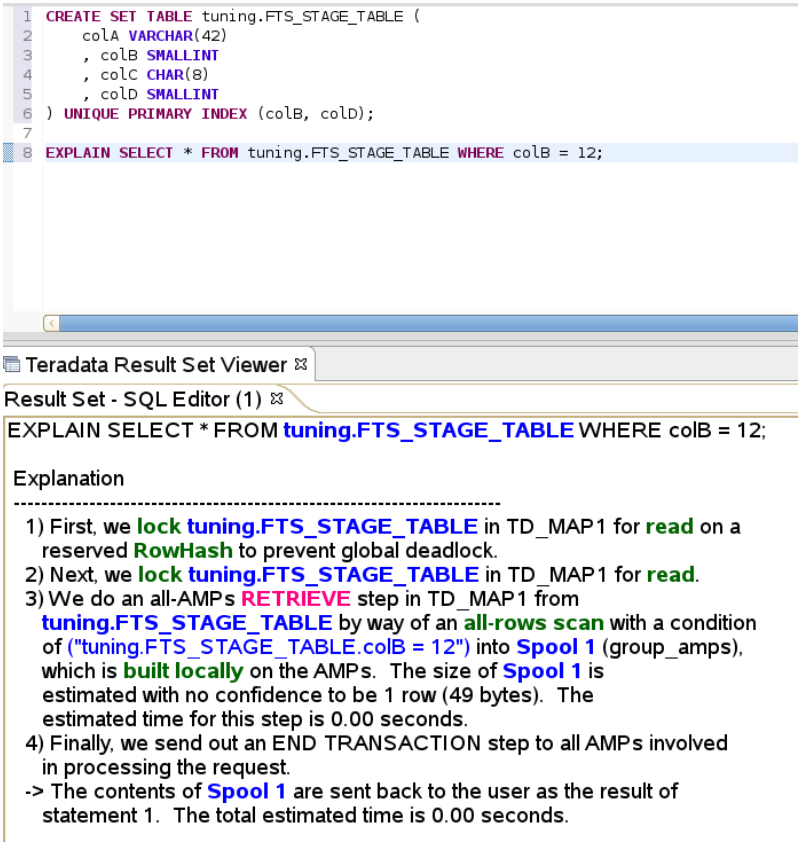

Figure 32 Data access path example not taking the primary index (column missing in the WHERE conditions)

Teradata allows a primary index to consist of up to 64 columns. The more the columns, the less generic will be the SQL queries that can use the primary index access path (contrast the examples in Figure 33 and Figure 34 with the ones in Figure 35 and Figure 36). This is a tuning decision to be made case-by-case depending on the actual queries: when taken, a wider primary index allows less data block transfers and single-AMP processing, but a narrower primary index offers more opportunities to take this access path and achieve single-AMP processing.

```
1  CREATE SET TABLE tuning.FTS_STAGE_TABLE (
2      colA VARCHAR(42)
3      , colB SMALLINT
4      , colC CHAR(8)
5      , colD SMALLINT
6      , colE SMALLINT
7  ) PRIMARY INDEX (colB);
8
9  EXPLAIN SELECT * FROM tuning.FTS_STAGE_TABLE WHERE colB = 12
10 ;EXPLAIN SELECT * FROM tuning.FTS_STAGE_TABLE WHERE colB = 12 AND colD = 24
11 ;EXPLAIN SELECT * FROM tuning.FTS_STAGE_TABLE WHERE colB = 12 AND colD = 24 AND colE = 36
```

Figure 33 Example of the one-column primary index and three SQL queries

```
EXPLAIN SELECT * FROM tuning.FTS_STAGE_TABLE WHERE colB = 12;

 Explanation
 -------------------------------------------------------------------------
   1) First, we do a single-AMP RETRIEVE step from
      tuning.FTS_STAGE_TABLE by way of the primary index
      "tuning.FTS_STAGE_TABLE.colB = 12" with no residual conditions
      into Spool 1 (one-amp), which is built locally on that AMP.  The
      size of Spool 1 is estimated with low confidence to be 1 row (51
      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.

EXPLAIN SELECT * FROM tuning.FTS_STAGE_TABLE WHERE colB = 12 AND colD = 24;

 Explanation
 -------------------------------------------------------------------------
   1) First, we do a single-AMP RETRIEVE step from
      tuning.FTS_STAGE_TABLE by way of the primary index
      "tuning.FTS_STAGE_TABLE.colB = 12" with a residual condition of (
      "tuning.FTS_STAGE_TABLE.colD = 24") into Spool 1 (one-amp), which
      is built locally on that AMP.  The size of Spool 1 is estimated
      with low confidence to be 1 row (51 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.

EXPLAIN SELECT * FROM tuning.FTS_STAGE_TABLE WHERE colB = 12 AND colD = 24 AND colE = 36;

 Explanation
 -------------------------------------------------------------------------
   1) First, we do a single-AMP RETRIEVE step from
      tuning.FTS_STAGE_TABLE by way of the primary index
      "tuning.FTS_STAGE_TABLE.colB = 12" with a residual condition of (
      "(tuning.FTS_STAGE_TABLE.colE = 36) AND
      (tuning.FTS_STAGE_TABLE.colD = 24)") into Spool 1 (one-amp), which
      is built locally on that AMP.  The size of Spool 1 is estimated
      with low confidence to be 1 row (51 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.
```

Figure 34 Primary index path taken for all the three queries

```
1  CREATE SET TABLE tuning.FTS_STAGE_TABLE (
2      colA VARCHAR(42)
3      , colB SMALLINT
4      , colC CHAR(8)
5      , colD SMALLINT
6      , colE SMALLINT
7  ) PRIMARY INDEX (colB, colD, colE);
8
9  EXPLAIN SELECT * FROM tuning.FTS_STAGE_TABLE WHERE colB = 12
10 ;EXPLAIN SELECT * FROM tuning.FTS_STAGE_TABLE WHERE colB = 12 AND colD = 24
11 ;EXPLAIN SELECT * FROM tuning.FTS_STAGE_TABLE WHERE colB = 12 AND colD = 24 AND colE = 36
```

Figure 35 Example of the three-column primary index and three SQL queries

```
EXPLAIN SELECT * FROM tuning.FTS_STAGE_TABLE WHERE colB = 12;

 Explanation
 -------------------------------------------------------------------------
   1) First, we lock tuning.FTS_STAGE_TABLE in TD_MAP1 for read on a
      reserved RowHash to prevent global deadlock.
   2) Next, we lock tuning.FTS_STAGE_TABLE in TD_MAP1 for read.
   3) We do an all-AMPs RETRIEVE step in TD_MAP1 from
      tuning.FTS_STAGE_TABLE by way of an all-rows scan with a condition
      of ("tuning.FTS_STAGE_TABLE.colB = 12") into Spool 1 (group_amps),
      which is built locally on the AMPs.  The size of Spool 1 is
      estimated with no confidence to be 1 row (51 bytes).  The
      estimated time for this step is 0.00 seconds.
   4) Finally, we send out an END TRANSACTION step to all AMPs involved
      in processing the request.
   -> The contents of Spool 1 are sent back to the user as the result of
      statement 1.  The total estimated time is 0.00 seconds.

EXPLAIN SELECT * FROM tuning.FTS_STAGE_TABLE WHERE colB = 12 AND colD = 24;

 Explanation
 -------------------------------------------------------------------------
   1) First, we lock tuning.FTS_STAGE_TABLE in TD_MAP1 for read on a
      reserved RowHash to prevent global deadlock.
   2) Next, we lock tuning.FTS_STAGE_TABLE in TD_MAP1 for read.
   3) We do an all-AMPs RETRIEVE step in TD_MAP1 from
      tuning.FTS_STAGE_TABLE by way of an all-rows scan with a condition
      of ("(tuning.FTS_STAGE_TABLE.colB = 12) AND
      (tuning.FTS_STAGE_TABLE.colD = 24)") into Spool 1 (group_amps),
      which is built locally on the AMPs.  The size of Spool 1 is
      estimated with no confidence to be 1 row (51 bytes).  The
      estimated time for this step is 0.00 seconds.
   4) Finally, we send out an END TRANSACTION step to all AMPs involved
      in processing the request.
   -> The contents of Spool 1 are sent back to the user as the result of
      statement 1.  The total estimated time is 0.00 seconds.

EXPLAIN SELECT * FROM tuning.FTS_STAGE_TABLE WHERE colB = 12 AND colD = 24 AND colE = 36;

 Explanation
 -------------------------------------------------------------------------
   1) First, we do a single-AMP RETRIEVE step from
      tuning.FTS_STAGE_TABLE by way of the primary index
      "tuning.FTS_STAGE_TABLE.colB = 12, tuning.FTS_STAGE_TABLE.colD =
      24
, tuning.FTS_STAGE_TABLE.colE = 36" with no residual
      conditions into Spool 1 (one-amp), which is built locally on that
      AMP.  The size of Spool 1 is estimated with low confidence to be 1
      row (51 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.
```

Figure 36 Primary index path taken only for the third query

Additional data access paths

Teradata supports row partition and secondary index definitions to extend the PDM and define additional data access paths. These paths are to be taken whenever the primary index path cannot be taken to achieve less data block transfers than an FTS would need.

Row partitions

Row partitioning is the primary means to reduce the volume of data transferred from disk to memory. When row partitions are defined for a table (i.e., a PPI table), the table rows are sorted first by the (multi-level) partition expression and then by the ROWHASH of each row. Partition expressions can include integer, date, timestamp, and character data types.

Rows that are partition-wise close are stored in the same or adjacent data blocks rather than spread across all the data blocks.

Partition elimination occurs when the WHERE clause of an SQL query includes all the partition-related columns of a table; then, only the partition data blocks are transferred from disk to memory, as depicted in Figure 37. This results in savings in I/O operations to transfer all table data blocks.

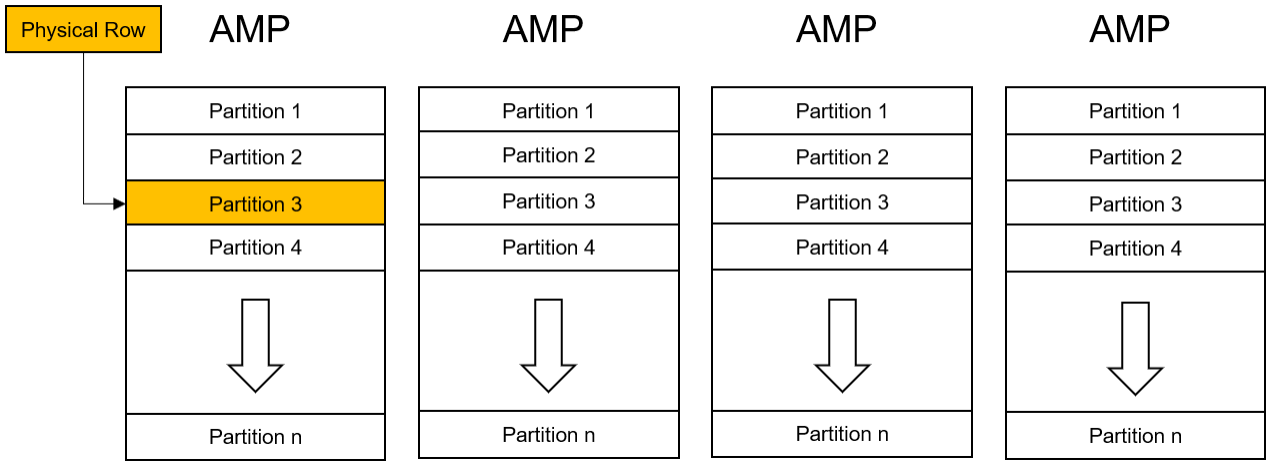

Figure 37 Searching a physical row with partition elimination in action

When the WHERE clause does not include them all or no partition is defined (i.e., an NPPI table), a single-AMP binary search cannot be performed anymore. At the same time, a single AMP will be involved in the search, and all its partition data blocks must be processed, as depicted in Figure 38.

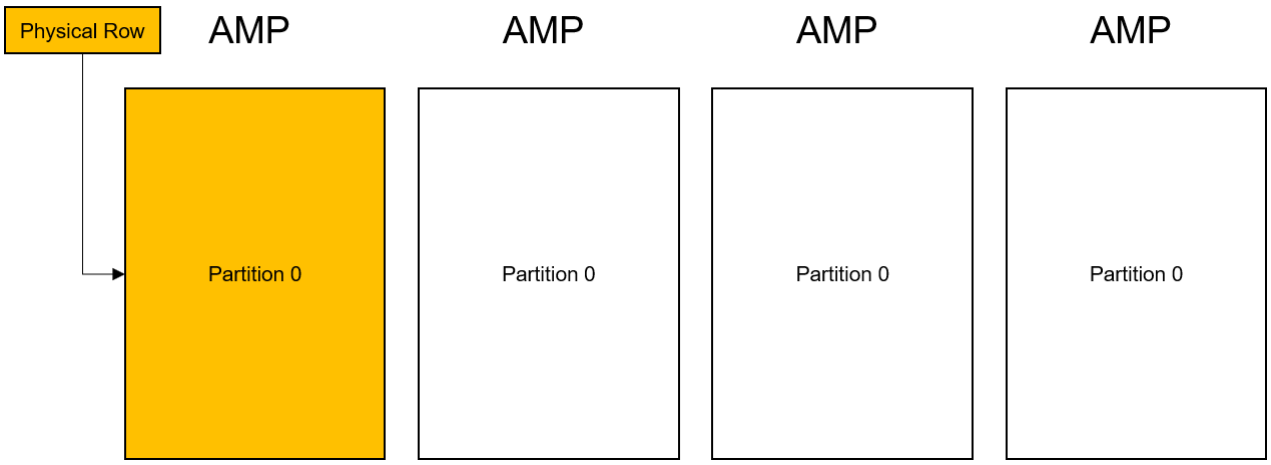

Figure 38 Searching a physical row in an NPPI table - all data blocks involved

Teradata assigns a unique combined partition number internally for each combination of a multi-level partition expression. For example, if the first level has four classes and the second level has five classes, then the table data are split and sorted in 20 partition identifiers.

Partition probing occurs when too fine-grained partitions are defined, while the WHERE clause of the SQL queries is not selective enough to specify the needed partition. Then, the system wastes resources by unnecessarily transferring and process data blocks. This occurs even if the defined partitions have resulted in data blocks being empty or near-empty; still, one data block per partition is transferred to check (probe) whether a row exists and then be excluded from further processing.

There is no one-size-fits-all row partition approach. Instead, one must analyze both the data and query demographics, and decide which approach to apply, including not having row partitions. The approach must be periodically assessed for its assumptions; if the data or query demographics change, a revision of the approach might be necessary. The next sections explore some typical use cases for row partitions.

Partition expression tuning

Teradata supports four types of partition expressions. *Direct partitioning* is the simplest-and-least-flexible of all the expressions. The partitioning column must have an INTEGER data type with values between 1 and 65,533. Two additional values are system reserved to mark a NULL value and a non-matching value. Direct partitioning is susceptible to partition probing.

Modulo partitioning overcomes the limitations of direct partitioning and allows us to define the exact number of needed partitions. A typical usage example is to define the partitions as the output of the expression `HASHBUCKET(HASHROW(column_name)`, as depicted in Figure 39. If the column values are evenly distributed, the partitions will contain an even number of table rows. Such an assumption must always be confirmed with actual data demographics. The artificial example in Figure 39 comes from the country numeric codes defined in ISO 3166-1 standard; the last digit of the code is heavily biased toward even numbers (here, shifted by one, due to the partition expression).

```
1  CREATE TABLE tuning.country_code (
2      country_name VARCHAR(100) CHARACTER SET  Latin NOT CASESPECIFIC
3      , country_2 CHAR(2) CHARACTER SET  Latin NOT CASESPECIFIC
4      , country_num INTEGER
5  ) PRIMARY INDEX (country_name)
6  PARTITION BY (1 + (country_num MOD 10));
7
8  SELECT PARTITION, count(DISTINCT country_NUM)
9  FROM tuning.country_code
10 GROUP BY 1
11 ORDER BY PARTITION;
```

Teradata Result Set Viewer

Result Set - SQL Editor 1 (1)

	PARTITI...	country ...
1	1	46
2	2	8
3	3	41
4	4	8
5	5	48
6	6	6
7	7	39
8	8	3
9	9	46
10	10	4

Figure 39 Example of skewed modulo partitioning

CASE_N and RANGE_N partitioning allow more fine-grained definitions of partitions, covering any column values. The CASE_N partition expression equals a CASE WHEN statement in SQL, as depicted in the example below:

```
CREATE SET TABLE database_name.table_name (
    transaction_number INTEGER,
    transaction_date DATE,
    item_number INTEGER,
    item_quantity INTEGER
) UNIQUE PRIMARY INDEX (transaction_number)
PARTITION BY CASE_N (item_quantity = 1, item_quantity = 5, item_quantity >= 10, NO CASE,
UKNOWN);
```

The list of conditions is evaluated until the first match is found. Teradata internally assigns a partition number (starting from 1) to each condition, effectively treating the conditions as a mapping to modulo partitioning. In the example above, should a value do not fall into any condition, the row is assigned to the NO CASE partition. A second partition, UNKNOWN, is reserved for NULL values. One can merge the two partitions into one using an expression like:

```
CREATE SET TABLE database_name.table_name (
    ...
) UNIQUE PRIMARY INDEX (transaction_number)
PARTITION BY CASE_N (item_quantity = 1, item_quantity = 5, item_quantity >= 10, NO CASE OR
UKNOWN);
```

An SQL INSERT statement will fail when a “NO CASE” partition is not defined and an input value matches no condition. It will also fail when an “UNKNOWN” partition is not defined, and the input value is NULL. It is then DWH-specific if these two cases are covered or not by additional partition(s). If yes, we consider a good practice to periodically check the number of rows falling into these partitions and confirm that it is within the expected ranges.

The RANGE_N partition expression applies to values that can be ordered and referenced as ranges. An internal number is assigned to each range, similarly to the CASE_N expression. Additional “NO RANGE” and “UNKNOWN” partitions can be defined with the same semantics as for a CASE_N expression. Typically, a RANGE_N expression is less processing-intensive compared to a CASE_N one and, thus, preferable.

A RANGE_N limitation is that the ranges must be sorted in the partition definition expression. Further, the ranges must not overlap with each other. With single-level partitions, Teradata detects the error at table creation time, as depicted in Figure 40, where the date June 30, 2020 falls into two value ranges.

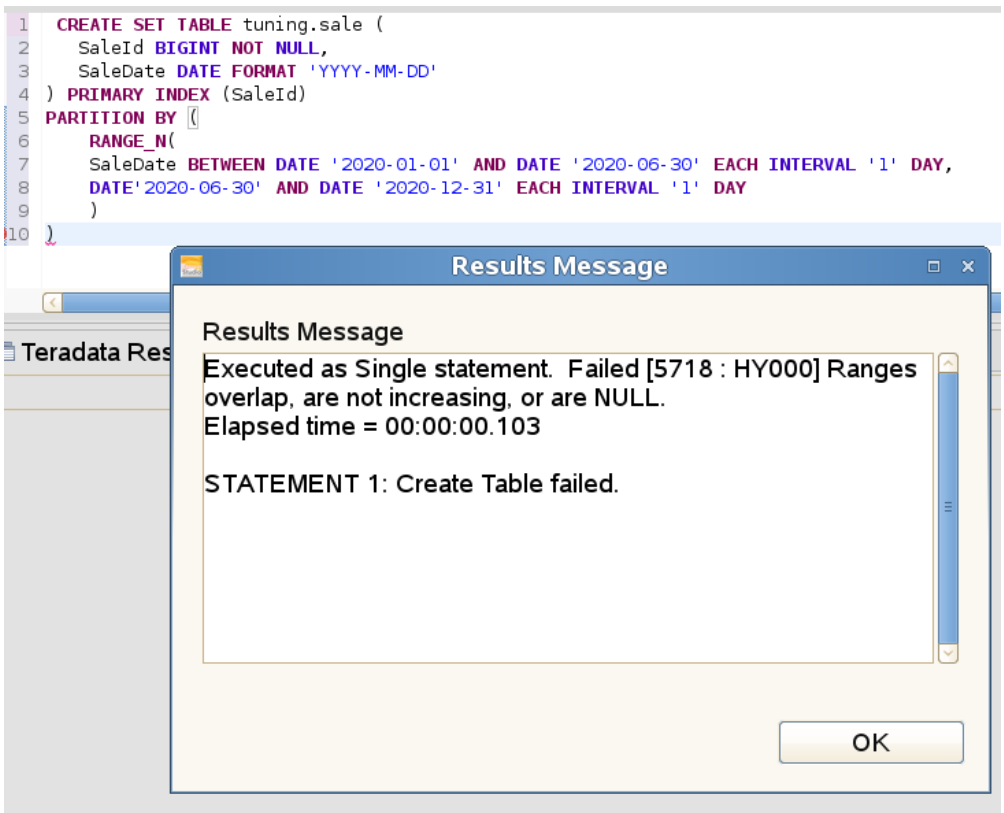

Figure 40 Single-level RANGE_N partitions with overlapping value ranges

With multi-level partitions, Teradata does not detect the error at table creation time, as depicted in Figure 41. While the syntax is almost identical, the CREATE statement defines

two levels of (RANGE_N) partitions, even on the same column. Hence, each level has non-overlapping ranges. However, their combination does have an overlapping range, which Teradata detects during the first INSERT attempt.

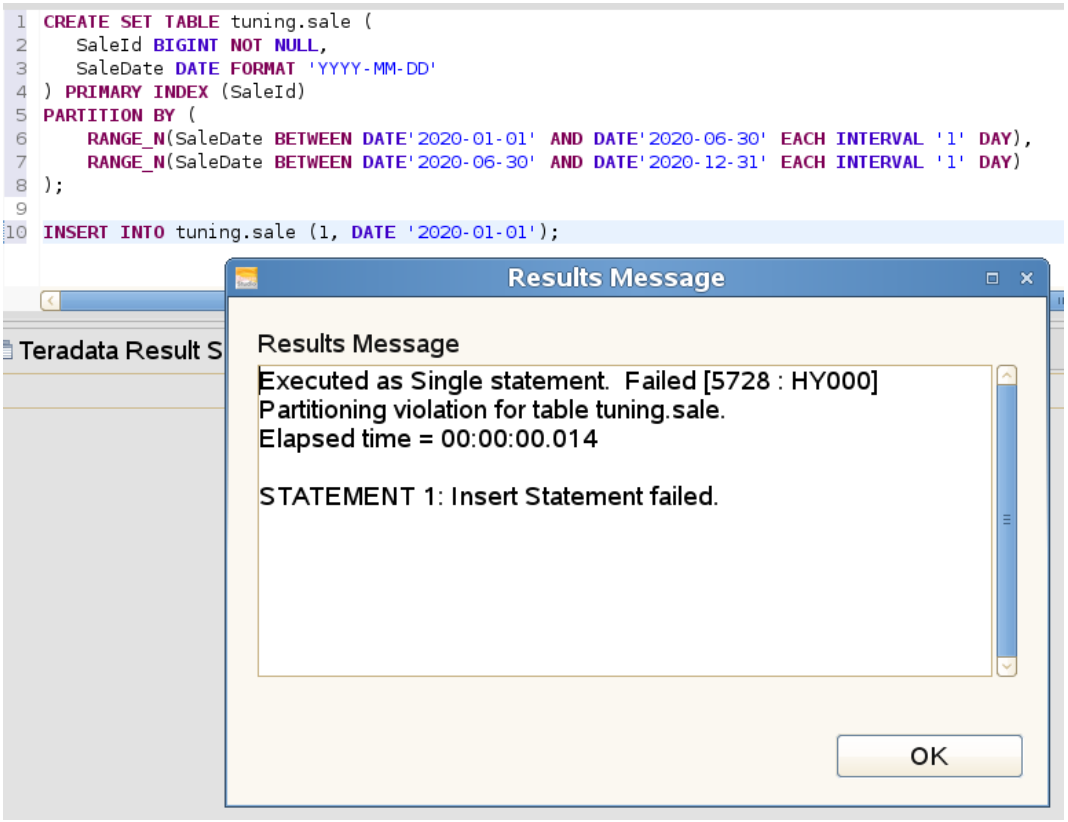

Figure 41 Multi-level RANGE_N partitions with overlapping value ranges

Teradata cannot perform partition elimination when character functions (e.g., substring) are included in the RANGE_N partition expression. Neither when a column is defined as CASE SPECIFIC but a case-insensitive query expression is provided (e.g., searching for "VIENNA" while a case-sensitive CASE_N range "Vienna" is defined).

Analyzing partition contents

Teradata internally maintains a table-level check constraint for the defined partitions of a table. An INSERT or UPDATE statement is checked for constraint violation, and an error is generated when the inputs result in an undefined partition. One can easily confirm the presence of a CHECK constraint by querying the `DBC.PartitioningConstraintsV` table, as in the example depicted in Figure 42.

```
CREATE SET TABLE tuning.Transactions (
    TransId BIGINT NOT NULL,
    TransType INTEGER NOT NULL,
    TransDate DATE FORMAT 'YYYY-MM-DD'
) PRIMARY INDEX (TransId)
PARTITION BY (RANGE_N(TransDate BETWEEN DATE'2020-01-01' AND DATE'2020-03-31' EACH INTERVAL '1' MONTH));

SELECT ConstraintText
FROM DBC.PartitioningConstraintsV
WHERE TABLENAME = 'Transactions'
AND DATABASENAME = 'tuning';
```

Teradata Result Set Viewer

esult Set - SQL Editor (1)

	ConstraintText
1	CHECK ((RANGE_N(TransDate BETWEEN DATE '2020-01-01' AND DATE '2020-03-31' EACH INTERVAL '1' MONTH)) BETWEEN 1 AND 00003)

Figure 42 Partition expression resulting in a three-value CHECK constraint

Evolving data demographics can cause sub-optimal performance, as the number of rows assigned to each partition (i.e., the *partition cardinality*) changes and does not serve the intended purpose anymore. This is especially true in the cases of direct and modulo partitioning. The partition cardinality must be periodically checked against expected values. One approach is to query the table partition expression, as in the example depicted in Figure 43.

```
INSERT INTO tuning.Transactions VALUES (1, 1, DATE '2020-01-15')
;INSERT INTO tuning.Transactions VALUES (2, 1, DATE '2020-01-16')
;INSERT INTO tuning.Transactions VALUES (3, 1, DATE '2020-02-16')
;INSERT INTO tuning.Transactions VALUES (4, 1, DATE '2020-02-16')
;INSERT INTO tuning.Transactions VALUES (5, 1, DATE '2020-03-16')
;INSERT INTO tuning.Transactions VALUES (6, 1, DATE '2020-03-16')
;INSERT INTO tuning.Transactions VALUES (7, 1, DATE '2020-03-16')
;INSERT INTO tuning.Transactions VALUES (8, 1, DATE '2020-03-16')
;

SELECT RANGE_N(TransDate BETWEEN DATE '2020-01-01' AND DATE '2020-03-31' EACH INTERVAL '1' MONTH) AS PART,
    count(*) as num_rows
FROM tuning.Transactions
GROUP BY PART
ORDER BY PART;
```

Teradata Result Set Viewer

Result Set - SQL Editor (1)

	PART	num_rows
1	1	2
2	2	2
3	3	4

Figure 43 The cardinality of the three partitions after eight INSERT statements

A simpler syntax is to use the PARTITION keyword, as in the example depicted in Figure 44.

```
INSERT INTO tuning.Transactions VALUES (1, 1, DATE '2020-01-15')
;INSERT INTO tuning.Transactions VALUES (2, 1, DATE '2020-01-16')
;INSERT INTO tuning.Transactions VALUES (3, 1, DATE '2020-02-16')
;INSERT INTO tuning.Transactions VALUES (4, 1, DATE '2020-02-16')
;INSERT INTO tuning.Transactions VALUES (5, 1, DATE '2020-03-16')
;INSERT INTO tuning.Transactions VALUES (6, 1, DATE '2020-03-16')
;INSERT INTO tuning.Transactions VALUES (7, 1, DATE '2020-03-16')
;INSERT INTO tuning.Transactions VALUES (8, 1, DATE '2020-03-16')
;

SELECT PARTITION AS PART, COUNT(*) AS NUM_ROWS, COUNT(DISTINCT TransDate) AS NUM_VALUES
FROM tuning.Transactions
GROUP BY PART
ORDER BY PART;
```

Teradata Result Set Viewer

Result Set - SQL Editor (1)

	PART	NUM_ROWS	NUM_VALUES
1	1	2	2
2	2	2	1
3	3	4	1

Figure 44 Number of rows and distinct values per partition after eight INSERT statements

This syntax can be expanded to cover the case of multi-level partitions, using the PARTITION#Ln keyword, where n is the level to be considered, as in the example depicted in Figure 45.

```
CREATE SET TABLE tuning.Transactions (
    TransId BIGINT NOT NULL,
    TransType INTEGER NOT NULL,
    TransDate DATE FORMAT 'YYYY-MM-DD'
) PRIMARY INDEX (TransId)
PARTITION BY (
    RANGE_N(TransDate BETWEEN DATE '2020-01-01' AND DATE '2020-03-31' EACH INTERVAL '1' MONTH),
    CASE_N(TransType <=2, TransType > 2)
);

SELECT PARTITION#L1 AS PART1, COUNT(*) AS NUM_ROW
FROM tuning.Transactions
GROUP BY PART1
ORDER BY PART1;

SELECT PARTITION#L2 AS PART2, COUNT(*) AS NUM_ROW
FROM tuning.Transactions
GROUP BY PART2
ORDER BY PART2;
```

Teradata Result Set Viewer

Result Set - SQL Editor (1)

	PART2	NUM_R...
1	1	8

Figure 45 Number of rows per first- and second-level partition level after eight INSERT statements

Historical data

Date ranges are a typical example of row partitioning in a DWH that stores historical data. A RANGE_N expression is used to assign rows in different partitions. Depending on the usage patterns, expressions can be defined, so older dates are grouped in a few partitions, while more recent dates are spread in more partitions. Such an approach can achieve a good balance between the overall number of partitions (less for better performance) and the volume of transferred data blocks for frequently-accessed rows (less data block transfers for better performance).

Let's consider a use case where the query demographics for a table reveal that table rows are mainly accessed with daily granularity for the current year but only with monthly for last year, and hardly ever for previous years. Then, a partition expression can be defined as in the example depicted in Figure 46 below.

```
CREATE SET TABLE tuning.sale (
    saleid BIGINT NOT NULL,
    saledate DATE FORMAT 'YYYY-MM-DD'
) PRIMARY INDEX (saleid)
PARTITION BY RANGE_N(
    saledate BETWEEN
    DATE '2019-01-01' AND DATE '2019-12-31' EACH INTERVAL '1' MONTH,
    DATE '2020-01-01' AND DATE '2020-12-31' EACH INTERVAL '1' DAY,
    NO RANGE OR UNKNOWN
);
```

Figure 46 A single-level, multi-granularity RANGE_N partition definition

Primary index and partition expressions

When the WHERE clause of a query includes the table's primary index, a single-AMP search will be performed, as the primary index access path is taken. The degree of partition elimination defines the number of partitions to be probed for data block transfers. If the primary index definition does not include all the columns of the partition expressions, then partition probing must take place. With few partitions defined, the probing overhead is negligible. In the case that many partitions are not eliminated, the overhead becomes noticeable.

When the primary index of the table is not mentioned in the WHERE clause of a query, an All-AMP search must be performed. Here, partition elimination, if applicable, can achieve better performance compared to a full-table scan. All the available AMP's will be involved in searching for matching rows, but each will look in a subset of its data blocks, depending on the clauses of the query.

Adapting to changing demographics

Teradata allows us to add or delete partitions at the beginning and the end of the partition expression. This fits nicely with the case of historical data partitioning, where old data are

dropped or aggregated (beginning of the partition expression), and new date ranges are introduced (end of the partition expression).

Drop an existing range

The DROP RANGE_N syntax of the ALTER TABLE statement can delete a range created by a RANGE_N partition expression. When table rows are already assigned in the partition to be dropped (including the NO RANGE and UKNOWN ones), then the DROP RANGE must also define the action to be taken for those rows: either DELETE or INSERT INTO another table, as depicted in the examples below:

```
ALTER TABLE Sale
MODIFY PRIMARY INDEX
DROP RANGE BETWEEN DATE '2020-07-01' AND DATE '2020-12-31' EACH INTERVAL '1' DAY WITH DELETE;

ALTER TABLE Sale
MODIFY PRIMARY INDEX
DROP RANGE BETWEEN DATE '2020-07-01' AND DATE '2020-12-31' EACH INTERVAL '1' DAY WITH INSERT
INTO SaleToClean;
```

The table for the latter case must already exist (i.e., created before the ALTER TABLE statement executes) and can have a different primary index and partitioning. Teradata transparently optimizes the RANGE_N expression after a DROP RANGE_N is applied. In the example below, Teradata created one (merged) partition covering the full date range.

```
/* starting RANGE_N */
DATE'2020-01-01' AND DATE'2020-06-30' EACH INTERVAL '1' DAY,
DATE'2020-07-01' AND DATE'2020-12-31' EACH INTERVAL '1' DAY,
DATE'2021-01-01' AND DATE'2021-12-31' EACH INTERVAL '1' DAY

/* DROP RANGE_N expression */
DROP RANGE BETWEEN
DATE'2020-01-01' AND DATE'2020-03-31' EACH INTERVAL '1' DAY WITH DELETE;

/* resulting RANGE_N */
DATE '2020-04-01' AND DATE '2021-12-31' EACH INTERVAL '1' DAY
```

The DROP RANGE_N syntax of the ALTER TABLE requires an exclusive lock to operate. When the to-be-dropped partitions contain many rows, the table is locked for a long time. We consider a good practice first to DELETE the rows of the partitions and only then DROP the emptied partitions. In this approach, only a table-level access lock (on partition level since Teradata 15.10) is required for the time-consuming DELETE step, and the table remains accessible for concurrent access for most of the time.

Add a new range

The ADD RANGE_N syntax of the ALTER TABLE statement can add a range created by a RANGE_N partition expression. Partitions can be added at the beginning or the end of the

partition expression. The resulting ranges need not be adjacent to each other (e.g., in a date range, there can be gap dates).

A typical use case for ADD RANGE_N is table partitioning based on a rolling date (e.g., tracking sales per day for the current year). There is a tuning challenge to balance the need to perform frequent table maintenance for adding new ranges (e.g., every workday) and devising an expression that results in partitions that remain empty for long times (e.g., for half a year).

We consider a good practice in such cases to use the expression CURRENT_DATE for the initial range definition and periodically update it, as in the example below:

```
/* initial range definition for table <tablename> */
DATE '2020-01-01' AND CURRENT_DATE + INTERVAL '365' DAY EACH INTERVAL '1' DAY

/* periodic update (one of the three) */
ALTER TABLE <tablename> TO CURRENT;
ALTER TABLE <tablename> TO CURRENT WITH DELETE;
ALTER TABLE <tablename> TO CURRENT WITH INSERT INTO <backup_table>
```

The CURRENT_DATE is replaced with the actual date on which the ALTER TABLE was last executed. This piece of information is available in column `DBC.IndexCOnstraintsV.ResolvedCurrent_Date` (Figure 47).

```
1  CREATE SET TABLE tuning.sale (
2      SaleId BIGINT NOT NULL,
3      SaleDate DATE FORMAT 'YYYY-MM-DD'
4  ) PRIMARY INDEX (SaleId,SaleDate)
5  PARTITION BY RANGE_N(SaleDate  BETWEEN DATE '2020-01-01' AND CURRENT_DATE + INTERVAL '365' DAY EACH INTERVAL '1' DAY
6  );
7
8  SELECT ResolvedCurrent_Date
9  FROM DBC.IndexCOnstraintsV
10 WHERE DATABASENAME = 'tuning'
11 AND TABLENAME = 'sale';
```

Teradata Result Set Viewer

Result Set - SQL Editor (1)

	ResolvedCurrent_Date
1	2020-07-31

Figure 47 Resolving CURRENT_DATE of an ALTER TABLE statement

UPI with row partitioning limitation

Teradata does not allow a unique primary index (UPI) definition for a table when not all the partition columns are part of the primary index as well (Figure 48). This is a technical limitation to benefit performance. The driving reason for this decision is that otherwise, all partitions could potentially store a specific ROWHASH. Then, it would be prohibitively expensive to check for duplicates at row insertion time.

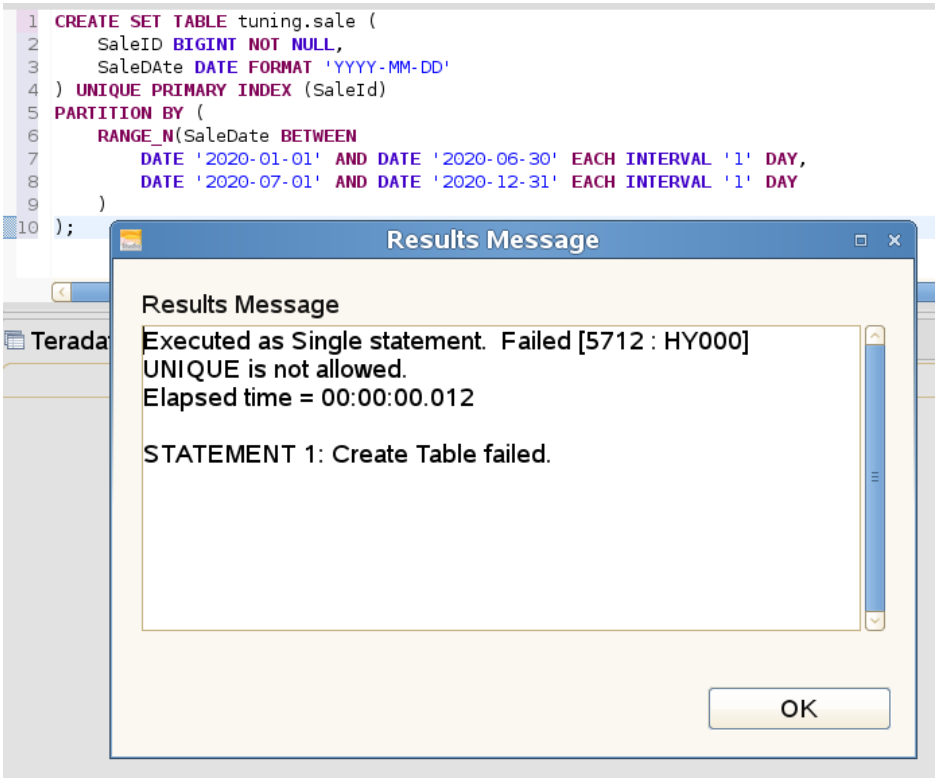

Figure 48 Error when defining a UPI without all partition columns

Secondary indices (USI and NUSI)

Secondary indices can improve performance without deviating from the Logical Data Model (LDM). This is an advantage over row partitions. On the other hand, a secondary index is stored in sub-tables (data blocks) separately from the base table ones. A secondary index occupies additional storage space and must be maintained each time a row of the base table changes. When the base table is defined as FALLBACK (the default since Teradata 16.10), then the secondary index is also FALLBACK-protected, doubling the needed storage space. A secondary index may require significantly more storage space than its base table itself, especially when block-level compression is not applied.

Teradata allows us to define up to 32 secondary indices per base table. Teradata supports Unique Secondary Index (USI) and Non-Unique Secondary Index (NUSI) definitions.

Unique Secondary Index (USI)

Teradata implements a USI for a (base) table as a (sub-) table with its TableID. The sub-table rows are distributed to AMP's by applying the Teradata hashing algorithm to the USI columns. This is similar to how the (base) table rows are distributed based on the Primary Index (PI) columns. The (base) table and sub-table rows are distributed to different AMP's, since the PI and USI column sets are different.

Two AMP's and two data blocks must be involved to access a base table row through a USI access path: one data block for the index row and one for the base row. The steps to retrieve a base table row (e.g., the one in Figure 49) are:

1. Teradata applies the hashing algorithm to the USI columns and derives the AMP for the sub-table.
2. The sub-table AMP consults its Master- and Cylinder Index to locate and transfer the data block with the indexed row.
3. The sub-table AMP extracts from the USI index row the ROW_ID of the base table row (Figure 50)
4. The sub-table AMP consults the hash map using the ROW_ID to derive the (base) table AMP of the searched row.
5. It suffices for the (base) table AMP to first locate in its Master- and Cylinder Index and transfer only the specific data block that contains the searched table row (Figure 51).

```
1  CREATE TABLE tuning.customer (
2      CustomerId BIGINT NOT NULL,
3      CustomerName VARCHAR(255) NOT NULL,
4      SocialSecurityNumder VARCHAR(255) NOT NULL
5  ) PRIMARY INDEX (CustomerId);
6
7  CREATE UNIQUE INDEX (SocialSecurityNumder) ON tuning.customer;
8
9  COLLECT STATISTICS COLUMN(CustomerId), COLUMN(SocialSecurityNumder) ON tuning.customer;
10
11 SELECT * FROM tuning.customer WHERE SocialSecurityNumder = '50';
```

Figure 49 Example of table definition with both a primary and a secondary index access path definition

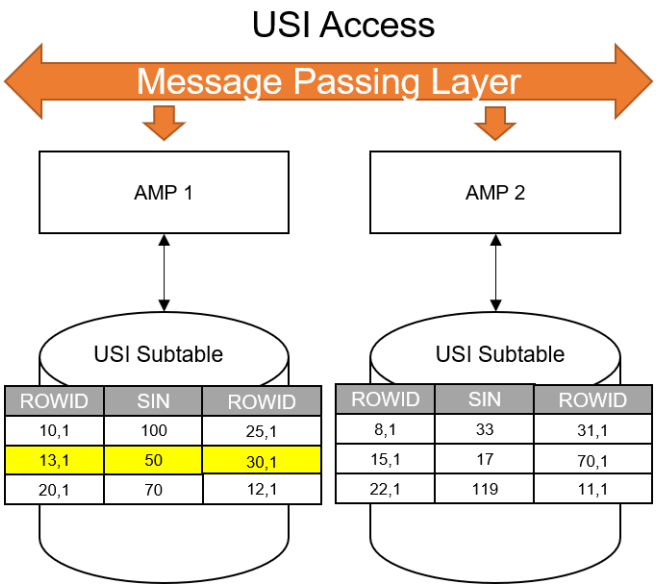

Figure 50 Locate base table ROW_ID through a USI access path (Steps 1-3)

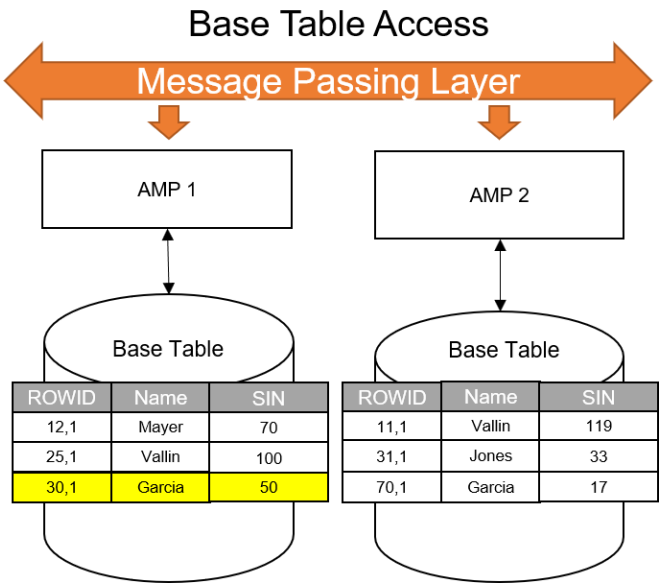

Figure 51 Direct access to base table row using the ROW_ID (Steps 4-5)

Although there are multiple steps involved in a USI access path, the overall performance is close to the one of a primary index access path. Among all possible access paths, Teradata always takes the USI unless a PI can be taken.

Usage scenarios

A unique secondary index is typically defined to provide an additional access path for specific queries that cannot utilize the primary index or the row partition access paths. One such case is when surrogate keys are utilized, but some queries need to access a table through its natural keys. Then, a USI on the natural key columns is an easy means to tune the performance of those queries.

A USI can be defined over a non-unique primary index (NUPI) set of columns to enforce value uniqueness on these rows. Such an approach achieves ROWHASH locking, which is more fine-grained than partition locking needed otherwise. Then, concurrency may improve, as a more significant portion of the table remains accessible during an SQL statement execution.

A USI definition allows us to bypass costly duplicate row checks on SET tables. Teradata implicitly enforces each unique constraint with a unique index. If a UPI is available, it is utilized to check for duplicates (Figure 52). If it is not available, Teradata transparently adds a USI when creating the table (Figure 53). This must be considered when defining the PDM, as there is an implicit impact on performance and storage space.

```
1  CREATE SET TABLE tuning.customer (
2      CustomerId INTEGER NOT NULL,
3      CustomerNumber INTEGER NOT NULL,
4      CONSTRAINT primary_key PRIMARY KEY (CustomerId),
5      SegmentId INTEGER NOT NULL,
6      GroudId INTEGER NOT NULL
7  ) UNIQUE PRIMARY INDEX (CustomerId);
8
9  SELECT DATABASENAME,TABLENAME,INDEXNUMBER,INDEXTYPE,INDEXNAME,COLUMNNAME,INDEXMODE
10 FROM DBC.INDICESV
11 WHERE DATABASENAME = 'tuning'
12 AND TABLENAME = 'customer';
```

Teradata Result Set Viewer

Result Set - SQL Editor (1)

	DatabaseName	TableName	IndexNumber	IndexType	IndexName	ColumnName	IndexMode
1	tuning	customer	1	K	primary key	CustomerId	null

Figure 52 A table UPI used for duplicate checks, no USI defined

```
1  CREATE SET TABLE tuning.customer (
2      CustomerId INTEGER NOT NULL,
3      CustomerNumber INTEGER NOT NULL,
4      CONSTRAINT primary_key PRIMARY KEY (CustomerNumber),
5      SegmentId INTEGER NOT NULL,
6      GroudId INTEGER NOT NULL
7  ) UNIQUE PRIMARY INDEX (CustomerId);
8
9  SELECT DATABASENAME,TABLENAME,INDEXNUMBER,INDEXTYPE,INDEXNAME,COLUMNNAME,INDEXMODE
10 FROM DBC.INDICESV
11 WHERE DATABASENAME = 'tuning'
12 AND TABLENAME = 'customer';
```

Teradata Result Set Viewer

Result Set - SQL Editor (1) | Result Set - SQL Editor (2)

	Databas...	TableName	IndexNumber	IndexType	IndexName	ColumnName	IndexMode
1	tuning	customer	1	P	null	CustomerId	null
2	tuning	customer	4	K	primary key	CustomerNumber	H

Figure 53 A table USI is added for duplicate checks (IndexType=K), distributed by hash value (IndexMode=H)

A secondary index can be added to or removed from a base table on demand, making it suitable for quickly testing the impact on an SQL query. In contrast, a change in the primary index or row partition typically requires the base table to be copied and populated again.

We consider a good practice to periodically check the defined secondary indices and drop them when there is no query utilizing them. Technically, this can be checked with a query similar to the one in Figure 54. In the example, a NUSI is defined (UniqueFlag = 'N'), which is never accessed (LastAccessTimestamp is NULL). It is an excellent candidate to drop and reclaim the storage space.

```
1 SELECT Databasename,Tablename,UniqueFlag,IndexMode,AccessCount,LastAccessTimeStamp
2 FROM DBC.IndicesV
3 WHERE IndexType = 'S'
4 AND DATABASENAME = 'tuning'
5 AND TABLENAME = 'TMP_MUL';
```

Teradata Result Set Viewer

Result Set - SQL Editor 1 (1)

	Databas...	TableName	UniqueFlag	IndexMode	AccessC...	LastAcce...
1	tuning	TMP_MUL	N	L	null	null

Figure 54 Example query for tables with secondary indices

DML statements

The benefit of a USI must be assessed, accounting also the cost to maintain it after each DML statement is applied to the base table. An INSERT or DELETE statement against a table with a USI additionally causes (a) a USI data block transfer to memory, (b) a "before image" entry in the Write-Ahead Log (WAL), (c) a change in the data block, (d) an "after image" entry in the WAL, (e) a data block transfer to disk, and (f) a Cylinder Index update and transfer to disk.

An UPDATE statement that does not involve a primary index column change requires five I/O operations for the base table. If the UPDATE affects one or more USI columns, then ten more I/O operations are required for the sub-table. If the primary index columns are touched, the distribution changes: ten I/O operations for the base table and five I/O operations for the sub-table. Still, the overall number remains the same: 15 I/O operations. Teradata groups the USI updates at the data block level rather than at the row-level and logs the changes in the Transient Journal at the data block level. Nonetheless, USI changes are first redistributed from the table AMP's to the sub-table AMP's.

Bulk data imports

The Teradata transactional load tools are not limited by the presence or absence of a USI at the expense of row-by-row processing. On the other hand, the high-performing Teradata bulk load tools (e.g., FastLoad) cannot import data into a USI table. As a secondary index can be dropped and recreated fast, a typical approach is first to drop all secondary indices, then perform the bulk import, and finally recreate the secondary indices.

The drop-bulk import-recreate approach must be assessed case-by-case. Maybe other queries are using the indices to be dropped and, thus, block the loading process. It is also possible with this approach that duplicate rows are loaded in the DWH as the USI is dropped. When this happens, the USI cannot be recreated (a check violation occurs), and the issue must be solved so the DWH can return to regular operation.

NUPI-to-UPI table conversion

This section is devoted to a particular usage scenario for USI: how to perform an in-place conversion of a non-unique primary index (NUPI) table with data into a table with a unique primary index (UPI) retaining its data. This is useful when the table cannot be copied and recreated, for example, due to space constraints.

Teradata does not allow to change the set of the primary index columns when the table already contains data (Figure 55). This limitation can be bypassed as follows. First, a USI is defined for the to-be primary index columns (Figure 56). If the step succeeds, the respective column values are guaranteed to be unique per row. Second, an ALTER TABLE is issued to convert from the as-is NUPI to the to-be UPI table. Then, Teradata not only converts the table to a UPI one but also silently drops the not-anymore-needed USI one (Figure 57).

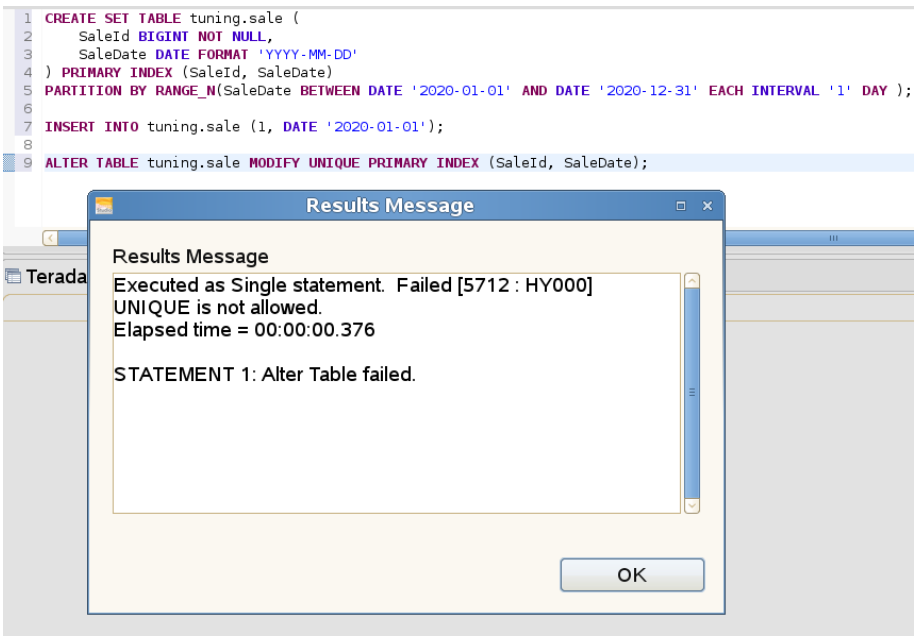

Figure 55 Error altering the primary index of an already-populated table

```
CREATE UNIQUE INDEX (SaleId, SaleDate) ON tuning.sale;

SHOW TABLE tuning.sale;

CREATE SET TABLE tuning.sale ,FALLBACK ,
     NO BEFORE JOURNAL,
     NO AFTER JOURNAL,
     CHECKSUM = DEFAULT,
     DEFAULT MERGEBLOCKRATIO,
     MAP = TD_MAP1
     (
      SaleId BIGINT NOT NULL,
      SaleDate DATE FORMAT 'YYYY-MM-DD')
PRIMARY INDEX ( SaleId ,SaleDate )
PARTITION BY RANGE_N(SaleDate  BETWEEN DATE '2020-01-01' AND DATE '2020-12-31' EACH INTERVAL '1' DAY )
UNIQUE INDEX ( SaleId ,SaleDate );
```

Figure 56 Step 1: Define a unique secondary index for the table

```
ALTER TABLE tuning.sale MODIFY UNIQUE PRIMARY INDEX (SaleId, SaleDate);

SHOW TABLE tuning.sale;

CREATE SET TABLE tuning.sale ,FALLBACK ,
     NO BEFORE JOURNAL,
     NO AFTER JOURNAL,
     CHECKSUM = DEFAULT,
     DEFAULT MERGEBLOCKRATIO,
     MAP = TD_MAP1
     (
      SaleId BIGINT NOT NULL,
      SaleDate DATE FORMAT 'YYYY-MM-DD')
UNIQUE PRIMARY INDEX ( SaleId ,SaleDate )
PARTITION BY RANGE_N(SaleDate  BETWEEN DATE '2020-01-01' AND DATE '2020-12-31' EACH INTERVAL '1' DAY );
```

Figure 57 Step 2: Replace NUPI with UPI (USI is dropped)

Non-unique Secondary Index (NUSI)

Teradata implements NUSI differently compared to a USI. The NUSI rows are not distributed to AMP's based on the Teradata hashing algorithm. Instead, they are stored in the same AMP as the base table rows but in another (sub-) table.

The searched NUSI column values can be in any AMP and, thus, Teradata employs an all-AMP binary search strategy. The only exception is the case when the NUSI columns match those of the primary index. Then, one AMP is queried for both the index value and the (base) table row. Although an all-AMP search is, from a performance point of view, not desirable, still an all-AMP NUSI search is better than an all-AMP search in the base table itself. Also, as the search is AMP-local, there is less network contention in the Message Passing Layer compared to a USI search that involves communication between the AMP's.

The steps to retrieve a base table row (e.g., the one in Figure 58) through a NUSI are:

1. All AMP's search their NUSI sub-table for a matching ROWHASH of the value.
2. Each AMP extracts from the NUSI row the ROW_ID of the base table rows (Figure 59).
3. Each AMP uses the extracted ROW_ID to locate and transfer the data blocks that contain the specific (base) table rows (Figure 60).

```
CREATE TABLE tuning.cutsomer (
    CustomerId  BIGINT NOT NULL,
    CustomerName VARCHAR(255) NOT NULL,
    CustomerSegment CHAR(01) NOT NULL
) PRIMARY INDEX (CustomerId);

CREATE INDEX (CustomerName) ON tuning.cutsomer;

COLLECT STATISTICS COLUMN(CustomerName) ON tuning.cutsomer;

SELECT * FROM tuning.cutsomer WHERE CustomerName = 'Garcia';
```

Figure 58 Example of table definition with both a NUPI and a NUSI access path definition; query on NUSI column

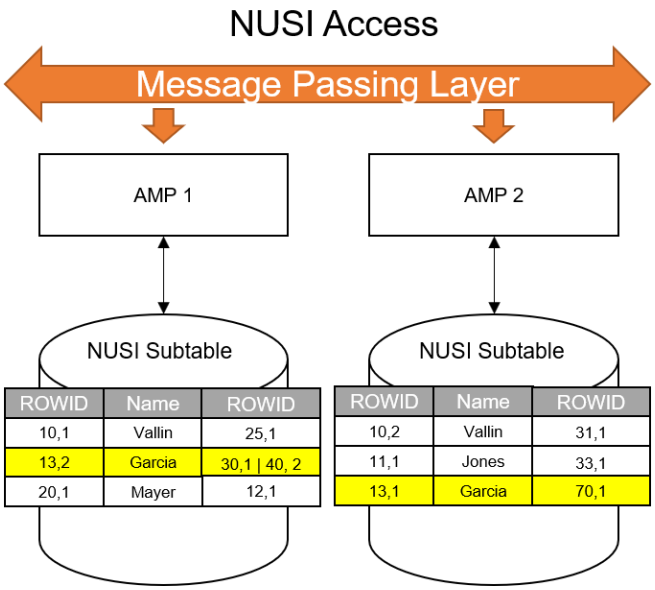

Figure 59 Locate base table ROW_ID through a NUSI access path (Steps 1-2)

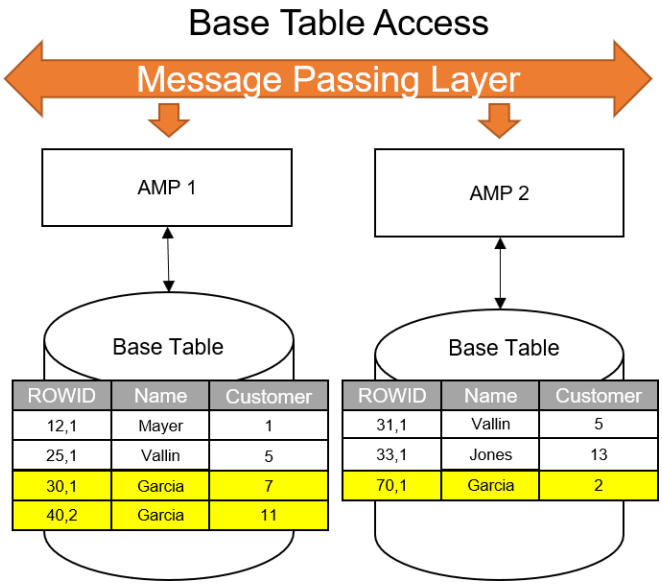

Figure 60 Direct access to base table row using the ROW_ID (Step 3)

Each (NUSI) index row may contain multiple base table ROW_ID's, up to a limit enforced by Teradata. Typically, an index row can hold a few thousand pointers to base table rows. The index row must store the internal partition number of the base table row for those tables that are row partitioned. Depending on the number of the defined partitions, two or eight more bytes are needed per index row, which increases the storage space requirements.

Index value ordering

The index rows can be stored inside the data blocks sorted according to three approaches: *default, hash, and value order*. Each approach serves better the needs of different usage scenarios.

Default ordering sorts the rows by the ROWHASH of all the index columns, similar to a primary index. This ordering is ideal for queries with equality constraints on all index columns. Queries on ranges of values can also benefit from a ROWHASH-sorted NUSI. However, the NUSI access path is rarely taken in such cases, as a full-table scan is considered more efficient (cf. Chapter 6). The example below demonstrates the creation of a ROWHASH-sorted NUSI:

```
CREATE INDEX (CustomerId) ON Customer;
```

Hash ordering sorts the rows by the ROWHASH of one selected index column. This ordering is ideal for queries with a multi-column NUSI where the important queries only search for value on the selected index column. The example below demonstrates the creation of a hash-ordered NUSI:

```
CREATE INDEX (CustomerId, CustomerName) ORDER BY HASH(CustomerName) ON Customer;
```

A value-ordered NUSI is the most efficient approach with queries on ranges of column values or with inequality constraints. Technically, each column value is mapped to a four-byte INTEGER value, and the index rows are sorted in a data block by this mapped value. The value ordering approach allows us to search only parts of the index (sub-) table. As date values are internally represented as integers, a value-ordered NUSI is well-suited for queries on ranges of dates. The example below demonstrates the creation of a value-ordered NUSI:

```
CREATE INDEX (CustomerId, BirthDate) ORDER BY VALUES(BirthDate) ON Customer;
```

DML statements

The benefit of a NUSI must be assessed, as in the case of USI, accounting also the cost to maintain it after each DML statement is applied to the base table. An INSERT or DELETE statement against a table with a NUSI additionally causes (a) a NUSI sub-table data block transfer to memory, (b) a change in the data block, and a transfer to disk, and (d) a Cylinder Index update and transfer to disk. In total, three I/O operations.

An UPDATE statement that does not involve a primary index column change requires five I/O operations for the base table. If the UPDATE affects one or more NUSI columns, then six more I/O operations are required for the sub-table: three for the NUSI sub-table data block and three for the new NUSI sub-table data block.

If the primary index columns are touched, the distribution changes: ten I/O operations for the base table (five to DELETE old rows from an AMP and give to INSERT new rows to a new AMP) and six I/O operations for the sub-table, as before. Despite the number of I/O operations for NUSI maintenance being the same, the number of involved AMP's is different in the latter case.

Single-AMP PI access path

The shared-nothing architecture of Teradata should be exploited to its full potential and evenly distribute table rows to all available AMP's. However, there are exceptional cases where it is more performant to concentrate by design all table rows in as few as one AMP. This is especially useful for tables with a low row count. It saves the effort to collect the otherwise dispersed rows into one memory block, which is then redistributed to all AMP's for further processing.

A single-AMP table row distribution can be achieved by defining a table primary index column with a BYTEINT data type (the least-space-consuming data type in Teradata) and storing a fixed value (e.g., 1) for all table rows. Then, the Teradata hashing algorithm will select the same AMP for all table rows. The related queries should then always include the fixed-value primary index column in their WHERE clause, effectively forcing a single-AMP access path through the PI.

The single-AMP PI design exhibits these advantages:

- All rows are at the same AMP
- All rows fit in one or a few data blocks
- Effectively, the whole table is kept in memory
- A ROWHASH-level lock suffices to read the table rows; no need for a table-level lock.
- One or a few I/O operations on only one AMP are sufficient for disk-to-memory transfers, instead of one I/O operation per available AMP.
- Also, fewer AWT's and AMP CPU cycles are needed to access the table, leading to more efficient resource usage.

When several single-AMP tables must be designed, we consider a good practice to use a different fixed value for each, so the tables are assigned to different AMP's rather than having one AMP responsible for them all. Newer releases of Teradata allow a database administrator to define dedicated hash maps for this purpose. The former approach allows anyone to define a dedicated single-AMP access path without requiring additional system privileges.

Access path tuning considerations

From a tuning perspective, a balance between the general and the special is necessary. On the one hand, there is the flexibility to define tailor-made row partitions and secondary indices to cover every SQL statement. This could lead to minimal data block transfers per

statement. On the other hand, such an approach could also increase overall resource consumption and implementation complexity.

We consider discussions about partitions and indices at the DWH design phase as premature and leading to suboptimal design choices driven by artificial, technical constraints. Further, we consider a good practice to regularly perform a cost-benefit analysis based on factual query demographics and assess whether a new row partition approach or secondary index would benefit the DWH.

In our experience, most tuning issues of a DWH in operation are solved by merely rewriting the SQL statements to utilize the already-defined indices and partitions properly. Seldom is the case that a new data access path must extend the PDM. The more the PDM differs from the LDM, the less flexible it is, and the more modeling inconsistencies must be addressed programmatically. The next sections discuss the design considerations and choices for defining a new data access path.

Row partitioning

Candidate columns for row partitioning are those that are often used in WHERE clauses. If no such columns exist, partition elimination cannot be applied, but space and processing overheads do apply. Those columns queried as ordered ranges (e.g., a date period, as in Figure 61) are well-suited for row partitioning.

```
1  CREATE SET TABLE tuning.event (
2      eventId BIGINT NOT NULL,
3      eventType BIGINT NOT NULL,
4      eventDate DATE FORMAT 'YYYY-MM-DD'
5  ) PRIMARY INDEX (eventID)
6  PARTITION BY (
7      RANGE_N(eventDAte BETWEEN DATE '2020-01-01' AND DATE '2020-12-31' EACH INTERVAL '1' MONTH)
8  );
9
10 SELECT *
11 FROM tuning.event
12 WHERE eventDate BETWEEN DATE '2020-08-01' AND DATE '2020-08-31';
```

Figure 61 Querying a date-range-partitioned table

A typical application for PPI tables is partitioning of tables by date. For example, if only one day is read from a table with 365 daily partitions of evenly-distributed rows, then only 1/365 of the table rows are read. Row partitioning is more beneficial for large tables, with many rows and many columns. In these cases, the table size in data blocks is large. A properly-defined row partitioning expression will cause significant savings in disk-to-memory transfers, i.e., I/O operations.

Row partitioning can improve performance even when the query does not support a primary-index access path for a table. Partition elimination may still apply, limiting the number of I/O operations for unnecessary data block transfers.

The optimal row-to-partition distribution depends on the query patterns to be served. The available options are: balance the number of rows per partition, balance the number of distinct values per partition independently of their relative frequency, and vary the number of rows based on other optimization criteria, even if the partitions end up with an imbalanced number of rows or distinct values.

Teradata does not allow to define a SET table with a non-unique partitioned primary index. Such a definition calls for duplicate checks, which can be prohibitively expensive. A MULTISET table is acceptable, as they can contain duplicate rows.

Row clustering and spread

The sort order enforced in a row partitioned table offers an opportunity for performance tuning by aligning load operations with table partition definitions. When rows are inserted on a fixed schedule (e.g., batch loading every day or week, or month), they are typically in that order in the source as well (e.g., every day the input for this specific day is received). A row partitioned table on that column (e.g., the data reference date) exhibits improved load performance when compared with an NPPI table. The reason is that all the rows end up in the same partition and the same or physically adjacent data blocks, which improves the probability of memory caching.

The sort order of row partitioned tables allows us to take advantage of the so-called "*row clustering*", provided that the data are loaded in a specific pattern. If data are inserted in a regular (e.g., daily, weekly) batch load, they usually arrive in a specific pattern (e.g., each day a specific date of reference). A table partitioned by this column has better load performance than an NPPI table because all rows are inserted into the same partition, as depicted in Figure 62. Their data blocks are physically located next to each other. This improves the probability that the data blocks are already cached in the AMP memory when it is time to be processed, reducing the number of transfers from the VDisk.

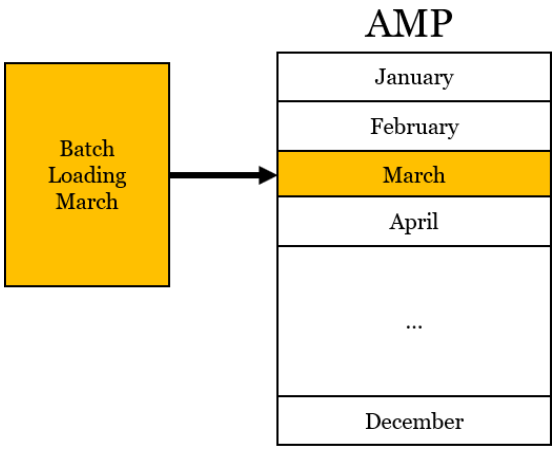

Figure 62 Load-driven row clustering

A partition expression can improve the distribution of a naturally-skewed primary index. A NUPI with many equal PI values achieves better insert performance when not all the partitioning columns are also primary index columns. Here, the NUPI values are assigned to different partitions, hence, spread to different disk-resident data blocks, as depicted in Figure 63. This is especially advantageous with SET tables, as the overhead for duplicate row checks and data block management is reduced to a minimum.

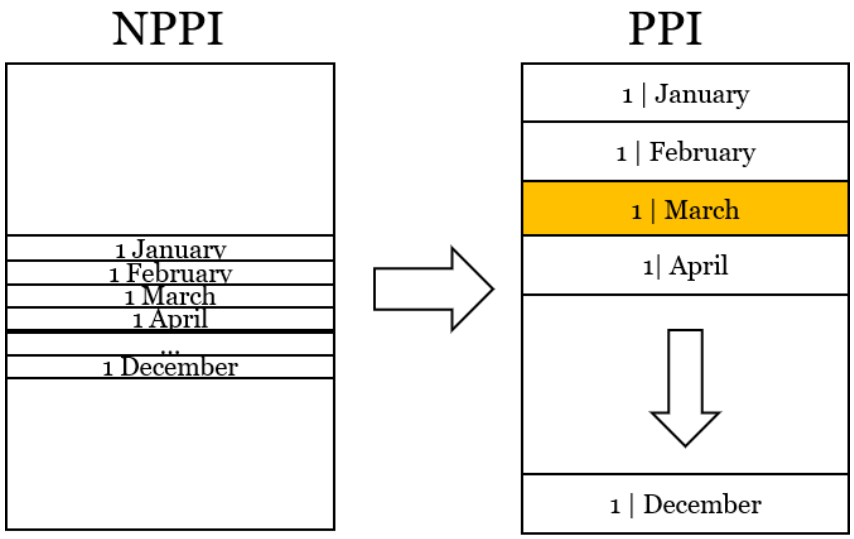

Figure 63 Spread rows to partitions through a NUPI

Let's assume the case of INSERT's into an NPPI SET table and the case of the same table but with a PPI defined, as depicted in Figure 64. A two-orders-of-magnitude improvement was observed in our experiments for both AMP CPU time (186 vs. 0.97) and I/O operations (59 million vs. 300 thousand) in the latter case, as duplicate row checks are avoided.

```
-- NPPI
CREATE SET TABLE tuning.Transactions (
    TransId BIGINT NOT NULL,
    TransType INTEGER NOT NULL
) PRIMARY INDEX (TransId);
-- Populate NPPI
INSERT INTO tuning.Transactions
SELECT RANDOM(1,5) AS TransId, ROW_NUMBER() OVER (ORDER BY 1) AS TransType
FROM SYS_CALENDAR.CALENDAR;
-- Drop NPPI
DROP TABLE tuning.Transactions;
-- PPI
CREATE SET TABLE tuning.Transactions (
    TransId BIGINT NOT NULL,
    TransType INTEGER NOT NULL
) PRIMARY INDEX (TransId)
PARTITION BY (RANGE_N(TransType BETWEEN 1 AND 10000000 EACH 1));
-- Populate PPI
INSERT INTO tuning.Transactions
SELECT RANDOM(1,5) AS TransId, ROW_NUMBER() OVER (ORDER BY 1) AS TransType;
-- Flush query log
FLUSH QUERY LOGGING WITH ALL;
```

Figure 64 Spreading NUPI-table rows in different partitions

Table locking

A PPI table can be potentially accessed with more fine-grained locks compared to an NPPI one. When the primary index or USI access paths are taken, then rowhash locking is sufficient for both cases. When such paths are not taken, but partition elimination can be applied, then partition-level locks are sufficient, which is advantageous for a PPI table. When neither PI nor USI nor partition elimination can be applied, then a full-table lock is necessary in both cases.

An exclusive table lock is needed when an ALTER TABLE statement drops an already populated partition. Teradata transparently optimizes DELETE and UPDATE statements that eliminate all the rows of a partition. In such cases, Teradata does not use Transient Journal, provided that it is a single-statement transaction or the last step of a multi-statement one. To reduce the time of the exclusive lock for the ALTER table is held, one can first issue a DELETE statement (only a write lock is necessary) to empty the partition and the ALTER TABLE afterward. This way, the non-dropped partitions remain accessible for all the time the DELETE statement executes.

Teradata can also bypass the Transient Journal when INSERT statement fills an empty partition, provided that a) no referential integrity is defined for the target table and b) if several empty partitions are filled at once (e.g., "1", "3", "4"), there is not an already-populated partition between them (e.g., "2"). If not, then it is better to issue separate INSERT statements.

Partitions instead of indices

Row partitioning can replace non-primary indices. This is beneficial as indices require additional storage space and maintenance during INSERT/UPDATE/DELETE statements. A multi-level row-partitioning (MLPPI) definition provides multiple access paths at once. Therefore, an MLPPI expression can replace multiple index definitions and significantly reduce storage space and index maintenance costs.

Row partitioning can improve disk access time in conjunction with Teradata Intelligent Memory and the multi-temperature concept discussed in Section Teradata Intelligent Memory technology of Chapter 2. Let's assume the example of a date-partitioned table in months. Let's further assume that more recent months are queried more often than old ones and that they have a higher temperature (e.g., they are "hot"). Then, Teradata places the rows of the recent months on a set of same-temperature cylinders. As new months are added over time, the temperature of the rows decreases, and Teradata moves them to cylinders of slower storage, freeing up the resources for the new "hot" rows.

Indices to improve PPI access

A UPI cannot always be defined for a row-partitioned table. When a primary index access path is taken, and the set of the partition columns does not match the set of the primary

index columns (Figure 65), then partition elimination cannot be applied (Figure 66). Potentially, all partitions contain the primary index and, therefore, must be probed, as depicted in Figure 67.

```
CREATE TABLE tuning.Schedule_PPI (
    ScheduleId BIGINT,
    StartDate DATE FORMAT 'YYYY-MM-DD'
) PRIMARY INDEX (ScheduleId)
PARTITION BY RANGE_N (StartDate BETWEEN DATE '1900-01-01' AND DATE '2100-12-31' EACH INTERVAL '1' MONTH, NO RANGE);

INSERT INTO tuning.Schedule_PPI
SELECT ROW_NUMBER() OVER (ORDER BY 1) AS ScheduleId, CALENDAR_DATE
FROM (
    SELECT *
    FROM SYS_CALENDAR.CALENDAR
    UNION ALL
    SELECT *
    FROM SYS_CALENDAR.CALENDAR
    UNION ALL
    SELECT *
    FROM SYS_CALENDAR.CALENDAR
) t01;

COLLECT STATISTICS COLUMN(ScheduleId) ON tuning.Schedule_PPI;
COLLECT STATISTICS COLUMN(StartDate) ON tuning.Schedule_PPI;
COLLECT STATISTICS COLUMN(PARTITION) ON tuning.Schedule_PPI;
```

Figure 65 A PPI table with different partition and primary index columns

```
EXPLAIN SELECT *
FROM tuning.Schedule_PPI
WHERE ScheduleId = 100;

Explanation
------------------------------------------------------------------------
  1) First, we do a single-AMP RETRIEVE step from all partitions of
     tuning.Schedule_PPI by way of the primary index
     "tuning.Schedule_PPI.ScheduleId = 100" 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 (33
     bytes).  The estimated time for this step is 0.01 seconds.
  -> 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.
```

Figure 66 Single-AMP primary index access path in all the partitions of the AMP

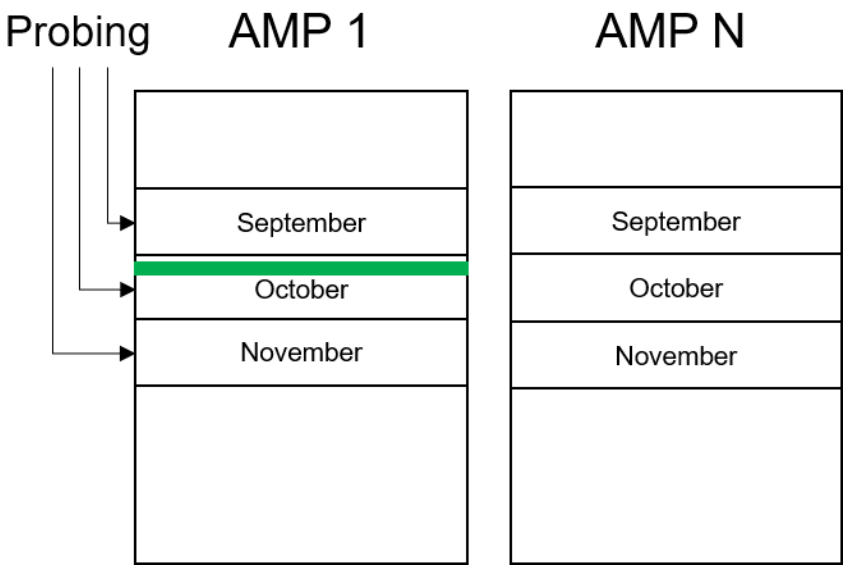

Figure 67 Partition probing as part of the primary index access path on a PPI table

Teradata allows us to define a secondary index (USI or NUSI) on top of the primary index to support partition elimination in such scenarios. As in the case of any secondary index, this comes at the expense of additional storage and index maintenance. When a USI is defined, Teradata involves two AMP's, as depicted in Figure 68: one for the sub-table for the USI and one for the base table for the PI. It is unnecessary anymore to probe all partitions. Instead, the USI points to those partitions that contain the PI rows. An additional benefit of this "*single AMP partition elimination with a USI*" is that rowhash-level locks on the table will now serve the request, increasing opportunities for concurrent execution with other statements.

```
1 CREATE UNIQUE INDEX (ScheduleId) ON tuning.Schedule_PPI;
2
```

Teradata Result Set Viewer

Result Set - SQL Editor (1)

```
EXPLAIN SELECT *
FROM tuning.Schedule_PPI
WHERE ScheduleId = 100;

Explanation
-----------------------------------------------------------------------
  1) First, we do a two-AMP RETRIEVE step in TD_MAP1 from
     tuning.Schedule_PPI by way of unique index # 4
     "tuning.Schedule_PPI.ScheduleId = 100" with no residual conditions.
     The estimated time for this step is 0.01 seconds.
  -> The row is sent directly back to the user as the result of
     statement 1.  The total estimated time is 0.01 seconds.
```

Figure 68 Single-AMP combined PI-USI access path for PPI table

Where a USI cannot be defined, single AMP partition elimination can still be achieved with a NUSI, as depicted in Figure 69. As in a USI case, rowhash-level locks are sufficient, and no partition probing is performed. A secondary advantage in these cases is that the Multiload bulk load tool can import data in a NUSI table but not in a USI one.

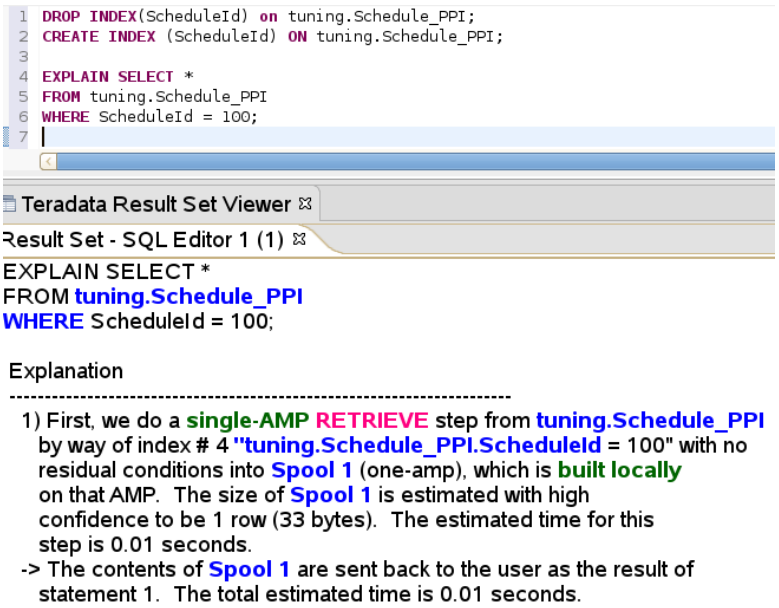

Figure 69 Single-AMP combined PI-NUSI access path for PPI table

Chapter summary

In this chapter, we turned our focus on the operational phase of the DWH and specifically on tuning the data block transfers from AMP VDisk to AMP memory. Typically, this is the most time-consuming processing step of an execution plan for serving a request. The volume of data block transfers heavily depends on both query and data demographics. As both evolve, a cost-benefit analysis must be periodically performed to identify areas of improvement.

Extending the PDM, especially in the design phase without hard evidence from demographics, can worsen the DWH performance. Rewriting SQL queries to better exploit the existing PDM is often sufficient to achieve a better execution plan and solve any performance challenges. When new data access paths must be defined, row partitioning is a better choice than secondary indices. As both options reduce the data block transfers at the expense of increased storage space and maintenance costs, their definitions must be carefully evaluated, always based on hard evidence. If they are not used anymore, they should be removed, simplifying the PDM and reducing the associated maintenance overheads.

The main takeaways of this chapter are:

- All data access paths in Teradata can be classified according to three criteria: the number of involved AMP's (one, multiple, or all); the ordering of the row pointers inside the data blocks; and the search strategy (full table scan, all-AMP search, single-AMP search, partition elimination, and sub-table access).
- Tall (many rows) and wide (many columns) tables benefit the most from row partitioning.
- The more the columns in a (N)USI, the more probable the query will be fulfilled by the sub-table itself, without accessing the base table (i.e., the (N)USI is a *covering index*). The fewer the columns, the more opportunities for a (N)USI to be used and avoid a full-table scan.
- The less distinct values in an index, the less the chances for the index to be used, as the cost of a sequential full table scan becomes comparable and exhibits less processing complexity. The same holds when the distinct values are many but heavily skewed in a few.

Chapter 6: Tuning the query execution plan

"Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it." Kernighan's law, named after Brian Wilson Kernighan

A key component for the high performance of Teradata is its ability to analyze multiple SQL statements of the incoming requests and, taking into consideration the overall status of the system, dynamically devise an optimal set of processing steps that exploit the MPP capabilities to serve all the requests in the best possible way, as depicted in Figure 70. The previous chapter focused on optimizing the data block transfers from AMP VDisk to memory. Here, the sole focus is the in-memory processing steps.

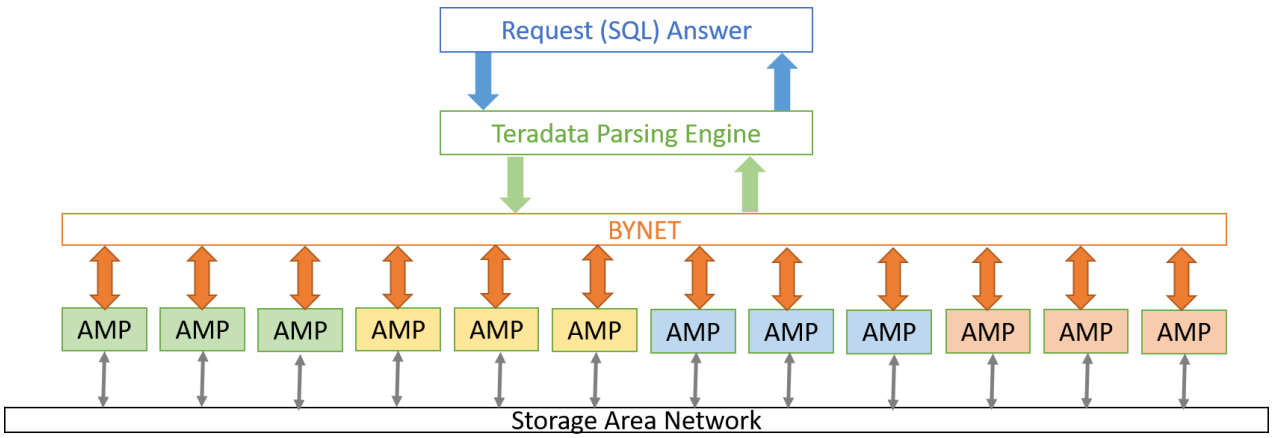

Figure 70 In-memory (thick arrows) and disk transfers (thin arrows) processing steps to answer a request

Request lifecycle

The Teradata Parsing Engine (PE) is responsible for the interaction with requests. All communication between Teradata and a requestor (e.g., a client tool) takes place via *parcels (packets)*. The requestor initiates the communication by sending at least one *request parcel* to Teradata, which contains one SQL statement (or more, in the case of a multi-statement request). When in Teradata session mode, each SQL statement represents exactly one transaction.

A request parcel is the elementary work unit for the parser; each parcel is parsed independently of any other one. Multiple *data parcels* may follow a request parcel. The data parcels are used to deliver different parameters for the statement contained in the request parcel. This approach allows the reuse of request parcels, reducing the communication overhead between Teradata and a requestor. For example, consider the

case of a request parcel calling a macro with parameters, followed by data parcels instantiating the macro call for specific input values, as depicted in Figure 71.

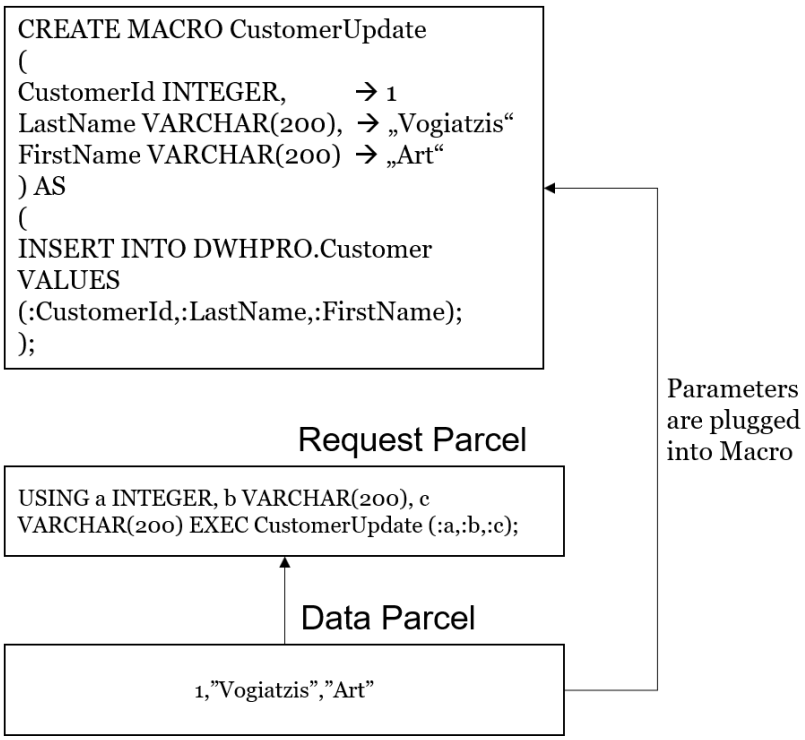

Figure 71 A macro call request parcel with parameters, followed by data parcel with parameter values

The parser

The *parser* component of the Teradata PE parses the SQL statement of a request parcel, checks its syntactic validity, converts database object names to internal identifiers using the Data Dictionary, and confirms that the requestor has the appropriate access rights for the database objects involved in the request. The output of the parser is an SQL statement *parse tree*.

The Teradata PE may cache a request parcel for many hours. Whenever the same parcel is received, all processing is skipped, but the access right checks. Then, the parse tree is passed for further processing using the cached information and, thus, saving processing time. When the cache is not user-specific, the request parcel caching may span multiple sessions, allowing further reuse and resource savings.

The `DBC.AccessRights` table and its view `DBC.AllRights` store the access rights per user, database, and database object (e.g., a table). When a simple database table is created, many rows are inserted (twelve in Teradata version 16.20) for the different access rights, as depicted in Figure 72.

```
SELECT UserName, DatabaseName, TableName, AccessRight, GrantorName, CreatorName
FROM dbc.allrights
WHERE databasename = 'tuning'
AND tablename = 'schedule_ppi';
```

Teradata Result Set Viewer

Result Set - SQL Editor (1)

	UserName	Databas...	TableName	AccessRight	GrantorName	CreatorName
1	DBC	tuning	Schedule_PPI	CG	DBC	DBC
2	DBC	tuning	Schedule_PPI	ST	DBC	DBC
3	DBC	tuning	Schedule_PPI	DG	DBC	DBC
4	DBC	tuning	Schedule_PPI	IX	DBC	DBC
5	DBC	tuning	Schedule_PPI	RS	DBC	DBC
6	DBC	tuning	Schedule_PPI	DP	DBC	DBC
7	DBC	tuning	Schedule_PPI	DT	DBC	DBC
8	DBC	tuning	Schedule_PPI	RF	DBC	DBC
9	DBC	tuning	Schedule_PPI	D	DBC	DBC
10	DBC	tuning	Schedule_PPI	I	DBC	DBC
11	DBC	tuning	Schedule_PPI	U	DBC	DBC
12	DBC	tuning	Schedule_PPI	R	DBC	DBC

Figure 72 Table access rights

Teradata defines the username (column `UserId`) as the primary index of `DBC.AccessRights` and the table rows are distributed to the AMP's accordingly. Then, heavy skewing may cause sub-optimal performance. One such scenario is when a single batch user creates a large number of data stage tables as part of the daily batch process. As the Teradata PE needs to confirm access rights for every request (even if cached), this task effectively becomes a single-AMP one.

The impact of this design choice can be mitigated as follows. By definition, the user creating an object implicitly owns all access rights. The records in `DBC.AccessRights` are, thus, redundant and can be deleted, as depicted in Figure 73. The parsing times are reduced once the access rights table is cleaned up. However, if the data stage tables are dropped and created as part of the ETL or ELT process at every batch load, the tuning improvements are lost; a new database object identifier is assigned to the table and, thus, the rows are added again in the access rights table. We consider a good practice to empty the data stage tables using a DELETE statement instead.

```
SELECT 'REVOKE ALL ON ' || TRIM(a.databasename) || '.' || TRIM(a.tablename) || ' FROM ' || TRIM(a.username) || ';' (TITLE '')
FROM dbc.allrights a
INNER JOIN dbc.tablesv b
ON b.DatabaseName = a.DatabaseName
AND b.tablename = a.tablename
AND b.tablekind = 'T'
WHERE a.grantorname = 'theUser' -- replace
AND a.DatabaseName IN (
    SELECT d.databasenameI
    FROM dbc.dbase d
    WHERE  d.rowtype = 'D'
)
AND a.username = a.grantorname
AND a.accessright NOT IN ('IX', 'RF')
GROUP BY 1;
```

Figure 73 Revoke user access rights on database tables

The optimizer

The *optimizer* component of the Teradata PE devises an *execution plan* for the parse tree, i.e., an ordered list of execution steps to calculate the response to the request and the database object locks necessary at each step. Every request can be fulfilled using one or more full-table scans. However, this is very inefficient. Teradata uses a cost-based optimizer to find the least expensive execution plan. The main cost factors considered are the number of I/O operations, CPU seconds, memory usage, and contention (load) on the message-passing layer.

The optimizer is sophisticated enough to perform SQL rewriting before devising an execution plan, i.e., restructure an SQL statement in an equivalent form that is less resource-intensive and exhibits better performance. Typically, the SQL statement rewriting includes:

1. Removal of non-referenced columns and expressions from a VIEW list. When a query runs over a view but references only a subset of its columns, like in the example below, the optimizer does not spool the residual columns to reduce or even eliminate spool usage.

```
CREATE VIEW myView AS
SELECT t1.col1, t2.col2, AVG(t2.col3) myAvg
FROM t1, t2
WHERE t1.pk = t2.pk
GROUP BY col1, col2;

SELECT Max(myAvg) FROM myView; -- optimizer does not spool col1 and col2
SELECT Count(*) FROM myView; -- optimizer does not spool at all; cylinder index suffices
```

2. Replacement of OUTER with INNER JOIN, if possible. When a query is satisfied by the inner join (e.g., there exists a WHERE condition on the right table of a LEFT JOIN, like in the example below), the optimizer rewrites the OUTER as INNER JOIN before further processing.

```
SELECT Distinct(col1)
FROM t1
LEFT OUTER JOIN t2
ON t1.pk = t2.pk
WHERE t2.col2 = 100; -- both t1.col1 and t2.col needed, optimizer performs INNER JOIN
```

3. View folding. When the query runs over a view, but it can be satisfied by directly accessing the underlying objects, like in the example below, the optimizer replaces the view objects with the latter to avoid additional processing overhead.

```
SELECT col2 FROM myView WHERE myAvg > 100; -- no view needed
SELECT col2 -- rewritten by the optimizer as
FROM t1, t2
WHERE t1.pk = t2.pk
GROUP BY t1.col1, t2.col2 HAVING AVG(t2.col3) > 100;
```

4. Predicate pushdown. When predicates (e.g., a WHERE clause) can be applied directly in a subquery, like in the example below, the optimizer rearranges ("pushes down") the predicates inside the subquery so that row filtering is applied as soon as possible.

```
SELECT Sum(theSum) theOverallSum
FROM (
       SELECT col1, col2, Sum(col3*col4) AS theSum
       FROM t1, t2
       WHERE t1.pk = t2.pk
       GROUP BY col1, col2
) t12
WHERE t12.col1 IN (1,2,3,4,5); -- the optimizer pushes this predicate inside the t12
subquery.
```

5. Elimination of SET operator branches and unneeded joins. When the query includes an unsatisfiable SET branch condition or an unneeded join, like in the examples below, the optimizer removes them from the query before devising the execution plan.

```
-- assume a CHECK CONSTRAINT on t2 that monthID is always equal to 2
SELECT * FROM t1 WHERE monthID = 1 UNION ALL SELECT * FROM t2 WHERE monthID = 1;
-- optimizer rewrites as
SELECT * FROM t1 WHERE monthID = 1

SELECT t1.* FROM t1 LEFT JOIN t2 ON t1.pk = t2.pk;-- LEFT JOIN on a table primary key
-- optimizer rewrites as
SELECT t1.* FROM t1
```

Teradata can process up to two tables at each step of an execution plan. When more are involved, the optimizer splits processing into more steps and places the intermediate output in temporary tables of the AMP spool area. These tables are then used as input to subsequent processing steps.

The optimizer may not always find the overall optimal execution plan. As the number of possible plans increases exponentially with the number of involved tables, the exhaustive search becomes prohibitively expensive in realistic usage scenarios. For a single-table SQL query, the optimizer must choose among all the possible table data access paths (e.g., FTS, partitioned PI, USI, and various NUSI), as discussed in Chapter 5. Additional data access paths may be available for a two-table SQL query, based on *join indices* discussed later in this chapter. Figure 74 depicts two of all the possible plans for a three-table SQL query, simplified in not considering the data access path combinations.

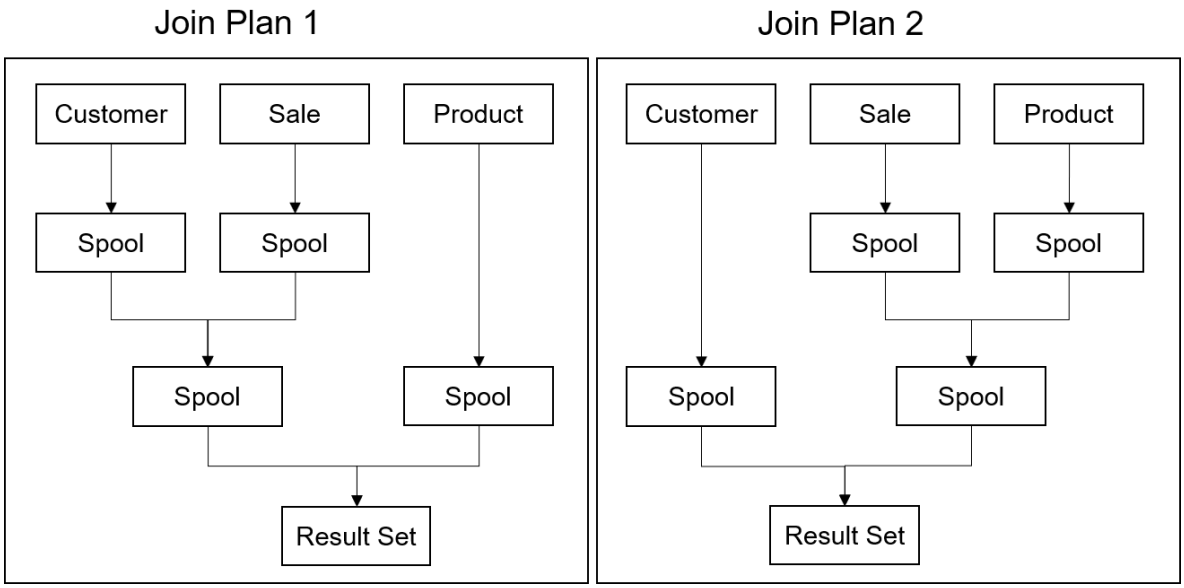

Figure 74 Two possible (abstract) execution plans for the same SQL query

As in the case of request parcels, Teradata may cache execution plans for many hours. When the parsing engine considers an execution plan independent of the underlying data demographics, it extends the cache residence of the latter for even longer than a request parcel. The assumption is that being generic enough, the calculated execution plan will be reused and, thus, achieve more resource savings. Thus, the SQL statement must remain untouched across different requests, as Teradata compares the statements byte by byte using a hash algorithm. Even a slight change, for example, an additional white space character, render the two statements different and, thus, the cached execution plan cannot be used. Another advantage of execution plan caching is that it allows the so-called *delayed partition elimination*, i.e., to detect the possibility of partition elimination while replacing the request parameters.

The least frequently used (LFU) execution plans are removed from the limited space of the cache memory whenever a new one must be added, and there is no space available. Also, changes in the underlying database object definitions invalidate all the related execution plans in the cache. DDL statements are never cached, as the expectation is that they execute only once.

The generator

The *generator* component of the Teradata PE performs administrative activities, such as generating the list of tasks for each AMP. These activities are of little interest from a performance point of view as they introduce a fixed, unavoidable overhead.

The dispatcher

The *dispatcher* component of the Teradata PE coordinates the execution of the steps and instructs the BYNET to send them to the AMP's progressively. Upon receiving a completion response from all involved AMP's and for all steps, the final processing of the answer to the request is performed.

Every execution step concurrently executes in all or a dynamically-defined subset of the available AMP's. This unique approach of Teradata allows achieving high performance. Each AMP can independently perform all the database tasks related to the data assigned to its VDisk, as depicted in Figure 75.

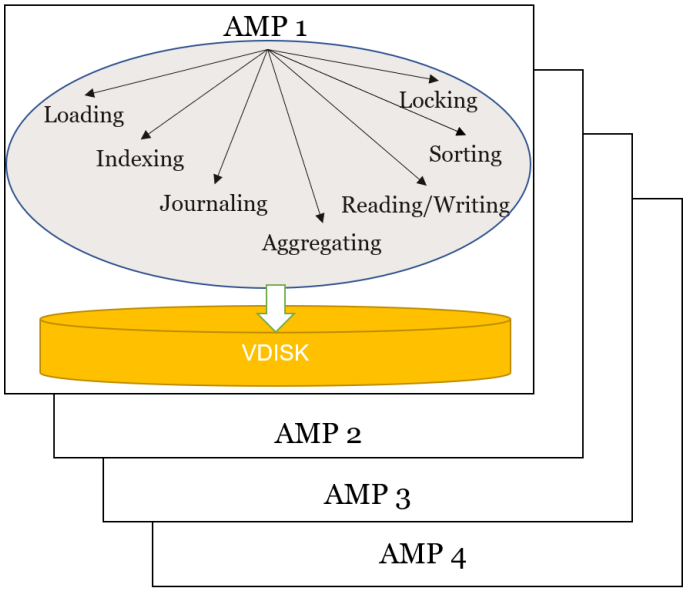

Figure 75 An AMP performs all tasks for its data

An execution plan may contain *serial* steps that must execute in sequence and *parallel* steps that may execute in parallel. Serial steps can only start when all AMP's working on a previous step have finished their task. A serial step following a set of parallel steps may start only after all parallel steps have finished. A third kind of steps are the common steps; these are exclusively used in multi-statement requests and allow a common intermediate result to be used multiple times with the request processing. The dispatcher can *pipeline* steps when partial input data are available for the next step, as depicted in Figure 76.

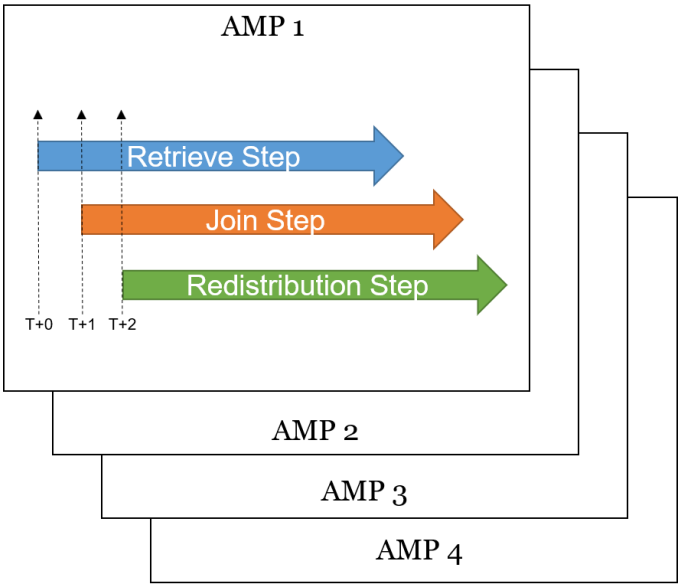

Figure 76 Step pipelining; rows processed by the next step as soon as fetched by the previous one

Spool space for execution steps

When an execution plan comprises more than one execution step, Teradata places in the *spool space* of the responsible AMP the intermediate output table. The subsequent step then uses this spooled table as its input.

Teradata allocates only fully-empty cylinders to spool space and does not permit cylinders to be shared between permanent and spool space. Then, it may be the case that no spool space can be allocated, despite space being available for use. The more the data are fragmented, the more the cylinders they occupy and, thus, reduce the free space to use for the spool.

Teradata enforces a spool space limit per user (i.e., a password-protected database, as discussed in Chapter 2). The current limit can be checked with a query similar to the one below:

```
SELECT SpoolSpace FROM DBC.databasesv WHERE DATABASENAME = 'tuning';
```

The allocated spool space of a database is equally split among the available AMP's. The more the AMP's in a system, the less the spool space per AMP, as depicted in Figure 77. It suffices just one AMP to run out of spool space to get a "*no more spool space*" error and halt processing (error message 2507).

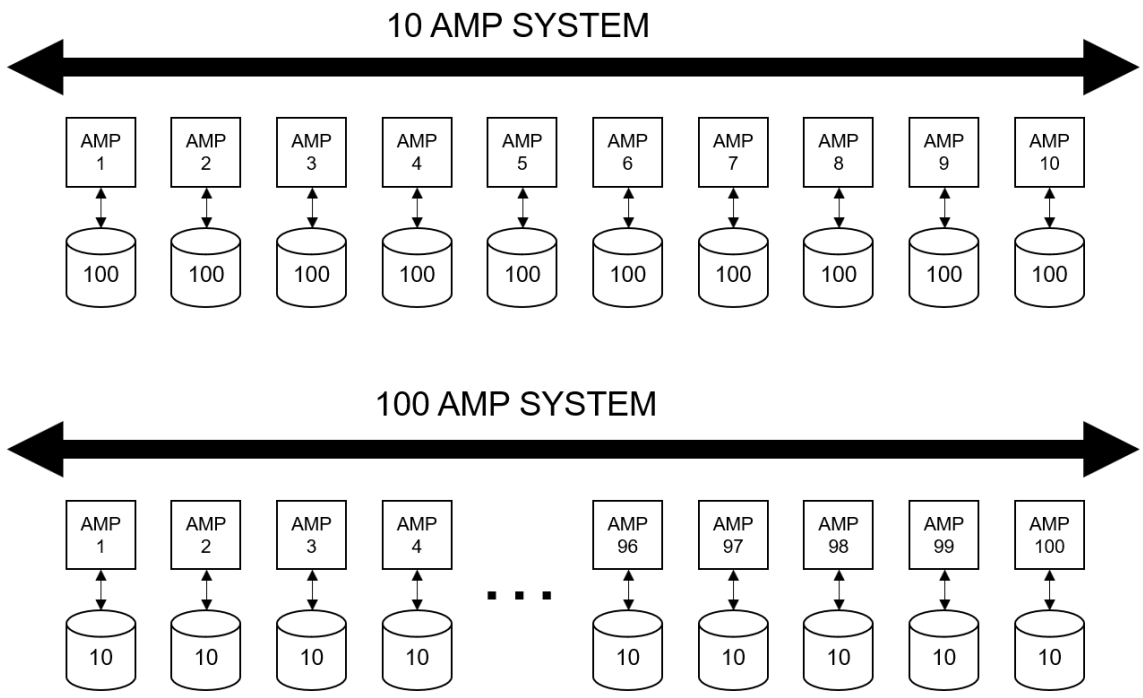

Figure 77 Sharing 100 TB of spool space among ten and 100 AMP's (10 and 1 TB per AMP, respectively)

Requests of the same user compete for the user's (database) spool space. As such, copying large tables in spool space increases the probability of spool exhaustion. Depending on the exact order of execution, any user query that concurrently executes is susceptible to a halt with a "no more spool space" error (error message 2646).

A skewed query or a badly-distributing primary index definition of an intermediate table is a typical cause of user-related error message 2646. A typical scenario of a system-related error message 2507 is an increase in system AMP's: requests that were successfully served before the upgrade are now halted due to spool space exhaustion. In either case, the root cause should be determined and fixed. A high spool usage does not necessarily imply that a request must be modified (e.g., tune a badly-written SQL query). Still, in our experience, there is a high correlation between these two.

Teradata distinguishes four types of spool space: *output*, *intermediate*, *volatile*, and *persistent*. The output spool space holds the final result of a request.

The intermediate spool space is typically used in *derived subqueries* of an SQL statement. Space is released as soon as it is no longer needed.

The volatile spool space is exclusively used for volatile tables. It is released once a session concludes. We consider a good practice to explicitly release volatile spool space (e.g., using a DROP statement for the respective volatile table) as soon as it is no longer needed within a session and reduce the risk of running out of spool space. An alternative to the DROP

approach is to wrap the use of the volatile table in a BEGIN/END transaction block, as in the example below:

```
CREATE VOLATILE TABLE RelevantCustomer (CustomerId BIGINT, CustomerDate DATE)
PRIMARY INDEX (CustomerId)
ON COMMIT DELETE ROWS; -- delete automatically

BEGIN TRANSACTION;
INSERT INTO RelevantCustomer (CustomerId, CustomerDate)
SELECT x, y FROM theTable;
-- one more or statements utilizing the volatile table
END TRANSACTION; -- volatile table is emptied
```

The persistent spool space is used to automatically restart *redrive-protected* incomplete SQL statements after a system restart or a node crash. Redrive protection can be activated on the session, user, account, or even system level, i.e., all SQL statements. On a session-level, the following statements can be used to activate and deactivate the use of persistent spool space:

```
SET QUERY_BAND = 'redrive=on;' FOR SESSION; -- activate
SET QUERY_BAND = 'redrive=off;' FOR SESSION; -- deactivate
```

At the expense of consumed spool space, redrive protection makes system failures transparent to the clients. This is beneficial for long-running, complex requests that cannot sustain to repeat all calculations from the very beginning.

Optimizer parameters

The optimizer considers several factors to calculate an optimal execution plan. Every single task is optimized, utilizing information about the system and data demographics. The primary system information considered in the optimization process is the number of system nodes, the number of AMP's, the available memory per AMP, the number and type of CPU's, and the message passing layer used (e.g., hardware BYNET).

The primary source of data demographics is *collected statistics* regarding information like the table size and the number of distinct values per column. When such statistics are not available or are considered outdated (stale), Teradata resorts to *dynamic AMP sampling* (for indexed table columns) and *heuristics* (for non-indexed table columns) to estimate the current data demographics and avoid full-table scans. The next sections explore the different types of data demographics and how they can affect the execution plan.

Collected statistics

Collected statistics equip the optimizer with the highest quality information for calculating an optimal execution plan. Statistics may be collected for tables and indices. Teradata can collect statistics on single-column, multi-column, and full-table level. The collected statistics are calculated over the whole or a fraction of the table (or index). A non-exhaustive list of key metrics includes table cardinality, average row length, value range of a column,

number of rows per column value, number of rows with a NULL-valued column, and skew information.

Teradata stores the collected statistics in *histogram intervals*, i.e., buckets with an approximately equal number of distinct column values (or distinct value combinations in the case of multi-column statistics). Teradata defines these intervals in a way that they represent *equal-height histograms*. This kind of histogram saves storage space and provide accurate-enough but not exact value demographics. The default value is 250 intervals and can be freely set to any number between 10 and 500.

Teradata determines the actual number of equal-height histograms used. When the table contains heavily-skewed values, Teradata may choose to collect their statistics in *high-bias intervals* instead, i.e., to store statistics about skewed values in dedicated structures, as depicted in Figure 78. High-biased intervals decrease the variance that equal-height histograms would exhibit and allow accurate estimations and better execution plans. Hence, depending on the table skew, the collected statistics may end up in only equal-height intervals (no skew is present); high-bias intervals (significant skew is present); or a combination of both (non-significant skew is present).

```
SHOW STATISTICS VALUES ON Customer;

COLLECT STATISTICS
            COLUMN ( CustomerId )
                ON DWHPRO.Customer
            VALUES
(
 /** SummaryInfo **/
 /* TimeStamp */ TIMESTAMP '2020-09-13 14:22:01-00:00',
 …
 /* NumOfBiasedValues      */ 1,
 /* NumOfEHIntervals       */ 250,
 /* NumOfHistoryRecords    */ 1,
 …
 /** Biased: Value, Frequency **/
 /*    1 */    99, 9999,
 /** Interval: MaxVal, ModeVal, ModeFreq, LowFreq, OtherVals, OtherRows **/
 /*      1    */ 10289, 10000, 1, 1, 289, 289,
  …
 /*    250    */ 73414, 73073, 1, 1, 341, 341
);
```

Figure 78 A dedicated histogram for high-bias intervals (highlighted in yellow)

An equal-height histogram contains the maximum value in the interval, the most frequent value in the interval, the cardinality of the latter, the number of distinct values, and the aggregate cardinality of these distinct values, as depicted in Figure 79. The more the intervals, the more the most-frequent (or skewed) values they can capture.

```
SHOW STATISTICS VALUES ON Customer;

COLLECT STATISTICS
            COLUMN ( CustomerId )
                ON DWHPRO.Customer
            VALUES
(
 /** Interval: MaxVal, ModeVal, ModeFreq, LowFreq, OtherVals, OtherRows **/
 /*     1    */ 298, 1, 1, 1, 297, 297,
 /*     2    */ 588, 299, 1, 1, 289, 289,
 …
 /*   249    */ 72988, 72423, 1, 1, 565, 565,
 /*   250    */ 73414, 72989, 1, 1, 425, 425
);
```

Figure 79 An example equal-height histogram for the Customer table

The collection of statistics comes with a performance overhead, as Teradata accesses through a full-table scan (FTS) the underlying object, sorts the collected metrics, and stores key figures in *histograms* for later use. Thus, the collection process is only manually initiated, either as part of an interactive request using a COLLECT STATISTICS statement or via scheduled maintenance jobs, as discussed later in this section. The performance overhead of a COLLECT STATISTICS statement can be checked using a query similar to the example below:

```
SELECT STARTTIME(DATE) theDate, Sum(TOTALIOCOUNT) Sum(AMPCPUTIME)
FROM dbc.dblogtbl
WHERE statementype = 'Collect Statistics'
GROUP BY theDate;
```

Teradata 14.10 introduced *sample statistics*, i.e., the possibility to collect statistics from only a fraction of the underlying object rather than by performing an FTS. This reduces the performance overhead at the expense of potentially less accurate information.

Sample statistics are an excellent replacement of FTS-based collected statistics when the underlying column values are not skewed. A UPI column is, by definition, fulfilling this constraint. In contrast, NUPI and NUSI columns are susceptible to skewing and, thus, the collection of sample statistics should be assessed on a case-by-case basis. When single-value outliers exist, sampling may result in entirely wrong estimates.

The sample size can be determined automatically by Teradata or set manually. In the former case, the following process is followed: the first time, FTS-based statistics are collected. Then, the sample size is reduced with every recollection, up to the point that a comparison with a random-AMP sample indicates that the estimates are inaccurate. At this point, the sampling size is frozen to the last one, still providing good estimates. After this point, Teradata periodically switches to FTS-based collection to confirm that the estimates remain accurate or the sample size must be readjusted. When deemed necessary, the sample size can be manually set with a statement similar to the example below:

```
COLLECT STATISTICS USING SAMPLE 10% COLUMN(CustomerId) ON Customer;
```

Tables with only a few rows, especially less than the number of AMP's, are, by definition, skewed and, thus, not a good choice for sampling. Let's assume a system with 400 AMP's and a table consisting of 40 rows that are evenly distributed to 40 AMP's. When a single AMP is sampled, there is a 90% probability for an empty table estimation and a 10% probability for a 400-row table estimation. In either case, the estimations are completely off and may guide the optimizer to choose a sub-optimal execution plan. We consider a good practice to collect full statistics for small tables, as the related performance overhead is negligible, and the risk of wrong estimations is eliminated.

Until Teradata 14.10, collected statistics, both FTS- and sample-based, were collected as in the example below:

```
COLLECT STATISTICS COLUMN(A) ON tuning.Customer;
COLLECT STATISTICS COLUMN(B) ON tuning.Customer;
COLLECT STATISTICS COLUMN(A,B) ON tuning.Customer;
```

This syntax results in one transaction per statement. An FTS retrieve- and a sort step is necessary for each transaction then. Teradata supports *synchronized scanning* to reduce the performance burden, i.e., allow multiple transactions running on different sessions to collect statistics on the same spool space. Starting with Teradata 14.10, the collection on the same object can be combined in one statement, as in the example below:

```
COLLECT STATISTICS COLUMN(A), COLUMN(B), COLUMN(A,B) ON tuning.Customer;
```

The new syntax allows a single transaction, and a single FTS retrieve step from the underlying table. All three statistics can be calculated from the same spool table. Moreover, Teradata can combine the results for single columns A and B to directly calculate the statistics for the column pair (A, B), achieving further performance improvements.

Dynamic AMP sampling

Dynamic AMP sampling only delivers table cardinality metrics. The optimizer uses dynamic AMP sampling when no statistics are collected on indexed columns, or the collected statistics of the indexed columns are considered outdated (stale). It is not a replacement for collected statistics but can offer sufficient accuracy in specific cases.

When a PI is defined, Teradata reads the first and last cylinder of the table from at least one of the AMP's (the exact number is configurable), counts the number of rows in the cylinders, and extrapolates the total number of rows across all AMP's. Should the table fit in only one cylinder, just this cylinder is read. The extrapolation formula is:

> *number of estimated table rows = (Average number of rows per data block in the sample cylinders) * (number of data blocks in the sample cylinders) *(number of existing cylinders of the table on the sample AMP) * (number of AMPs)*

When a USI is defined, the number of the distinct values of the USI column in the sampled AMP's matches the number of rows in the USI sub-table. The optimizer assumes that the overall number of distinct values matches the table cardinality. Thus, it extrapolates the latter by multiplying the cardinality on the sampled AMP with the number of available AMP's.

When a NUSI is defined, one or two cylinders of the sampled AMP's are read. The optimizer assumes that for each NUSI column value, there is exactly one NUSI index row. This assumption holds when the index is highly-selective, and all the pointers to the base table fit in one NUSI index row. The cardinality estimation is otherwise inaccurate.

Teradata uses, by default, a single AMP for dynamic AMP sampling. The sampling AMP is determined by hashing the TABLE_ID. On the one hand, this hashing approach allows to equally distribute the sampling effort of different tables across all AMP's. Further, it ensures a consistent execution plan at every request invocation, provided that the data demographics are stable. On the other hand, this approach results in consistently inaccurate extrapolations when the tables are skewed. Typically, the dynamic AMP samplings are stored for several hours in the data dictionary cache of the Parsing Engine to reduce the performance burden to recollect them. Teradata uses all-AMP sampling when a volatile table, a single table aggregate join index, or a single-table sparse join index database object is involved. Join indices are discussed in the Section "Table join strategy" below.

Table skew on index columns causes the most significant challenges in dynamic AMP sampling. In those cases that the root cause of skewness cannot be fixed, Teradata allows changing the number of sampled AMP's from one to two or five or all node AMP's, or even all system AMP's. At the expense of increased resource consumption, the skew effect is smoothened. Such a change must be carefully assessed as it may improve some SQL queries but worsen the performance of others. We consider a good practice to consider only single-AMP or all-AMP sampling. In our experience, anything in between (i.e., multi-AMP sampling) causes additional overhead without noticeable improvements in the estimations. An advantage of all-AMP sampling is that the resulting metrics are permanently stored in histograms, similarly to collected statistics.

Heuristics

The optimizer resorts to *heuristics* when it has no statistics for non-index columns. Heuristics are simple rules of thumb for estimating how many rows of a table will be processed by an execution step. The table cardinality is a basis for the estimates; the more it deviates from the actual count of rows, the more inaccurate all the estimations will be. In decreasing accuracy, the table cardinality is derived by collected table summary statistics, collected PI statistics, or PI statistics via dynamic AMP sampling.

When a single non-index column is included in a WHERE clause, the optimizer heuristic estimates that 10% of the table rows will match. When two or more non-index columns are combined with an AND condition, the heuristic assumes 75% of each additional column

will match the overall intersection. For example, the heuristic assumes 7.5% for a two-column AND but 5.625% for a three-column AND.

When different columns are combined with an OR condition, the heuristic adds 10% for each additional column, i.e., 20% for two columns and 70% for seven columns, up to 100% when ten or more different columns are present in the WHERE clause. When non-consecutive values of the same non-index column are combined with an OR condition, the heuristic adds 10% for the first value, 10% for the second value, and 1% for all values (including the first two) thereafter. For example, the heuristic estimates 20% for two non-consecutive values, 23% for three, and 25% for five.

When ranges of values are combined with an OR condition (e.g., through a plain OR of consecutive values, an IN list, or a BETWEEN x AND y construct), the heuristic estimates 20% for the first range, no matter how many values are contained in the range, 40% if a second range is included (again, independently of how many values each range contains). When three or more ranges are included with an OR condition, the heuristic switches to another approach: the first two ranges are estimated to contribute 10% each (instead of 20% before), and then a 1% is added for each distinct value contained in all ranges, including the first two. For example, the heuristic estimates 33% in the case of three ranges, where the first contains four values, the second seven values, and the third just two values.

The optimizer prioritizes those non-index columns that collected statistics exist when they are mixed with ones lacking statistics; the estimations are then more accurate. Let's assume the following example case of non-index columns:

```
COLLECT STATISTICS COLUMN(Age) ON Customer;
SELECT * FROM Customer WHERE Segment = 1 AND Age = 25;
```

Further, let's assume that 50% of the table rows contain value 25 for column Age. The optimizer uses this piece of information as a starting point and estimates that the total rows of the query will be 37.5% of the base table, i.e., 50% times 75% due to the AND condition. If no statistics were available, the estimation would have been six times less, as discussed before.

When statistics are available for more than one non-index column, the optimizer chooses the column with the highest selectivity as a starting point and ignores the rest to keep the heuristic calculation simple. Let's assume the following example case:

```
COLLECT STATISTICS COLUMN(Age), COLUMN(Gender) ON Customer;
SELECT * FROM Customer WHERE Age = 25 AND Gender = 'U';
```

Further, let's assume that 50% of the table rows contain value 25 for column Age, and 5% contain the value U for column Gender. The optimizer uses the latter as a starting point and estimates that the intersection will be 3.75% of the base table, i.e., 5% times 75% due to the AND condition.

A similar prioritization approach applies when the non-index columns are combined with an OR condition. Let's assume the following example case:

```
COLLECT STATISTICS COLUMN(Age) ON Customer;
SELECT * FROM Customer WHERE Age = 25 OR Gender = 'U';
```

Further, let's assume that 50% of the table rows contain value 25 for column Age. The heuristic adds to this 50% a 10% for the Gender column, for a total estimation of 60%.

The examples above indicate that heuristics can support estimating the total number of table rows to be processed. Being heuristics, they cannot be as accurate as collected statistics and may lead to sub-optimal execution plans. We consider a good practice always to collect statistics on all non-index columns involved in WHERE clauses.

Extrapolated statistics

When the collected statistics become outdated (stale), the optimizer uses *extrapolation* to estimate the current table cardinality and histogram information (specifically, count of distinct values, count of NULL rows, count of rows containing the most frequent value, and the maximum value of the histogram). Extrapolation allows for balancing the cost of recollecting statistics using outdated information that may lead to sub-optimal execution plans.

Extrapolation requires sufficient historical records for summary statistics to identify trends. When such records are not available, Teradata resorts to heuristics. Therefore, we consider a good practice not to drop and recreate tables but rather delete their rows and populate again. In the former case, Teradata assigns a new TABLE_ID, even when the human-readable table name is the same and all the historical records are lost.

Extrapolation cannot be disabled but can be skipped when collected statistics are up to date. The extrapolated estimates can be checked with a statement similar to the one in the example below:

```
HELP CURRENT STATS; -- extrapolated estimates
HELP STATS; -- actual, non-extrapolated estimates
```

The optimizer uses dynamic AMP sampling, histogram summary statistics, and historical histograms as extrapolation inputs. When the underlying table is skewed, the optimizer avoids dynamic AMP sampling, as the latter may lead to inaccurate estimates. If deemed necessary, this can be switched to an all-AMP dynamic sampling rather than the default random single-AMP one. Such a switch must be carefully assessed, as all-AMP sampling may introduce an additional performance burden to the system. The switch may be beneficial for tables with natural skew. In all cases and based on the column value change patterns, the optimizer chooses among three methods to extrapolate statistics: *rolling*, *static*, and *hybrid extrapolation*.

The *rolling extrapolation* is applied for columns with a stable number of existing rows per value but with new rows with new values added. Prime examples are date columns, timestamp columns, and columns that contain more than 95% unique values. The optimizer considers by default all date columns as rolling. This kind of extrapolation is useful for tables where periodically new rows are added per day, week, or month. Rolling extrapolation allows estimating the data demographics for date ranges that span beyond the latest contained in the already collected statistics and, thus, are not present in any histogram interval, as depicted in Figure 80.

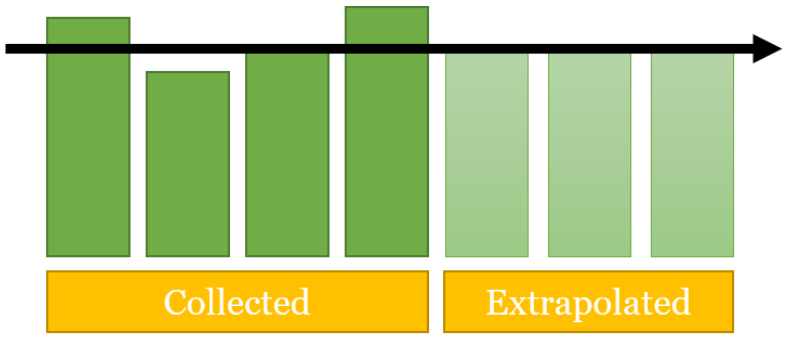

Figure 80 Extrapolating row counts (vertical axis) for new dates

Teradata determines the maximum extrapolated future date using the following formula based on a dynamic AMP sample:

> *Maximum Extrapolated Date = Maximum Date in Collected Statistics + ((Maximum Date in Collected Statistics – Minimum Date in Collected Statistics) * ((Distinct Random AMP Sample Dates * Average Rows Per Date from Collected Statistics) / (Distinct Dates from Collected Statistics * Average Rows Per Date from Collected Statistics))*

Let's assume an example case with an average of 1,000 rows per date between January 1, 2020 and March 30, 2020 (collected statistics) and ten distinct values per date (dynamic AMP sampling). Plugging these numbers in the above formula, the maximum extrapolated date is calculated to be April 9, 2020. For an SQL query for the range from March 25, 2020 until April 9, 2020, the optimizer uses exact figures from collected statistics for the first six days (until March 30, 2020) and extrapolated figures for the next ten days (until April 9, 2020) using the extrapolation formula from the dynamic AMP sampling section, as depicted in Figure 81.

Figure 81 Extrapolation for dates after the collected statistics

The estimation quality of rolling extrapolation is highly correlated with the skew of the underlying table and how the queried range overlaps with the ranges covered by collected statistics. Let's assume the three example ranges in Figure 82. Better estimations are expected for Range 1, which fully overlaps within the joined collected and extrapolated periods. Range 2 mostly overlaps with the extrapolated period; the estimations will be less accurate then. Range 3 does not overlap; then, none of the estimation formulas apply, and the quality is very low.

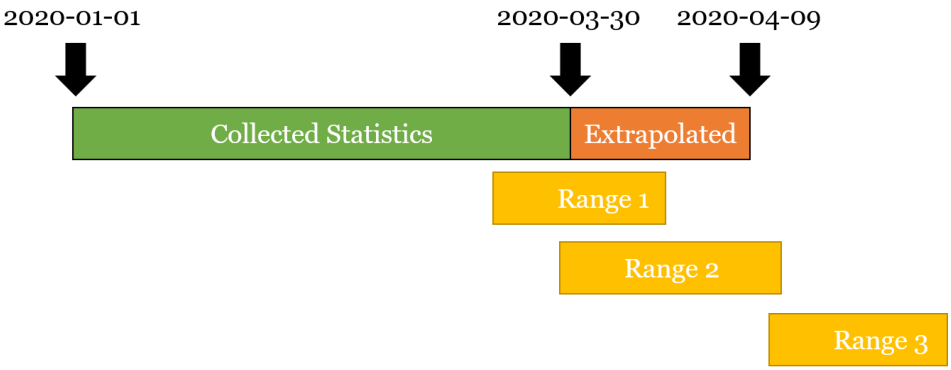

Figure 82 Range overlap with collected statistics

The *static extrapolation* is applied for columns with a stable count of distinct values but with an increasing count of rows per distinct value, as depicted in Figure 83. A prime example is a customer table with a contract type column; a more-or-less stable number of contract types exists over time while new customers are added, changing the number of customers per contract type.

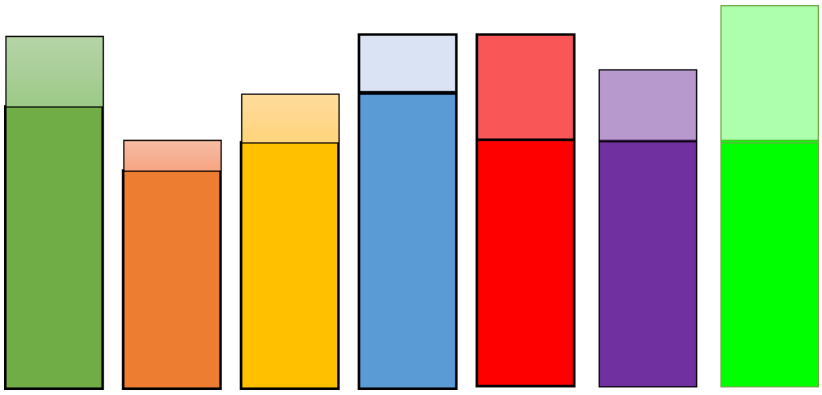

Figure 83 Stable count of distinct values and increasing row count per value

The *hybrid extrapolation* is applied when both the count of distinct column values changes and the count of rows per value grow. In such cases, Teradata applies a combination of rolling extrapolation (increase in column value counts) and static extrapolation (increase in counts of rows per column value).

Object Use Counts

Teradata 14.10 introduced the Object Use Counts (OUC) feature to aim collection of statistics. OUC logs the database object access frequency for tables, views, columns, indices, statistics, and the *UDI* counts, i.e., the count of changed rows caused by update, insert, and delete statements. Teradata distinguishes between system- and user OUC. System OUC can only be reset by Teradata, while user OUC can be reset by the user itself. OUC are stored in the `DBC.ObjectUsage` table and can be activated for a database with a statement as in the example below:

```
BEGIN QUERY LOGGING WITH USECOUNT ON tuning; -- the database name
```

OUC may be collected for *statistics* and *indices* since Teradata considers both of them as database objects. The `DBC.ObjectUsage` logs how often these objects are used. The usage of indices is additionally recorded in `DBC.IndicesV`. Usage metrics are necessary when assessing whether, from a performance point of view, it is beneficial to maintain specific statistics or indices with their associated costs.

When statistics are dropped as database objects, the related rows in `DBC.ObjectUsage` are also deleted, as depicted in Figure 84 (before the drop) and Figure 85 (after the drop). We consider a good practice to delete table rows and update their associated statistics instead of a drop-and-recreate table approach. When the latter approach is followed, the OUC are effectively reset and provide information only since the last statistics object creation.

```
1  COLLECT STATISTICS COLUMN(CustomerId) ON tuning.Customer;
2  SELECT * FROM tuning.Customer WHERE CustomerId = 1; -- Select first time
3  SELECT * FROM tuning.Customer WHERE CustomerId = 1; -- Select second time
4  FLUSH QUERY LOGGING WITH ALL; -- Write cache immediately to DBC tables
5
6  SELECT
7      t02.DatabaseName AS DatabaseName
8      , t03.TVMName AS TableName
9      , Coalesce(t04.StatsName, t04.ExpressionList, 'Summary') AS theSummary
10     , t01.UserAccessCnt AS AccessCnt
11     , CAST(t01.LastAccessTimeStamp AS DATE) AS LastAccessDate
12     , CURRENT_DATE - CAST(t04.CreateTimeStamp AS DATE) CreatedDayAgo
13
14     FROM DBC.ObjectUsage t01
15
16 INNER JOIN DBC.Dbase t02
17 ON t01.DatabaseId=t02.DatabaseId
18
19 INNER JOIN DBC.TVM t03
20 ON t01.ObjectId=t03.TVMId
21
22 INNER JOIN DBC.StatsTbl t04
23 ON t01.DatabaseId=t04.DatabaseId
24 AND t01.ObjectId=t04.ObjectId
25 AND t01.FieldId=t04.StatsId
26
27 WHERE t01.UsageType='STA'
28 AND DATABASENAME = 'tuning'
29
30 ORDER BY 1,2,3;
31
```

Teradata Result Set Viewer

Result Set - SQL Editor (1)

	DatabaseName	TableName	theSummary	AccessCnt	LastAccessDate	Created...
1	tuning	customer	CustomerId	2	2020-08-26	1

Figure 84 Creation and initial use (statistics accessed twice in queries)

```
DROP STATISTICS COLUMN(CustomerId) ON tuning.Customer;
COLLECT STATISTICS COLUMN(CustomerId) ON tuning.Customer;
SELECT * FROM tuning.Customer WHERE CustomerId = 1; -- Select third time, first after drop
FLUSH QUERY LOGGING WITH ALL; -- Write cache immediately to DBC tables

SELECT
    t02.DatabaseName AS DatabaseName
    , t03.TVMName AS TableName
    , Coalesce(t04.StatsName, t04.ExpressionList, 'Summary') AS theSummary
    , t01.UserAccessCnt AS AccessCnt
    , CAST(t01.LastAccessTimeStamp AS DATE) AS LastAccessDate
    , CAST(t04.CreateTimeStamp AS DATE) CreateDate

    FROM DBC.ObjectUsage t01

INNER JOIN DBC.Dbase t02
ON t01.DatabaseId=t02.DatabaseId

INNER JOIN DBC.TVM t03
ON t01.ObjectId=t03.TVMId

INNER JOIN DBC.StatsTbl t04
ON t01.DatabaseId=t04.DatabaseId
AND t01.ObjectId=t04.ObjectId
AND t01.FieldId=t04.StatsId

WHERE t01.UsageType='STA'
AND DATABASENAME = 'tuning'

ORDER BY 1,2,3;
```

Teradata Result Set Viewer

Result Set - SQL Editor (1) | Result Set - SQL Editor (2)

	DatabaseName	TableName	theSummary	AccessCnt	LastAccessDate	CreateDate
1	tuning	customer	CustomerId	1	2020-08-26	2020-08-26

Figure 85 Drop and use for the third time (access count is reset to 1)

The optimizer utilizes the UDI counts to enhance extrapolated data demographics with information about the table growth (i.e., changes in row counts due to INSERT and DELETE statements) and the number of distinct values for columns (i.e., changes in column values due to UPDATE, DELETE, and INSERT statements). When OUC is not activated, the optimizer uses dynamic AMP sampling to estimate the table size growth as the aggregated effect of INSERT and DELETE statements on the random sample checked. In contrast, changes in column values due to UPDATE statements cannot be detected at all. Furthermore, the estimates are susceptible to skew effects on the selected sample. In contrast, when OUC is activated, fine-grained information across all AMP's are available through the UDI counts, resulting in more accurate estimates.

Let's assume an example where a table contains only values "A" and "B" in a column for which statistics are collected. Further, assume a query for value "C", which is not present in the column. Before Teradata 14.10 and OUC, the optimizer estimates that 10% of the table rows contain value "C". Depending on the table cardinality, this may be already a million of rows estimation. When OUC is activated, Teradata is aware that no changes occurred since the last collection of statistics and, thus, the queried value is not present. In this case, the optimizer estimates that one table row contains the value to allow for concurrency effects and proper plan execution.

The UDI counts are cached and only periodically stored (by default, every ten minutes or when the cache gets full) for improved performance. Due to caching, the counts may become stale after an aborted session or a system restart. The optimizer implements a detection logic during the execution step generation to identify stale UDI counts. It compares the UDI-based table cardinality estimation against the other available methods, i.e., collected summary statistics, dynamic AMP sampling, and extrapolation from historical summary statistics. When the UDI-based estimates deviate from the latter by more than a system-defined margin, the optimizer marks the UDI counts as stale and does not consider them when creating the execution plan. Additionally, it resets them with the next collections of statistics.

The OUC feature consumes negligible resources, both for processing and storage. We consider a good practice always to activate OUC for all critical databases, as the benefits outperform the maintenance costs.

Execution steps

There are four types of execution steps that are essential from a performance point of view: *retrieve*, *join*, *aggregate*, and *sort*. By and large, the execution plans are static. Starting with Teradata 14.10, the optimizer may choose to use a *dynamic execution plan*, where the execution steps are progressively decided. In either case, the optimizer always ranks its decisions for each execution step in *confidence levels*.

Retrieve steps

The primary optimization objective for a retrieve step is to choose the most efficient data access path. The optimizer always prefers the direct access path through a PI or a USI, even when statistics are not available. This is rational, as no other access path can be better. When these paths cannot be taken, alternative ones, such as a NUSI, are carefully weighed against the cost of a full-table scan.

Join steps

Typically, the vast majority of Teradata requests are SQL queries that combine data from different tables. The optimizer can choose from many options to form a *join strategy* with the primary aim to keep the number of I/O operations and then the volume of row transfers across AMP's as low as possible.

The table join strategy is a critical optimization task; its tuning is among the top priorities from a performance point of view. Given the broadness and criticality of the topic, a dedicated section below discusses the table join strategy in detail.

Aggregation steps

Aggregation steps are typically used for SQL aggregations, i.e., the GROUP BY, SUM, MIN, MAX, AVG, and DISTINCT operators. Aggregation in a shared-nothing architecture is a

resource-intensive task, as rows must move on the same processing unit (i.e., an AMP in Teradata). Additionally, when the rows must be redistributed to different AMP's, a negative performance impact may occur. First, because of the relocation process itself, and second, the process will conclude only when all AMP's conclude their aggregation part. If the rows are not evenly distributed but are heavily skewed towards specific AMP's, then the processing time on those AMP's guides the overall performance.

Teradata implements unlimited parallelism for aggregation based on the so-called *Aggregation, Redistribution, Sort, and Aggregation (ARSA) algorithm*. Let's assume a two-AMP system and an example query of summing up a value column (e.g., the count of visitors) per another non-index column of a table (e.g., the country code). Then, both AMP's may hold values per country code, as depicted in Figure 86.

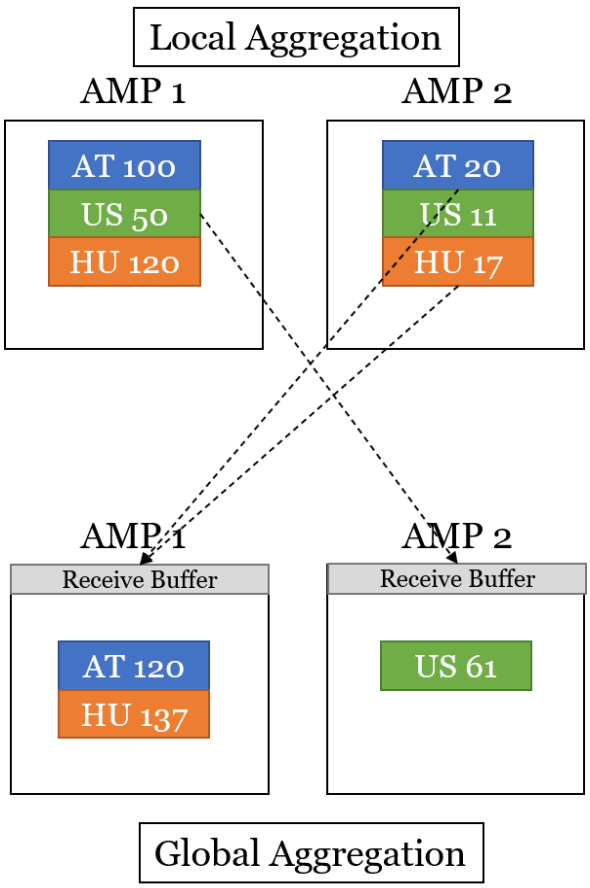

Figure 86 ALSA in action summing up amounts per country code (simplified)

In the first step (Local Aggregation), each AMP locally sums up per country code the counts in its table rows. Teradata uses a dedicated cache memory only for local aggregations. It devotes 90% of its capacity for the so-called *main cache*. The main cache is progressively filled and updated as the table is scanned for matching rows, assuming that the most

frequent group column values will be stored there (e.g., the country codes with the most rows for visitors). The remaining 10% of the capacity is devoted to the so-called *overflow cache*. A second local aggregation step may be needed for those group column values, especially when the aggregation involves many distinct values that overflow the main cache. When the overflow cache is full, the current contents are swapped to the disk to make space for new cache entries. As soon as the main cache aggregation is ready, Teradata recalls all the overflow cache snapshots from the disk, sorts the entries, and performs a second and final aggregation step over all overflow rows.

In the second step (Redistribution), the rows are assigned to an AMP and redistributed as necessary. The assignment is done by calculating the ROWHASH of all the aggregation columns (in the example: the country code). The target AMP asynchronously receives the rows from the other AMP's and buffers them in a *receive buffer* (in the example: AMP2 receives the "US 50" row from AMP1 and sends the "AT 20" and "HU 17" rows).

In the third step (Sort), the rows from the receive buffer are sorted by the aggregation columns to aid the next processing step. Finally, in the fourth step (Global Aggregation), the now-sorted rows are summed up to reach the final result (in the example: AT 120, HU 137, and US 61).

Teradata skips the redistribution, sort, and global aggregation steps under certain conditions to avoid waste of resources and achieve better performance. One such case is when the aggregation columns include the primary index of the table. Then, the rows are only locally located in the AMP's, and there is no need for a new ROWHASH and redistribution. The same holds when the ARSA input is from the AMP spool space, and the rows were already hash-distributed to the proper AMP's. When there are additional (non-primary-index) columns to be grouped, Teradata still performs the ROWHASH-based aggregation first and then applies any residual conditions, as depicted in Figure 87.

```
EXPLAIN SELECT CustomerId,CustomerNumber,COUNT(*)
FROM Customer
GROUP BY 1,2;

  1) First, we lock DWHPRO.Customer in TD_MAP1 for read on a reserved
     RowHash to prevent global deadlock.
  2) Next, we lock DWHPRO.Customer in TD_MAP1 for read.
  3) We do an all-AMPs SUM step in TD_MAP1 to aggregate from
     DWHPRO.Customer by way of an all-rows scan with no residual
     conditions, grouping by field1 (DWHPRO.Customer.CustomerId
     ,DWHPRO.Customer.CustomerNumber).  Aggregate Intermediate Results
     are computed locally, then placed in Spool 3 in TD_Map1.  The
     input table will not be cached in memory, but it is eligible for
     synchronized scanning.  The size of Spool 3 is estimated with high
     confidence to be 2,716,318 rows (100,503,766 bytes).  The
     estimated time for this step is 29.94 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.  The size of Spool 1 is estimated with
     high confidence to be 2,716,318 rows (89,638,494 bytes).  The
     estimated time for this step is 5.57 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 35.51 seconds.
```

Figure 87 Example of aggregation performed already in local AMP

The approach for handling DISTINCT aggregations has evolved over Teradata versions. There was no pre-processing performed in the past, i.e., a first local aggregation step to remove duplicate rows. Instead, all the matching local rows were immediately distributed to the target AMP for processing. When the aggregation columns contain many distinct values, this approach for DISTINCT was outperforming the equivalent GROUP BY one, as the local aggregation step of the latter was skipped. Based on the number of distinct values on the aggregation columns, as estimated by statistics or a random data sample, newer Teradata versions assess whether a local aggregation step should be performed to reduce the number of to-be-transferred rows or this would result in an additional processing delay.

Sort steps

Teradata honors the shared-nothing architecture and performs any sorting task included in a request (i.e., an ORDER BY clause) at each processing level (AMP, Node, BYNET, and Parsing Engine) and entirely in parallel, without any row redistribution. At first, the rows of the table are sorted on each AMP, as the last step of an execution plan. The more the AMP's, the higher the performance, as the sorting effort is shared among them, provided the rows are already evenly distributed.

Once the AMP's finish the local sort step, the result is placed in their local spool table. Teradata creates a *global buffer* with the first part of the merged result and notifies the requestor that the result is available. The requestor progressively fetches rows from the buffer. Concurrently, Teradata fetches more rows from the AMP spool space and through the BYNET sorts and fills the global buffer with the next part of the result. The whole process continues until the requestor stops requesting additional parts or the result has been delivered in full.

The performance advantage is that only rows actually consumed by the requestor are sorted in the last global merge step. Hence, no resources are used for a potentially unneeded result. Typically, this is the case of interactive (human-driven) sessions, where result fetching is aborted after inspecting a few hundred rows.

Incremental Planning and Execution

Incremental Planning and Execution (IPE) is an advanced optimizer option available since Teradata 14.10. IPE allows creating adaptive (dynamic) execution plans to enhance the static ones discussed until now. IPE can adjust the execution plan based on information that becomes available only during plan execution. A typical application example is complex SQL queries; IPE splits them into fragments and progressively optimizes each next fragment. Opportunities for row partition elimination, new data access paths, simplified query predicates, improved cost estimates based on dynamic statistics sampling, or even table join elimination are examples of dynamic plan execution.

The optimizer activates IPE only when the estimated run time exceeds one minute and the parsing time is less than 10% of the estimated run time. Additionally, the optimizer must

be able to receive dynamic feedback from within the request. When all these conditions are met, IPE is activated and, as a first step, the optimizer calculates a static plan. Then, it decides whether an IPE would be beneficial. In this case, the optimizer utilizes all available system information and data demographics to calculate a so-called *plan fragment*. The plan fragment is immediately dispatched for execution, and the optimizer utilizes the execution feedback (i.e., result spool and *dynamic statistics*) to calculate one or more subsequent plan fragments. This process repeats until the final plan fragment concludes execution, and the result is returned to the requestor.

Dynamic statistics are different from the collected statistics of the base tables discussed earlier in this section. The dynamic statistics are calculated from the plan fragment spool space. As such, they do not cause additional disk I/O operations. Further, they are not permanently stored but instead discarded once the execution concludes. Dynamic statistics include row count number, count of NULL values, count of unique values, and the most frequent values.

An SQL statement can be checked for IPE eligibility by analyzing whether its EXPLAIN contains the phrases "*This request is eligible for incremental planning and execution (IPE). The following is the static plan for the request*" or "*This request is eligible for incremental planning and execution (IPE) but does not meet cost thresholds. The following is the static plan for the request.*" Both are eligible, but the latter means that a dynamic execution plan will not be used, as the estimated run time is less than one minute.

The dynamic execution plan can be checked by prepending the EXPLAIN statement with the keyword DYNAMIC, as in the example below:

```
DYNAMIC EXPLAIN SELECT * FROM tuning.schedule_PPI;
```

There are two main phrases in the explain plan to identify a dynamic one: "*The following is the dynamic plan for the request*" and "*We send an END PLAN FRAGMENT step for plan fragment n*". When an intermediate plan fragment feedback is returned, the phrase "*We do an All-AMPs FEEDBACK RETRIEVE step*" is included while dynamic statistics include the phrase "*We do an All-AMPs FEEDBACK RETRIEVE step from spool n of spool m statistics*"

We consider a good practice to refrain from using DYNAMIC EXPLAIN as a shortcut to check if a statement is IPE-eligible. The optimizer does execute the plan fragments, causing resource usage.

A dynamic execution plan may improve performance for specific workloads, but it may also introduce performance overhead. A static execution must always be calculated as a basis for IPE. Dynamic statistics are also calculated to eliminate additional I/O operations; however, the latter cannot always be avoided. Thus, the use of IPE must be carefully assessed on a case-by-case basis. The following statements can be used to control how the

optimizer utilizes IPE, even for requests that do not meet dynamic execution eligibility criteria:

```
SET QUERY_BAND = 'DynamicPlan=SYSTEM;' FOR SESSION; -- let the system decide (default)
SET QUERY_BAND = 'DynamicPlan=SYSTEMX;' FOR SESSION; -- bypass one-minute threshold check
SET QUERY_BAND = 'DynamicPlan=OFF;' FOR SESSION; -- disable IPE
```

Confidence levels

For every retrieve, join, or aggregate step to be taken, the optimizer ranks its decision in one of the four confidence levels: *No Confidence*, *Low Confidence*, *High Confidence*, and *Index Join Confidence*. The lower the confidence, the more conservative the optimizer will be in selecting a step action. This is necessary to avoid wrong estimates and costly step executions.

A "No Confidence" level occurs when the optimizer is missing statistics for non-index columns, or it has no information about the queried column and, thus, it must rely on heuristics. A no-confidence step may be improved by collecting the missing statistics, as it will be discussed in the next sections.

A "Low Confidence" level may occur:

- For a Retrieve step: it might be the case that the statistics do not match the filter condition (e.g., the WHERE clause), but they can still be used (e.g., the case of multi-column statistics discussed in a next section).
- In a Join step, if statistics exist independently on both tables for the joined columns. In this case, the optimizer cannot know how many rows the joined table will contain.
- When statistics were not collected on the base table columns, but a NUSI can be used in the WHERE clause of the statement. In this case, Teradata uses dynamic AMP sampling to extract estimates.

A "High Confidence" level is the desired one from a performance point of view. This level is achieved when a filter condition involves indices or columns, and statistics are available for them. It allows the optimizer to generate the most optimistic execution plan to achieve the best possible performance. Still, there is no guarantee that the actual execution will match the assumptions of the optimizer. Instead, a "High Confidence" means that the optimizer considers the available statistics to represent the actual data demographics. This might not always be the case. For example, the optimizer treats as equal the sampled and the full-table statistics. As sampled statistics are skew-sensitive, they can provide high quality but still inaccurate estimates. Transient effects in between the time data demographics change (e.g., due to a massive data import) and the time when Teradata marks the available statistics as stale are another case of mismatch between the reality and the assumptions of the optimizer.

An “Index Join Confidence” level is a special one for only join steps. It is achieved when an index can be used for one of the two joined tables, even when statistics are not available for this table, irrespective of the existence of an index or statistics of the second table.

Typically, the optimizer does not achieve a “High Confidence” level for all the execution plan steps. As the confidence level of a step propagates to the subsequent steps, the more the execution steps, the less the confidence may be, down to “No Confidence”. As Teradata treats most of the times “High Confidence” and “Low Confidence” with equal importance, this is not necessarily a problem.

The statistics are a starting point for the estimates of the optimizer. When calculating an execution plan, additional parameters are considered, such as table CHECK constraints, referential integrity on columns, and query predicates. All these result in so-called *derived statistics* stemming from the initial estimations. Additionally, the optimizer incorporates additional information into the estimates of a step. Retrieve, join, and aggregate steps typically provide improved estimates for the subsequent steps. For example, a filter “WHERE columnX IN (1,2,3)” implies that the step output will contain at most three distinct values; this piece of information is then passed as a starting point to subsequent execution steps.

Table join strategy

Teradata joins two tables at a time, and the table rows to be joined must reside on the AMP. The primary objective of the optimizer when deciding a table join strategy is to minimize the use of spool space. Teradata considers multiple factors when deciding a join strategy, including:

- How many rows must be compared for the join?
- How many bytes must move from AMP to AMP?
- How many rows must be sorted after the move?
- How often must a data block be copied from AMP VDisk to its memory?
- How many data blocks must be copied in total?

A join strategy comprises two components: a *join preparation* and a *join method*. The former is about optimizing the redistribution of needed table rows to the target AMP’s, where the joined table row will be produced. The latter is about the algorithm used by the target AMP’s to combine the rows of the two tables into one joined table.

The total cost for a table join is the sum of the join method cost plus the join preparation cost. As the two join strategy components are tightly coupled, the two costs cannot be considered independently. There are join methods that incur minimal, if any, preparation costs but have high processing costs, while other join methods have low processing costs and offload the bulk of the work to the join preparation. Some join methods require the rows to be sorted by ROWHASH before processing, while others can process the rows as received.

There is no absolute best or worse join strategy; each performs better in specific usage scenarios. The performance tuning priority should always be that the optimizer is equipped with accurate information about table data demographics so that it can choose the most appropriate join strategy.

Join preparation tuning

The join preparation is skipped when the table rows to be joined reside already on the same AMP. This is the case when two or more tables share the same primary index, which highlights the importance of a proper logical and physical design model, as discussed in Chapter 3. When the logically-connected rows are on the same AMP, the costly cross-AMP transfers are avoided. When they are on different AMP's, the join preparation must transfer the rows of one or even both tables to other AMP's. Depending on the selected join method, the join preparation may either copy intact the whole table to all the AMP's or perform *row rehashing*, i.e., calculate a new primary index matching the join columns and redistribute the rows accordingly to AMP's.

The optimizer always distributes the smaller-in-bytes table to the larger one, based on available statistics or heuristic estimations. This allows reducing the traffic volume across AMP's. The to-be-distributed table is identified by multiplying the count of qualifying rows by the number of bytes all the selected columns occupy. We consider a good practice to filter for the qualifying rows (e.g., using a WHERE clause) the earliest possible so that they are not moved to the spool space only to be removed later on. We also consider a good practice to refrain from using SELECT * out of convenience; it increases the spool space consumption during joins and may mislead the optimizer which table is effectively smaller. The example case in Figure 88 achieves a 95% reduction of spool space just by selecting the desired column compared to issuing a SELECT * statement.

```
1  CREATE MULTISET TABLE tuning.SpoolSpaceTest  (
2       Id INTEGER
3       ,Description CHAR(500)
4  ) PRIMARY INDEX (Id);
5
6  INSERT INTO tuning.SpoolSpaceTest
7  SELECT ROW_NUMBER() OVER (ORDER BY 1) AS Id, 'RandomString'
8  FROM SYS_CALENDAR.CALENDAR;
9
10 SET QUERY_BAND = 'Columns=All;' FOR SESSION;
11 SELECT * FROM tuning.SpoolSpaceTest;
12
13 SET QUERY_BAND = 'Columns=1;' FOR SESSION;
14 SELECT Id FROM tuning.SpoolSpaceTest;
15
16 SET QUERY_BAND = NONE FOR SESSION;
17 SELECT Queryband,SpoolUsage
18 FROM DBC.DBQLOGTBL
19 WHERE QUERYBAND LIKE '%Columns%';
20
21 --
22 -- QueryBand        SpoolUsage
23 -- =S> Columns=All; 52 649 984
24 -- =S> Columns=1;    2 613 248
25
```

Figure 88 Spool space savings by selecting only needed columns

The join preparation actions depend on the row demographics in AMP spool space and the join method selected by the optimizer. The next sections discuss the different join methods available and their requirements for join preparation.

Join indices

The *join index* is a Teradata object that can be defined to avoid costly data access paths to base tables during joins. A join index is more like a base table itself rather than an index. Primary indices, secondary indices, and row partitions can be defined over a join index. In contrast to secondary indices, the join index objects are stored in a user-defined database. A list of the defined join index objects in a database is available with a query similar to the example below:

```
SELECT * FROM DBC.INDICESV WHERE DATABASENAME = 'tuning' AND INDEXTYPE = 'J';
```

The primary use of a join index is to perform *pre-joins of tables*, i.e., perform a table join only once at an optimal step of the execution plan; afterward, the optimizer reuses the joined table result and avoids costly re-execution of the table joins. The larger the pre-joined tables, the more the performance gain. When two tables are to be joined, the optimizer assesses whether and which join index can be utilized, considering different factors, such as I/O operations, data distribution, and index coverage. Depending on the query structure, a join index might result in duplicate rows in the output. To account for this, the optimizer adds a sort or delete execution step at the end to filter them out. The cost of this step is also accounted for when assessing the use of a join index.

A join index may prove beneficial in usage scenarios beyond table pre-joins, including:

- Scanning a covering join index as a replacement for a full table scan of the base table. Significant performance improvements can be achieved when the join index data blocks are fewer than those of the base table.
- Represent a base table through a different primary index. In this case, the rows of the join index are redistributed to different AMP's. This eliminates the need to redistribute the table rows on every single execution plan repeatedly. A join preparation step is avoided, as the rows are already accessible in the target AMP.
- Perform pre-aggregations based on one or more tables. The join index stores the aggregation result and dynamically updates it on every change of the underlying data. Teradata performs the re-aggregation only on those parts necessary.
- Early resolution of complex expressions. Like pre-aggregations, the results are readily available as columns of the join index object and dynamically updated. Further, it becomes possible to collect statistics over these columns and help the optimizer choose a better join strategy.
- Enhance row partitioning for improved joins without impacting the base table. As these are two different database objects, different partition strategies may be applied to each one. Still, using a row-partitioned join index over a non-partitioned base

table must be carefully assessed; typically, such a design has a negative performance impact.

- Maintain a normalized PDM while improving performance. A join index can offer a denormalized view of the PDM without changing the underlying PDM. The system automatically resolves in the background all the complexities of a denormalized PDM (e.g., those for INSERT and UPDATE statements) during the join index maintenance when the underlying base tables change.

The next sections explore the different types of join indices, their applicability in different usage scenarios, and the associated tuning considerations.

Sparse join index

A *sparse join index* allows for indexing only those rows that are important for selected WHERE conditions, such as filtering out NULL values. The optimizer may use a sparse join index when the WHERE condition matches the join index definition. A sparse join index increases access performance and reduces maintenance costs and space consumption. Any single-table, multi-table, or aggregate join index can be defined as sparse, using a statement similar to the example below:

```
CREATE JOIN INDEX Customer_JI AS SELECT CustomerId FROM Customer WHERE LastName IS NOT NULL
PRIMARY INDEX (CustomerId);
```

The sparse join index should be used with caution, as it might exhibit unintended behavior. The WHERE conditions may render impossible to take the data access path over the primary index of the join index; then, the optimizer may resort to a full table scan of the sparse join index, as depicted in Figure 89.

```
DROP JOIN INDEX Customer_JI;

CREATE JOIN INDEX Customer_JI AS
SELECT CustomerId
FROM Customer
WHERE CustomerId = 1
PRIMARY INDEX (CustomerId);

EXPLAIN SELECT CustomerId
FROM Customer
WHERE CustomerId=1;

  1) First, we lock DWHPRO.CUSTOMER_JI in TD_MAP1 for read on a
     reserved RowHash to prevent global deadlock.
  2) Next, we lock DWHPRO.CUSTOMER_JI in TD_MAP1 for read.
  3) We do an all-AMPs RETRIEVE step in TD_MAP1 from DWHPRO.CUSTOMER_JI
     by way of an all-rows scan with no residual conditions into Spool
     1 (group_amps), which is built locally on the AMPs.  The size of
     Spool 1 is estimated with high confidence to be 1 row (25 bytes).
     The estimated time for this step is 0.00 seconds.
  4) Finally, we send out an END TRANSACTION step to all AMPs involved
     in processing the request.
  -> The contents of Spool 1 are sent back to the user as the result of
     statement 1.  The total estimated time is 0.00 seconds.
```

Figure 89 A sparse join index resulting in a full-table scan

A sparse join index on top of a row-partitioned base table may achieve a positive performance impact as Teradata accesses only those partitions necessary when creating the join index. Let's assume the row-partitioned base table and sparse join index creation depicted in Figure 90.

```
CREATE SET TABLE DWHPRO.Sale_PPI
(
     SaleId INTEGER NOT NULL,
     CustomerId INTEGER NOT NULL,
     ProductId INTEGER NOT NULL
) PRIMARY INDEX ( SaleId )
PARTITION BY (RANGE_N (SaleID BETWEEN 1 AND 1000 EACH 1));


CREATE JOIN INDEX Sale_JI
AS
 SELECT * FROM Sale_PPI
 WHERE SaleId IN (5,6,7);
```

Figure 90 PPI table and sparse join index definition

The execution plan for the join index creation is depicted in Figure 91. The optimizer creates a three-value lookup table for the SaleId column (Step 7). This lookup table is then joined with the three partitions of the base table (Step 9), using a merge join. Thus, the join index is created without resorting to a full table scan of the base table.

```
7) We do an INSERT step into Spool 2 with rows (SaleId) values (7), (
     6), (5), which is redistributed by the hash code of (SaleId) to
     all AMPs.
  8) We do an all-AMPs RETRIEVE step in TD_Map1 from Spool 2 (Last Use)
     by way of an all-rows scan into Spool 3 (all_amps), which is built
     locally on the AMPs.  Then we do a SORT to partition Spool 3 by
     rowkey.  The size of Spool 3 is estimated with high confidence to
     be 3 rows (51 bytes).  The estimated time for this step is 0.00
     seconds.
  9) We do an all-AMPs JOIN step in TD_Map1 from Spool 3 (Last Use) by
     way of a RowHash match scan, which is joined to 3 partitions of
     DWHPRO.Sale_PPI by way of a RowHash match scan with no residual
     conditions.  Spool 3 and DWHPRO.Sale_PPI are joined using a
     rowkey-based merge join, with a join condition of ("SaleId =
     DWHPRO.Sale_PPI.SaleId").  The result goes into Spool 1 (all_amps),
     which is built locally on the AMPs.  Then we do a SORT to order
     Spool 1 by the hash code of (DWHPRO.Sale_PPI.SaleId).  The size of
     Spool 1 is estimated with low confidence to be 2 rows (50 bytes).
     The estimated time for this step is 0.00 seconds.
 10) We do an all-AMPs MERGE step in TD_MAP1 into DWHPRO.Sale_JI from
     Spool 1 (Last Use).  The size is estimated with low confidence to
     be 2 rows.  The estimated time for this step is 1 second.
 11) We create the index subtable on DWHPRO.Sale_PPI.
```

Figure 91 Execution plan for sparse join index creation (excerpt)

Compressed join index

A *compressed join index* stores its physical rows in a compressed form to reduce the disk storage space requirements. The index groups one or more columns into two fields. The

first is the *grouping field* and is used to compress common column values (i.e., define a group). The second is the *repeating field* and stores the non-compressed column values (i.e., define the group members). Let's assume an example of a table storing a segment-customer relationship, as follows:

```
CREATE SET TABLE Customer (CustomerId BIGINT NOT NULL, SegmentId BIGINT NOT NULL, CategoryId
BIGINT NOT NULL) PRIMARY INDEX (CustomerId);
CREATE JOIN INDEX Seg_Cust_JI AS SELECT (SegmentId),(CustomerId) FROM Customer;
CREATE JOIN INDEX Cust_Seg_JI AS SELECT (CustomerId),(SegmentId) FROM Customer;
CREATE JOIN INDEX Cust_NoCompress_JI AS SELECT CustomerId,SegmentId FROM Customer;
```

The first compressed join index definition (Seg_Cust_JI) groups the customers per different segment, as depicted in Figure 92, while the second one (Cust_Seg_JI) groups the segments per customer. A rational assumption is that the number of segments is significantly less than the number of customers (or else, segmentation would not be needed in the first place) and that each customer belongs to one segment (or just a few). Under these assumptions, the first index achieves significant savings compared to the second one:

```
INSERT INTO Customer SELECT ROW_NUMBER() OVER (ORDER BY 1) AS CustomerId, RANDOM(1,10) AS
SegmentId, RANDOM(1,100) AS CategoryId FROM SYS_CALENDAR.CALENDAR;
SELECT TABLENAME,SUM(CURRENTPERM) FROM DBC.TableSize WHERE DATABASENAME = 'tuning' AND TABLENAME
LIKE '%Cust%JI%' GROUP BY 1;
-- Seg_Cust_JI 442368
-- Cust_Seg_JI 1982464
-- Cust_NoCompress_JI 1236992
```

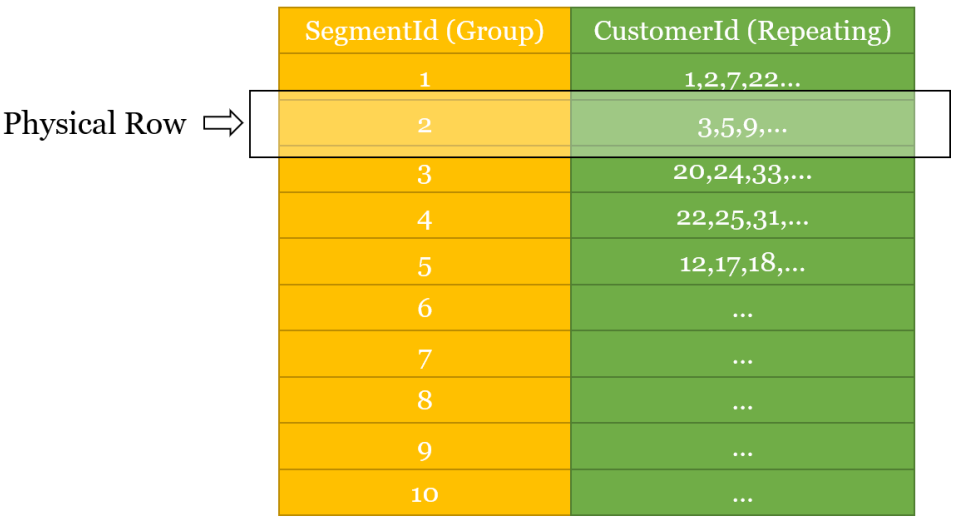

Figure 92 Compressed join index example

The second compressed join index definition requires five times more the storage space of the first one and 50% more space than an uncompressed join index due to its complex structure. Thus, it is unsuitable for this example.

The definition of a compressed join index must be carefully assessed and consider the actual data demographics and base table definitions. A topic of attention is the definition of the primary index of a compressed join index object. The primary index columns must also belong to the set of grouping columns (i.e., first field). Should this be not the case, Teradata returns an error message "[5564] Error in Join Index DDL, A primary index's field does not belong to field 1."

A compressed join index may be defined only for multi-table, single-table, and aggregate join indices, which are discussed below. Neither the primary index of a compressed single-table join index nor of a multi-table one can be unique. A UPI on the grouping field would not allow any compression, as the relation between the grouping field and the repeating one would be one-to-one. However, as there are technical limitations on the size of the physical row, there might be multiple join index rows with the same grouping field value. Should a definition includes a UPI, as in the example below, Teradata returns an error message "[5564] Error in Join Index DDL, Only primary index of non-compressed single-table JI can be unique":

```
CREATE JOIN INDEX Customer_Compressed_JI3 AS SELECT (SegmentId),(CustomerId) FROM Customer
UNIQUE PRIMARY INDEX (SegmentId); -- results in error 5564
```

Multi-table join index

The *multi-table join index* is a join index that pre-joins two or more base tables. The primary use is to fully cover the query needs in table columns so that access to the base tables is avoided and this reduces join processing down to a join index table access. For this to happen, the index must contain all the columns mentioned in the SELECT and the join conditions.

A multi-table join index can only have a NUPI. This is a technical limitation enforced by Teradata for performance reasons. A UPI would require a costly duplicate check during both the initial creation of the index and on every change of the contents of the involved base tables.

There are several considerations for designing an as-generic-as-possible multi-table join index. By and large, the join index definition must match the types of the query joins. A join index definition based on inner joins can be utilized only by queries containing inner joins of the respective tables and columns. In contrast, both outer and inner joins can utilize a definition based on outer joins. In contrast, when the query contains left joins, an inner join index cannot be utilized, as not all rows of the base tables are included in the index definition. We consider a good practice to define a multi-table join as an outer join rather than an inner join and ensure wider applicability at the expense of increased storage space.

A multi-table join index definition based on outer joins must contain at least one not-nullable column of each joined table. This is a necessary restriction; otherwise, it is impossible to separate between a no matching inner-join table and a matching table where

all rows contain NULL-valued columns. An attempt to create such an index, similar to the example below, results in an error "[5564] Error in Join Index DDL, All selected columns of an inner table are nullable":

```
CREATE JOIN INDEX CustomerSegment_JI AS
SELECT c.CustomerId, c.SegmentId, s.SegmentText
FROM Customer c
LEFT JOIN Segment s
ON s.SegmentId = c.SegmentId
AND s.SegmentText LIKE '%12%'
PRIMARY INDEX (SegmentId); -- results in error 5564
```

An outer-based join index definition must specify the join direction, either LEFT or RIGHT. A FULL outer join is prohibited, and any attempt to define one results in error "[5464] Error in Join Index DDL, Full Outer Joins are not allowed."

The rows of the multi-table join index may be sorted inside the data blocks by the ROWHASH of the primary index or by the values of any other columns for increased performance. This *value ordering* is similar to the case of a NUSI and is only available for 4-byte numeric columns, such as integer and date data types. A value-ordered multi-table join index can be used for range scans. In the example case below, the optimizer decides that the join index can be utilized; therefore, adds one more range condition, as depicted in Figure 93:

```
CREATE JOIN INDEX CustomerSegment_JI AS
SELECT c.CustomerId, c.SegmentId, s.SegmentDate, s.SegmentText
FROM Customer c
INNER JOIN Segment s ON s.SegmentId = c.SegmentId
ORDER BY (SegmentDate) -- force value ordering
PRIMARY INDEX (SegmentId);
```

```
EXPLAIN
SELECT c.CustomerId,c.SegmentId,S.SegmentDate,S.SegmentText
FROM
    Customer c
INNER JOIN
    Segment s
ON
    segmented = s.SegmentId
WHERE SegmentDate BETWEEN DATE'2019-01-01' AND DATE'2019-03-31';


1) First, we lock DWHPRO.CUSTOMERSEGMENT_JI in TD_MAP1 for read on a
reserved RowHash to prevent global deadlock.
2) Next, we lock DWHPRO.CUSTOMERSEGMENT_JI in TD_MAP1 for read.
3) We do an all-AMPs RETRIEVE step in TD_MAP1 from
DWHPRO.CUSTOMERSEGMENT_JI by way of an all-rows scan with a
condition of ("(DWHPRO.CUSTOMERSEGMENT_JI.SegmentDate <= DATE
'2019-03-31') AND (DWHPRO.CUSTOMERSEGMENT_JI.SegmentDate >= DATE
'2019-01-01')") into Spool 1 (group_amps), which is built locally
on the AMPs. The input table will not be cached in memory, but it
is eligible for synchronized scanning. The size of Spool 1 is
estimated with no confidence to be 5,073,697 rows (1,253,203,159
bytes). The estimated time for this step is 25.45 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 25.45 seconds.
```

Figure 93 Value-ordered multi-table join index used for date range search

The use of a multi-table join must be carefully assessed. On the one hand, it allows performing denormalization without changing the underlying PDM, achieving reduced complexity and improved read performance for the related queries. On the other hand, this comes at the expense of increased processing overheads when the base table contents change, e.g., through INSERT or UPDATE statements. Also, it requires adjustments in the data import processes, as bulk load tools cannot operate on tables with a join index defined. Increased storage space requirements must not be overlooked; the use of spare or compressed index forms may reduce the impact on storage.

Single-table join index

The *single-table join index* is a special case of a multi-table comprising columns of the same table. Typically, a single-table join index is utilized to realize an intermediate "bridge" table to help the optimizer skip a join preparation step. The tables are joined over the conveniently-defined primary index, and unneeded rows are automatically filtered out. A similar result may be achieved using a NUSI. The main advantage of a single-table join index over a NUSI is that the latter always requires table-level locks, while the former may only need ROWHASH locks. Then, concurrency is improved.

In the example below, a single-table join index is defined, and the specific query following can take advantage of the newly-defined index to avoid the join preparation step:

```
CREATE JOIN INDEX Customer_SJI AS
SELECT c.CustomerId, c.SegmentId FROM Customer c
PRIMARY INDEX (SegmentId); -- a different primary index from the base table

SELECT c.CustomerId, c.SegmentId, s.SegmentDate, s.SegmentText
FROM Customer c
INNER JOIN Segment s
ON s.SegmentId = c.SegmentId; -- matching primary index with the single-table join index one
```

A single-table join index may be further enhanced to perform a *join back* to the base table and access any missing columns for a query. A join back is realized by adding the ROWID in the single-table join index, similar to the example below:

```
CREATE JOIN INDEX Customer_JBI AS
SELECT c.CustomerId, c.SegmentId, ROWID FROM Customer c -- added ROWID
PRIMARY INDEX (SegmentId); -- a different primary index from the base table

SELECT c.CustomerId, c.LastName, c.SegmentId, s.SegmentDate, s.SegmentText
FROM Customer c
INNER JOIN Segment s
ON s.SegmentId = c.SegmentId
WHERE c.CustomerId <= 1000; -- additional Customer columns accessed via join back
```

The join back is performed in three steps. The two tables are joined utilizing the single-table join index and its primary index in the first step. In the second step, the matching rows are hashed again using the primary index of the base table and redistributed to the

responsible AMP's. Finally, each AMP utilizes the ROWID to retrieve the additional columns for the matching rows and produce the final result.

A single-table join index can be value-ordered for improved performance on value range queries, as in the case of multi-table; to support join back too, the ROWID may also be added in the index, similar to the example definition below:

```
CREATE JOIN INDEX Segment_VOJI AS
SELECT s.SegmentID, s.SegmentDate, ROWID FROM Segment s ORDER BY SegmentDate
PRIMARY INDEX (SegmentId); -- value ordered and ROWID added

SELECT s.SegmentID, s.SegmentDate, s.SegmentText -- last column accessed via index join back
FROM Segment s
WHERE s.SegmentDate BETWEEN DATE '2020-01-01' AND DATE '2025-12-31' -- value-range search in
index
```

The use of a single-table join index must be carefully assessed. On the one hand, such an index requires additional storage space to the base table and increases the maintenance cost for UDI operations. Since a different primary index is used, the index rows reside in different AMP's to the ones of the base table. On the other hand, join preparation may be skipped and ROWHASH locks instead of table-level ones increase concurrency. When the index covers the query columns, no base table access is necessary, improving both performance and concurrency.

We consider a good practice to consider in the analysis the base table row counts, the index frequency use, and the query demographics to make an informed decision whether a single-table join index should be defined, a NUSI should be preferred, or no additional index should be defined at all. In our experience, a single-table join index is an excellent choice for tables storing relationships between two objects, where queries may be performed from either side. In such cases, a primary index defined over the two columns of the relation cannot be used for most of the queries. Respectively, a primary index over one of the two columns does not support the queries for the other side of the relationship. A single-table join index allows to effectively define *two* primary indices for the same table: one on the base table, covering the first side of the relationship, and one on the index table, covering the second side.

Aggregate join index

The *aggregate join index* provides pre-calculated aggregate data over single or multiple tables, aiming to avoid costly expression calculations on each query execution. An aggregate join index can be defined using a statement similar to the example below:

```
CREATE JOIN INDEX CustomerSegment_AJI AS
SELECT c.CustomerId, c.SegmentId, Count(*) AS theCount
FROM Customer c INNER JOIN Segment s ON s.SegmentId = c.SegmentId
GROUP BY 1,2
PRIMARY INDEX (SegmentId);
```

An aggregate join index can only perform INNER JOIN between tables with at least one equality condition. An OUTER JOIN may cause unmatched rows in the index, as described earlier, for the case of multi-table join indices. An attempt to use an OUTER JOIN results in an error message "[5464] Error in Join Index DDL, Outer join is not allowed in aggregate join indexes."

An aggregate join index definition may only contain MAX, MIN, COUNT, and SUM as an aggregation expression. An attempt to use another one results in an error message "[5464] Error in Join Index DDL, Only columns and SUM, COUNT, MIN, MAX or non-aggregate expressions with aliases are allowed in select list." Furthermore, the definition cannot include HAVING or QUALIFY expressions. An attempt to use them results in an error message "[5464] Error in Join Index DDL, Having clause is not allowed."

The aggregate join index may be utilized in aggregations not mentioned in its definition. For example, the row count is always stored in the index for maintenance purposes. A SUM-based join index can also serve an equivalent AVG-based query: Teradata divides the stored SUM by the stored COUNT to derive the requested AVG.

The join index may also be utilized for queries of a subset of its columns, similar to the example below:

```
SELECT c.CustomerId, c.SegmentId
FROM Customer c
INNER JOIN Segment s
ON s.SegmentId = c.SegmentId
GROUP BY 1,2 -- may utilize CustomerSegment_AJI

SELECT c.CustomerId, Count(*) AS theCount
FROM Customer c
INNER JOIN Segment s
ON s.SegmentId = c.SegmentId
GROUP BY 1,2 -- may utilize CustomerSegment_AJI and sum up theCount per SegmentId, avoids FTS
```

The aggregate join index cannot be used if its grouping columns are only a subset of the ones of the query. In such cases, the aggregation granularity of the index is not sufficient to extract the query result, as in the example below:

```
SELECT c.CustomerId, c.SegmentId, c.Lastname, Count(*)
FROM Customer c
INNER JOIN Segment s
ON s.SegmentId = c.SegmentId
GROUP BY 1,2,3 -- cannot utilize CustomerSegment_AJI, c.Lastname not part of the index
```

Similar constraints apply for the WHERE conditions of the query. An aggregate join index cannot be used when the WHERE clause contains conditions on columns that are not part of the index definition, as in the example below:

```
SELECT c.CustomerId, c.SegmentId, Count(*)
FROM Customer c
INNER JOIN Segment s
ON s.SegmentId = c.SegmentId
WHERE c.Lastname = 'Smith'
GROUP BY 1,2 -- cannot utilize CustomerSegment_AJI, c.Lastname not part of the index
```

An aggregate join index can be row-partitioned to exploit partition elimination and improve performance. However, the aggregated columns themselves cannot be used in the partitioning expression. An attempt to create such an index, like in the example below, results in an error message "[5714] Invalid partitioning expression for PARTITION BY":

```
CREATE JOIN INDEX Customer_RPAI AS
SELECT c.CustomerId, c.SegmentId, Count(*) AS theCount
FROM Customer c
GROUP BY 1,2
PRIMARY INDEX (CustomerId)
PARTITION BY (theCount);
```

A typical usage scenario for an aggregate join index is replacing manually-maintained summary tables in data marts. The latter approach allows for shifting the population of such data mart tables when the impact on DWH operations and performance is unnoticeable. On the other hand, the latter approach must cope with multiple challenges. First, a permanent summary table must always be maintained. Then, any changes in to-be-aggregated data require a full recalculation from scratch. Also, the data mart queries must be maintained and redirected to use the provided summary tables rather than the ones already defined in the PDM of the DWH. Furthermore, the summary tables reflect the base data when the summary table was populated; the current DWH data may have changed since then.

Teradata transparently maintains an aggregate join index and addresses all the challenges mentioned above, as the index is updated each time the underlying data change. This is a tradeoff to assess for any join index definition: the access performance is improved at the expense of increased processing of DML statements.

Design considerations

A join index has many similarities with a secondary index. From a performance tuning perspective, the need for an additional index (join or secondary) must be periodically assessed based on actual data and query demographics, and system and operational constraints.

Teradata prohibits the definition of a join index over large-object columns (e.g., CLOB and BLOB). Similarly, set operations (e.g., UNION, INTERSECT, and MINUS) and subqueries (e.g., IN, NOT IN, EXISTS, and NOT EXISTS) cannot be part of a join index definition. An attempt to create such an index results in an error message "[5464] Error in Join Index DDL, Derived table is not allowed."

A join index requires additional storage space and additional maintenance costs. The latter is more important when the join index definitions include complex query expressions or aggregations. Similar to the case of secondary indices, a join index prohibits the use of bulk table loading (e.g., via the FastLoad and MultiLoad tools). When a join index must be used, all the join indices must be dropped before the bulk load and recreated afterward. This may become a prohibitively expensive operation from a tuning perspective. We consider a better alternative to performing the bulk load on an empty, NOPI staging table and then performing the DML statements to the target table. Teradata can then properly update the related join indices during the statement execution. It may also reuse intermediate results (i.e., spool tables) to maintain multiple indices, further improving the performance.

A secondary index is stored in the same database as its base table. In contrast, a join index can be stored in any database. Thus, a single-table join index may be defined even in cases where a secondary one cannot.

The collection of statistics on the join index object is essential for devising execution plans that optimally utilize the object. We consider a good practice to collect statistics on the primary and secondary indices defined for a join index object. As an exception, statistics collection on a non-sparse, single-table join index can be skipped, as the object inherits the collected statistics from its base table. Proper collection of statistics on base tables is also necessary to minimize the index maintenance costs for insert, delete, and update operations on the base table.

The use of a join index is recorded in the `DBC.DBQLOGTBL` table. When the join index is not utilized in the execution plans, it should be dropped to free the storage space and eliminate the processing costs for index maintenance.

Typically, a join index is designed to avoid accessing the base table, i.e., to be a *covering index* and provide all columns needed by the queries of interest. A non-covering join index may be an acceptable alternative, exchanging some read performance for less storage space requirements. In this case, the base table must be accessed to retrieve the additional columns, which are not included in the join index definition. The more the columns in the join index, the more the queries that can be covered. Still, the design aim should be to cover those critical queries for the DWH operation rather than to cover all the DWH queries.

A join back to the base table can be tuned by including the ROWID in the join index definition, emulating the operation of a secondary index. When the join index definition already includes the primary index of a base table (e.g., in a multi-table join), it is unnecessary to add a ROWID column. The already-available primary index can be used to transparently derive the ROWID and perform the join back to the base table, thus, saving the space for storing an additional column in the index.

The optimizer considers three options to join back to the base table when a non-covering index data access path is taken:

1. When it is a UPI-based table, and the UPI columns are included in the join index definition, Teradata can hash the columns from the join index and recover the ROWID to access the base table.
2. When the base table is a NUPI one, and the join index definition includes the ROWID pseudo-column, Teradata can directly join back to the base table using the available ROWID, as depicted in the example of Figure 94.
3. The third option is explored under three conditions: the base table is a NUPI one; a USI is defined; the USI columns are included in the join index definition. Then, Teradata can hash the USI columns and recover the ROWID to access the base table, similar to the first option.

```
Explain
SELECT * FROM Customer WHERE CustomerNumber = 1;

  1) First, we lock DWHPRO.Customer in TD_MAP1 for read on a reserved
     RowHash to prevent global deadlock.
  2) Next, we lock DWHPRO.Customer in TD_MAP1 for read.
  3) We do a single-AMP RETRIEVE step from DWHPRO.CUSTOMER_JI by way of
     the primary index "DWHPRO.CUSTOMER_JI.CustomerNumber = 1" with no
     residual conditions into Spool 2 (all_amps), which is
     redistributed by the hash code of (DWHPRO.CUSTOMER_JI.Field_1026)
     to few AMPs in TD_Map1.  Then we do a SORT to order Spool 2 by the
     sort key in spool field1.  The size of Spool 2 is estimated with
     low confidence to be 2 rows (54 bytes).  The estimated time for
     this step is 0.00 seconds.
  4) 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 DWHPRO.Customer by way
     of an all-rows scan with no residual conditions.  Spool 2 and
     DWHPRO.Customer are joined using a row id join, with a join
     condition of ("Field_1 = DWHPRO.Customer.ROWID").  The input table
     DWHPRO.Customer will not be cached in memory.  The result goes
     into Spool 1 (group_amps), which is built locally on the AMPs.
     The size of Spool 1 is estimated with low confidence to be 2 rows
     (16,058 bytes).  The estimated time for this step is 0.01 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 0.01 seconds.
```

Figure 94 Explain plan of a join back utilizing a join index containing the ROWID of the base table

When the base table is a NUPI, and neither the ROWID pseudo-column nor USI columns are available in the join index definition, Teradata resorts to a full-table scan, as depicted in the example of Figure 95.

```
Explain SELECT * FROM Customer WHERE FirstName = 'Art';

  1) First, we lock DWHPRO.Customer in TD_MAP1 for read on a reserved
     RowHash to prevent global deadlock.
  2) Next, we lock DWHPRO.Customer in TD_MAP1 for read.
  3) We do an all-AMPs RETRIEVE step in TD_MAP1 from DWHPRO.Customer by
     way of an all-rows scan with a condition of (
     "DWHPRO.Customer.FirstName = 'Art'") into Spool 1 (group_amps),
     which is built locally on the AMPs.  The input table will not be
     cached in memory, but it is eligible for synchronized scanning.
     The size of Spool 1 is estimated with high confidence to be 1,000
     rows (8,029,000 bytes).  The estimated time for this step is 47.80
     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 47.80 seconds.
```

Figure 95 Explain plan for accessing a NUPI table with neither ROWID nor USI columns available in the join index

Letter-case-specific columns in base tables (e.g., a column defined as VARCHAR(500) CASE SPECIFIC) require special treatment when used in a join index definition. Teradata stores the column values in the join index object in uppercase by default and performs the so-called *case-blind comparison*. A secondary index with the qualifier "ALL" in the join index definition must be included, similar to the example below, to allow *case-sensitive comparisons*. When the qualifier is used, the join index storage requirements increase, as case-specific values must be stored, albeit having a common uppercase representation.

```
CREATE JOIN INDEX Customer_JI AS
SELECT CustomerId, FirstName
FROM Customer
PRIMARY INDEX (CustomerId)
INDEX ALL (FirstName); -- a NUSI over a CASE SPECIFIC column
```

Merge join for NPPI tables

The merge join of NPPI tables is the most robust and frequently-used join method. The optimizer prefers it over other join methods, even when the latter are more performant, or statistics are missing, or available statistics are of "No Confidence" level. Merge joins can only be used if at least one of the join conditions is an *equijoin*, as in the example below:

```
SELECT c.CustomerId, c.SegmentId, S.SegmentDate, S.SegmentText
FROM Customer c
INNER JOIN Segment s
ON c.SegmentId = s.SegmentId; -- an equijoin
```

The merge join algorithm compares the ROWHASH of both table rows for possible matches, hence the equijoin requirement. When additional columns are included in the JOIN conditions, they are used as additional filters, which are applied after the ROWHASH match. Thus, the execution time estimations are increased to account for the additional post-match filtering.

The merge join algorithm requires that the rows are already located on the same AMP based on the ROWHASH of the join columns and that the rows of both tables are sorted in ascending ROWHASH order. When the join columns are also the primary index for both tables, then both the conditions are by definition met, and no join preparation is necessary. This is a so-called *direct merge join*. When this is not the case, a join preparation is necessary to relocate the qualifying rows of one or both tables.

Row redistribution can be done in two ways: copy all the qualifying rows to all the AMP's, i.e., *row duplication*, or calculate the new ROWHASH over the join columns. When row duplication is performed, the smaller-in-bytes table is copied and then sorted by ROWHASH on all AMP's. When the large-in-bytes table must also be rehashed over the join columns, then copies of the table rows are transferred to the responsible AMP, then locally placed in spool space, and finally sorted by the new ROWHASH.

Once the join preparation part concludes, the merge join algorithm starts. Teradata provides two variants of the algorithm, namely *slow-path* and *fast-path merge join*. The slow-path merge join is used when one of the tables can be read without resorting to the FTS data access path, e.g., by utilizing an index. The fast-path merge join is used when an FTS is needed for both (physical or spooled) tables. In either case, the rows are already sorted and, thus, a binary search is performed, as depicted in Figure 96.

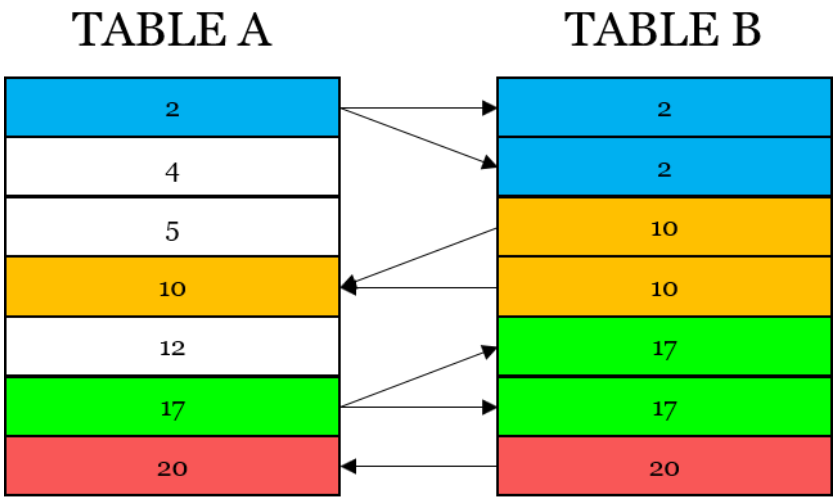

Figure 96 Binary search of sorted rows for merge join

The merge join is the most robust of all join methods when it comes to stale statistics. As the rows are sorted, unnecessary comparisons are avoided, and each data block of each table is visited only once. Teradata may use full-cylinder reads on both tables to further speed up processing. Additionally, none of the joined tables must entirely fit in AMP memory, which turns into a clear advantage over the other join methods discussed in the next sections. The total number of I/O operations needed for a merge join (excluding join preparation) is a straightforward sum of the data blocks of the qualifying rows of the two tables: $M + N$, while the time complexity is log-linear (quasilinear) to the number of the involved data blocks: $O(M \log M + N \log N)$.

The optimizer may decide to use a covering NUSI instead of directly accessing one or both the base tables and further improve the merge join performance in the form of reduced I/O operations and less volume of data transferred from disk to AMP memory. This is under the assumption that the NUSI sub-table data blocks store more rows than the base table due to the limited number of columns they cover. It is then essential to collect statistics on the NUSI columns so that the optimizer explores this opportunity.

Merge join for PPI tables

The newer version of Teradata support row partitioning, which, among others, has an impact on the merge join approach, as the rows are now sorted by the ROWKEY (which

considers the partition identifier) rather than ROWHASH alone. An NPPI table is logically equivalent to a PPI table with a single partition. As such, when two NPPI tables must be joined, the previously-described merge join can still be used in the newer Teradata versions, i.e., perform a *direct merge join* on the primary index columns of the two tables, which conveniently reside on the same AMP and share the same ROWHASH.

When two PPI tables must be joined, and they have the same primary index definition, the same partitioning expression, and both the PI and all the partitioning columns are included in the join columns, then the direct merge join can be used again. In these cases, it is called *ROWKEY-based merge join*. As in the case of NPPI merge join, this method can only be used for equi-joins. The performance of a ROWKEY-based merge join is similar to an NPPI direct merge join.

When the partitioning expression involves columns with character data types, it must be ensured that they are of the same character set. Otherwise, one of the tables must be spooled to make the columns of the same character set (e.g., convert from LATIN to UNICODE). Hence, it is essential to consistently define the partitioning expressions across logically-linked tables to allow the optimizer to choose performant join methods.

When two PPI tables with different partitioning expressions or one PPI and one NPPI table must be joined, it becomes more complicated to exploit the benefits of a direct merge join approach. As a prerequisite, both tables must share the same primary index definition. Teradata introduced with row partitioning three variants of a *sliding window merge join* as an extension to the direct merge join for PPI tables.

The *single-window merge join* variant may be used for joining an NPPI and a PPI table. This variant assumes that the PI columns are also the join columns and that one data block of the NPPI table and one data block for each PPI table partition fits in memory. The variant is more likely to be considered by the optimizer when the PPI table consists of only a few partitions, or many partitions can be eliminated before the join. The algorithm works as follows (see Figure 97 for a schematic representation):

1. The first data block of the NPPI table is transferred from disk to memory.
2. One data block per each partition of the PPI table is transferred from disk to memory.
3. The NPPI data block rows are compared against the transferred rows of the PPI table.
4. The remaining PPI table data blocks are progressively read and compared.
5. The next data block of the NPPI table is transferred from disk to memory.
6. The whole process repeats until all NPPI and PPI data blocks have been processed.

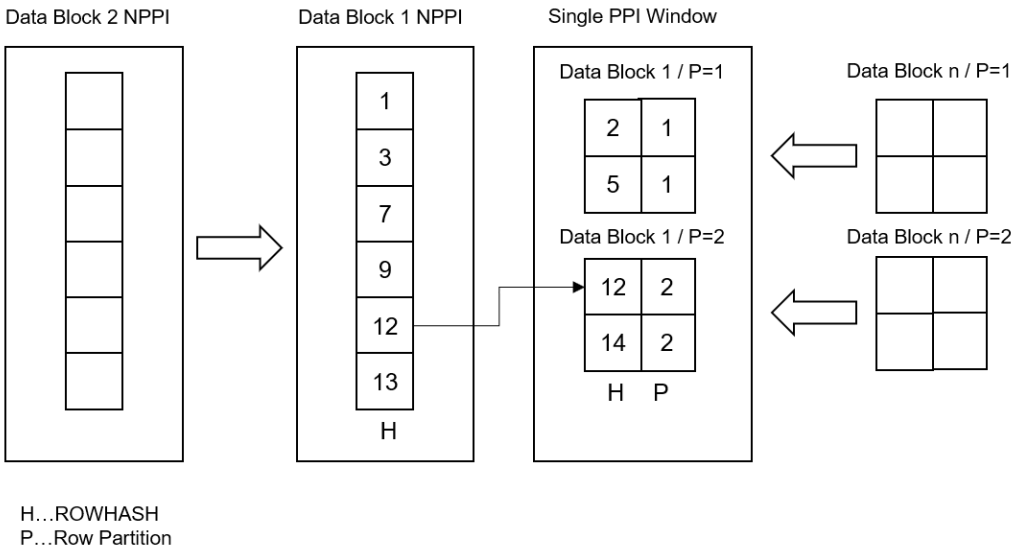

Figure 97 Single-window merge join

In total, the sum of the data blocks of both the NPPI and PPI tables will be transferred from disk to memory exactly once. Typically, the number of I/O operations matches the one of a ROWKEY-based merge join. The single-window merge join requires some more memory, as it initially transfers one data block per partition, and a slightly more complex processing logic.

The *multiple-window merge join* variant may be used for joining an NPPI and a PPI table when there is not sufficient memory to fit one data block for each PPI table partition. Then, the partitions must be processed in non-overlapping-in-time windows. The variant may be used for joining an NPPI and a PPI table. The algorithm works as follows, assuming that *k* partitions are concurrently in memory (see Figure 98 for a schematic representation):

1. The first data block of the NPPI table is transferred from disk to memory.
2. One data block per each of the *k* partitions of the PPI table is transferred from disk to memory.
3. The NPPI data block rows are compared against the transferred rows of the *k* partitions of the PPI table.
4. The remaining PPI table data blocks of the *k* partitions are progressively read and compared in blocks of at a time.
5. The next data block of the NPPI table is transferred from disk to memory.
6. The whole process repeats until all NPPI and PPI data blocks of the *k* partitions have been processed.
7. The next processing window of *k* partitions starts from step 1 above but reading the next data block of the NPPI table.
8. The process repeats until all the windows of the PPI table have been processed.

By and large, the processing costs for the multiple-window merge join are higher than those of the single one. Typically, significantly more I/O operations are necessary for a multiple-window merge join in comparison with a ROWKEY-based one. While each data block of the PPI table is transferred exactly once, the data blocks of the NPPI table must be transferred p/k times, where p is the count of non-eliminated partitions and k as before. From a performance point of view, it is crucial to keep this ratio as close as possible to 1. A ratio of 20 implies that an FTS will be performed 20 times for the NPPI table, i.e., it will take 20 times as long as a single-window merge join.

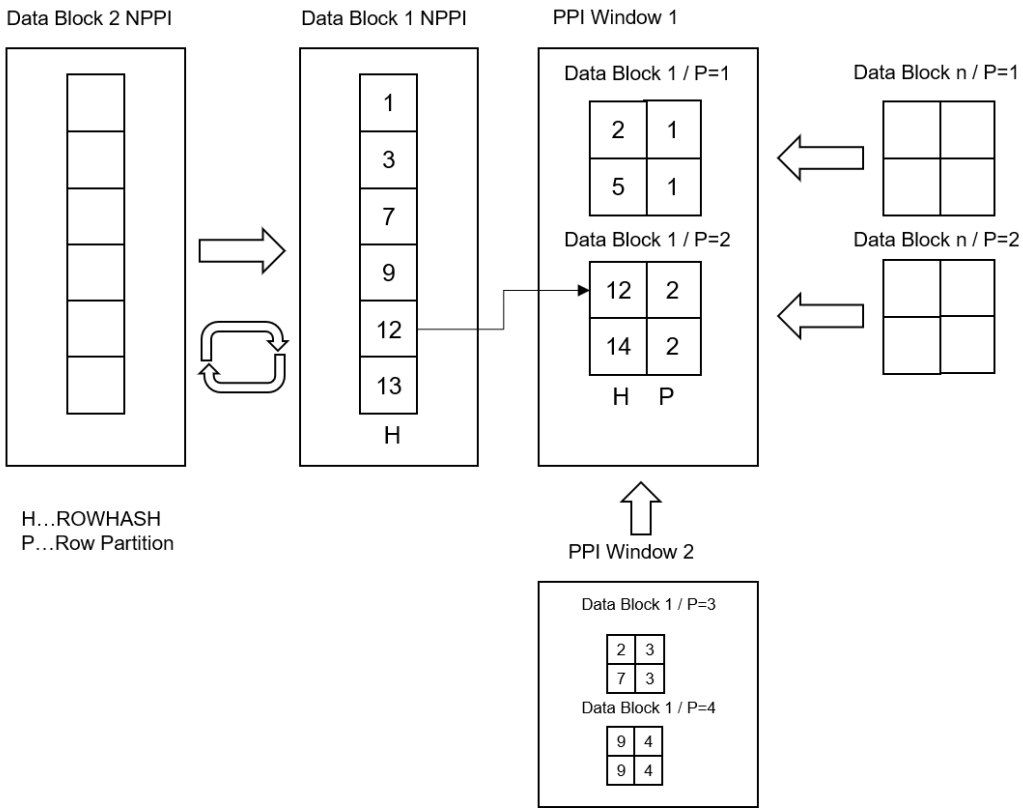

Figure 98 Multiple-window merge join

The *sliding window merge join* variant may be used for joining two PPI tables that have different partitioning expressions. It is one of the most inefficient methods for merge join, as the join is performed via multiple, sliding windows for both tables. The number of I/O operations is $p_1/k_1 * B_1 + p_2/k_2 * B_2$, where $B_1$and B_2 are the total data blocks of the first and second PPI table, respectively. As before, the performance tuning aim should be to keep the two ratios as low as possible; as the denominator cannot be directly controlled without changing the system parameters (e.g., a memory upgrade), it is essential to keep the number of involved partitions as low as possible. The performance of the sliding window merge join becomes comparable to the one of ROWKEY-based when most of the partitions can be eliminated.

Hash join

The hash join method may be used instead of a merge join in those cases that one of the two tables entirely fits into the AMP memory. In principle, a hash join has the same requirements as a merge join for equijoins. The advantage over the merge join is that the rows of the larger table need not be sorted by ROWHASH. Teradata implements multiple variants of hash join algorithms; while the implementation details are proprietary, a generic approach applies to all of them.

The *classic hash join* is the most elementary variant, on top of which more advanced ones are built. The algorithm considers the smaller-in-bytes table fitting in memory like the *build-* or *hash table* and the large-in-bytes table as the *probe table*. The probe table needs not fit in memory and may remain unsorted. The hash join algorithm consists of two phases, namely the *build-* and the *probe phase*.

In the build phase, the rows of the build table are distributed into *hash buckets* based on a hashing algorithm over the join columns. Each hash bucket may contain multiple rows, and all columns needed for the join are stored in the hash bucket along with their hash values. The memory footprint for the build table depends on both the number of qualifying rows and the selected columns.

The probe table is processed row-by-row in the probe phase using an FTS; then, all table data blocks are progressively copied from disk to AMP memory. When possible, Teradata reads whole cylinders to speed up the processing. Like the build phase, a hash value is calculated over the join columns of the qualifying rows. The hash value of the probe table is then compared against all the rows of the corresponding hash bucket for possible matches.

The total number of I/O operations for the classic hash join is $M + N$, like in the case of a merge join. The time complexity is linear to the number of the involved data blocks: $O(M + N)$. However, skew can have a considerable impact on parallel efficiency and cause performance degradation. To take full advantage of the classic hash join, each hash bucket of the build table must have the same small *load factor*, i.e., contain approximately the same small number of rows.

If sufficient memory is available to hold the build table, the classic hash join performs up to 40% better than the merge join. When the estimation of the optimizer proves inaccurate and the build table does not fit in available memory, the probe phase is adjusted as follows. First, an FTS is performed matching the probe table rows with the ones of the build table fitting in the hash buckets. Then, the data blocks of the build table are removed from memory. A new set of data blocks is transferred from the disk, a new build table is built, and the qualifying rows are distributed in the new hash buckets. Another FTS is performed matching the probe table rows with the new build table. The process repeats until all the build table data blocks are processed. By and large, the optimizer avoids a hash join as the run time may explode due to the multiple FTS of the large-in-bytes probe table.

A *hybrid hash join* variant is used by Teradata to cope with the shortcomings discussed in the paragraphs above (e.g., multiple FTS, stale statistics, and table skew) and achieve better performance. The difference with the classic algorithm is that both the build and the probe tables are split into *hash join partitions* (not to be confused with row and column partitioning of tables). The hybrid variant is based on a divide-and-conquer algorithm, ideally suited for MPP and parallel database systems.

In the divide (or *build*) phase, Teradata estimates the optimal *fanout*, i.e., the number of hash partitions to split the build table into. The estimation is based on the table cardinality and the skew information available in the statistics. The partitions are chosen so that they entirely fit in memory but do not fully occupy it. Some free space is kept as a buffer for the next partitions. The first partition of the build table is placed in memory. The remaining partitions go to the buffer area and are swapped to the disk when the buffer becomes full.

In the conquer (or *probe*) phase, the probe table is partitioned, and, in parallel, the rows of the first probe table partition are joined with the rows of the first build table partition already in memory using the classic hash join variant, as depicted in Figure 99. This is now possible as the build table partition meets the requirement to fit in memory entirely. The rest of the probe table partitions are progressively placed in the buffer area and swapped to the disk whenever the buffer becomes full. Once the first pair of partitions are processed, the process repeats for the partitions in the buffer, swapping back from disk whenever necessary. The assumption is that the first build table partition contains a sufficiently large count of rows to be reused for the joins to reduce costly disk I/O operations.

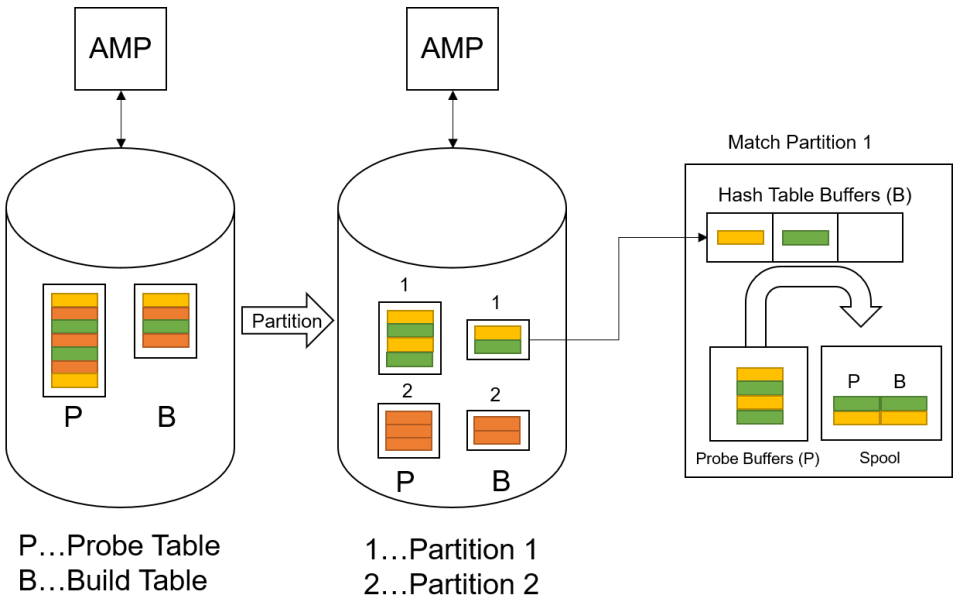

Figure 99 Grace hash join algorithm with partitioned build and probe tables

The *dynamic hash join* variant is used to avoid spooling of big probe tables. This variant is useful when the probe table must be rehashed because the join columns do not include the primary index columns. In such a case, the whole table must be spooled with the new ROWHASH, stressing available resources. When the dynamic hash join variant is used instead, the probe table is not spooled at once. Instead, its data blocks are dynamically (progressively) transferred from disk to memory, the new ROWHASH is dynamically calculated (on-the-fly) for the qualifying rows, and the ROWHASH is probed (searched) immediately in the build table.

The dynamic hash join requires that the build table is copied in full to all AMP's, so that all AMP's can directly compare with the probe table. It can only be used with the following equijoin types: INNER JOIN, LEFT/RIGHT/FULL OUTER JOIN, INCLUSION/EXCLUSION JOIN. The dynamic hash join algorithm comprises the following steps:

- Duplicate the smaller-in-bytes table to all the AMP's
- Locally at each AMP:
    - Create the build table
    - Sort by the ROWHASH calculated over the join columns
    - Read a row from the probe table and calculate its ROWHASH over the join columns
    - Search for build table ROWHASH matches.
    - Repeat the last two steps until all the probe table rows are processed.

The *in-memory hash join* applies for tables stored in columnar format and combines several techniques to improve performance. Starting with Teradata 16, single-instruction-multiple-data (SIMD) CPU instructions, if available, can be utilized to process multiple column entries in parallel to improve the columnar format processing. The optimizer chooses to use in-memory hash joins only when each Teradata system node has at least 512 GB of main memory and if this is assessed as the most efficient of all the join methods. The in-memory hash join approach can also be used in the classic and dynamic hash join and perform an *in-memory (dynamic) hash join*. In this case, SIMD instructions are further utilized to calculate the ROWHASH values of multiple probe table rows at once.

The in-memory hash join is similar to the classic hash join. The smaller-in-bytes table must entirely fit into the AMP memory. When the base table is stored in columnar format, Teradata uses SIMD instructions for bulk qualification of tables rows; else, it resorts to row-by-row qualification. The data in the spool space are always stored in columnar format for both the join tables and consist of up to three containers. The first holds the primary index columns, the second the join columns, and the third is optional, holding any additional columns needed for the result. The three containers form the so-called *in-memory optimized spool*, which is optimized for parallel column processing. SIMD instructions are used to perform the probing for multiple rows at once.

Nested join

The nested join method may be used only when the two tables are equi-joined and can be accessed via an index. The nested join is the most performant join method as it requires just one or a few AMP's.

A nested join requires a constant-value row qualifier (e.g., a WHERE clause) for one of the tables that results in a data access path through a UPI, a USI, or an almost-unique NUPI, which returns only one or a few rows. Additionally, it requires that the second table has an index (e.g., UPI, NUPI, USI, NUSI) that matches the join columns. A join index containing the ROWID column of its base table is also acceptable.

The nested join method works as follows. Teradata reads a qualifying row from the one table with the WHERE clause (the "left table") and, through it, identifies the AMP containing the matching rows of the other table (the "right table"). Then, it proceeds with one of the two possible variants: *local-* or *remote nested join*.

The local nested join has strict requirements to guarantee that only one row is processed. The index of the left table must be unique, i.e., either a UPI or a USI, and the index of the right table must not be a NUSI. The algorithm performs the following steps, as depicted in Figure 100:

1. The qualifying row of the left table is read, and the ROWHASH over the join columns is calculated.
2. The left-table AMP uses the ROWHASH to derive the right-table AMP holding the respective row of the second table. A request for the row is sent from the left-table AMP to the right-table AMP.
3. The right-table AMP sends the searched row back to the left-table AMP.
4. The join is performed in the left-table AMP, and the result is copied to the spool space for further processing.

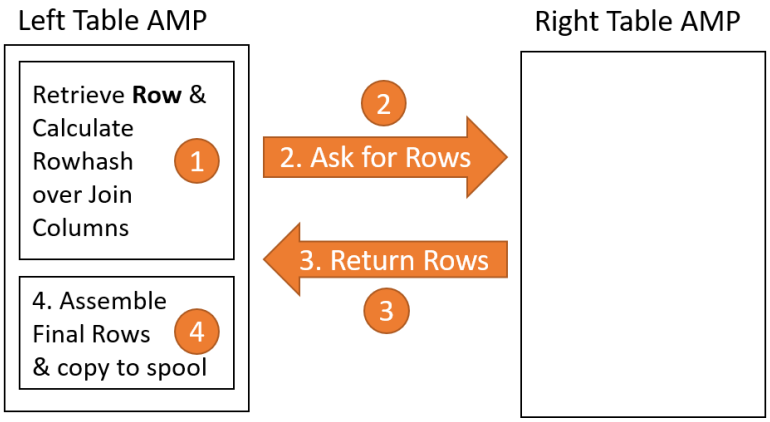

Figure 100 The steps of the local nested join algorithm

The remote nested join has less strict requirements than the local one. It can be used even when the index of the left table is not unique (e.g., is an almost-unique NUPI), or the index of the right table is a NUSI. The algorithm performs the following steps:

1. The qualifying row (or few rows) of the left table are redistributed by the ROWHASH calculated over the join columns. If the index of the right table is a NUSI, then the rows are instead duplicated to all AMP's, as the row may be on any AMP.
2. A join between the qualifying rows and the index of the right table is performed. The base table ROWID's of the right table are extracted through the index and placed in an intermediate spool table.
3. The collected intermediate spool table is the input for a second join step performed on a ROWID basis. The left table ROWID values are directly extracted from its index sub-table.
4. The actual base table rows are extracted from either the same AMP (in the case of a NUSI) or a different AMP (in the case of a USI).

The remote nested join requires two joins with physical tables: one for the right table index sub-table and another for the right table itself). Despite its complexity and increased resource usage, the remote variant is still very efficient.

Product join

The product join method is the only one that can resolve any type of join, including INCLUSION (IN and EXISTS) and EXCLUSION ones (NOT IN and NOT EXISTS). At the same time, it consumes the most resources of all the join methods. The algorithm simply compares the ROWHASH of the qualifying rows of the left table (say *M* in total) with all the respective ones of the right table (say *N* in total), as depicted in Figure 101.

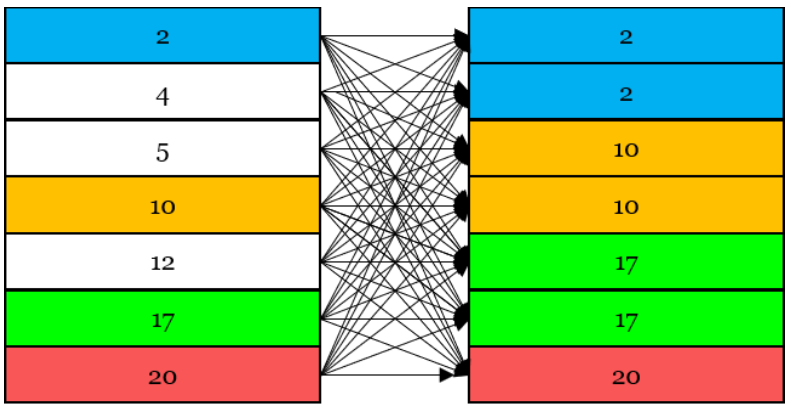

Figure 101 A product join example

The optimizer may choose a product join when one of the two tables contains a tiny number of rows, or there are OR-combined join conditions, or the join conditions are not equijoins (e.g., for a greater than or a NOT conditional). An unintended product join may occur

during the execution of a merge join step if the ROWHASH accesses are frequent, as depicted in Figure 102.

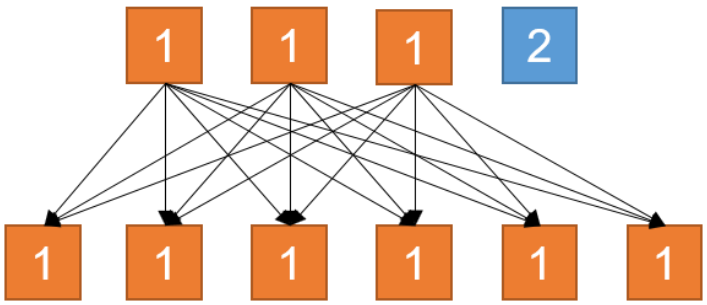

Figure 102 A merge join effectively being a product join

The choice of a product join may also result from a logical error in the SQL statement. Two common error patterns, in our experience, are an unconditional join of a reference table and skipped table aliases, as in the examples below:

```
SELECT a.col1, a.col2, b.col3, c.col1, c.col4
FROM tuning.customer a
INNER JOIN tuning.sale b
INNER JOIN tuning.provider c
WHERE a.col1 = b.col1 AND b.col2 = DATE '2020-08-31' AND a.col3 = 'Austria'
AND a.col2 = DATE '2020-08-31' AND c.col2 = DATE '2020-08-31'
AND c.col3 = 'Teradata' -- no equijoin of c with a or b

SELECT cust.col1, cust.col2, dale.col3, prov.col1, prov.col4 -- CROSS JOIN data
FROM tuning.customer cust
INNER JOIN tuning.sale sale
INNER JOIN tuning.provider prov
WHERE a.col1 = b.col1 AND b.col2 = DATE '2020-08-31' AND a.col3 = 'Austria'
AND a.col2 = DATE '2020-08-31' AND c.col2 = DATE '2020-08-31'
AND c.col3 = 'Teradata' AND a.col1 = c.col1 -- equijoin of c with a
```

The product join performs well when one of the tables is small enough to entirely fit in memory (i.e., the *M* is small). Then, only a few comparisons are made, and each data block of the small table is transferred only once from disk to memory. When the small table does not entirely fit in memory, its data blocks must be transferred multiple times from disk to memory to compare against different chunks of the big table. Despite being the most efficient join strategy, a product join heavily depends on reliable statistics. Stale statistics may lead to underestimating the table cardinality of the small table, resulting in a dramatic increase of run times caused by the continuous swapping of data blocks of the small table.

When the resource consumption is low, a product join may outperform even a merge join, accounting for all the join strategy factors. While the merge join requires significantly fewer comparisons than a product join, the join preparation of the former to sort and redistribute rows may be very resource-demanding. In contrast, the join preparation for a product join requires just copying the small table to all AMP's. In essence, the two methods complement

each other in terms of cost factors: product join may have a low preparation but a high comparison cost, while merge join may have a high preparation but a low comparison cost.

Dynamic partition elimination (DPE) is a Teradata feature to improve partition elimination at runtime when a PPI table is involved in a product join. The optimizer may use DPE under three conditions. First, at least one equality constraint exists between a partitioning column of the PPI table and a column of the other table. Second, the PPI table is not using 8-byte (wide) partitions. Last, the product join is not an exclusion one (i.e., NOT IN or NOT EXISTS). Starting with Teradata 15.10, exclusion product joins are allowed, provided that the partition definition comprises RANGE_N expressions at all levels.

The product join with DPE works as follows. The left table is row-partitioned the same way as the PPI (right) one. The first partition of the left table is then taken and compared with the first partition of the right table. This repeats for the rest until the left partition is compared with all the partitions of the right table. After this, the second partition of the left table is taken, and the whole comparison process with the partitions of the right table repeats. The advantage of this approach is that not all but rather only non-eliminated partitions of the right table are processed for the product join.

The optimizer is very conservative in choosing a product join with the DPE approach, as inaccurate estimates can have a significant performance impact. The optimizer demands that statistics are available for the primary indices of both the tables, the partitions of the PPI table, and the equijoin column(s) that enable DPE in the first place. The more up-to-date the statistics are, the more possible the optimizer will use DPE in joins. While DPE is primarily aimed for product joins, it may also be utilized by the optimizer for hash joins as well, including the dynamic hash join.

Tuning arsenal

Many different sources may reach a *performance tuner* (i.e., a person assigned the role of tuning the performance of the DWH) calling for execution plan tuning with an issue description of varying quality. A performance tuner must be equipped with the tools and the knowledge to assess the DWH status before and after any tuning activity takes place, including a cost-benefit analysis of whether any tuning is necessary in the first place. The previous sections approached the execution plan from a system point of view. The next sections approach it from a performance tuner point of view. For readability, the case of (SQL) *queries* will be considered, but the discussion is equally valid for all types of SQL statements.

There are several indicators of an execution plan that underperforms. What is not an indicator is the execution time measured in real-world (wall clock) seconds. Teradata is a massively parallel processing system and can concurrently handle thousands of requests. In such an environment, the order of the requests, the database locks, and data availability, and the overall system load may significantly affect such a metric. Furthermore, the execution time perceived through a client tool interfacing a Teradata system adds more

processing layers (e.g., network infrastructure, computing system hosting the client tool, and other software running on the client system). All these layers may also affect the perceived execution time. Nonetheless, consistently slow execution time should not be ignored but rather serve as an initial point for further analysis.

EXPLAIN and Teradata Viewpoint

The output of an EXPLAIN statement (or, more accurately, "*an EXPLAIN request modifier for an SQL statement*") is the most elementary and readily accessible tool in the performance tuning arsenal. It should be utilized already from the development time by all involved persons to check the (dynamic) execution plan devised by the optimizer and the confidence levels for each step. When the (DYNAMIC) EXPLAIN output does not match the expectations, appropriate actions should be considered to remedy the situation. Potential reasons for deviation include, among others, queries that overlook the PDM; lack of or inaccurate statistics; and a suboptimal PDM. In our experience, 80% of the performance issues can be addressed by adequately rewriting the SQL to adhere to the defined PDM.

We consider a good practice to be familiar with the EXPLAIN output and able to reason about it; little if any performance tuning can be achieved without a good understanding of the EXPLAIN output. A key characteristic is that EXPLAIN offers a consistent textual description of each step and step action, independently of the SQL query to be executed. This simplifies the analysis to a large extent. The Appendix provides a list of standard phrases typically mentioned in the execution steps. The provided list is short enough to be memorized but broad enough to cover the most common cases. In our experience, the Pareto principle holds: knowing 20% of the possible phrases is sufficient to understand and address 80% of the performance problems.

A typical use of EXPLAIN is to compare the two outputs before and after tuning is applied. A tuning might be anything like a rewritten query, altered data demographics, updated statistics, new indices or partitions, or even a system reconfiguration. As the textual description is consistent, a direct comparison of the two text outputs suffices to identify whether the optimizer devised a new execution plan. The EXPLAIN output reports also run estimates; as mentioned already, these numbers should not be considered absolute, wall-clock times but rather as logical time units. Typically, when the new estimations are lower, the new execution plan will perform better than the old one, provided that the latter does not introduce a skewed step.

The Teradata Viewpoint allows monitoring of the execution of the plan steps in real-time. The information is similar to an EXPLAIN output but with additional metrics, such as the difference between the estimate and the actual number of rows involved in every execution step. The additional data provided by Viewpoint can be used to identify those execution steps that are skewed in either execution time or spool space.

The portlet “Query Monitor” is of particular interest for performance tuning. It provides a real-time overview of all session queries and allows to replay executed ones. As depicted in Figure 103, the summary display is updated by default every 30 seconds and contains a configurable set of metrics for the Utility Workload (e.g., FastLoad and MultiLoad) and the running SQL requests. In our experience, the count of CPU seconds and I/O operations since the last sample (i.e., the Delta CPU and Delta I/O), Snapshot CPU, I/O Skew, Blocked Time, Spool, Duration, and the Product Join Indicator (PJI) comprise a list that provides good indicators for performance tuning tasks. Additional filters can be applied to group the requests based on the user and the status (e.g., Active, Blocked, Idle, Delayed, Abort, or Responding).

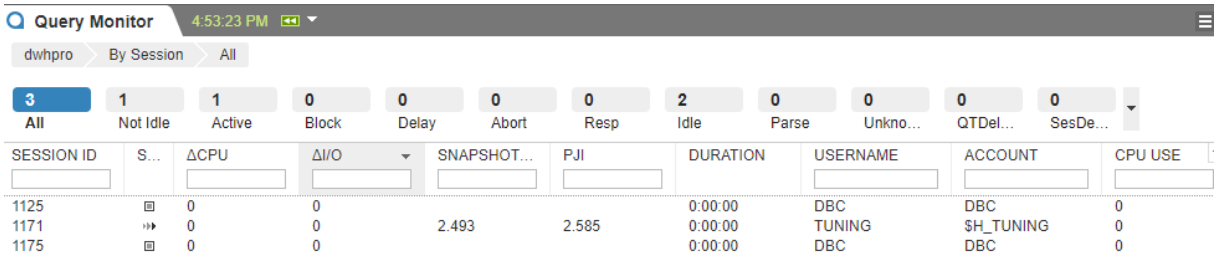

Figure 103 Teradata Viewpoint Query Monitor summary display

The Overview tab of the Query Detailed View, as depicted in Figure 104, provides the most crucial performance indicators of a request. The “Snapshot CPU skew” and “Snapshot I/O skew” as well as “Spool space”, “Request CPU”, and “Request I/O” are of particular interest for real-time performance monitoring and analysis, especially for long-running queries. The SQL tab displays the exact SQL statement being executed. When a QUERY_BAND is defined for the session, as discussed in the next section, then the Query Band tab displays all the key-value pairs in real-time.

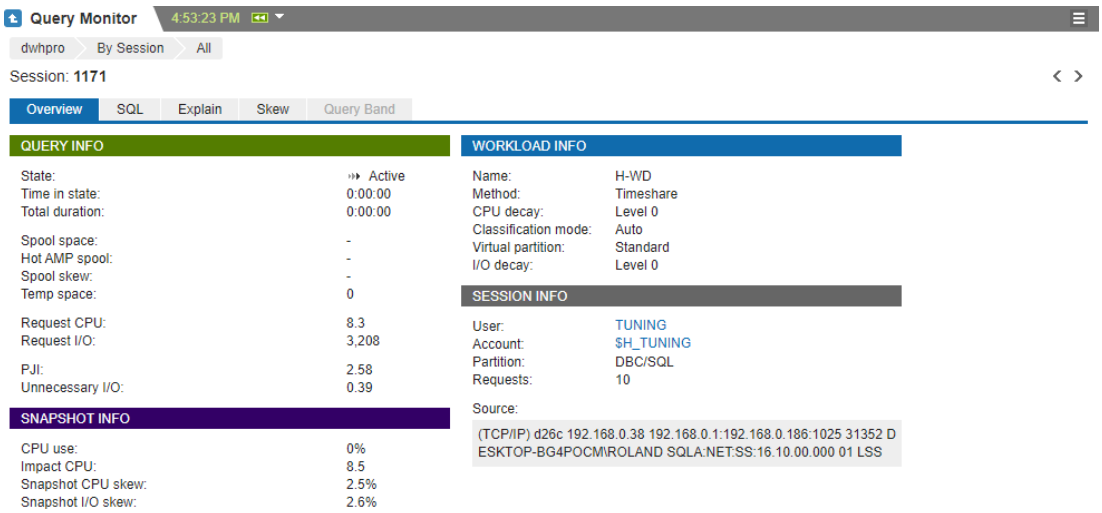

Figure 104 Teradata Viewpoint Query Overview tab

The Explain tab of the Query Detailed View, as depicted in Figure 105 Teradata Viewpoint Query Explain tab, provides the EXPLAIN output. When a skew problem is identified in the Overview tab, switching to the Explain tab allows performing a more thorough analysis of the exact steps that contribute to the skew. The Explain tab also provides information about the actual time and actual rows involved in each execution step. When these metrics significantly deviate from the expected ones, performance issues may arise, as the optimizer did not use estimations that reflect the actual data demographics.

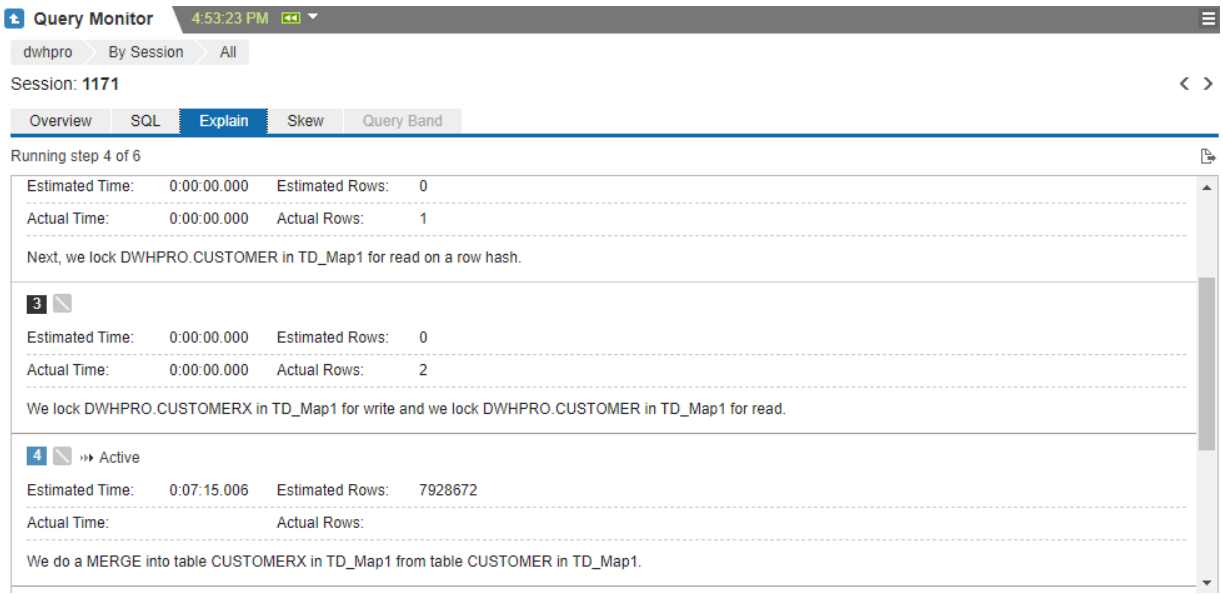

Figure 105 Teradata Viewpoint Query Explain tab

DBQL tables and QUERY_BAND

Teradata offers the so-called DBQL tables, which log information related to the actual execution of queries. These tables complement the EXPLAIN estimations and the real-time logging of Viewpoint. They allow a posthoc analysis of already-executed queries. Logged information includes (among other) response time, resource consumption, consumed spool space, answer set size, accessed database objects, any workload delays, as well as the start and end time of the query execution.

The DBQLOGTBL table logs information about individual queries and is typically activated on a Teradata system. The table provides information about (among others) the start and end execution time of a query, the resource consumption (I/O operations, CPU cycles, and spool space), workload-related information (e.g., assigned workload group and any observed delays), the SQL query itself (typically, a truncated part of it due to size constraints), and any error codes, and UDI counts.

The DBQLSQLTBL table logs the whole SQL text of a query. Typically, it is not activated due to the increased storage space requirements.

The DBQLOBJTBL table logs usage information for all database objects. This includes tables, columns, indices, and databases themselves.

The DBQLSTEPTBL table logs detailed information about each step of an execution plan. It can be considered an extension of DBQLOGTBL, as the last logs information on the query level.

The DBQLExplainTBL and DBQLXMLTBL tables store the whole EXPLAIN output of the query in text and XML format. Typically, such logging is not activated due to the increased storage space requirements.

The DBQLSUMMARYTBL table logs information about query execution but aggregated on a user (or account) level. As such, logging is of lower granularity than the contents of DBQLOGTBL; the logged information is the count of queries per summary group and the aggregate CPU usage and I/O operations.

The BEGIN QUERY LOGGING statement is used to define which information should be collected per query, which in turn drives the selection of the appropriate DBQL table and storage space consumption. We consider a good practice to log in the DBQLOGTBL those long-running workloads used in the DWH ELT (or ETL) process. During performance tuning, it might also be helpful to additionally activate logging in DBQLOBJTBL and DBQLSQLTBL tables, using a statement like the one in the example below:

```
BEGIN QUERY LOGGING WITH SQL, OBJECTS LIMIT SQLTEXT =0 ON ALL ACCOUNT = 'DWHPRO';
```

When a mix of long- and short-running workloads exists, we consider a good practice to use *threshold logging* to split logging into detailed and summarized. Then, the storage space consumption is reduced. The statement in the example below logs all queries which consume less than two CPU seconds (the threshold) in the DBQLSUMMARYTBL table (aggregated on account level) and the rest in the DBQLOGTBL, DBQLSQLTBL, and DBQLOBJTBL tables (detailed):

```
BEGIN QUERY LOGGING WITH SQL, OBJECTS LIMIT THRESHOLD = 200 CPUTIME AND SQLTEXT =10000 ON ALL
ACCOUNT = 'DWHPRO';
```

The DBQL tables can be utilized to identify queries and query execution steps that exhibit bad performance and their cause. The so-called *product join indicator* is a metric that can be derived from DBQLOGTBL columns, using an expression like the one in the example below:

```
CASE WHEN TOTALIOCOUNT = 0 THEN 0 ELSE (AMPCPUTIME * 1000)/TOTALIOCOUNT END
```

When the product join indicator has a value of five or more, it is a hint of a potential product join negatively affecting the query performance. A product join might not always be the cause. CPU-intensive operations, such as string manipulation functions and complex CASE statements (typically used in reporting tools), might also be the root cause for a high

product join indicator. The inverse product join indicator metric can be used to identify huge table scans, which might serve as a hint for adding an index-based access path (e.g., a join index or a NUSI).

Skew is one of the most important metrics when it comes to performance tuning. A skew metric can be easily extracted from DBQLOGTBL, using an expression like the one in the example below:

```
/* CPU skew */
CASE WHEN (AMPCPUTIME / (HASHAMP()+1) = 0 THEN 0 ELSE MAXAMPCPUTIME / (AMPCPUTIME /
(HASHAMP()+1) END
/* I/O skew */
CASE WHEN (TOTALIOCOUNT / (HASHAMP()+1) = 0 THEN 0 ELSE MAXAMPIO / (TOTALIOCOUNT / (HASHAMP()+1)
END
```

Ideally, the skew metric should be less than one. Any number higher than one implies the presence of skew. Typically, skew problems arise when either metric exceeds 1.5; up to this value, the skew is considered present but tolerable.

By and large, a Teradata DWH serves numerous clients per day and executes thousands of queries, often many times each. It then becomes hard to isolate in the DBQL tables a specific query instance for further analysis. Teradata allows to set and record a QUERY_BAND in the form of key-value pairs stored in the DBQLOGTABLE.QUERYBAND column.

QUERY_BAND logging can be activated for transactions and whole sessions, as depicted in Figure 106 and Figure 107, respectively. In our experience, session-level is more useful for performance tuning activities. The `DBC.SessionsInfoV` view shows all query bands for the currently active sessions.

```
SET QUERY_BAND = 'key1=value1;key2=value2;...' FOR SESSION;
```

Figure 106 Example of session-level query band logging

```
SET QUERY_BAND = 'key1=value1;key2=value2;' FOR TRANSACTION;
```

Figure 107 Example of transaction-level query band logging

The key-value pairs can be freely selected. They can also be updated after they are set using an UPDATE keyword, and new key-value pairs may also be added, as depicted in Figure 108. The deletion of key-value pairs is not allowed. Instead, the query band must be deleted and recreated.

```
SET QUERY_BAND = 'key1=value1;key2=value2;' FOR SESSION;

SELECT QueryBand
FROM DBC.SessionInfoV
WHERE SessionNo = 1043;

--> key1=value1;key2=value2;

SET QUERY_BAND = 'key1=value1;key2=value3;' UPDATE FOR SESSION;

SELECT QueryBand
FROM DBC.SessionInfoV
WHERE SessionNo = 1043;

--> key1=value1;key2=value3;

SET QUERY_BAND = 'key1=value1;key2=value2;key3=value3;' UPDATE FOR SESSION;

SELECT QueryBand
FROM DBC.SessionInfoV
WHERE SessionNo = 1043;

--> key1=value1;key2=value2;key3=value3;

SET QUERY_BAND = 'key1=value1;' UPDATE FOR SESSION;

SELECT QueryBand
FROM DBC.SessionInfoV
WHERE SessionNo = 1043;

--> key1=value1;key2=value2;key3=value3;
```

Figure 108 Adding and changing key-value pairs

The stored values are retrieved using the `GetQueryBandValue` function, as depicted in the example of Figure 109. The first function parameter specifies the query band preference to cover those cases where a key is defined at both transaction and session levels. A zero value indicates no preference, a value of one indicates the priority for a transaction, and a value of two is for the session.

```
SELECT GetQueryBandValue(0,'key1') FROM DBC.SessionInfoV;
WHERE SessionNo = 1043;

--> value1
```

Figure 109 Retrieving a query band value

We consider a good practice to always include a QUERY_BAND definition in batch scripts and other time-triggered artifacts. Application names, job names, script names and versions, and batch loading identifiers are good key-value choices. They can simplify and accelerate the DBQL-based post-hoc analysis and tuning effort. Ideally, every query in an operational DWH should have a QUERY BAND attached.

A typical use of a QUERY_BAND during performance tuning is depicted in Figure 110. First, a query band is activated, the key named "CheckedQuery" is set to a value, and the query is executed. Then, a new value is set, and the tuned query is executed. Finally, the query band is disabled so that no further queries can influence the analysis. A posthoc analysis can then query the DBQL tables using the specific session and key identifiers to restrict the metric to only those of the two queries involved in the tuning process.

```
SET QUERY_BAND = 'CheckedQuery=Old;' FOR SESSION;

SELECT * FROM CustomerView_Old;

SET QUERY_BAND = 'CheckedQuery=New;' FOR SESSION;

SELECT * FROM CustomerView_New;

SET QUERY_BAND = NONE FOR SESSION;
```

Figure 110 Use of query bands for performance tuning

HELPSTATS

The EXPLAIN output is not always sufficient to identify the root cause of an underperforming query. The output only mentions the steps that the optimizer decided to take and the confidence level for each step. The Teradata optimizer can record if instructed so, the conditions under which it would have chosen a better execution plan. The `DIAGNOSTIC HELPSTATS ON FOR SESSION` statement enables session-level diagnostic information for the execution steps and use (or lack of) statistics for retrieve and join steps, i.e., the most important from a performance point of view.

Figure 111 depicts a usage example of the statement, where the EXPLAIN output is enriched with recommendations for what statistics to collect for a better execution plan and the confidence level of the optimizer for this suggestion, i.e., how important it considers their collection. The optimizer chose with a low confidence a data access path through an indexed column (DESCR) and estimated two columns in the original execution plan. The optimizer recommended collecting statistics on column DESCR with high confidence.

Figure 112 depicts the new execution plan after the recommendation was taken. The optimizer again chooses the index column data access path, but it now estimates with high confidence the size of the result.

```
1 DIAGNOSTIC HELPSTATS ON FOR SESSION;
2 EXPLAIN SELECT * FROM tuning.TMP_MUL WHERE DESCR = '123';
```

Teradata Result Set Viewer

Result Set - SQL Editor (1)

```
EXPLAIN SELECT * FROM tuning.TMP_MUL WHERE DESCR = '123';

Explanation
------------------------------------------------------------------------
 1) First, we lock tuning.TMP_MUL in TD_MAP1 for read on a reserved
    RowHash to prevent global deadlock.
 2) Next, we lock tuning.TMP_MUL in TD_MAP1 for read.
 3) We do an all-AMPs RETRIEVE step in TD_MAP1 from tuning.TMP_MUL by
    way of index # 4 "tuning.TMP_MUL.DESCR = 123" with no residual
    conditions into Spool 1 (group_amps), which is built locally on
    the AMPs.  The size of Spool 1 is estimated with low confidence to
    be 2 rows (58 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.
    BEGIN RECOMMENDED STATS FOR FINAL PLAN->
    -- "COLLECT STATISTICS COLUMN (DESCR) ON tuning.TMP_MUL" (High
    Confidence)
    <- END RECOMMENDED STATS FOR FINAL PLAN
```

Figure 111 EXPLAIN output enhanced with recommended statistics

```
1 COLLECT STATISTICS COLUMN (DESCR) ON tuning.TMP_MUL;
2 EXPLAIN SELECT * FROM tuning.TMP_MUL WHERE DESCR = '123';
```

Teradata Result Set Viewer

Result Set - SQL Editor (1)

```
EXPLAIN SELECT * FROM tuning.TMP_MUL WHERE DESCR = '123';

Explanation
------------------------------------------------------------------------
 1) First, we lock tuning.TMP_MUL in TD_MAP1 for read on a reserved
    RowHash to prevent global deadlock.
 2) Next, we lock tuning.TMP_MUL in TD_MAP1 for read.
 3) We do an all-AMPs RETRIEVE step in TD_MAP1 from tuning.TMP_MUL by
    way of index # 4 "tuning.TMP_MUL.DESCR = 123" with no residual
    conditions into Spool 1 (group_amps), which is built locally on
    the AMPs.  The size of Spool 1 is estimated with high confidence
    to be 1 row (29 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.
```

Figure 112 Changed execution plan after statistics are updated

The combined output of the DIAGNOSTIC HELPSTATS and EXPLAIN is a mere recommendation for a specific query and should be treated as such. The actual impact of

the suggested actions on the overall DWH operation should be carefully assessed. Even at the query level, it can be the case that the recommendations are of residual importance and, thus, do not help the optimizer choose a better execution plan. The next section discusses tuning the collection of statistics in the context of a DWH-side strategy.

Collect statistics strategies

The availability of statistics is the primary source of information for devising an optimal execution plan. Naturally, the question arises: What should be the DWH strategy for the statistics? In theory, the answer is simple: the ones that help the optimizer devise better execution plans; any additional ones are redundant and a waste of resources. In practice, this is hard to achieve.

A typical DWH serves many thousands of requests per day. Some are processed in batches (e.g., overnight or monthly processing), some periodically during the workday, while many others on an ad hoc basis. By and large, the query landscape is only partially known, and the data demographics may dynamically change through bulk imports from third systems. Collecting accurate statistics for all queries is neither feasible nor advisable. The associated costs and impact should be assessed on a system level.

A cost-benefit analysis may reveal that it is better to have inaccurate or no statistics for some queries (e.g., the rarely-executed ones and are not marked as business critical). The following sections analyze four statistics dimensions: *column type*, *value lookup*, *sampling volume*, and *trigger*. In our experience, these are the important ones to be considered when devising a strategy for collecting statistics.

Column type

Statistics may be collected on single columns, column combinations, and column expressions. By and large, the more specific the object of statistics, the less their overall impact and the more the resources to maintain them. In contrast, simple ones, such as single-column statistics, are generic and may improve multiple execution plans.

Column expression statistics should be carefully considered and always in conjunction with the output of the EXPLAIN request modifier. The Teradata optimizer is quite sophisticated and can rewrite SQL statements with constant expressions to better utilize existing columns and indices.

The first example depicted in Figure 113 benefits from a column expression statistic on the day of the month. The second example in Figure 113 includes an SQL query with a constant value expression on the column. The Teradata optimizer can rewrite the query in a functional-equivalent form that considers the column itself rather than an expression over it. The provided EXPLAIN output confirms this; the request differs from the condition of the retrieve step. As such, collecting statistics over the column expression is redundant and

wastes resources. It is better to collect single-column statistics, which are used both by the specific query and potentially many others that include a condition on this specific column.

```
1  -- Example 1: expression on monthday - no SQL rewrite, no recommendation
2  COLLECT STATISTICS COLUMN(EXTRACT(DAY FROM eventDate)) AS theDay ON tuning.event;
3  EXPLAIN SELECT * FROM tuning.event WHERE EXTRACT(DAY FROM tuning.event.eventDate) = 1;
4
5  -- Example 2: SQL rewritten by the optimizer; redunant collected statistics
6  COLLECT STATISTICS COLUMN(eventDate + 1) AS nextEventDate ON tuning.event;
7  EXPLAIN SELECT * FROM tuning.event WHERE eventDate + 1 = DATE '2020-09-01'; -- original query
8  COLLECT STATISTICS COLUMN(eventDate) ON tuning.event; -- suggested STATISTICS
```

Teradata Result Set Viewer

Result Set - SQL Editor (1)

```
EXPLAIN SELECT * FROM tuning.event WHERE eventDate + 1 = DATE '2020-09-01';

Explanation
------------------------------------------------------------------------
 1) First, we lock tuning.event in TD_MAP1 for read on a reserved
    RowHash in a single partition to prevent global deadlock.
 2) Next, we lock tuning.event 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 tuning.event with a condition of ("tuning.event.eventDate =
    DATE '2020-08-31'") with a residual condition of (
    "tuning.event.eventDate = DATE '2020-08-31'") into Spool 1
    (group_amps), which is built locally on the AMPs.  The size of
    Spool 1 is estimated with low confidence to be 1 row (41 bytes).
    The estimated time for this step is 0.00 seconds.
 4) Finally, we send out an END TRANSACTION step to all AMPs involved
    in processing the request.
-> The contents of Spool 1 are sent back to the user as the result of
    statement 1.  The total estimated time is 0.00 seconds.
```

Figure 113 Example of column expression statistics and SQL rewrite

Statistics may be collected for specific SQL functions, namely: SUBSTR, UPPERCASE, LOWERCASE, EXTRACT, CASE, and data type conversions. Such statistics are quite specific and, thus, their applicability is limited. However, they can improve the execution plan for specific queries. The example below supports queries over years for a date column:

```
COLLECT STATISTICS COLUMN (EXTRACT YEAR FROM OpenDate) AS OpenYear ON Customer;
```

Starting with Teradata 14.10, the number of bytes used to build a statistics histogram is configurable. The default value is 25 bytes, i.e., Teradata considers the first 25 bytes as significant for the histogram. This length may prove insufficient for multi-column statistics and result in inaccurate estimations. Let's assume the case of two multi-column statistics over two VARCHAR(20) columns with CHARSET LATIN (i.e., one byte per character). Then, the values of the first column are considered in full but only five characters of the second column. This may result in histograms that are highly biased towards the first column. The same holds for single-column statistics for columns that are wider than 25 bytes (e.g., the case of a column storing IBAN's; these can be up to 39 characters). A statement similar to the example below can be used to extend how many column bytes should be considered as significant for collecting statistics:

```
COLLECT STATISTICS USING MAXVALUELENGTH 50 COLUMN (CustomerName) ON Customer;
```

Typically, statistics on the PARTITION pseudo-column of PPI tables should always be collected. Their maintenance costs are low, and the probability of using DPE for PPI tables increases. This is especially useful for complex queries that involve multiple execution steps and product joins.

```
COLLECT STATISTICS COLUMN(PARTITION) ON Customer;
```

The optimizer assumes that table columns are not correlated. Hence, when statistics are available for different columns of the same table, they are assessed independently. When the columns exhibit a noticeable correlation, the collection of multi-column statistics on the combined columns may help the optimizer choose a better data access path and table join strategy. The recommended approach to collect multi-column statistics is an expression similar to the one of the example below:

```
COLLECT STATISTICS COLUMN(A,B) ON Customer; -- multi-column statistics on combined column (A,B)
```

The increased accuracy of multi-column statistics is offered at the expense of increased cost to collect and maintain them. The multi-column statistics cannot be used for querying the individual columns independently except in one case: when single-column statistics are not available, and the first column appears in the WHERE condition, the optimizer may utilize the multi-column statistics to improve its estimation. In this case, the optimizer reports a "low confidence" level for its decision instead of "no confidence" when they are not used at all. In contrast, single-column statistics result in "high confidence" levels and allow the optimizer to choose more aggressive execution plans. By and large, single-column statistics are more generic and can be used for a larger query population, but their collection must be assessed on a case by case basis. It might be the case that for a specific DWH, the queries always include both columns; it may then be a better strategy to collect one set of multi-column statistics rather than two sets of single-column statistics as the latter may never be used.

The optimizer and its DIAGNOSTIC HELPSTATS always recommend collecting all possible combinations of multi-column statistics. Such a recommendation must not be blindly followed. Rather, once single-column statistics on all columns are collected, rarely, if ever, the execution plan is improved by additionally collecting multi-column statistics.

Summary (or table-level) statistics also have low maintenance costs and should be typically collected. Summary statistics include the table row counts, the UDI counts (if activated), a dynamic AMP sample, the average data block size, information about block-level compression, and the data temperature. The summary statistics are automatically collected and updated whenever any column-level statistics are collected. If necessary, an independent collection can be performed and shown using a statement similar to the one in the example below:

```
COLLECT SUMMARY STATISTICS ON Customer; -- Collect
SHOW SUMMARY STATISTICS VALUES ON Customer; -- Show
```

In the past, it was recommended to collect partition-level statistics for the single partition of an NPPI table. Starting with Teradata 14.10, this can be replaced with a statement like the one in the example above.

By and large, summary statistics are the minimum information that should always be available to the optimizer. Teradata stores the previous summary statistics in a compressed form to build a history of records and derive the table growth trends with every recollection. Teradata does not blindly trust the latest summary statistics when assessing the costs of retrieve and join steps. Instead, it considers the following additional information to derive table row count estimates:

- When UDI counts are available, the insert and delete counts of the latest summary statistics are compared with the previous ones to adjust the row count estimates.
- The two dynamic AMP samples are compared to perform row count adjustments upward, if necessary.
- When sufficient historical summary statistics records are available, an established trend is used to extrapolate the table growth.
- All the calculations above are compared to each other to assess the most reliable one. When the UDI counts differ less than 5% with either the dynamic AMP sample or the table growth trend, the optimizer considers the UDI counts reliable. Otherwise, the dynamic AMP sample counts are considered in the final estimation.

The optimizer additionally considers dynamic AMP sampling to detect stale statistics. When the counts deviate significantly from the collected ones, the optimizer considers the dynamic AMP sample counts.

The optimizer does not consider the additional CPU seconds needed to decompress a data block when the table is block-level compressed. Therefore, the optimizer calculates a shorter processing time in this case, which may cause, in extreme cases, a change in the execution plan.

Value lookup

The optimizer stores the exact count of rows for those column values that are the most frequent in an interval. As such, the cardinality of a query with an equality constraint on the specific column-value pair is readily available by a lookup on the collected statistics "*The most frequent value in the interval*" and "*The cardinality of the most frequent value*". Any other cardinality is an estimation by the optimizer.

The estimated cardinality for an equality-based value search results from dividing the "*Number of distinct values in the interval*" by the "*Cardinality of other values in the*

interval". When the search value spans multiple intervals, the results over all containing intervals are summed up.

In the cases of non-equality and range queries, the optimizer divides "*The cardinality of other values in the interval*" by two at each interval containing the non-equality or range values and sums up, if necessary. If the most frequent value is contained in the range, the optimizer considers the exact figure for this specific value.

The optimizer treats differently the case where the queried value lies outside the value range of all intervals. The optimizer estimates then the cardinality to be equal to the count of distinct column values in the entire table. This is an overestimation of the cardinality, a purposefully conservative choice. A more aggressive choice may result in an underestimation, which can end up in performance degradation (e.g., due to insufficient memory for join strategies that rely on in-memory processing of full tables, as in the case of product joins).

Sampling volume

The collection of statistics can be based on full table scans (i.e., *full statistics*), on volume sampling (i.e., *sample statistics*), or on sampling a selected AMP and dynamically extrapolate the statistics over all AMP's (i.e., *dynamic AMP sampling*). Data demographics, table skewing, and column types impact the selection of a proper sampling volume approach.

By and large, full statistics should be the preferred collection method, as it ensures accurate metrics irrespective of the data demographics. However, the maintenance costs are higher than the other two alternatives. Thus, when the resource usage for full statistics is not considered tolerable, the alternative options should be considered, as long as they do not harm estimation accuracy.

Full statistics collection may be skipped for *UPI columns* used in equality-based row filtering conditions. In such cases, sample statistics or even dynamic AMP sampling is sufficient. However, full statistics are necessary when the UPI columns are used in inequality-based filtering conditions. When range-based conditions are used but are broad enough, the optimizer considers a better strategy to perform an FTS and filter on the range values.

A generic recommendation for a *NUPI column* is not possible; it must be assessed on a case by case basis. Depending on data demographics and actual table skewing, sample statistics, or even dynamic AMP sampling, may be sufficient. The optimizer estimates that the NUPI values are 75% unique. Thus, any NUPI that does not heavily deviate from this number does not impact the execution plan choice. As a conservative strategy, full statistics should be collected, provided that their maintenance cost is acceptable.

Both a UPI and a USI cannot be value-skewed by definition. As such, a dynamic AMP sampling always reflects actual data demographics and can be used instead of costly full statistics collection.

Full statistics are necessary for a *NUSI column,* or else the optimizer does not consider its use at all. They are also necessary for *non-indexed columns* used in filtering conditions for qualifying table rows. The alternative is dynamic AMP sampling; its estimation typically results in conservative, sub-optimal execution plans.

Full statistics must always be collected for small tables, especially for those with a row cardinality less than the count of available AMP's. In such cases, the table is definitely skewed. Any sampling attempt is bound to provide inaccurate estimates, including extrapolation of zero rows overall, when dynamic AMP sampling is performed on an AMP storing no rows of the table. The maintenance cost for full statistics on small tables is negligible, and avoiding it is unjustifiable from a risk perspective.

PPI tables require full statistics on the PARTITION pseudo-column. The optimizer utilizes these statistics for detecting empty partitions and estimating the row count per partition. Full statistics are also required for the combinations of the PARTITION pseudo-column, the Primary Index column(s), and the partitioning columns themselves. The PARTITION and PI combination provide the optimizer the information about ROWHASH distribution to assess the possibility for DPE, sliding window merge join, and ROWKEY-based merge join. The triple combinations allow the optimizer to estimate the number of distinct values after (dynamic) partition elimination for the PI and partitioning column combinations.

In principle, an execution plan with non-stale collected statistics, either full or sampled, is always better or at least no worse than an execution plan based on dynamic AMP and heuristics. Additionally, when statistics of the former type are not available, Teradata excludes optimization techniques such as DPE and partial redistribution of rows during joins.

Trigger

Collected statistics must be periodically refreshed, or else they become stale. Stale statistics lead the optimizer in choosing underperforming execution plans and hamper performance. In the past, tuning the refresh frequency was tedious, requiring a lot of manual work and human judgment. Multiple factors had to be taken into account on a case by case basis, including:

- Should all statistics be collected at once?
- How often and when during the day should they be collected?
- How should requirements for statistics be handled in SQL batch scripts?
- Can the recollection process be postponed for a later time, or the statistics are stale already?

To put this into perspective, let's assume the case of a night batch ETL process executing every working day after business hours. Should the statistics be refreshed after each bulk import? Should the statistics be refreshed after a transformation step, the results of which are used by multiple follow-up transformations? Should the ETL process be delayed or interrupted for refreshing the statistics? Should the statistics be collected after the ETL process concluded and for supporting the business hour queries? The questions only multiply when ETL processes of mixed periodicities (e.g., daily and weekly batches) are considered. Should the collection occur on a daily or weekly basis? Or split into two parts? What if a table is updated by both a daily and weekly process? All questions resorted to finding an acceptable balance between resource consumption and the accuracy of estimates.

Starting with Teradata 14.10, the so-called THRESHOLD options are available to automate the statistics collection process and realize *threshold statistics*. Still, it remains a human decision which statistics to collect in the first place. In principle, the THRESHOLD options guide Teradata to skip a COLLECT STATISTICS statement and save the associated resource consumption for an unneeded collection. We consider a good practice to enable the THRESHOLD options to automate the collection of statistics and avoid stale statistics.

A *time-based threshold* skips a COLLECT STATISTICS until a defined number of days (the threshold) has passed since the last collection. On the other hand, a *change-based threshold* skips collection when the data demographics have not changed more than a defined percentage (the threshold) since the last collection of statistics. The threshold percentage can be either calculated or estimated using various techniques discussed below.

The THRESHOLD options can achieve significant resource consumption savings. In the past, every COLLECT STATISTICS statement was executed and consumed resources, even if the statistics did not change at all since the last collection. The THRESHOLD options allow now to inject COLLECT STATISTICS statements at any point in the ETL process deemed necessary, with the assurance that collection will be only be activated if necessary, i.e., when the defined thresholds are exceeded. Teradata implements three types of thresholds: *system*, user, and *global*.

The system thresholds are determined per table by Teradata itself. They can only be changed-based thresholds. When activated, the optimizer estimates the growth of the table since the last time statistics were collected. The estimation is used to decide whether a recollection should occur. The estimation relies on the established trend derived from the historical records of summary statistics. Thus, it is crucial that tables are not dropped and recreated; when this happens, the historical records are lost, as the recreated table is considered a new one and is assigned a new TABLEID. System thresholds can be activated if OUC are activated, or else the optimizer falls back to the full collection of statistics. The optimizer relies on the UDI counts of the OUC to detect the changes in the table data demographics. We consider a good practice to always active OUC and system thresholds, as they significantly reduce the maintenance burden for collecting table statistics.

The user thresholds are defined per table. They should be enabled only when they provide better statistics than the system thresholds. User thresholds do not rely on table growth estimates. On the other hand, both time- and change-based thresholds can be defined, either individually or combined, using statements similar to the examples below:

```
COLLECT STATISTICS USING THRESHOLD 7 DAYS COLUMN CustomerId ON Customer; -- time-based
COLLECT STATISTICS USING THRESHOLD 5% AND THRESHOLD 7 DAYS COLUMN CustomerId ON Customer; --
combined
```

Global thresholds can only be set by the database administrator and apply to the whole Teradata system. Global thresholds can be both time- and change-based. Teradata uses two parameters to guide global thresholds, DefaultTimeThreshold and DefaultUserChangeThreshold, that can be switched off and on by setting them to a zero and a non-zero value, respectively.

The DefaultTimeThreshold parameter defines the count of days to act as a change threshold. When these days have passed, the next COLLECT STATISTICS statement is not skipped but executed.

The DefaultUserChangeThreshold treats the percentage as a change threshold; when this percentage is exceeded, the next COLLECT STATISTICS statement is not skipped but executed. In contrast to system thresholds, OUC need not be activated for this parameter to be used; the optimizer can revert to dynamic AMP sampling in this case, with the associated challenges of potential skew.

A typical usage scenario for global thresholds is when a system upgrade from a version older than Teradata 14.10 occurs. A tailor-made statistics collection framework was used for time-based collection. If a transition to change-based threshold statistics is considered too risky, global thresholds on both time- and change can be transparently activated to allow for a smoother transition.

Teradata prioritizes the use of the various threshold statistics. A user-defined change- or time-based threshold takes precedence over any other threshold. When both a user and a global threshold are defined, the global threshold complements the missing user one (if missing in the first place). For example, the user change threshold of 5% is combined with a global time threshold of seven days in the example below:

```
COLLECT STATISTICS USING THRESHOLD 5% COLUMN CustomerId ON Customer; -- user change threshold
-- DefaultTimeThreshold=7 -- global time threshold of seven days
-- DefaultUserChangeThreshold=10 -- global change threshold of 10%
```

System thresholds are used only when user thresholds are not defined, and global thresholds are switched off. A statement similar to the example below can be used to switch to system thresholds when user thresholds are defined:

```
COLLECT STATISTICS USING SYSTEM THRESHOLD COLUMN(CustomerId) ON Customer;
```

We consider a good practice to always opt for system thresholds with OUC. Global thresholds are useful in the transition phase from an older system and its statistics collection framework. User thresholds should be used only when system thresholds are not sufficiently accurate for a specific statistic. In principle, change-based thresholds should be preferred over time-based ones, as they can better capture dynamic shifts in data demographics and are less resource-intensive.

A statement similar to the one in the example below can be used to force the collection of statistics at the next execution temporarily:

```
COLLECT STATISTICS USING NO THRESHOLD FOR CURRENT COLUMN(CustomerId) ON Customer;
```

When the FOR CURRENT option is not present, the threshold method is permanently updated. As such, it will be used for every future collection of the specific statistic.

A Pareto-inspired strategy

We consider a good practice to follow a bottom-up and step-wise approach to collecting statistics. The initial step is to rank the query population based on a metric that considers the execution frequency, the system impact, and the business criticality. Then, the top-20% of the queries are selected and analyzed further.

The first analysis step is to check SQL for syntax-level improvements. In our experience, significant performance improvements can be achieved by rewriting SQL statements to better utilize the defined PDM and the available statistics, without the need to change either. The next section will explore this tuning concept in more detail and with concrete examples; here, it is assumed that the SQL statements are already tuned to the extent possible.

In the second step of the analysis, *single-column statistics* that the HELPSTATS recommend with "*high confidence*" should be collected in small batches (or even one-by-one, when the query population is not significant). On every batch, the impact of collected statistics on the whole sample of 20% should be assessed. The reason is that cascade effects are possible: a single-column statistics collection performed during a batch might also improve the execution plans of statements of follow-up batches, i.e., HELPSTATS does not offer any recommendations anymore. In this case, the additional queries are tuned "for free" and can be removed from the list.

In the third step of the analysis, *multi-column high-confidence statistics* should be collected using the same approach as before. The rationale is the same; small batches ensure that the cascade effect can occur and clean the list for free without stressing to collect multiple types of statistics that end up being redundant.

If necessary, a fourth step is performed for "medium confidence" recommendations for statistics, respecting the rank order. As before, single-column statistics should be tried first, and when the situation does not improve, then multi-column ones should be considered.

This Pareto-inspired strategy works quite well in our experience: resource savings of more than 80% can be attributed to tuning just 20% of the queries by simply rewriting SQL and collecting mostly single-column statistics.

SQL rewriting

The optimizer performs SQL rewriting whenever it detects statements that can be reformulated and reduce the number of accessed tables and columns. Towards this direction, providing clean SQL is the most critical factor for devising optimal execution plans with simplified expressions that may even surpass the knowledge and experience of any performance tuner.

In our experience, once refreshed statistics are collected, rewriting SQL is sufficient to solve most tuning challenges without changing the PDM or introducing additional database objects, such as new join indices. Apart from the resource savings, an SQL rewrite is less complicated and time-consuming to perform. Furthermore, it requires no additional maintenance after it is changed.

The next sections explore SQL tuning topics, in ascending order of complexity to analyze and change. As an empirical rule, the less complex the analysis, the more the performance improvement.

Star considered harmful

Nothing harms more the performance than a "SELECT * FROM". It may be convenient and speed up development, but DWH operations suffer forever.

A star selection consumes more spool space, rendering "out of spool" error message more probable. Further, it may cause contention when transferring unnecessarily-wide spooled tables from AMP to AMP. A star selection may also rule out the use of join indices, force unnecessary accesses to the base tables (join backs), rule out specific join methods, and result in suboptimal join strategies.

BTEQ scripts are not the only place where a star selection may reside. Typically, view objects are defined as star selections. They may also contain filters. In such cases, it may be beneficial to define two views: a general-purpose one (e.g., for cross-database access) with a star selection and a tailor-made one to serve those queries relying on the filter conditions. The latter should then include only the columns necessary to fulfill the specific queries.

Monoliths and branches

A large, monolithic query is susceptible to hitting the resource limits, especially to exhausting the spool space, as more and more results pile up. We consider a good practice to split such queries into smaller processing chunks; this may also benefit testing and debugging. UNION and UNION ALL are candidate points to replace with multiple INSERT statements grouped in multi-statement requests.

Complex subqueries are good candidates to materialize in volatile tables before further processing. Such volatile tables can be enriched with proper indices and statistics to help the optimizer select optimal execution plans. Volatile tables can also be used to split complex table joins, especially those not utilizing access via an index (e.g., when joining a relationship table to its both member tables).

The presence of functions in equi-join conditions prohibits using the efficient ROWHASH-based data access path and most of the index-based ones unless they are built specifically for this condition. In our experience, the presence of functions, similar to the example below, is a strong indicator of suboptimal data modeling, low data quality, or both. Ideally, the underlying cause should be addressed as a priority and eliminate the function-based equi-joins.

```
-- TRIM applied in the equi-join condition; use of surrogate keys may be a better approach
SELECT a.*, b.* FROM a INNER JOIN b ON a.SUBSCRIBER_ID = TRIM(LEADING '0' FROM b.SUBSCRIBER_ID)
```

Each table join results in either a full table scan or an index-based data access path. Since the optimizer can handle up to two tables at each step, the less the tables involved in a query, the less the execution steps and resource consumption, especially spool space. Multiple joins on the same table should be revised whenever possible. Query rewriting using subqueries, volatile tables, or ordered analytical functions may be used to eliminate the multiple joins, as in the example below:

```
-- Query before rewrite, multiple accesses to table tblCall
SELECT A.SUBSCRIBER_ID, B.VALUE AS VAL_IN, C.VALUE AS VAL_OUT
FROM A
LEFT JOIN tblCall B
ON B.SUBSCRIBER_ID = A.SUBSCRIBER_ID AND B.IO_CD = 'IN'
LEFT JOIN tblCall C
ON C.SUBSCRIBER_ID = A.SUBSCRIBER_ID AND C.IO_CD = 'OUT'

-- Query after rewrite, pivot tblCall in a subquery, single access
SELECT A.SUBSCRIBER_ID, D.VAL_IN, D. VAL_OUT
FROM A
LEFT JOIN (
    SELECT SUBSCRIBER_ID
        , Max(CASE WHEN IO_CD = 'IN' THEN VALUE END) AS VAL_IN
        , Max(CASE WHEN IO_CD = 'OUT' THEN VALUE END) AS VAL_OUT
    FROM tblCall
    GROUP BY SUBSCRIBER_ID
) D
```

Typically, column value conditionals are needed, e.g., assign an output column value based on specific input column value or value ranges. By and large, these conditionals are realized with CASE expressions. COALESCE, and NULLIF expressions are ANSI-SQL-compliant alternatives, which achieve equivalent results, may improve code readability, and allow better testing, as in the examples below:

```
SELECT CASE WHEN myAvg IS NOT NULL THEN 100*myAvg ELSE 0 END;
SELECT Coalesce(100*myAvg, 0);

SELECT CASE WHEN col2 <> '' THEN col2 END;           -- CASE silently returns NULL
SELECT CASE WHEN col2 <> '' THEN col2 ELSE NULL END; -- less compact, clear CASE return
SELECT NULLIF(col2, ''); -- more compact, return col2 but NULL if blank

SELECT CASE WHEN col2 IS NOT NULL THEN col2 WHEN col3 IS NOT NULL THEN col3 ELSE 'AAA' END;
SELECT Coalesce(col2, col3, 'AAA');
```

Teradata offers the GREATEST and LEAST expressions for numerical comparisons, including DATE data types, since Teradata internally represents the latter as numbers. These expressions are extensions to ANSI SQL but can greatly simplify the minimum and maximum calculation over a set of columns, as in the examples below:

```
SELECT CASE WHEN col1 >= col2 THEN col1 ELSE col2 END AS maxVal;
SELECT GREATEST(col1, col2) AS maxVal;

SELECT CASE WHEN col1 <= col2 AND col1 <= col3 THEN col1 WHEN col2 <= col1 AND col2 <= col3 THEN
col2 WHEN col3 <= col1 AND col3 <= col2 THEN col3 ELSE NULL END AS maxVal;
SELECT LEAST(col1, col2, col3);
```

Number casting and rounding errors

Data input checks, especially in ELT processes, sooner than later involve data type checks. Typical conversion targets are numbers with and without decimals, as different sources may represent differently such data. Before Teradata 14.10, such a check is typically realized via a workaround based on the letter-case comparison, similar to the example below:

```
-- not always correct, CAST afterward if isNumeric=1
SELECT CASE WHEN UPPER('3.14') = LOWER('3.14') THEN 1 ELSE 0 END AS isNumeric;
```

Starting with Teradata 14.10, the function TO_NUMBER offers a more reliable and efficient check. However, being a UDF (user-defined function), the optimizer does not consider it when defined in an index, even when statistics are collected for it. Respectively, the optimizer does not consider the underlying number structure for row filtering in a query. Instead, the optimizer converts the number to a VARCHAR data type and resorts to a full table scan, as depicted in Figure 114. Note that the return type of the UDF is a VARCHAR(32000), which further stresses resource consumption and may lead to memory errors when multiple explicit or implicit conversions are cascaded in a statement.

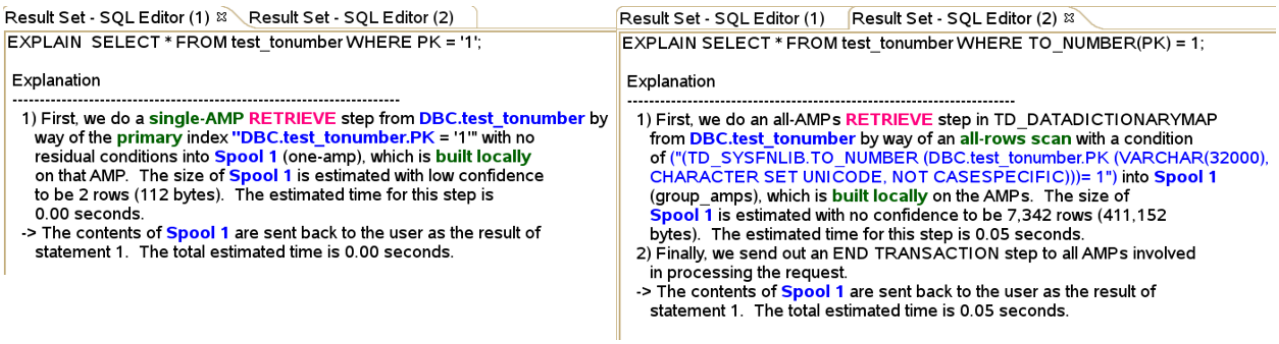

Figure 114 Full table scan and Unicode conversion due to TO_NUMBER in the WHERE clause

The Teradata SQL extension TRYCAST is available since version 15.10 and allows a robust check against any Teradata supported numerical type. In contrast to CAST, this extension does not fail but instead returns NULL when the casting fails, further simplifying code logic:

```
-- v14 onward, CAST afterward if isNumeric=1
SELECT CASE WHEN TO_NUMBER('3.14') IS NOT NULL THEN 1 ELSE 0 END AS inNumeric;
-- v15.10 onward, result directly available
SELECT Trycast('3.14' AS DECIMAL(3,2)) AS resCol;
```

A common pattern in expressions over numeric types is rounding errors in divisions. In the simplest form, when a full percentage (100%) is split into three parts (rows), then each row is assigned 33.3%, and the sum of the three rows is not a full percentage anymore. There are multiple approaches on how to cope with this issue with varying implementation complexity and performance overhead. In our experience, the following example provides an elegant solution that is also a performant one:

```
SELECT theMonth, Sum(val) AS theSum -- check the total sum of the split
FROM (
    SELECT 1 AS theMonth, colAmount/colMonth FROM soureTable -- assume correct type conversion
here
    UNION ALL SELECT 2 AS theMonth, colAmount/colMonth FROM soureTable
    UNION ALL SELECT 3 AS theMonth, colAmount/colMonth FROM soureTable
    -- assign the rounding error, if any, to the last row
    UNION ALL SELECT 3 AS themonth, colAmount - (colAmount/colMonth * colMonth) FROM sourceTable
) X
GROUP BY theMonth
ORDER BY theMonth ASC

-- Example output for (colAmt, colMonth) = (10.25, 3)
1 3.42
2 3.42
3 3.41
```

Strings

A DWH typically integrates information from dispersed source systems, each with its conventions for information processing. On a technical level, this is evident in the variety of codepages and encodings used to represent the same for strings of characters. On a

business level, this is evident in the variety of norms for free-text columns, including human-queried ones (e.g., a customer name, an address, a telephone number, a product name, or a description). The CHAR2HEXINT function can be used to detect representation issues and incorrect translations that harm performance. This is especially true when data are translated from source encoding (e.g., EBCDIC) into target encoding (e.g., ASCII), and variants of codepages are involved. SQL rewriting offers ample potential for performance improvement of character string processing.

Implicit Latin-to-Unicode conversions are a typical cause of index bypassing when comparing character-based columns. The EXPLAIN output of the statement can reveal that Teradata implicitly converts a string containing ASCII characters to Unicode. Albeit the internal representation of a Teradata LATIN string being the same as in Teradata UNICODE, an implicit conversion via a function occurs always. The Latin-to-Unicode conversion can cause significant performance degradation, as demonstrated in the example below.

```
DELETE FROM LatinTable
WHERE Col1 IN (SELECT Col1 FROM UnicodeTable);
```

In the initial design, the LatinTable is defined with a LATIN column col1 and contains many rows, while the UnicodeTable contains a single row with a UNICODE column col1. The execution plan selected by the optimizer involves three steps: first, spool all the LatinTable rows fanned out in hash join partitions; then convert col1 of each row into Unicode; finally, hash join with the single-row UnicodeTable.

In the tuned design, the CREATE TABLE UnicodeTable statement is rewritten to define col1 as LATIN character set instead. The execution plan now considers a dynamic hash join, which requires no spool space at all, as depicted in Figure 115. This simple SQL rewrite resulted in 99.96% less I/O operations and 96% fewer CPU seconds in an actual tuning case one of the authors worked.

```
EXPLAIN
DELETE FROM LatinTable
WHERE Col1 IN (SELECT Col1 FROM LatinVolatileTable);

5) We do an all-AMPs JOIN step in TD_MAP1 from DWHPRO.LatinTable by
     way of an all-rows scan with no residual conditions, which is
     joined to Spool 5 (Last Use) by way of an all-rows scan.
     DWHPRO.LatinTable and Spool 5 are joined using a inclusion dynamic
     hash join, with a join condition of ("DWHPRO.LatinTable.Col1 =
     Col1").  The result goes into Spool 2 (all_amps), which is built
     locally on the AMPs.  Then we do a SORT to order Spool 2 by the
     hash code of (DWHPRO.LatinTable.ROWID) the sort key in spool
     field1 eliminating duplicate rows.  The size of Spool 2 is
     estimated with index join confidence to be 2,716,348 rows (
     48,894,264 bytes).  The estimated time for this step is 15.59
     seconds.
```

Figure 115 Rewritten CREATE TABLE results in dynamic hash join and no Latin-to-Unicode conversion

Typically, the LIKE operator results in a full table scan. There are some strategies to reduce the processing burden and improve performance. First and foremost, the selection should not mention unneeded columns (i.e., avoid a star selection) to help the optimizer perform a full-table scan over a defined secondary or join index table instead of the base table.

A tailor-made NUSI or single-table join index definition may help the optimizer avoid the full-table scan and instead perform index-based data access. However, this is a hard-to-achieve data path; in our experience, the optimizer dominantly favors the full-table scan. When the searched pattern resides on the left side of the expression (i.e., the clause reads similar to "WHERE col LIKE 'pattern%'"), the optimizer may consult the available statistics for row count estimates and choose an alternative data access path. In all other cases (e.g., when the clause reads similar to "WHERE col LIKE '%pattern%'), the optimizer considers less costly, hence, preferable, to perform a full table scan.

A third approach may prove beneficial by trading off increased storage space and maintenance for reduced full-table scans and increased performance. Let's assume a wide table A which queries frequently need to access with a LIKE operator. The first step is to create an auxiliary table B with the same primary key columns as A and collect statistics. The queries are then rewritten, similar to the example below, to access both tables but with the condition on the auxiliary table B.

```
SELECT a.*
FROM a
INNER JOIN b -- contains the columns of the primary key and the LIKE operator of table a
ON a.pk = b.pk
AND b.col LIKE '%pattern%'
```

The approach of joining two tables instead of accessing a single table appears counter-intuitive from a performance tuning point of view. Upon closer inspection, it can be seen that the optimizer is forced to perform a full-table scan on the auxiliary table and only move to the spool those rows of the table that satisfy the LIKE operator. In a second step, the optimizer performs an AMP-local primary index merge join with the matching rows of table A. This execution plan achieves a significant reduction in I/O operations for transferring the data blocks of A. In a sense, the approach emulates the operation of a single-table join index but with guaranteed access. As a further enhancement, the auxiliary table can be constructed to contain only those rows matching specific LIKE expressions.

There are multiple approaches to string matching, including SUBSTR (ANSI-compliant), SUBSTRING (Teradata extension to ANSI SQL), LEFT, LIKE, and regular expressions. In our experience, there is no clear performance advantage when using one or another operator but rather depends on data demographics and the structure of the searched pattern. A performance advantage is actually achieved when the SQL is rewritten. The operators are applied as early as possible and as close as possible to their target table; the optimizer then spools only the necessary characters for further processing rather than the whole column. When indices or partitions are available for the target table, we consider a

good practice to rewrite the SQL queries to match the index (partition) definitions rather than rely on the optimizer to identify the expression equivalence and rewrite the SQL itself.

Lengthy VARCHAR columns can be used to store long text descriptions (e.g., a product description). A column with a Latin character set can hold up to 64,000 characters, while a Unicode one has at most 32,000 characters. The more the characters are stored in such a column, the fewer the table rows can fit in a single data block. Then, full-table scans become more expensive in terms of I/O operations. Let's assume a table definition similar to one in the example below:

```
CREATE TABLE longTable (
    id INTEGER NOT NULL
    , col1 CHAR(1)
    , desc VARCHAR(30000)
) PRIMARY INDEX (id);
```

When a sub-table index is not available, any query on this table results in a suboptimal number of data block transfers from disk to AMP memory, even for accessing a single column (e.g., "id"). The situation may be improved by defining a covering index as a sub-table. Then, the same amount of information becomes available in the AMP memory with fewer data block transfers. An alternative and more efficient approach is to use the CLOB data type for the lengthy "desc" column: Teradata transparently stores any CLOB column in a separate sub-table and not in the base table. Thus, the same performance is achieved without any additional index definition and maintenance.

Time periods

Date calculations are typically needed in a DWH environment storing temporal data. In the past, complex calculation logic had to be defined in SQL queries to combine data of specific validity in time. The latest versions of Teradata implement OVERLAPS defined in ANSI SQL:2011, which simplifies calculations of intervals relations. We consider a good practice to replace any custom logic for accessing temporal data with the standardized OVERLAPS.

Teradata represents internally a DATE data type as an integer using the formula: *10,000*(year-1,900) + 100 * month + day*. As such, a cast from an integer to a date type is possible, and Teradata automatically performs a validity check, as depicted in the example of Figure 116. We consider a good practice to use date data types rather than perform date calculations over integer arithmetic. The latter harms readability and, in our experience, leads to complex and hard to debug expressions (e.g., how to calculate the last day of the next month over the formula mentioned above?).

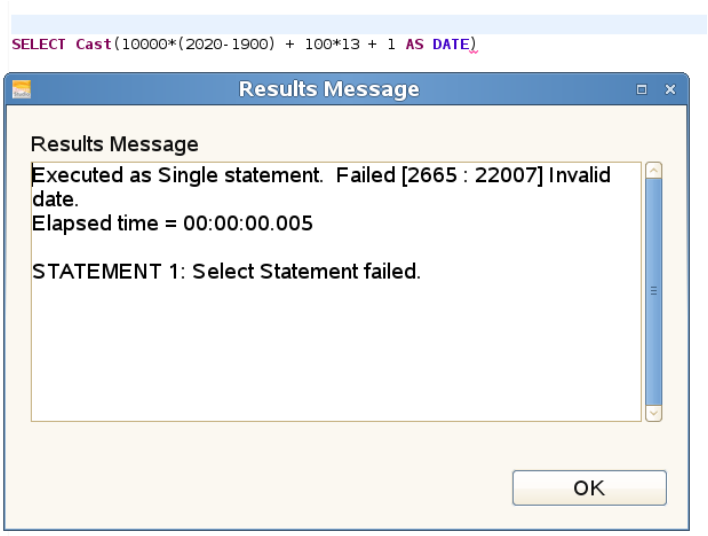

Figure 116 Teradata checks an integer-to-date cast validity

By and large, date calculations exhibit better performance when the EXTRACT function is used instead of string operations (e.g., a SUBSTR or SUBSTRING) on a date column. A common date calculation is to find the first day of the month or the last day of another month. In our experience, TRUNC and OADD_MONTHS (instead of ADD_MONTHS) respectively exhibit both better readability and performance compared to equivalent expressions using EXTRACT and ADD_MONTHS, as in the examples below:

```
-- first day of the month
CURRENT_DATE - (EXTRACT(DAY FROM CURRENT_DATE) - 1)
SUBSTRING(CURRENT_DATE FROM 1 FOR 8) || '01' (DATE)
TRUNC(CURRENT_DATE, 'MM')

-- last day of next month (go backward n-1 days, go forward 2 months, go backward 1 day)
ADD_MONTHS(CURRENT_DATE - (EXTRACT(DAY FROM CURRENT_DATE) - 1), 2) - 1
-- more readable version with OADD_MONTHS
-- (go backward to last day of the previous month, go forward 2 months)
OADD_MONTHS(CURRENT_DATE - EXTRACT(DAY FROM CURRENT_DATE), 2)
```

When indices or partitions are available for the target table, similar to the case of strings, we consider a good practice to rewrite the SQL queries to match the definitions rather than relying on the optimizer to identify the expression equivalence itself.

Sets and bags

SQL operates on bags rather than sets, as discussed in Chapter 3. Rarely, if ever, a bag is the desired query response from a DWH. Converting a bag to a set typically involves a DISTINCT or a GROUP BY operator. The performance of the two options can be quite different. When a DISTINCT is applied, Teradata redistributes qualifying rows without any pre-aggregation taking place. Then, if only a few duplicate rows exist, the local processing

savings justify the increased traffic caused by the duplicates. On the other hand, when a GROUP BY is applied, Teradata locally performs a pre-aggregation step to sort the rows and remove any duplicates before redistributing the rows to the AMP's. Hence, which operator to prefer is a matter of actual data demographics.

As a cautionary note, when DISTINCT is applied, and the data are heavily skewed, then one or a few AMP's may receive a large share of rows and run out of spool space. If the underlying skewing issue cannot be addressed (e.g., by using a more appropriate primary index), replacing DISTINCT with GROUP BY may be an alternative solution.

Set operations may also result in duplicate rows. Whether the query response contains a bag or a set depends on the SQL statement used to define the set operation. UNION ALL and INTERSECT ALL do not filter out duplicates, i.e., they respond with bags. On the other hand, UNION and INTERSECT filter out duplicate rows, i.e., their response is a set.

Teradata redistributes the rows of a UNION- or INTERSECT-based query so that ones with a common primary index reside at the same AMP. This is a necessary preparation for the next step: each AMP sorts the received rows and removes duplicate ones. This may negatively impact performance due to the volume of I/O operations needed and the volume of data to be spooled at each AMP. As mentioned already, a star selection can only worsen the situation. We consider a good practice always to replace UNION with UNION ALL and INTERSECT with INTERSECT ALL when it is guaranteed that the data will not contain any duplicate rows.

Before Teradata 16, the optimizer first collected in a common spool all the branches of a UNION ALL and only then performed any reductions to each branch (e.g., filter rows based on a branch-specific WHERE condition). Starting with Teradata 16, the optimizer may first perform any reductions (including aggregations and joins with other tables) in a local spool and only afterward collect the output in a common spool for further processing. The new approach may use significantly fewer resources; the more the branches combined with a UNION ALL, the more the savings. This is one more argument to rewrite, when possible, UNION as UNION ALL. The availability of up-to-date statistics heavily influences the decision of the optimizer to use the new or the earlier approach.

The following example demonstrates the benefit of the Teradata 16 approach. Let's assume a query like "SELECT count(*) FROM (SELECT * FROM a UNION ALL SELECT * FROM b UNION ALL SELECT * FROM c) x". Using the new approach, the optimizer simply counts the rows on the Cylinder Index of each table and sums them up to derive the final count. In contrast, the old approach collected the rows from all the three tables in a common spool and counted them one by one to derive the final count.

A UNION ALL may also be utilized to perform an unpivot of two or more columns of a single table. The established approach results in two full table scans:

```
SELECT OpenDate FROM Customer
UNION ALL
SELECT CloseDate FROM Customer
```

There is an alternative approach that requires just one full table scan and, thus, results in halving the costly I/O operations at the expense of additional memory consumption during the calculation:

```
SELECT CASE WHEN t02.NBR = 1 THEN t01.CloseDate WHEN t02.NBR = 2 THEN t01.OpenDate ELSE NULL END
FROM Customer t01 -- accessed once, each row is duplicated through the CROSS JOIN
CROSS JOIN (
    SELECT x AS NBR
    FROM (SELECT 1 AS X) AS X
    UNION ALL
    SELECT x AS NBR
    FROM (SELECT 2 AS X) AS X
) t02
WHERE t02.NBR IN (1,2)
```

Grouping expressions

Aggregations typically require to collect all related columns on the same AMP as a preliminary step. They can significantly impact the performance in terms of lengthy execution plans, cross-AMP traffic, and memory consumption. Grouping over functions and expressions should be rewritten whenever possible. Even the most straightforward expression may cause a redistribution of the spooled rows based on a new ROWHASH needed for the row aggregation. This holds even for value concatenation, as in the example below:

```
-- Not recommended, new ROWHASH created after string concatenation.
SELECT COALESCE (SUBSCRIBER_ID, 0) || COALESCE(DESCRIPTION ,'') AS subscriber_description
FROM theTable
GROUP BY COALESCE (SUBSCRIBER_ID,0) || COALESCE(DESCRIPTION ,'');
-- or GROUP BY 1 OR GROUP BY subscriber_description

-- Recommended form, may reuse existing columns and ROWHASH.
SELECT COALESCE (SUBSCRIBER_ID, 0) || COALESCE(DESCRIPTION ,'') AS subscriber_description
FROM theTable
GROUP BY COALESCE (SUBSCRIBER_ID, 0), COALESCE(DESCRIPTION ,''); -- change || to a comma
```

By and large, the more clearly-written the GROUP BY statements, the more probable the optimizer will be able to rewrite the query and devise a better execution plan. *Early aggregation* and *partial aggregation* are two approaches explored by the optimizer toward this aim. The former approach moves the aggregation steps early in the execution plan, assuming that *data reduction* will be exhibited, i.e., the cardinality of the result set of the early step will be significantly smaller than the one when performed as a later step. The latter approach performs some parts of the requested aggregation early in the execution plan, aiming for data reduction. Whether early and partial aggregations are performed is decided by the optimizer based on the calculated costs for the involved operations against the expected benefits of introducing additional steps in the execution plan.

In the example below, the SQL query is compact but, in our opinion, not clean and readable. Still, the optimizer can rewrite it and perform a partial aggregation on columns t01.b and t01.d before proceeding with the joins with the rest of the tables. A final aggregation is performed after all the joins.

```
SELECT SUM(t01.a)
FROM t01,t02,t03
WHERE t01.b = t02.b AND t02.c = t03.c
GROUP BY t01.d
```

Various forms of aggregations and grouping expressions are typically used in data marts to simplify the DWH information in a convenient and familiar representation to specific groups and functions of an enterprise (i.e., the DWH consumers). In our experience, such aggregations tend to be tailor-made to cover specific needs on an ad hoc approach, incorporating unnecessarily complex expressions, which exhibit poor performance and become a friction point between the DWH operations groups and the DWH consumers. We consider a good practice to rewrite such SQL queries by exploring the advanced Teradata grouping functions GROUP BY ROLLUP (group columns over a single dimension), GROUP BY SETS (group set of columns over a dimension), and GROUP BY CUBE (group over multiple dimensions). The examples below demonstrate the effectiveness, readability, and performance benefit of using the advanced grouping functions.

```
-- Example table definition storing number of flights per calendar date and aircraft
CREATE MULTISET TABLE FlightRecord (
    aircraft AS BIGINT NOT NULL
    , calDate AS DATE NOT NULL
    , numFlights AS INTER NOT NULL
) PRIMARY INDEX (aircraft);

-- calculate the total number of flights per year, per month, and aircraft (before rewrite)
SELECT aircraft, Sum(numFlights) AS numFlightsPerAircraft
FROM FlightRecord
GROUP BY aircraft;

SELECT Extract(YEAR FROM calDate) AS theYear, Sum(numFlights) AS numFlightsPerYear
FROM FlightRecord
GROUP BY theYear;

SELECT Extract(MONTH FROM calDate) AS theMonth, Sum(numFlights) AS numFlightsPerMonth
FROM FlightRecord
GROUP BY theMonth;

-- Rewrite to collect all at once and reuse later
SELECT aircraft, Extract(YEAR FROM calDate) AS theYear, Extract(MONTH FROM calDate) AS theMonth,
Sum(numFlights) AS numFlights
FROM FlightRecord
GROUP BY GROUPING SETS (aircraft, theYear, theMonth);
```

Ordered analytical functions

Ordered analytical functions and window aggregate functions allow to perform calculations that use column values of an ordered set of results, e.g., the running total of the last three rows in a date-ascending-ordered result set, similar to the example below:

```
SELECT Sum(numFlights) OVER (PARTITION BY aircraft ORDER BY calDate ROWS BETWEEN 2 PRECEDING AND
CURRENT ROW) AS threeDaysRunningTotal
FROM FlightRecord;
```

A compressed join- or hash join index is not possible when an order analytical function is calculated. Before the function is applied, the rows may need to be redistributed to AMP's based on the order analytical function partition expression. From a performance point of view, an ordered analytical function may lead to suboptimal execution plans, AMP skewing, and out of spool errors.

The optimizer chooses between two strategies for the redistribution of rows. When the number of different partition expression values is high compared with the number of the available AMP's, the optimizer calculates the ROWHASH over the columns comprising the partition expression. When the ratio is small, the optimizer calculates the ROWHASH over value buckets (ranges) over the columns of the partition expression and, additionally, considers the columns of the ORDER BY definition. This strategy aims to achieve a better balance between the AMP's and reduce the probability of a skewed distribution. The phrase "*redistributed by value to all AMPs*" in the EXPLAIN output indicates using the second strategy.

When suboptimal performance is identified for a query involving an ordered analytical function, SQL rewrite may aid the optimizer switch from the first to the second strategy and achieve a less skewed distribution of rows. This is achieved by enriching the partition expression with order columns that do not change the query semantics. In this case, the optimizer may consider the additional ORDER BY columns to improve the row distribution transparently and, thus, improve performance.

```
-- Query before rewrite (heavily skewed due to data demographics, I/O spreads between 102 and
237)
SELECT SUM(amount) OVER (PARTITION BY customerId ORDER BY dueDate ASC) AS totSum
FROM Ranges;
-- Query after rewrite (same semantics but the skew is reduced, I/O spreads now between 191 and
231)
SELECT SUM(amount) OVER (PARTITION BY customerId ORDER BY dueDate ASC, aGoodColumn ASC) AS
totSum
FROM Ranges;
```

DDL statements

The DWH table definitions are typically stable over time; new business requirements primarily drive structural enhancements (e.g., adding new columns or changing column data types) and performance tuning activities (e.g., adding new partitions). Albeit rarely

needed, DDL statements can cause severe performance issues, especially in those cases where massively-populated tables must be changed.

A common pattern in the ETL and ELT processes, especially when third-party applications are used, is to DROP the stage table, then CREATE it, and finally load the data in it. As discussed at the beginning of this chapter (cf. section "*Request lifecycle*"), this approach negates any tunning performed for the `DBC.AccessRights` table. It also increases the contention for locks on data dictionary tables. Starting with Teradata version 15.10, the `DBC.AccessRights` table is now partitioned by database (databaseId) and table (tvmId). Thus, table-level lock is no longer necessary; a partition- or even rowkey-level lock suffices, and the contention is minimized.

In our experience, structural changes in standardized stage tables rarely occur, and even then, only in a coordinated fashion with the involved systems. We consider a good practice to replace any DROP/CREATE statements in the ETL and ELT processes with a DELETE statement to start staging on an empty table.

There are different strategies for realizing a change in the table structure, including ALTER and INSERT-SELECT approaches. In the case of stage tables, if a dynamic (i.e., DWH-driven) set of columns must be added and filled as part of the staging process (e.g., columns for surrogate keys), the process should be set up so that the emptied table structure is first enhanced with the dynamic set of columns (ALTER), and filled with data. This approach reduces the processing time for the ALTER statement and the space requirements to perform the change of an otherwise populated table.

Typically, an ALTER TABLE statement does not require spool space but operates on a Cylinder Index level. Still, there are specific cases (e.g., changing a row-partitioned table without a NO RANGE partition) that spool space is used by an ALTER TABLE statement. We consider a good practice to always check the EXPLAIN output of an ALTER TABLE statement before executing it for the potential use of spool space and indications of skew. An ALTER TABLE can cause significant performance issues and cannot be stopped once it starts to execute.

Table cloning is necessary on multiple occasions, including table structure change scenarios. A typical table cloning approach involves three steps: retrieve the original table DDL statement, create a copy of the table using this DDL, run an INSERT-SELECT into the new table. A more efficient means to achieve the same functionality is using a statement similar to the example below:

```
CREATE TABLE tblClone AS tblOrig WITH [NO] DATA [AND STATISTICS]
```

This approach allows cloning the table and its data in one statement. From a performance point of view, the single-statement approach allows copying the collect statistics definitions to the new one. Starting with Teradata version 13, a two-statement approach, similar to the

example below, allows copying the collected statistics. This eliminates the need to perform a resource-intensive recollection step.

```
-- Clone the table contents and its statistics (version 13 onward)
CREATE TABLE tblClone AS tblOrig WITH DATA AND STATISTICS; -- statistics defined but empty
COLLECT STATS ON tblClone FROM tblOrig; -- statistics for tblClone are now avalable
```

The CREATE TABLE approach can be utilized to emulate column attribute changes via an ALTER TABLE statement, similar to the example below. In this case, the primary index of the new table must be explicitly defined. Otherwise, Teradata will transparently use as primary index the first column of the SELECT statement satisfying the primary index definition criteria.

```
-- Clone a table and change the data type of col1 to INTEGER, col1 becomes UPI
CREATE TABLE tblClone AS (
    SELECT Cast(col1 AS INTEGER), col2, col3, ..., colN
    FROM tblOrig
) WITH DATA;

-- Create a new table and add a NOT NULL constraint on col1, col1 becomes UPI
CREATE TABLE tblClone (col1 NOT NULL) AS (
    SELECT col1
    FROM tblOrig
) WITH DATA;
```

Table updates are typically performed using UPDATE statements. A better alternative is to use MERGE INTO statements. The MERGE INTO approach requires no spool space, consumes fewer CPU seconds, and performs less disk I/O operations. The reason for these improved performance characteristics is that MERGE INTO operates on the data block level rather than a single row. The performance gain of MERGE INTO over UPDATE becomes greater when secondary indices must be maintained during the updates; again, the operations are performed for all involved data block rows at once, without additional I/O operations.

We consider a good approach also to replace INSERT-SELECT with MERGE INTO statements, whenever possible. An example case is when a table structure is cloned and then filled using a non-trivial SELECT statement.

Spool space

Typically, DML statements consume permanent space, and SELECT statements consume spool space. By and large, when storage space becomes a system constraint, it is better to transiently run out of spool space rather than out of permanent space. The reason is that a DML statement will need to perform a rollback and consume more resources while SELECT statements rarely, if ever, need to be rolled back. We consider a good practice to always define a dedicated *spool reserve database* and only use it in an emergency. Other measures to be explored on a case-by-case basis include table compression, disabling fallback protection, and deleting indices. These measures must be carefully assessed for both the

associated processing costs and the long-term impact on the DWH operations. In our experiences, these measures should be taken only in transitional periods until a system upgrade addresses the root cause of the problem.

NoPI tables exhibit even distribution when used by bulk loading tools. However, transactional INSERT statements and joins with PI tables may result in out-of-spool errors. Teradata does not redistribute locally-spooled rows in these cases and, thus, NoPI tables can become skewed. Even distribution can be forced in such cases by rewriting the SQL statements and include the phrase "HASH BY RANDOM", similar to the example below.

```
CREATE tblClone AS (SELECT * FROM tblOrig) WITH DATA NO PRIMARY INDEX; -- tblClone is a NoPI
INSERT INTO tblClone
SELECT *
FROM tblSecondOrig -- imbalanced primary index causing the out-of-spool error
HASH BY RANDOM; -- force redistribution on a data-block level, use RANDOM((1,1000) for row-level
```

Transient journal

The transient journal must be maintained for every DML statement and consumes additional resources. The space occupied by the transient journal can be checked using a statement similar to the example below:

```
SELECT Sum(CurrentPerm)/(1024**3) AS currentPermGB
FROM dbc.allspacev
WHERE databasename = 'dbc' AND tablename = 'transientjournal'
```

There are cases where the use of a transient journal can be reduced or eliminated without sacrificing the integrity of the data. If all rows of a PPI table partition must be deleted (e.g., a set of old records according to the EU GDPR), the transient journal can be bypassed entirely, provided that a NO RANGE partition is not defined.

Figure 117 depicts an example case of the same ALTER TABLE applied to two similar PPI tables with the same count of rows; their only difference is that the first one had a NO RANGE partition defined. Changing the table structure of this one required an order of magnitude more I/O operations and CPU seconds. Also, it was necessary to use a large portion of the spool space. In contrast, the one lacking a NO RANGE partition used the absolute minimal resources necessary. We consider a good practice to define NO RANGE partitions only when needed and not as a failsafe option.

```
SELECT QUERYBAND,TOTALIOCOUNT,AMPCPUTIME,SPOOLUSAGE FROM DBC.DBQLOGTBL
WHERE QUERYBAND LIKE '%NO_RANGE%'
AND STATEMENTTYPE = 'Alter Table';
```

QueryBand	TotalIOCount	AMPCPUTime	SpoolUsage
=S> NO_RANGE=Y;	149 140,00	74,76	831 893 504
=S> NO_RANGE=N;	280,00	0,06	0

Figure 117 Performance of ALTER TABLE for a PPI table with and without a NO RANGE partition

The transient journal use can be bypassed in the case of populating an otherwise empty table via multiple statements. An example case is a newly-created table (e.g., a cloned one), which is then filled by combining INSERT-SELECT statements. By utilizing the BTEQ multi-statement syntax, all statements are wrapped in a single transaction. Then, the as-is status of the table (before the transaction) is an empty table, and the to-be status (after the transaction) is the table filled by all the statements. In contrast, when multiple transactions are involved, the transient journal is bypassed only for the first of them. Thus, by changing a semi-colon position, like in the example below, significant resource savings are achieved, as depicted in Figure 118.

```
-- Start from an empty table and fill with a multi-statement from three source tables
DELETE FROM tblClone;
INSERT INTO tblClone SELECT col1, col2, col3 FROM tblSource1
;INSERT INTO tblClone SELECT col1, col2, col3 FROM tblSource2
;INSERT INTO tblClone SELECT col1, col2, col3 FROM tblSource3
;
```

QueryBand	Sum(TotalIOCount)	Sum(AMPCPUTime)
=S> OPT=0;	10 179,00	5,74
=S> OPT=1;	6 100,00	2,19

Figure 118 I/O operations and CPU seconds for three- (OPT=0) and a multi-statement (OPT=1)

Case study: Tactical workloads

The primary role of a DWH, as discussed in the previous chapters, is to serve strategic workloads, and the performance tuning efforts focus on this. In some cases, the DWH must serve *tactical workloads* too. In essence, tactical workloads complement OLTP (online transaction processing), while the latter is operational and predominantly involves write operations. Tactical workloads involve read operations to support decision making. The main requirement for a DWH tactical query is the fast response (a couple of seconds at most) to requests targeted to a small volume of data, down to a single row result.

The optimization goal of tactical query performance tuning is to minimize the number of I/O operations. This involves reducing the number of execution steps and involved AMP's and elimination, if possible, of row copies and redistribution. In an optimal case, a tactical decision support execution plan localizes the work in only one AMP and requires just one data block transfer from disk to memory, as depicted in Figure 119.

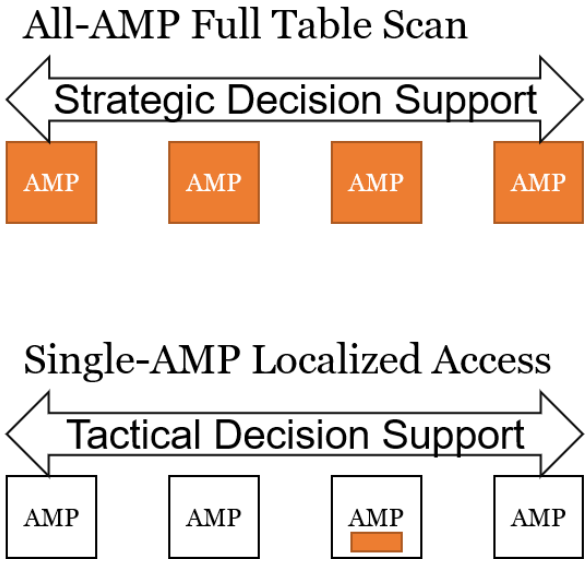

Figure 119 Strategic vs. Tactical Decision Support

Teradata features

The Teradata Priority Scheduler (part of TIWM and TASM, discussed in Chapter 2) provides a tactical tier. Teradata reserves a portion of AWT's specifically for tactical workloads. When a request is assigned to the tactical tier, it executes in these dedicated AWT's with high priority to ensure fast response time. However, this is not sufficient for a performing tactical workload. If the execution plan proves inappropriate, Teradata moves the request to a lower priority tier to avoid misuse of critical resources. Thus, it is essential to support the optimizer in devising a proper execution plan to fulfill the tactical workload requirements under these constraints.

The query run time will be the same for each execution, as long as the execution plan remains unchanged; the Teradata workload management identifies it as "tactical" and schedules it in the tactical tier priority. Only nested and merge joins can qualify for a tactical workload, as they are the only ones that need to involve a single or a few AMP's. All other join strategies involve all AMP's. The nested join is the optimal join method and, as such, should be the target for tactical workload tuning.

The advantage of single-AMP operations is that the parsing engine embeds the ROWHASH-level locking inside the retrieve step. Thus, a separate locking step is not needed. Furthermore, ROWHASH-level locks allow for better concurrency, as more queries can access the locked table at the same time and result in better performance.

Single-AMP operations require a primary index data access path. The most useful data access paths for single-AMP queries are UPI, NUPI, and single-table join with a proper selection of its primary index, i.e., the join must be performed over a common primary

index. The qualified rows must be based on a filter on the primary index columns, as in the example depicted in Figure 120.

```
EXPLAIN
SELECT *
FROM
    Sales t01
INNER JOIN
    SalesDetails t02
ON
    t01.SalesId = t02.SalesId -> Join on primary index of both tables
WHERE
    t01.SalesId = 100; -> Restriction on the primary index column


First, we do a single-AMP JOIN step from DWHPRO.t01 by way of the primary index
"Sales.t01.SalesId = 100" with no residual conditions,
which is joined to Sales.SalesDetails by way of the primary index "Sales.t02.SalesId = 100"
with a residual condition of ("Sales.SalesDetails.SalesId = 100").
Sales.Sales and Sales.SalesDetails are joined using a merge join,
with a join condition of ("Sales.Sales.SalesId = Sales.SalesDetails.SalesId").
The result goes into Spool 1 (one-amp), which is built locally on that AMP.
The size of Spool 1 is estimated with high confidence to be 151 rows.
```

Figure 120 Tactical query example – single-AMP merge join

A USI data access path may also serve the needs of a tactical query, as it is a two-AMP data access path with ROWHASH-level locking. On the other hand, a NUSI is always an all-AMP data access path requiring table-level read locks. As such, a NUSI is not a good candidate in tactical queries to achieve single-AMP access. However, its use is not prohibited; instead, it comes with a performance penalty and impacts system concurrency.

A join of a UPI table with a NUPI table can be achieved with a single-AMP merge join, while a UPI- with a USI tables needs a three-AMP nested join. A USI table join with either a UPI or a NUPI also needs a three-AMP nested join, while a USI/USI join needs a four-AMP nested join.

One more Teradata feature to support tactical workloads are the group-AMP operations. When used, such operations can help the optimizer downgrade the needed locks from table-level to ROWHASH-one and, thus, reduce resource usage and improve concurrency. The use of group-AMP operations is version-specific. As an empirical rule, less than half of the AMP's may be involved in a step execution for group-AMP operations to be used. The probability of use increases when statistics are collected for the join columns. By and large, a product join anywhere in the execution plan prohibits any use of group-AMP operations.

Hash joins prohibit the use of few-AMP or group-AMP operations; only nested and merge joins are allowed. Should an analysis indicated that the latter is necessary for improved performance or for experimentation purposes, a statement like the one in the example below can be used to disable hash joins for a session (this statement is not recommended for general use and might not be available in the future):

```
DIAGNOSTIC NOHASHJOIN FOR SESSION;
```

Starting with version 16.10, Teradata provides *sparse maps*. This feature allows placing all rows of small tables into a single AMP or few AMP's (one AMP per node) for most efficient access. One can define additional sparse maps, if necessary, and use them for an unlimited number of small tables. The advantage of sparse maps is that they reduce all-AMP accesses to a single- or few-AMP ones. Additional resource savings are achieved by having only one or a few AMP's maintaining the data blocks to store the table rows. Single-AMP and tailor-made sparse maps are ideal for supporting tactical queries and storing all table rows in a selected set of AMP's by design.

Design principles

The main challenge from a performance tuning perspective regarding tactical queries is to decide which design choices may result in a single-, few-, or group-AMP steps for an execution plan. Matching primary index definitions are an obvious choice. Should a query does not fully utilize the primary index columns, it should be rewritten, if possible, or define an appropriate covering join index to allow for nested joins.

Single-table join indices can be used in tactical queries for multiple needs. They may offer an improved primary index for a direct access path. They may be designed as *hashed NUSI*, allowing a non-unique index without the disadvantages of a NUSI, i.e., table-level lock and all-AMP access. A hashed NUSI can be defined to join back to the base table using ROWID pointers, primary index columns of the base table, or USI columns to retrieve the base table ROWID.

When row-partitioned tables are involved, all partitioning columns should be part of the primary index to avoid costly partition probing. When this is not feasible, the number of partitions should be kept to a minimum. This can be achieved by defining a USI or a single-table hashed NUSI on the primary index columns to allow direct row access. A sparse join index should be defined for single-table join indices to reduce I/O operations.

Lock granularity is an essential consideration for tactical queries. High concurrency is achieved when the execution plan allows for ROWHASH-level locks. When this is not feasible, access locks on underlying tables can be explored to increase concurrency if data consistency is not a hard requirement. ROWHASH locks should be applied to the join index object to the extent possible, avoiding locks on the base tables to improve concurrency further.

Multi-statement requests are another design choice for tactical queries, as they can reduce processing overheads. If necessary, each statement of the request can be prepended with a proper access lock modifier to improve concurrency and performance further.

Summary statistics are a preferred choice for tactical queries, as the latter are tunes for non-all-AMP access, have a short run time, and small resource consumption. The overhead for

detailed logging is not justified and may impact the query execution time in case the statistics become stale.

Last but not least, designing queries for caching should always be considered. When a query can be cached, the plan generation and optimization steps are skipped for subsequent executions. Only the security checks for access rights on needed database objects are performed, reducing the processing time. As mentioned earlier, a cached execution plan is reused only when the query text is identical at the byte level to the cached one. Macros with parameters (i.e., through the USING clause) are an excellent choice for tactical queries, which are executed repeatedly with a different parameter (e.g., querying customer information one customer at a time).

Chapter summary

In this chapter, we explored the multiple aspects of execution plan tuning. The Teradata optimizer uses a variety of techniques to decide the best possible execution plan. The cost calculation function considers factors such as the available data access paths, the cardinality of physical and spooled tables, join preparation, and join methods. The optimizer is sophisticated enough to transparently rewrite SQL statements to reduce the number of rows processed at each step and overall. However, human intervention is also needed to guide the optimizer in the right direction. A performance tuner must develop their tuning arsenal and sharpen their skills. A good understanding of the EXPLAIN output, ensuring up-to-date statistics, and performing SQL rewrites are typically sufficient to address the performance tuning challenges.

The main takeaways of this chapter are:

- The wall-clock execution time is neither proof of a performance problem nor a performance improvement.
- Skewed data are typically the root cause of performance problems in a mature DWH.
- Real-time monitoring tools help identify deviations between the estimates of the optimizer and the actual workload of each execution step.
- The final confirmation for a successful change only comes from the target (production) system environment.
- When tuning, perform one change at a time, keep the EXPLAIN output before and after the change, compare the two outputs, and document all changes performed. Repeat as necessary.
- The correct use of available index and row partitioning definitions is the key to a data model that stands the test of time without significant redesign.
- Statistics collections must always keep up with evolving data demographics for optimal performance.
- The use of spool space must be reduced as much as possible and freed as soon as possible.

- Simple SQL rewriting is typically, in our experience, sufficient for performance tuning.
- SELECT * is a bad practice and hampers performance in too many different ways.

Chapter 7: Beyond query tuning

"Any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization's communication structure." Conway's law, named after Melvin E. Conway

The final chapter of the book departs from query tuning towards business change and human bias. By and large, these are the hardest to tune in a DWH environment and subjective by nature. For this, we switch to a personal tone in these concluding sections, drawing lessons learned and stating opinions from our experience.

Operations and data model fixes

Performance tuning on a DWH in operation may evolve into fixing a wrong data model. Strong management and stakeholder support are always necessary, as a change in the data model impacts all DWH customers across the whole enterprise. A data model fix is like time traveling. The performance tuner must return to the DWH design phase, study the business requirements, and question how the requirements were transformed into the defined data model. Challenging the design decisions is neither an easy task nor always feasible, as *tribal knowledge* might have been lost in the meantime.

In this journey in time, the performance tuner may be confronted with uncooperative behavior. People tend to defend their petty areas of expertise and approach questions as a personal attack. A wrong data modeling may also be the outcome of lack of budget, wrong assignment of people to roles, or project delays that suffocated the time planned for proper analysis.

In our experience, a data model redesigned should be considered only after validating that the redesign will address the performance issues. A cost-efficient prototype is always necessary to demonstrate the improvement tangibly and measurably. Only then, stakeholder commitment should be requested to proceed with the data model change; else, the dreaded response can be expected: "*would be nice, but we have no budget for it.*"

DWH change project methodologies

The established project methodology for implementing changes in a DWH is to follow the *waterfall model* comprising five sequential phases: requirements, design, implementation, verification (i.e., testing), and maintenance (i.e., deployment, operation, and end of life support). In our experience, this model is outdated, even for DWH environments. Research findings also support this: more than 70% of the DWH project implementations that used the waterfall model failed. Some of the reasons behind the failures are:

- The requirements must be predicted, often with a horizon of many years in the future. The waterfall model assumes that the requirements are perfect and complete.

This is far from the truth for any non-trivial DWH project. The same holds for the translation between phases; the waterfall model assumes it is a perfect one.

- Due to time constraints, problems in the design phase are typically passed to the implementation phase; then, developers lacking the business knowledge are forced to invent workarounds due to their deadlines.
- Misunderstandings of the requirements, either in the design or the implementation, are only detected during the verification; then, it is too late for corrective actions. More workarounds are invented to cope with pressing deadlines. The same equally holds on the reserve directions; business users may only realize during the verification that their understanding of the system is incomplete and, hence, the requirements were based on wrong assumptions.

From a performance tuning point of view, the major drawback of the approach with phases enforced by the waterfall model is that the final responsibility is moved to the last link of the chain, which, in our experience, is the developers. However, the later the problem is discovered, the more expensive it becomes to fix it, as the costs increase exponentially.

The design phase is the most critical one in a waterfall DWH project. Several things may go wrong in this phase. The following list includes the most common ones to our experience:

- *User requirements are accepted uncritically*. The task of an analyst is to analyze the requirements together with the business users and guide them in the right direction. When this does not happen, it is discovered during verification that the implementation was done as specified but what was specified was not what the business users actually wanted. In our experience, the excuse by an analyst, "*but business wanted it like this*" is simply unacceptable. It is the responsibility of the analyst to figure out the actual user requirements, knowing the DWH environment and the waterfall model approach.
- *Combining functional and non-functional requirements*. Only functional requirements must be formulated in collaboration with business users. Non-functional requirements (e.g., the details of the DWH PDM) are to be discussed only in those very exceptional cases that may significantly impact business users. A requirement like "*the report must be fast*" is vague and unacceptable. In contrast, a functional requirement "*the report must be ready until 08:00 every working day of the week*" is acceptable, and the project implementation must fulfill it.
- *Considering the design phase in the requirement phase*. It is acceptable to consider some design constraints in the requirement phase already. However, it is unacceptable that design considerations dominate the requirements, as they unnecessarily constraint the landscape of possible solutions.
- *Claim a waterfall model and start different phases in parallel.* Short timelines are often used as an excuse to initiate the design or even the implementation phase from day one of a DWH project. In our opinion, this is neither waterfall nor agile. Instead, it is a Wild West style not prescribed in any project management methodology for

complex projects. In our experience, such approaches lead to failures in scope, time, budgets, or all of them and responsibility ping-pong games.
- *Passing responsibility for design decisions to business users*. The latter can state what they want from a DWH project but not how to do it.
- *Isolating communication*. Placing a communication firewall between business users and the developers introduces an additional layer of translation. This may be confusing, especially when the communication intermediary is an expert on none of the domains. In our experience, mediation and facilitation by intermediaries is more than welcomed but firewalling should be unacceptable.

In the late 2010's, the *agile methodology* became popular for DWH projects hoping that the agile mindset would improve the success rate. In our opinion, agile is not about certification and being religious on specific frameworks. Instead, it means dividing a DWH project into small tasks defined and confirmed by business users in short development cycles. When the tasks are implemented and accepted, they are progressively deployed to production systems. In theory, agile projects exhibit a lower failure rate. In practice, empirical evidence from time-tested DWH implementation is lacking. While agile may reduce development times in the short term, it remains to be seen their long-term impact on the DWH design and operation. It could be the case that the close interaction of business and IT improves common understanding and delivered quality. But it could also be the pressure to deliver short-term results in losing the "big picture" and discourage proper design decisions with a long-term impact. In our opinion, an automated, process-oriented approach, such as *DataOps*, based on continuous integration and deployment principles, has a better potential for DWH environments, bridging the DWH providers and consumers, removing *red tape*, and allowing faster feedback loops.

How DWH projects fail

Over the many years of working in DWH environments, we have identified the "*Seven Sins of DWH Projects*", i.e., seven common patterns of significant mistakes made over and over again when designing and implementing a DWH:

1. *Not knowing or losing sight of the strategic business objectives*. Most DWH projects start with much enthusiasm but fail because their outcome does not meet the business objectives. The best technical design and highly-tuned DWH are still useless if the DWH customer is not satisfied.
2. *Not setting up a measure to evaluate success*. Knowing the business objective is a prerequisite but not a guarantee for success by itself. Every clear business objective must be measurable. When it cannot be quantified, then the associated costs cannot be justified; the DWH and its people are then treated as an expense rather than an enterprise asset.
3. *Overspecialization*. A Teradata DWH operation is more like a mass-production assembly line. Over-specialization becomes then a problem. Industry sectors with decades of history are well-aware of the adverse effects of extreme labor division,

i.e., “Taylorism”. The Kaizen idea that was popular in the 1980’s and 1990’s repeats in the DWH environment. When each project or team member contributes a very narrow range of skills, lacks understanding of the overall DWH operation, and is discouraged from growing, the DWH project is bound to fail. It is cost-effective, at least at the beginning of the project, to create more specialized roles. In our experience, this never pays off in the end. Instead, it shifts the associated costs towards the end of the project, where the time constraints cause additional stress and pressure. The costs are then shifted and increased, as experienced DWH professionals must be engaged to bring the project on track again.

4. *Aiming for a DWH but realizing an ODS*. It is tempting to replace the functionality of a DWH with one of an operational data store (ODS). An ODS avoids costly source data transformation to achieve a unified and harmonized representation for the whole enterprise like a real DWH does. Then, development times shrink, and costs are reduced since even entry-level developers can implement an ODS. Bypassing a sound data model, skipping a surrogate key generation, or adopting source-system naming conventions in the DWH is a recipe for failure. In some cases, the project might survive the delivery, but this just makes things worse: the technical debt created will be paid during the whole lifetime of the DWH.
5. *Elimination of essential roles in the DWH project setup*. DWH operations span beyond the short life of ad hoc projects. Excluding critical roles, such as the DWH Architect, the Data Modeler, or the Performance Tuner, may seem cost-effective when adopting a single-project point of view. Still, this choice is more expensive when assessed at the DWH operation level and series of projects involved. When combined with inexperienced developers and project managers lacking communication with business departments, project delays and failures are likely.
6. *Lacking in-depth knowledge of the target architecture and the involved tools*. A data warehouse is, at its core, a highly-technical topic. On many occasions, the technical dimension is devalued and outsourced to low-cost locations. Solid education and training of involved people are required at multiple levels; in-depth knowledge and fluency in at least Teradata, ETL applications, and reporting tools are necessary. Too often, the interaction of these technical components does not get proper attention from a project. Even worse, performance, being a non-functional requirement, is almost always completely ignored at the beginning of the project. The outcome of such choices is a minefield of workarounds, hacks, and ad hoc fixes, causing never-ending issues with every new software release.
7. *Budgetary dishonesty and an atmosphere of dissimulation*. When a DWH project is set up with assumptions, amounts, or milestones that are illusional and outside the suggestions of subject-matter experts, the question is not whether the project will fail but by when. Sooner than later, people attend unpleasant meetings, cautiously stretching to add a layer of realism to the project plan. When the DWH owner and stakeholders suffer from the *naked Emperor syndrome* and do not value clarity and transparency (pun intended), the real problems multiply. Then, bureaucratic dissimulation tactics surface and absorb more and more of the available resources and energy. By this time, the very best project members exit and

seek better opportunities. This vicious cycle continues till, one way or another, the project ends.

Cloud-based MPP – a friend or foe?

The established approach of operating a DWH is using a hardware-based, on-premises infrastructure. Some organizations are reluctant to migrate to a cloud-based architecture or to consider alternative technologies. Other organizations are more willing to subscribe to this idea. Teradata adopts a neutral position on this, offering on-premise, hosted-, and public cloud solutions. From a performance point of view, some benefits of hardware-based infrastructure (e.g., hardware BYNET) must be traded off for increased scalability and flexibility of cloud-based infrastructure (e.g., hardware node procurement is replaced by simply activating additional EC 2 instances on an AWS cloud).

In the early 2010's, the established practice of SQL-based RDBMS was heavily questioned. It was fashionable to consider it a relic of the past, for systems unable to cope with the data volumes increase. Alternative approaches were proposed and popularized, collectively referred to as NoSQL. In the late 2010's, it became evident that SQL exhibits too many advantages to be ignored, while NoSQL technologies did not succeed in delivering their promise in the enterprise environments. The driving reasons behind this failure were the lack of joins to support analytical queries and of a standardized language across the different product offerings. Each vendor offered a different NoSQL variant to query their database, increasing the learning curve and making it harder to migrate across different technologies.

At the same time, the commoditization of the cloud computing resources gave birth to the so-called *SQL-based Massively Parallel Processing Data Warehouse (MPP DWH)*. Major cloud infrastructure providers added MPP DWH in their offerings: Google BigQuery, Amazon Redshift, and Microsoft Azure Data Warehouse, to name but a few.

Effectively, the decades-old, field-proven, shared-nothing MPP architecture of Teradata was rediscovered in the cloud environment. Thus, many Teradata concepts and terms exist under different names in other RDBMS. For example, the Teradata Primary Index and AMP are the Distribution Key and the SPU (Snippet Processing Unit) in IBM Netezza. Amazon Redshift Slices are computing resources similar to Teradata nodes and AMP's. The primary index of Teradata is the DISTKEY in Redshift. Even the DDL syntax is almost identical:

```
-- Teradata definition
CREATE TABLE customer (
    customerId INTEGER NOT NULL
    , customerName VARCHAR(100)
) PRIMARY INDEX (customerId)
-- Amazon Redshift definition
CREATE TABLE customer (
    customerId INTEGER NOT NULL
    , customerName VARCHAR(100)
) DISTKEY (customerId)
```

The significant difference between Teradata and cloud-based MPP is found in the storage system and the inferior support of Teradata for column partitioning when compared to systems engineered as column stores from the very beginning.

As the key concepts and design principles are the same, we expect a Teradata performance tuner to transfer their experience and expertise in these new environments successfully. Still, one needs to understand the similarities and, most importantly, the architectural and implementation differences that will drive the envisioned performance improvements.

Final thoughts on performance tuning

We are often questioned, when does a DWH need a performance tuner? We prefer to answer with an analogy. When do you need a fire brigade most? Yes, when the house is on fire. But when is the best time to establish the fire brigade? Not when the fire starts. A fire brigade must be established as soon as the town or the city is founded, even before the first houses are built. And when is the best time to perform a fire safety inspection? When the building is laid down, and construction progresses or after the building is ready and you have to tear down the wings and rebuild it?

Performance tuning specialists are a good fit for any DWH team at the DWH inception and any next stage. This is an insight after many years of experience rather than an attempt to lobby for performance tuners. When projects claim cost reasons to rule out performance tuning, they implicitly accept the considerable risk of failure and multiplied costs to introduce such roles late in the project implementation. As the DWH design must stand the test of time and affects too many parts of an enterprise, design and implementation decisions lock the DWH in a non-reversible status for its whole lifecycle. The role of the performance tuner is to anticipate the upcoming bottlenecks and work with the team members to remove or bypass them before the DWH operations hit them.

The strategic dimensions of performance tuning must be communicated and understood by all stakeholders early on and before debates over measurements and results surface. Performance tuning must serve the following strategic goals:

- A net improvement of the system resource usage is achieved.
- The key stakeholders of the DWH achieve their goals faster, more often, with less friction and less intervention.
- Changes and enhancements stand the test of time, even long after a performance tuner is no longer there.

Performance improvements must be quantifiable and demonstratable. Work on performance tuning is goal-oriented. As such, an improvement on a set of defined metrics is the only proof for claiming a success. A performance improvement cannot succeed when it merely transfers the problem to another part of the DWH.

Performance issues cannot be solved purely on a technical level. Inaccurate data models, unclear business specifications, and incomplete mappings are often the root cause of the performance issue. When the root cause is not addressed, workarounds are invented; these may work in the short-term and under specific conditions but increase the overall costs in the long-term; the more the workarounds are piled up, the harder the implementation of a correct solution at a later time. Thus, a performance tuner must serve several roles in a DWH: a business analyst, a data modeler, and a developer in one person.

Performance tuners must never neglect to sharpen their knowledge and skills as technical experts in performance tuning. We hope that this book helped you with this, and we are grateful that you made it to this point! An online supplement of the book is available at https://www.dwhpro.com/tuning-book. There, we provide additional resources (e.g., ready-to-use scripts to automate performance tuning tasks) and real-world performance tuning case studies that do not fit the format of a book. We invite you to join us and continue together with the journey in the world of Teradata performance tuning.

Stay tuned, and thank you!

Roland and Artemios, September 2020

Appendix: Execution plan glossary

The EXPLAIN output of a statement describes in great detail the execution plan devised by the Teradata optimizer. The following sections provide a list of standardized phrases contained in the EXPLAIN output, which, in our experience, are the most important to know for performance tuning. The list is organized in groups based on the different phases of request processing and execution steps involved for easy reference.

Retrieve steps

- *We do an all-AMPs RETRIEVE step from <database>.<table> by way of an all-rows scan into spool 1*
  - A full table scan is performed, and the rows are placed in an intermediate spool table.
- *By way of the primary index*
  - The rows are retrieved via the primary index access path (NUPI).
- *By way of the unique primary index*
  - The rows are retrieved via the unique primary index access path (UPI).
- *with a condition of (<condition>)*
  - A condition is applied during the retrieve step, creating a spool out of the qualifying rows.
- *By way of a traversal of index # n with a range constraint of (<constraint>) extracting row ids only into Spool* n
  - A value-ordered NUSI is traversed. The base table ROWID's are extracted for further processing (for example, in a join).
- *By way of index # n "<condition>" with no residual conditions into Spool n*
  - A NUSI is used to access base table rows via their ROWID's.
- *we do a two-AMP RETRIEVE step from <table> by way of unique index # n "col1 = 1"*
  - A USI is used to access the unique row (2-AMPs are always involved).
- *we do a BMSMS... (bit map set manipulation step)*
  - Two indices with weak selectivity are combined, as together they have high selectivity.

Join preparation steps

- *which is redistributed by the hash code of (<column>) to [few|few or all| all] AMPs*
  - The rows of a table or spool are redistributed based on the hash value of the mentioned column(s)
- *which is redistributed by hash code to all AMPs*
  - The rows of a table or spool are redistributed based on the hash value of all columns
- *which is duplicated on all AMPs*

    - All rows of a table or spool are copied to all AMPs. This is always the smaller table and used, for example, in product joins.
- *which is built locally on the AMPs*
    - All AMPs create in parallel a local spool file
- *fanned out into n hash join partitions, which is duplicated on all AMPs*
    - The probe table of a hash join is created and duplicated to all AMPs

Join method steps

- *We do an all-AMPs JOIN step*
    - All AMPs are used in the join step
- *single-AMP JOIN step by way of the unique primary index*
    - A row is selected from an AMP via primary index access, the hash calculated on the join column(s); the row is distributed to the AMP containing the other table's row and being joined.
- *Spool 1 and Spool 2 are joined using a hash join of n partitions, with a join condition of ("col1 = col1")*
    - A hash join is done via join column "col1". The Optimizer uses n hash join partitions
- *by way of a row hash match scan …are joined using a merge join*
    - A merge join is done. During the run time, the AMP's internally decide which variant, slow- or fast-path, to use; this piece of information is not included in the EXPLAIN output.
- *into Spool N (all_amps), which is duplicated on all AMPs*
    - a merge join on a PPI table is performed
- *a rowkey based*
    - A rowkey-based merge join is done, as there are equality constraints on all partitioning columns and the primary index
- *are joined using an in-memory dynamic hash join*
    - an in-memory dynamic hash join is performed
- *are joined using an in-memory hash join of n partitions*
    - an in-memory hash join is performed
- *are outer joined using a dynamic hash join*
    - a dynamic hash join is performed
- *are joined using a nested join, with a join condition of ("(1=1)*"
    - the nest join method is used
- *are joined using a product join*
    - the optimizer performs a product join
- A pair of *are right outer joined using a hash join of n partitions* and *are left outer joined using a single partition hash join (classic)*
    - a hybrid hash join is performed
- *enhanced by dynamic partition elimination*
    - indicates the potential use of the DPE feature

Sort steps

- *we do a SORT to order Spool n by the hash code of (<column>)*
  - A spool table is sorted by the ROWHASH of one or several columns; for example, a join preparation step for the merge joins binary search.
- *SORT to partition by rowkey*
  - The spool is sorted by ROWKEY to prepare for a rowkey-based merge join.
- *SORT to order Spool n by row hash and the sort key in spool field1 eliminating duplicate rows*
  - The rows of a spool are sorted by ROWHASH, duplicates are identified based on *field1* and removed. In the case of set operations (UNION, MINUS, INTERSECT, EXCEPT) or a DISTINCT over all columns, *field1* is the concatenation of all columns; still, *field1* can be as well a set of other columns used for removing duplicates.
- *SORT to order Spool n by the sort key in spool field1*
  - The rows of a spool are sorted directly by the value of *field1*. *Field1* can be any combination of columns that are defined in the ORDER BY clause.

Aggregation steps

- *Aggregate Intermediate Results are computed globally*
  - The aggregation takes place in several steps. Each AMP does a local pre-aggregation; rows are distributed by hash value to their target AMPs; rows are sorted, removing duplicates; the final (global) aggregation is done.
- *Aggregate Intermediate Results are computed locally*
  - Aggregate Intermediate Results are computed locally. The aggregation is done locally on each AMP, and as the aggregation columns match the primary index columns, no redistribution is needed.
- *We do an all-AMPs partial SUM step*
  - An explicit partial GROUP BY optimization is done for a sum step.
- *we do a SORT/GROUP*
  - An implicit partial GROUP BY is performed before a join takes place.

Partition elimination

- *n partitions of*
  - The optimizer applies partition elimination, and only *n* internal partitions are read (n >= 2).
- *a single partition of*
  - The optimizer reads exactly one internal partition of a row-partitioned table.
- *all partitions of*
  - The optimizer reads all internal partitions of a row-partitioned table, doing primary index access of a single AMP.
- *enhanced by dynamic partition elimination*

- During a join, the join columns are used to apply partition elimination.

Step execution

- *(Last Use)*
  - The last time the spool file is used in the plan. It will be released after this step.
- *with no residual conditions*
  - All applicable conditions have been applied for row qualification.
- *with a residual condition*
  - When a WHERE condition cannot be resolved with index access, row qualification is done after rows have been retrieved.
- *eliminating duplicate rows*
  - This phrase appears when row duplicates from spool tables are removed (e.g., during a UNION operation).
- *by the hash code of (<table>.<column>)*
  - A spool is sorted by the ROWHASH of the mentioned column.
- *we execute the following steps in parallel*
  - Two or more steps of the execution plan are done in parallel, completely independent from each other.
- *(compressed columns allowed)*
  - The target spool can keep multi-value compressed columns compressed.
- *estimated size*
  - The estimated size in bytes for a spool file, based on statistics.
- *The estimated time for this step is NN seconds*
  - The estimated logical time units needed for the step (not wall-clock seconds).
- *The estimated time is NN seconds*
  - The estimated time for the whole request.
- *lock*
  - A lock will be placed on a database object used in the execution plan.
- *Spool N*
  - Identifies each spool used in a plan step operation.
- *we do a SMS (set manipulation step)*
  - Indicates a set operator taking place (union, minus, intersect, except).
- *(all_amps)*
  - The plan step involves all AMPs.
- *(group_amps)*
  - The plan step involves a group AMP operation since the optimizer estimated that less than 50% of the AMP's are needed and built a dynamic AMP group.