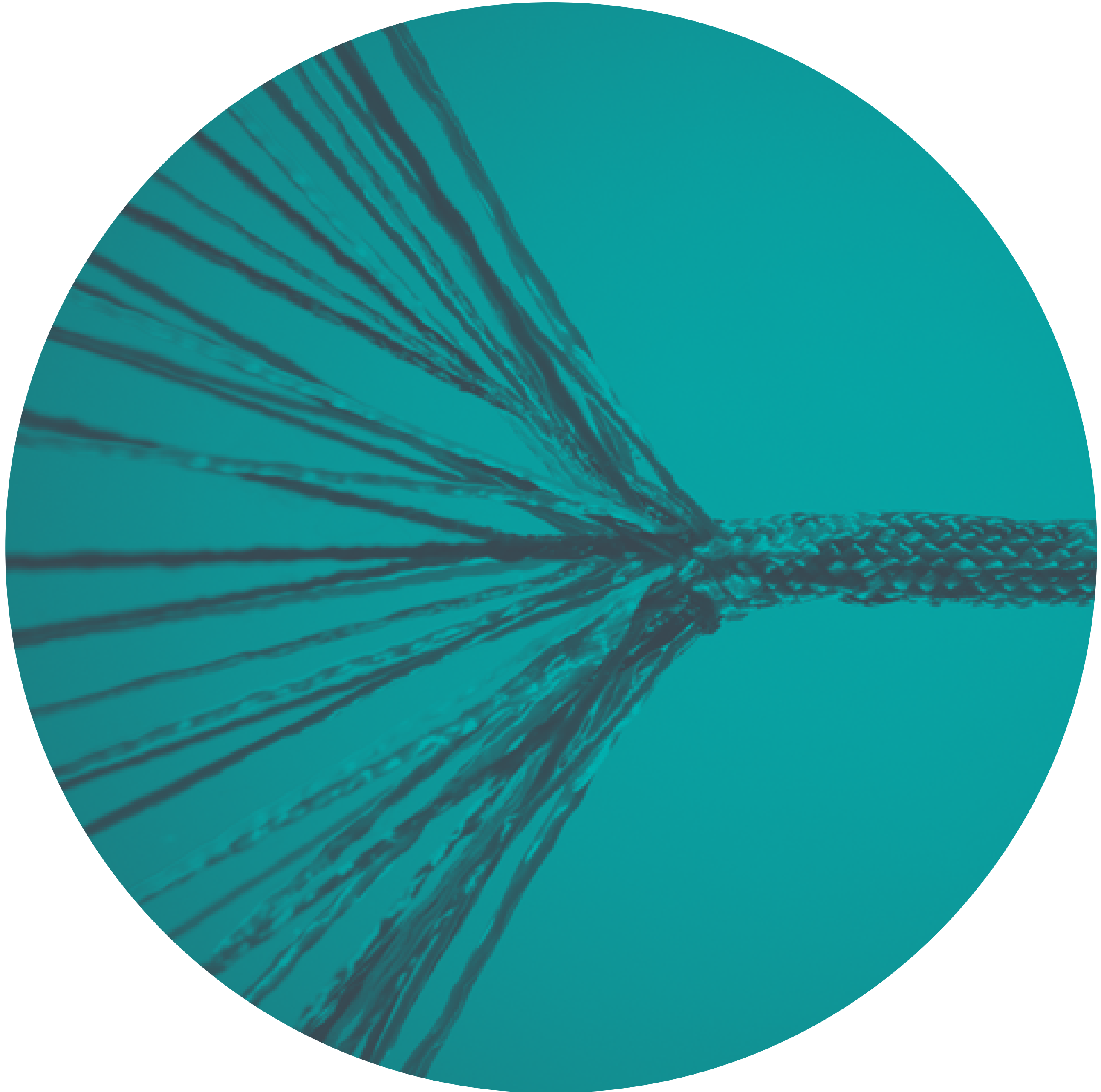
WHITE PAPER

Born to Be Parallel, and Beyond



By Carrie Ballinger, Sr. Technologist, Technology & Innovation Office

03.19 / EB3053 / TERADATA VANTAGE / WHITE PAPER

teradata.

Table of Contents

2 Introduction

3 Multidimensional Parallel Capabilities

6 Parallel-Aware Optimizer

8 The Bynet's Considerable Contribution

11 A Flexible, Fast Way to Find and Store Data

17 Workflow Self-Regulation

20 Workload Management

22 Conclusion

23 Endnotes

Introduction

Teradata's Enduring Performance Advantage

To survive and thrive in today's competitive business environment, organizations must be capable of managing all of the data, all of the time—to deliver analytics that matter. This happens when you have a unified, integrated analytics environment that delivers the best analytic functions and engines at scale.

Providing the most powerful analytics platform that supports advanced analytics and Pervasive Data Intelligence—so you can analyze anything, deploy anywhere, and deliver actionable answers fast—the Teradata Database has become the NewSQL Engine component in Teradata Vantage.

The basic building blocks that were initially put in place still remain, and they continue to elevate and extend the major advantages of the NewSQL Engine today. The surprisingly enduring performance advantage of the NewSQL Engine is a direct result of these early, somewhat unconventional design decisions made by a handful of imaginative architects.

This paper describes and illustrates some of these key fundamental components of the NewSQL Engine that are as critical to performance now as they were then, and upon which today's new features and capabilities rest.

Discussions of these specific areas are included in this paper:

- Multidimensional parallel capabilities
- A parallel-aware query optimizer
- The BYNET's considerable contribution
- A flexible and fast way to find and store data
- Internal self-regulation of the flow of work
- Managing the flow of work externally with Workload Management

The scope of this whitepaper is limited to important, foundational components of the database. It is not a comprehensive discussion of all the aspects of the Teradata NewSQL Engine.

teradata.

There have been several important capabilities introduced over the years that rest on top of that foundation, but that are not discussed in this paper. They include user-defined functions, table operators, data types like XML, JSON/BSON/UBJSON for semi-structured data, geospatial, columnar, temporal, Teradata Intelligent Memory, and in-memory optimizations.

Multidimensional Parallel Capabilities

Emerging in the late 1970's, Teradata Database was the first commercially available SQL-based "parallel processing" machine designed from the base up to support user business queries. Since then, parallel processing has become a necessity for any serious database offering, as demand for data analytics continues to drive even higher volumes, greater numbers of users, and more real-time performance.

With a design goal of eliminating single-threaded operations, the original architects of Teradata Database parallelized everything, from the entry of SQL statements to the smallest detail of their execution. The database's entire foundation was constructed around the idea of giving each component in the system many look-alike counterparts. Not knowing where the future bottlenecks might spring up, early developers weeded out all possible single points of control and effectively eliminated the conditions that can breed gridlock in a system.

Limitless interconnect pathways, and multiple optimizers, host channel connections, gateways, and units of parallelism are supported in Teradata, increasing flexibility and control over performance that is crucial to large-scale data analytics today.

The Teradata basic unit of parallelism is the AMP (Access Module Processor), a virtual processing unit that manages all database operations for its portion of a table's data. Many AMPs are typically configured on a given node. From 20 to 40 or more AMPs per node is common today.

Once configured, data loads, backups, index builds—in fact everything that happens in a Teradata system—is distributed across a pre-defined number of AMPs. The parallelism is predictable and understandable.

Each AMP acts like a microcosm of the database, supporting such things as data loading, reading, writing, journaling and recovery for all the data that it owns (see Figure 1). The parallel units also have knowledge of each other and work cooperatively together behind the scenes. This teamwork among parallel units is an unusual strength of the NewSQL Engine, driving higher performance with minimal overhead. When executing analytics on the Machine Learning Engine co-processors, each AMP in the SQL Engine participates in sending data-to-be-processed to the Machine Learning Engine, and then receiving a share of the results that comes back.

Types of Query Parallelism

While the AMP is the fundamental unit of parallelism, there are two additional parallel dimensions woven into the NewSQL Engine, specifically for query performance. These are referred to here as "within-a-step" parallelism, and "multi-step" parallelism. The following sections describe these three dimensions of parallelism:

Parallel Execution Across AMPs

Probably the most recognizable type of parallelism is parallel execution across multiple AMPs. It involves breaking the request into subdivisions, and working on each subdivision at the same time, with one single answer delivered. Parallel execution can incorporate all or part of the operations within a query, and can significantly reduce the response time of an SQL statement, particularly if the query reads and analyzes a large amount of data.

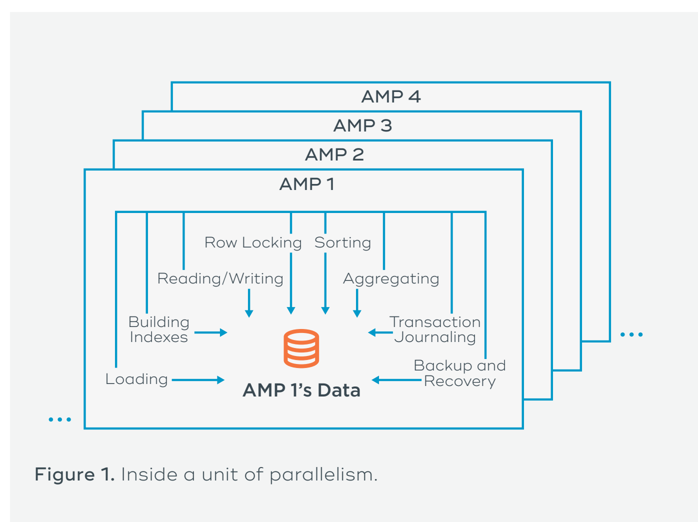


Figure 1. Inside a unit of parallelism.

teradata.

Parallel execution is usually enabled in Teradata by hash-partitioning the data across all the AMPs defined in the system. Once data is assigned to an AMP, the AMP provides all the database services on its allocation of data blocks. All relational operations such as table scans, index scans, projections, selections, joins, aggregations, and sorts execute in parallel across the AMPs simultaneously. Each operation is performed on an AMP's data independently of the data associated with the other AMPs.

Within-a-Step Parallelism

A second dimension of parallelism that will naturally unfold during query execution is an overlapping of selected database operations referred to here as within-a-step parallelism. The optimizer splits an SQL query into a small number of high level database operations called "steps" and dispatches these distinct steps for execution to the AMPs, one after another.

A step can be a small piece or a large chunk of work. It can be simple, such as "scan a table and return the result" or complex, such as "scan two tables and apply predicates to each, join the two tables, redistribute the join result on specified columns, sort the redistributed rows, and place the redistributed rows in an intermediate table."

Within each of these potentially large chunks of work that we call steps, multiple relational operations can be processed in parallel by pipelining. While a table scan is taking place, rows that are selected from the scan can be pipelined into a join process immediately.

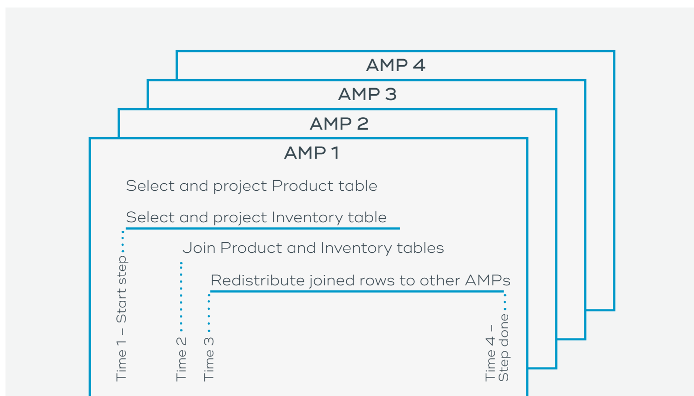


Figure 2. Pipelining of 4 operations within one query step.

Pipelining is the ability to begin one task before its predecessor task has completed and will take place whenever possible within each distinct step (see Figure 2).

This dynamic execution technique, in which a second operation jumps off of a first one to perform portions of the step in parallel, is key to increasing the basic query parallelism. The relational-operator mix of a step is carefully chosen by the Teradata optimizer to avoid stalls within the pipeline.

Multi-Step Parallelism

Multi-step parallelism is enabled by executing multiple "steps" of a query simultaneously, across all the participating units of parallelism in the system. One or more tasks are invoked for each step on each AMP to perform the actual database operation. Multiple steps for the same query can be executing at the same time to the extent that they are not dependent on results of previous steps.

Figure 3 is a representation of how all of these three types of parallelism might appear in a query's execution.

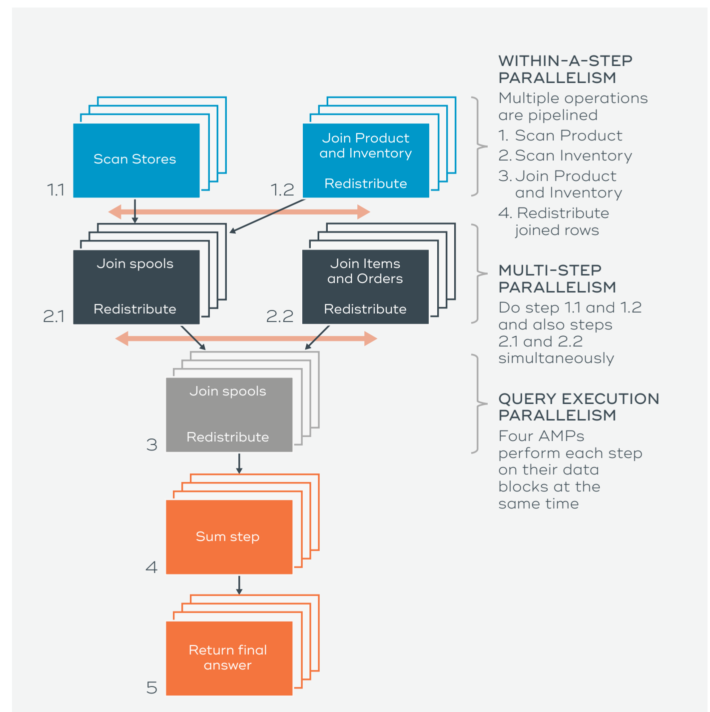


Figure 3. Multiple types of parallelism combined.

teradata.

The figure shows four AMPs supporting a single query's execution, and the query has been optimized into 7 steps. Step 1.2 and Step 2.2 each demonstrate within-a-step parallelism, where two different tables are scanned and joined together (three different operations are performed). The result of those three operations is pipelined into a sort and then a redistribution, all in one step. Steps

1.1 and 1.2 together (as well as 2.1 and 2.2 together) demonstrate multi-step parallelism, as two distinct steps are chosen to execute at the same time, within each AMP.

This multifaceted parallelism is not easy to choreograph unless it is planned for in the early stages of product evolution. An optimizer that generates three dimensions of parallelism for one query, such as described here, must be intimately familiar with all the parallel capabilities that are available and know how and when to use them. But most importantly, the NewSQL Engine applies these multiple dimensions of parallelism automatically, without user intervention or special setup.

Multi-Statement Requests

In addition to the three dimensions of parallelism shown in Figure 3, the NewSQL Engine offers an SQL extension called a Multi-Statement Request that allows several distinct SQL statements to be bundled together and sent to the optimizer as if they were one unit. The NewSQL Engine will attempt to execute these SQL statements in parallel, as long as there are no dependencies among the statements.

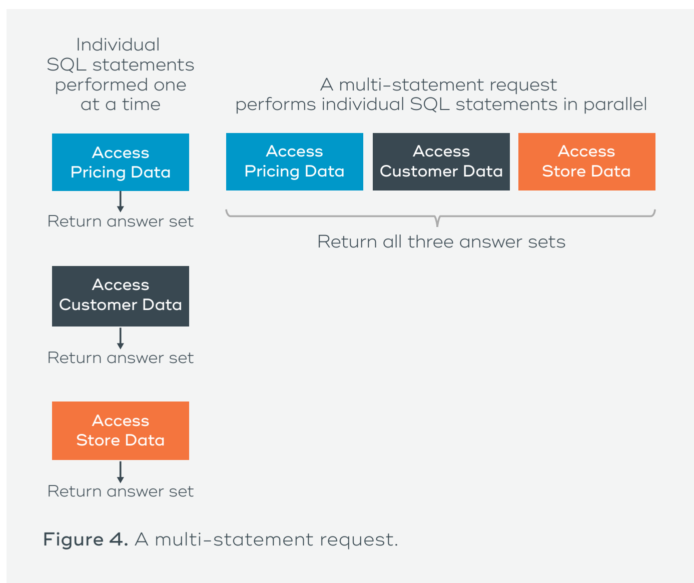


Figure 4. A multi-statement request.

When this feature is used, any sub-expressions that the different SQL statements have in common will be executed once, and the results shared among them. Known as common sub-expression elimination, this means that if six select statements were bundled together into a single request, and all contained the same sub-query, that sub-query would only be executed once. Even though these SQL statements are executed in an interdependent, over-lapping fashion, each query in a multi-statement request will return its own distinct answer set (see Figure 4).

Evolution

Many features have been added to the NewSQL Engine over the years that take advantage of the database's inherent parallelism. Things like the FastExport utility, which pulls large volumes of data out of the database across all AMPs in parallel, or ordered analytic functions, that perform complex windowing on top of the parallel foundation, are a few examples.

Teradata QueryGrid, a more recent feature, relies on the parallelism of the NewSQL Engine as well when accessing rows from a foreign server. The QueryGrid feature provides a single interface to combine data across different systems (including heterogeneous systems), minimizing the need for data duplication.

For example, with QueryGrid you can issue queries from Teradata that access, filter, and return rows from a Hadoop platform. You can then join data from Teradata tables to those rows brought in from Hadoop, if required, all in a single SQL statement.

The parallelism of the NewSQL Engine contributes to the performance of QueryGrid when a moderate or a large number of rows are being accessed from a foreign server. Multiple streams of data can be brought over from Hadoop in parallel, directly connecting to and coming to rest on different AMPs in the Teradata configuration. Each AMP receives and spools its subset of the Hadoop data.

All AMPs are working in parallel as though taking in and processing the data that just arrived from Hadoop was part of just another query step. Without the indigenous parallelism of the NewSQL Engine, QueryGrid could not offer the same level of efficiencies when importing data from a foreign server.

teradata.

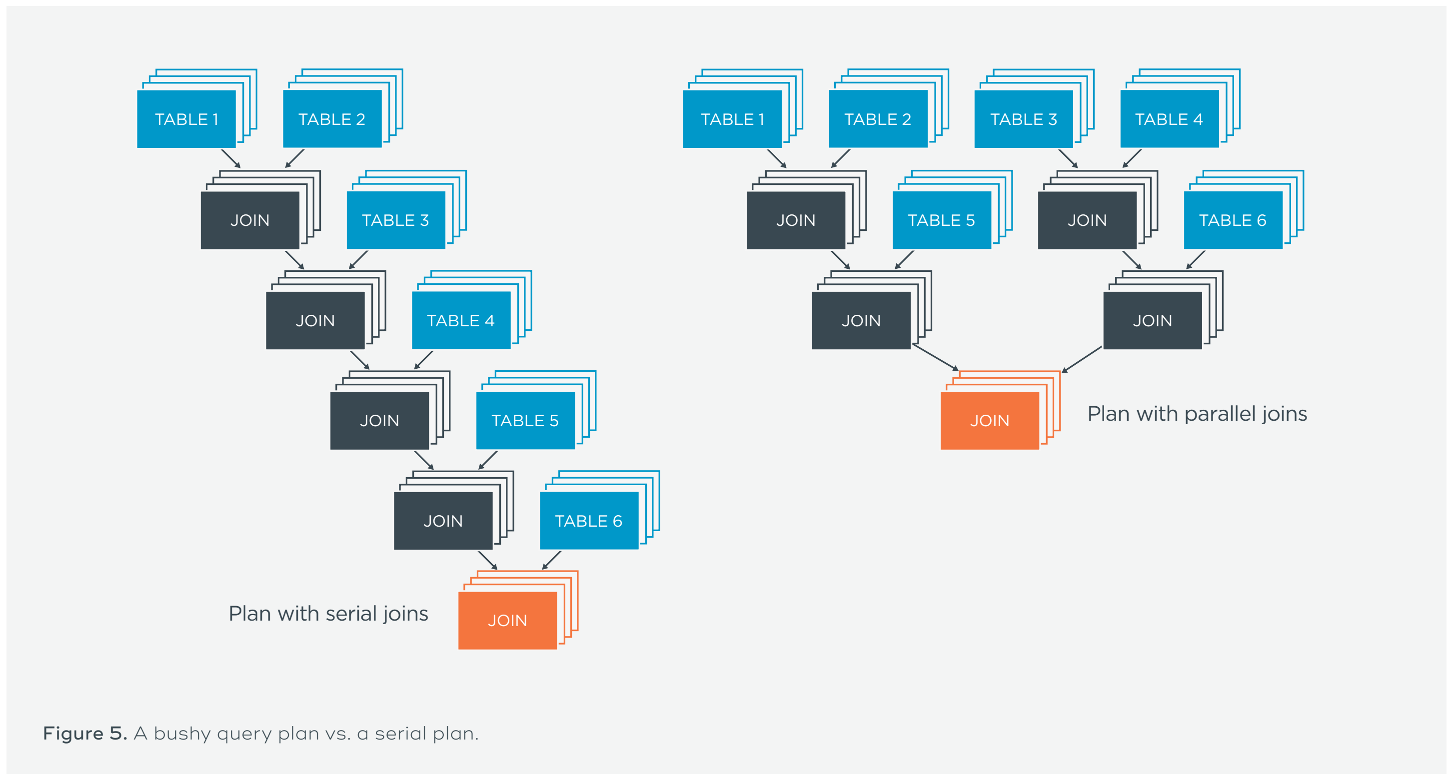


Figure 5. A bushy query plan vs. a serial plan.

Parallel-Aware Optimizer

Having an array of parallel techniques can turn into a disadvantage if they are not carefully applied around the needs of each particular request. Envisioning the power of these combined dimensions of parallelism, early architects of Teradata Database constructed a query optimizer that was fully in tune with these choices and had the smarts to know when and how to apply them.

Join Planning

When the optimizer begins the task of building a query plan, one of its goals is to maximize the throughput of this particular piece of work. Think of a query that has to access six tables to build its answer set. One of the jobs of the optimizer is to determine which tables to access and join first, and which tables to access and join later in the plan. It also has to consider what type of join to use to bring the rows of two tables together, and the method of accessing each table (indexed access or table scan, for example).

One unique capability built into the original Teradata optimizer is the ability to access and join multiple tables simultaneously, constructing a wide or "bushy" query plan, rather than a serial one-join-at-a-time plan. Those six tables discussed above could be joined in a strictly linear fashion: join table1 to table2, then join their result to table3, then join their result to table4, etc. as shown in Figure 5. This will spread the resource usage required by the query over a longer period of time, potentially impacting elapsed time.

The Teradata optimizer seeks out tables within the query that have logical relationships between them, such as Items and Orders in Step 2.2 of Figure 3. It also groups tables that can be accessed and joined independently from the other subsets of tables. Those are often candidates to execute within parallel steps. Figure 5 illustrates the differences when optimizing a six-table join between a plan that is restricted to linear joins, and one that has the option of performing some of the joins in parallel.

teradata.

Sizing up the Environment

But understanding the dimensions of parallelism by itself was found to be inadequate when it came to creating an optimizer suitable for complex queries on a parallel database. Early optimizer architects made sure that other "environmental" factors were considered during plan building as well.

The optimizer knows about the number of AMPs and the number of nodes in the current configuration. It considers the processing power of the hardware, and uses costing algorithms in devising estimated costs of potential plans. Putting all this information together, the optimizer comes up with a price in terms of resources expected to be used for each of several candidate query plans, then picks the least costly candidate. The lowest cost plan is the plan which will take the least system resources to execute. These final cost estimates are externalized in prose-like query "explain" text, for the user to read.

An important piece of information that the optimizer always looks for when building a plan is statistics: data demographics about the tables and columns that participate in a query. This statistical data, which is stored as histograms in the data dictionary, helps to determine what the best order of joins, and it helps the optimizer assess the size of the data set that results from joining two tables. This information is used to select the best method of implementing the joins.

Thinking in Parallel

The Teradata optimizer was born into a parallel world. Because it was built on top of a shared-nothing architecture, it has been forced to think with a completely parallel mind set.

For example, before it settles on a step that does a table duplication1 the optimizer evaluates the number of AMPs that the table will be copied to, estimates the number of rows that will be fanned out, and considers the total data load across the BYNET. Then it gives that an estimated cost and compares it to other alternatives that might involve less movement of data or a different join order. Its entire focus is to deliver a query plan that will execute a user-submitted query with the least possible effort.

Hiding Complexity

One thing customers have always liked about Teradata Database's optimizer is that it alleviates the user who submits the query from having to get involved in directing the query plan. Query optimization is completely automated, and other than collecting statistics, the user has no influence over this process. Query optimization happens behind the scenes and the user only need be concerned with what statistics to collect. There is complete freedom to submit very complex ad hoc analytic queries, or canned tactical dashboard queries, or quick single-row look-up queries, because the optimizer will adjust to whatever is thrown at it.

Evolution

Many years of feedback and experience with Teradata users has helped developers discover ways to enhance the optimizer capabilities to better meet real-world needs. During this evolution, existing components are often expanded or used in a new way, without the need to start all over or change the underlying foundation. Row-IDs, for example, were repurposed so they could support no primary index tables and were expanded from the original 8 bytes to 16 bytes in order to support new table partitioning opportunities.

There has been a continuous stream of new optimizer features over the last 30 years, all building on the original foundation. Some of the key ones include:

- More sophisticated statistics collection options, including sampling, thresholds for recollections, and optimizer-initiated statistics collection skipping.
- Extrapolation of statistical information at run-time when statistic collections are outdated.
- Join indexes (materialized views) synchronized with base tables whose access is managed by the optimizer.
- New types of joins, including a star join that brings together unrelated small tables first before joining to a fact table, and in-memory hash joins to take advantage of in-memory processing possible on today's large memory processors.
- Incremental Planning and Execution (IPE), where the optimizer builds a partial plan, executes that first fragment, then based on the output of the first fragment, builds and optimizes the second and subsequent fragments.

The BYNET's Considerable Contribution

The Teradata solution was designed as a shared-nothing architecture, the hardware as well as the software. AMPs and parsing engines2 (PE) operate in isolation from one another. Messages are the means of communication among these many moving parts and the interconnect is the glue that holds the pieces together and speeds along all of the parallel activities that are running on the NewSQL Engine (see Figure 6).

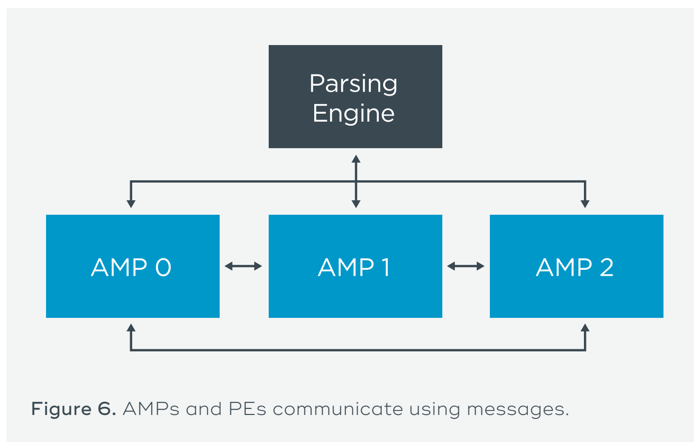


Figure 6. AMPs and PEs communicate using messages.

From the beginning, the interconnect was treated as something more than a delivery device for messages. Instead, the design of SQL Engine widely exploited the interconnect to increase performance and simplify user interaction with the database wherever possible.

Functionality provided by the original Teradata interconnect, known as the YNet, lived in the hardware and the hardware driver code on the individual nodes.

The YNet did much more than a standard interconnect usually does. The BYNET, in use today, inherited these same capabilities. Beyond just passing messages, the BYNET is a bundle of intelligence and low level functions that aid in efficient processing at practically each point in a query's life. It offers coordination as well as oversight and control to every optimized query step.

This section explores distinctive characteristics that were built into the YNet and that were passed on to the current BYNET.

These original YNet benefits were essential for unveiling the original Teradata Database, and have proven to be equally indispensable today:

- **Message Delivery:** Sends, optimizes, and guarantees message arrival
- **Multi-AMP Coordination:** Oversees step completion and error handling when multiple AMPs are working on the same query step
- **Final Answer Set Ordering:** Efficient, dynamic merge of final answer set across parallel units, bypassing expensive sort/merge routines
- **Isolation:** Insulates the database from configuration details, recognizes and adjusts to hardware failures
- **Resource Conservation:** Streamlines message traffic
  - Identifies and minimizes the number of AMPs involved in receiving a given message by setting up dynamic BYNET groups
  - Buffers up multiple small messages going to the same AMP or node, sending fewer large messages
- **Congestion Control:** Regulates high-volume messages to prevent overruns or bottlenecks
- The following sections provide more detail about a few of these specific benefits performed by the BYNET.

Messaging

A key role of the BYNET is to support communication between the PEs and AMPs and also from AMPs to other AMPs. This communication comes in many forms, some of it very straightforward.

- Sending a step from the dispatcher module on the PE to AMPs in order to initiate a query step
- Redistributing rows from one AMP to another in order to support different join geographies
- Sorting a final answer set from multiple AMPs

These simple message-passing requirements are performed using a low level messaging approach, bypassing more heavyweight protocols for communication. For example, making a costly TCP/IP connection between AMPs and PEs every time a message needs to be sent is never required.

teradata.

There is no connection setup or teardown cost whenever a process like row redistribution or table duplication needs to happen. And when rows are being redistributed or duplicated, outbound rows are never sent one-at-a-time. Like carpoolers, individual messages are bundled up so fewer messages ever need to be sent.

Even though message protocols are low-cost, the NewSQL Engine goes further by minimizing interconnect traffic. Same AMP, localized activity is encouraged wherever possible. AMP-based ownership of data keeps activities such as locking and some of the simple data processing local to the AMP. Hash partitioning that supports co-location of to-be-joined rows reduces data transporting prior to a join. All aggregations are ground down to the smallest possible set of sub-totals at the local (AMP) level first before being brought together globally via messaging. And even when BYNET activity is required, use of dynamic BYNET groups (originally called dynamic YNet groups) keeps the number of AMPs that must exchange messages down to the bare minimum.

Even More Performance Benefits

The NewSQL Engine always uses broadcasts across the BYNET in cases where all or most of the AMPs in the system require the same information, such as duplicating the rows of a table, or sending a dispatcher message for an all-AMP operation such as applying a table-level lock. But the cheaper point-to-point messaging option is considered when a smaller subset of AMPs is required.

The BYNET point-to-point communication is similar to a standard phone call over the public telephone network. These mono-cast circuits connect one sender node to one receiver node. Generally known as a non-collision architecture, this approach minimizes the total volume of data in motion. And because it understands the hardware configuration and which AMPs are on which node, the BYNET can further optimize the process by delivering only one message to each physical node with pointers to each of the AMPs on that node. This reduces message sending tremendously. And it eliminates the need for receiving and collating delivery confirmations from each AMP in the system.

Dynamic BYNET Groups

The Dynamic BYNET Group is simply an on-the-spot association of the AMPs that will be working on one specific query step. It is possible for many of these BYNET groups to exist at any point in time. When a query is optimized and the first step is ready to be dispatched, a message will be automatically sent across the BYNET, but directed only to the AMPs that are actually needed in doing that step's work. This may be all the AMPs in the system, or it may be a subset, or just one. Receipt of this step message causes the AMP to be automatically enrolled in the BYNET group without the database software having to initiate a separate communication.

Group AMP functionality is an optimizer opportunity that takes advantage of dynamic BYNET groups to better service tactical queries. This feature eliminates some all-AMP steps and replaces them with a step that engages just a subset of the AMPs, if the subset is all that the query requires. This reduces the resources required for such a query, and frees up unneeded AMPs to service other work, thus increasing throughput.

Semaphores

Even though the BYNET uses a light touch with message passing, it offers an even less intrusive technique called channels which it uses for behind-the-scenes inter-AMP coordination during the execution of a query step.

As a step begins to execute, one or more channels are established that loosely associate all AMPs in the dynamic BYNET group that is executing the step. The channels use monitoring and signaling semaphores in order to communicate things like the completion or the success/failure of each participating AMP. Semaphores are parallel infrastructure objects that are globally available because they live within the BYNET. Each completion semaphore, for example, contains a count that reflects how close that BYNET group's AMPs are to completing that step, as shown in Figure 7.

The semaphores' jobs are to signal when the first AMP in the group completes the optimizer step being worked on, and when the last AMP in the group completes the same step. This eliminates the need for every AMP to send a message to the dispatcher and having a bunch of code to collate and figure out when everyone has responded.

teradata.

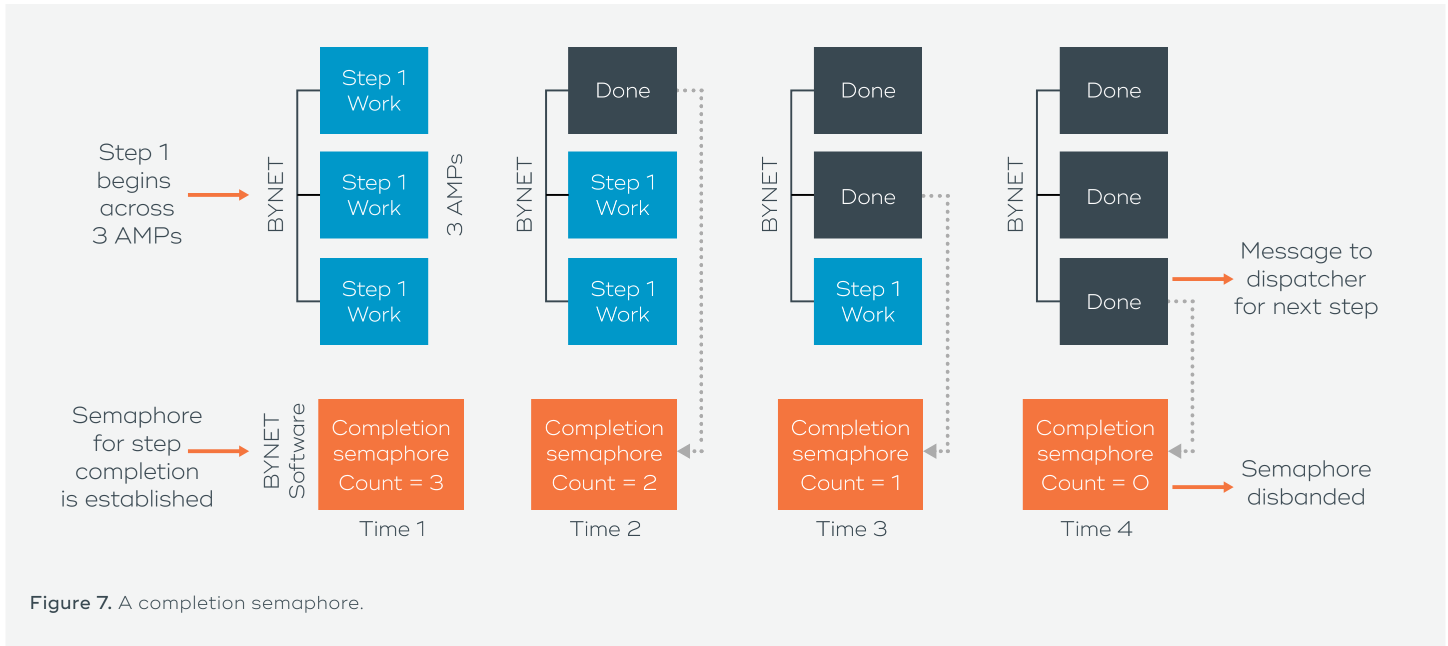


Figure 7. A completion semaphore.

The completion semaphore's count is reduced by one when an AMP reports in and will be reduced to zero when the last AMP completes. When the final AMP completes its work, it sees that the semaphore is registering zero and it knows it is the last one done. Because it is the last participant to complete this step, this AMP sends a message to the dispatcher to send out the next optimized step for that query. This "last-done" message is the only actual message sent back to the parsing engine concerning this step, whether the dynamic BYNET group is composed of three or 3000 AMPs.

Some queries fail to complete. If the cause of a failure is limited to a single AMP, it is important that the other participating AMPs hear about the failure immediately and stop working on that query. If a tight coordination does not exist among AMPs in the same BYNET group, then the problem-free AMPs will continue to work on the doomed query step, eating up resources in unproductive ways.

Semaphores provide the means of alerting all AMPs in the group if one AMP should, for example, run out of spool,3 or otherwise terminate the step, using signaling across the BYNET in lieu of messaging.

Without the BYNET's ability to combine and consolidate information from across all units of parallelism, each AMP would have to independently talk to each other AMP in the system about each query step that is underway. As the configuration grows, such a distributed approach to coordinating query work would quickly become a bottleneck.

Final Answer Set Sort/Merge

Never needing to materialize a query's final answer set inside the database has long been a Teradata differentiator. The final sort/merge of a query takes place within the BYNET as the answer set rows are being funneled up to the client.

Three levels participate in returning a sorted answer set (see Figure 8):

- **Level 1 (AMP-level):** Each AMP performs a local sort in parallel with other AMPs and creates a spool file for its part of the answer set.
- **Level 2 (Node-level):** Each node merges and sorts one buffer's worth of data from all its contributing AMPs.
- **Level 3 (PE-level):** The PE receives and sorts node-level buffers, building one buffer's worth of sorted data to return to the client.

teradata.

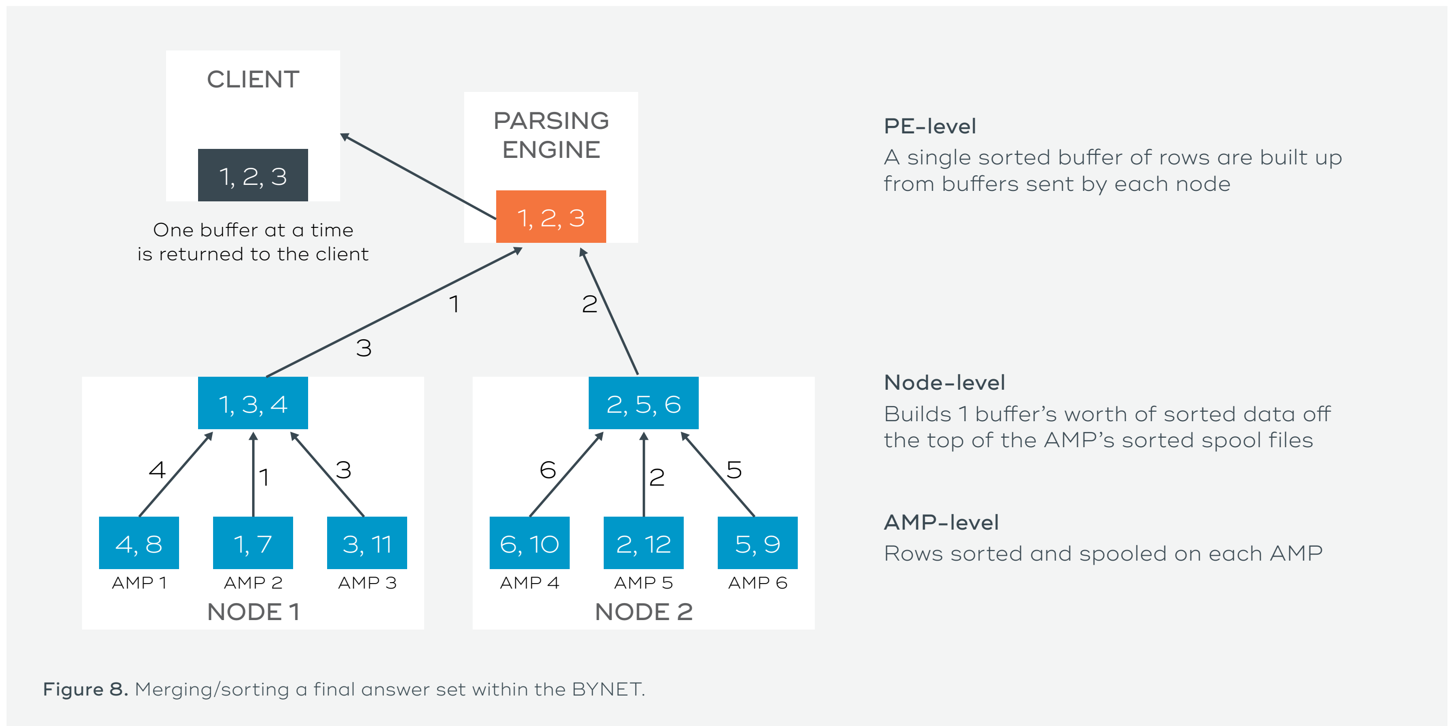


Figure 8. Merging/sorting a final answer set within the BYNET.

The highest values are sent up through the 3 tiers first, while the part of the answer set that contains the lower values remains in the spool files at the AMP level until the higher values have been returned the client.

During this process there is minimal physical I/O performed. For a given row, only a single write—into the AMP's spool table—is ever performed. And only a single read of that same row is ever needed—to retrieve the data for the user in the correct order.

A big benefit of this approach is that the final answer set never has to be brought together in one location for a possibly-large final sort. Rather, the answer set is finalized and returned to the client a buffer at a time. A potential "big sort" penalty has been eliminated; or actually, it never existed.

Evolution

The original architecture of the YNet interconnect easily evolved into the current BYNET, and as it did, it underwent several transformations. The BYNET brought greater availability and reliability, supporting multiple virtual broadcast trees and multiple paths.

Recent enhancements to BYNET functionality include:

- Converting YNet hardware functionality into software capabilities: A virtualized BYNET allows the NewSQL Engine to embrace whatever is the optimal general purpose interconnect functionality at any point in time.
- Moving BYNET hardware into software allows for transparent and consistent inter-AMP communications, making it possible for AMPs on the same node to contact each other without going over the interconnect, thereby reducing delays and traffic congestion.
- Support for multiple Teradata systems sharing the same network infrastructure, with intra-system communications isolated over private partitions.

A Flexible, Fast Way to Find and Store Data

The previous sections discussed the original functionality of the NewSQL Engine in terms of the parallelism, the optimizer, and the BYNET.

teradata.

Another very important factor behind the enduring Teradata performance is how space is managed. What is the effort of locating a row in the database? What happens when a row is inserted and there is no room in the data block where it belongs?

This section will focus on the original architecture of the sub-system that handles space management in the database, a sub-system called the "file system." The file system is responsible for the logical organization and management of the rows, along with their reliable storage and retrieval.

At first glance, managing space seems like a trivial exercise, something a robot could easily be programmed to do. But the file system in Teradata was architected to be extremely adaptable, simple on the outside but surprisingly inventive on the inside. It was designed from Day One to be fluid and open to change. The file system's built-in flexibility is achieved by means of:

- Logical addressing, which allows blocks of data to be dynamically shifted to different physical locations when needed, with minimal impact to active work.
- The ability for data blocks to expand and contract on demand, as a table matures.
- Reliance on an array of unobtrusive background tasks that do continuous space adjustments and clean-up.

Teradata was architected in such a way that no space is allocated or set aside for a table until such time as it is needed. Rows are stored in variable length data blocks that are only as big as they need to be. These data blocks can dynamically change size and be moved to different locations on the cylinder or even to a different cylinder, without manual intervention or end user knowledge.

This section takes a close look at how file system frees up the administrator from mundane data placement tasks, and at the same time provides an environment that is friendly to change.

How Data is Organized

Teradata permanently assigns data rows to AMPs using a simple scheme that lends itself to an even distribution of data—hash partitioning. As initially designed, the value found in the columns selected by the DBA to be that table's primary index are put through a hashing algorithm and two outputs are produced (see Figure 9), often referred to as the "hash" of the row:

- A hash bucket, which maps to one AMP when applied against a pre-defined hash map.
- A hash-ID, which becomes part of the row's unique "row-ID."[4]

In addition to being a distribution technique, this hash approach to data placement serves as an indexing strategy, which reduces the amount of DBA work normally required to set up direct access to a row. To retrieve a row, the primary index data value is passed to the hashing algorithm, which generates the two hash outputs: 1) the hash bucket which points to the AMP; and 2) the hash-ID which helps to locate the row within the file system structure on that AMP. There is no space or processing overhead involved in either building a primary index or accessing a row through its primary index value, as no special index structure needs to be built to support the primary index.

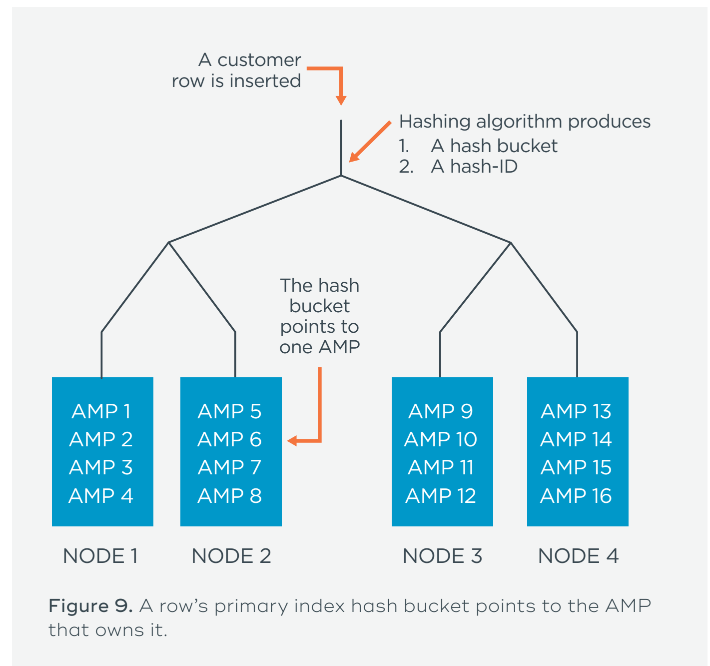


Figure 9. A row's primary index hash bucket points to the AMP that owns it.

teradata.

Hashed data placement is very easy to use and requires no setup. The only effort a DBA makes is the selection of the columns that will comprise the primary index of the table. From that point on, the process is completely automated. No files need to be allocated, sized, monitored, or named. No DDL needs to be created beyond specifying the primary index in the original CREATE TABLE statement. No unload-reload activity is ever required.

Once the owning AMP is identified by means of the hash bucket, the hash-ID is used to look up the physical location of the row on disk. Which cylinder and sector holds the row is determined by means of a tree-like three-level indexing structure (as shown in Figure 10) and further explained in detail in following sections.

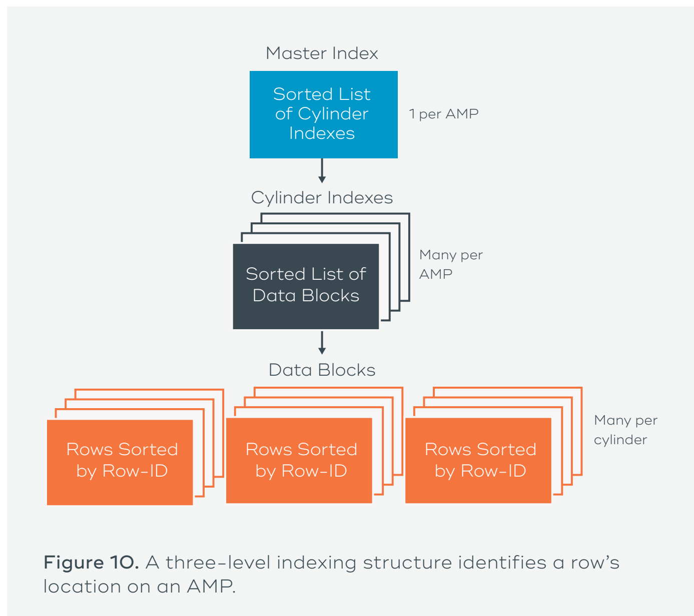


Figure 10. A three-level indexing structure identifies a row's location on an AMP.

Master Index

The master index (MI) is the topmost level of the tree-like structure. An instance of the MI is memory-resident on each AMP. The MI lists all cylinders managed by that AMP in sorted ascending sequence, and contains a series of cylinder index descriptors. These descriptors contain the lowest table-ID and row-ID, and the highest table-ID and row-ID for the data residing on each individual cylinder. Each descriptor provides a pointer to the physical location of the cylinder index for that cylinder. The MI also includes a list of free cylinders available on that AMP.

Cylinder Index

The cylinder index (CI) is the second level. Contrary to the master index, cylinder indexes may or may not be held in AMP memory. While there is only a single MI per AMP, there may be thousands of CIs, many of which may rarely be used.

Physically, the cylinder index is part of the cylinder itself. Think of it as the first data block in the cylinder, as it can be read into memory and updated just like any other data block. There are actually two cylinder indexes on each cylinder, so that when a CI is being updated, another version is available to read, and no blocking will take place.

A CI is composed of a sorted list of data block descriptors. These data block descriptors hold the first table-ID and row-ID for each data block, the number of rows in the block, its physical location and length, as well as a free block list indicating where free space exists on the cylinder.

Data Blocks

The third level in the tree is the data block itself. Data blocks contain a series of rows that are from the same table. The rows within the data blocks are always sorted by hash. No space is allocated ahead of time for a data block, but rather space is dynamically made available at the time data is loaded or a new row is inserted.

Although there is an effort to keep all data blocks from one table ordered physically by hash on a cylinder, growth that causes blocks to exceed their maximum allowable size and to split sometimes makes that difficult. But because the entries are logically sorted in the master index and the cylinder index, any data block within a cylinder can be directly accessed even when it is physically out of sequence with other data blocks. See Figure 12 for an example.

The data block is the physical I/O unit of work in the Teradata file system. Block sizes are not expected to be uniform in length, which alleviates the need for the administrator to spend time optimizing storage or tuning row sizes.

teradata.

But each table has a block size specification, which acts as an upper limit on the size of a multi-row data block. Although this may increase from release to release,the system will enforce a maximum data block size that will cap how large a physical data block may be; this maximum is at least as large as the largest row size allowed.

The administrator may choose a smaller data block size as the maximum multi-row data block size, on a table-by-table basis.

Easy Accommodation of Data Growth

The NewSQL Engine is built using a logical addressing model as a low impact way to adjust to data growth. Data in a Teradata system is stored in flexibly-sized data blocks that are loosely anchored together via logical-to-physical indexing, and are able to be floated from one physical disk location to another as needed. Multi-row data blocks that grow beyond a DBA-specified maximum size automatically split to make room for more rows. If a particular block needs to grow beyond the space it has on its cylinder, that block can be moved to a different location on the same cylinder, or to a different cylinder. When this happens, the appropriate cylinder index is updated to reflect the new physical location change. Figure 11 explains this behavior visually.

This adaptable behavior delivers numerous benefits. Random growth is accommodated at the time it happens. Rows can easily be moved from one location to another without affecting in-flight work or any other data objects that reference that row. There is never a need to stop activity and re-organize the physical data blocks, or adjust pointers.

This flexibility to consolidate or expand data blocks anytime allows the NewSQL Engine to do many space-related housekeeping tasks in the background and avoid table unloads and reloads common to fixed-sized page data-bases. This advantage increases database availability and translates to less maintenance tasks for the DBA.

Contrast this to the fixed-size page approach to storing rows. The physical address of such rows can only be changed by removing and re-inserting the rows, which usually requires the database to be brought down for a data reorganization procedure. And if overflow pages are required (a concept unknown to the NewSQL Engine), this can cause performance degradation because overflow pages add additional I/O to accessing and inserting rows.

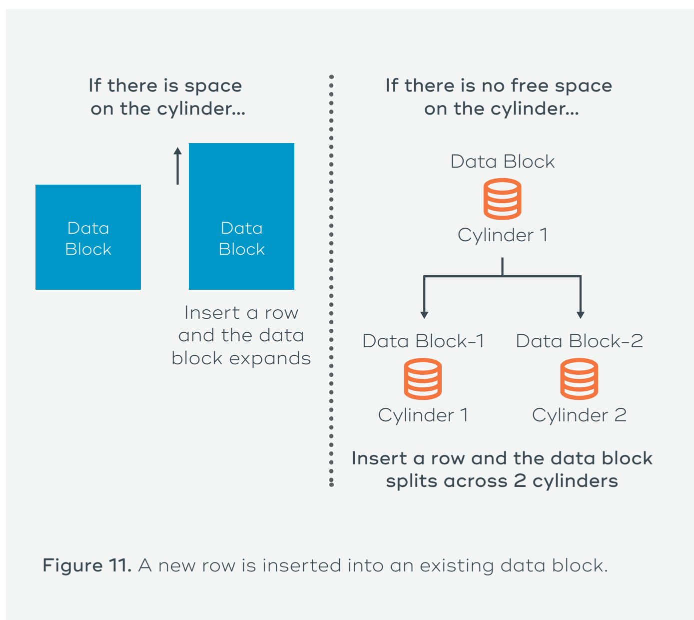


Figure 11. A new row is inserted into an existing data block.

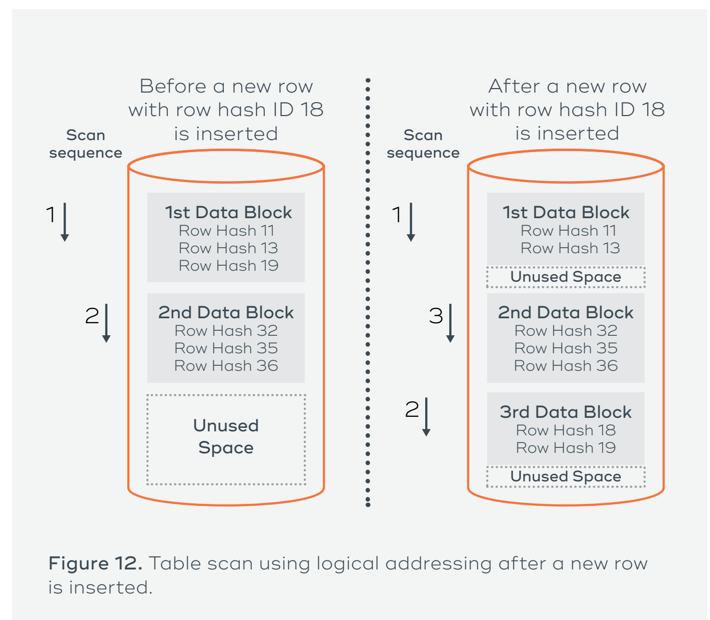


Figure 12. Table scan using logical addressing after a new row is inserted.

teradata.

Table Scan Efficiency with Logical Addressing

This section illustrates an example of how logical addressing works when a query is scanning a table whose data blocks have matured over time. Every effort is made by the file system to keep rows from the same table physically co-located and in row hash sequence. But as a table grows, and some blocks split because they get too large, strict physical ordering is not always sustainable. This is where the advantage of logical addressing arises.

In the example (Figure 12), the first data block and the second data block are immediately adjacent to one another. Data block 1 has no room for growth, and therefore has to split into two smaller data blocks in order to accommodate a new row and still maintain row hash order within the data block.

To keep a semblance of order across what could end up being a messy free-for-all if logical addressing ran amok, the file system imposes several internal conventions on the ordering of rows within data blocks and across cylinders:

1. A data block can only contain rows from one table.
2. The row hash values present in a given data block are either all greater than, or all less than, the row hashes maintained on any other data block of the same table.
3. The key values for a cylinder's cylinder index are either all greater than, or all less than, the values maintained on any of the other cylinders within an AMP.

As a result, when a table is scanned in row hash order, all the rows in one data block can be processed beforemoving onto the next data block. And while the next data block may not be physically adjacent to the current data block within the cylinder, all data blocks on one cylinder can be read before moving on to the next cylinder. There is never a need to go back and read part of an already-read cylinder. And no pointer chains are ever needed to locate logically adjacent data blocks. This also means that when using a special I/O-saving feature known as a cylinder read, all data blocks that are on the cylinder can be processed in the correct order using a single I/O.

Background Clean-up Tasks

From the very first instance of SQL Engine, background tasks have played an important role. Transparent to the user and DBA the database continuously performs clean-up tasks, such as consolidating small pockets of free space on a cylinder (called "defrag"). This housekeeping work is done only when it is required, and at a low priority, so as not to impact other active work. Consequently, DBA intervention or system down time is typically not required to keep the rows organized.

The list below shows a few of these self-managing tasks:

- **Defrag:** Consolidate fragmented space, one cylinder at a time—the fuller the disks, the more work the task will do.
- **AutoCylPack:** Combines adjacent, sparsely filled cylinders, tends to run these "cylpacks" when the system is idle.
- **Transient journal purging:** Removes transaction journal entries from disk periodically, after the commit has taken place, rather than holding up transaction completion for this work.
- **FSG[5] cache purging:** Looks for older data blocks that have been updated but are resident in the FSG cache, and writes them to disk.

Evolution

In the original Teradata release, each AMP logically and physically owned and managed its own data on its own physical set of disks. As time passed and more advanced disk technologies became available, the logical whereabouts of the data as seen by the file structure components and its actual physical address on disk became abstracted. Logical addressing, which was already in place as the key file system approach for accommodating every-day data growth, made this decoupling straightforward.

One of the most noteworthy features contributing to virtualization of the file system is called Teradata Virtual Storage (TVS) which, when it was introduced, enhanced multi-temperature capabilities by optimizing data placement on the disks to speed access to the most

teradata.

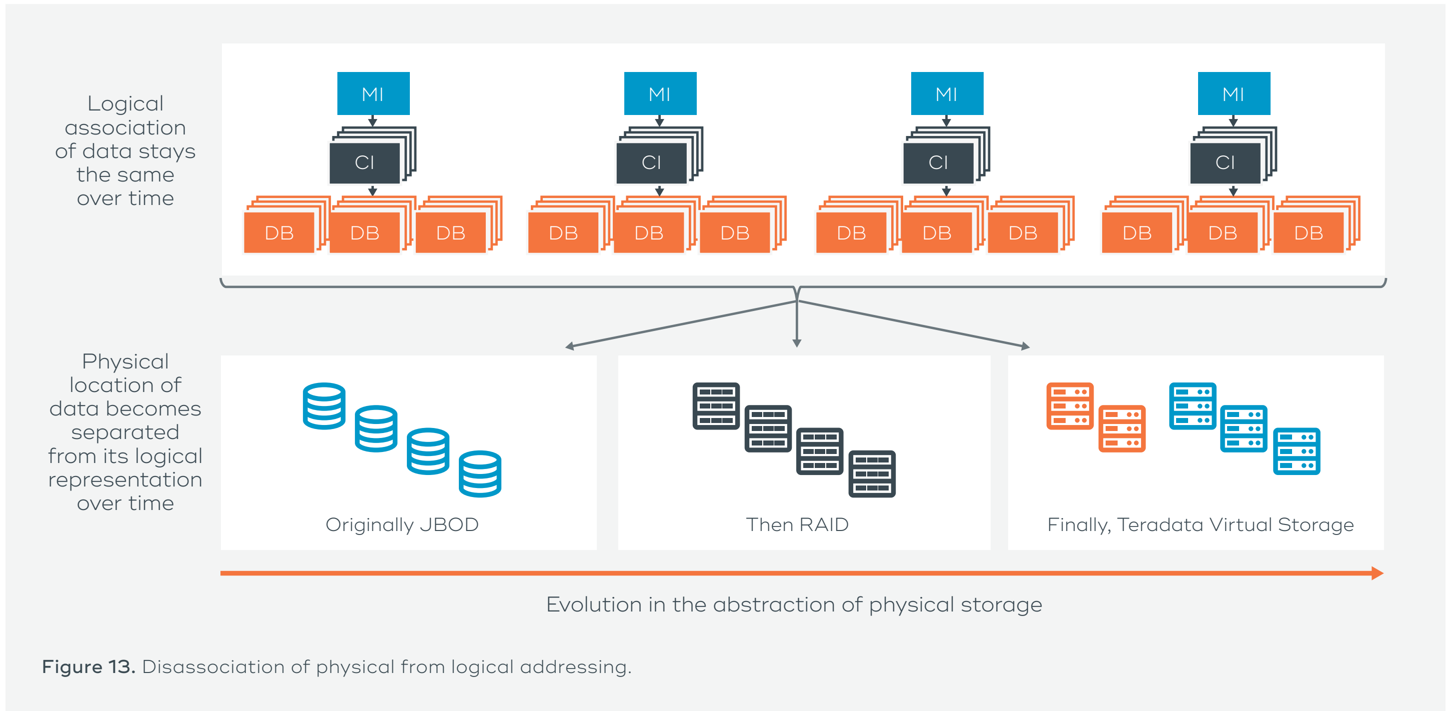


Figure 13. Disassociation of physical from logical addressing.

frequently used hot data. Within the file system TVS has taken over all responsibility for the physical organization/ management of data on disk. Using TVS, for example, a cylinder might be migrated to a different physical storage device depending on its access pattern.

TVS enables the mixing of drive sizes and device types (e.g., spinning and solid state disks) within a single Massively Parallel Processing (MPP) system. If there is a mix of devices, hot data is moved to fast storage, while cold data is moved to slower devices. All of this relocation activity happens behind the scenes, in the background, by TVS. TVS is integrated with the NewSQL Engine and is data object aware: it is able to understand, for example, the difference between temporary spool files and user table data.

This decoupling, this virtualization of the disk space management (as shown in Figure 13), makes the NewSQL Engine even more scalable and higher performing. And it builds on, rather than replaces, the original way of doing things. The DBA still defines the table logically and the database takes care of allocating file space and where the data resides at any given time.

Some of the features in addition to TVS, which were built on and rely upon the original file system architecture, include:

- Table partitioning that supports row, column or a combination of both approaches and dynamic partition elimination, with all partitioned rows stored and manipulated using the same original internal file system conventions, including the row-ID layout.
- Block-level compression, in which all rows within a data block are compressed as a unit.
- Temperature-based block-level compression (compress on cold), an extension of block level compression where only data blocks residing on infrequently accessed cylinders undergo compression.
- No Primary Index tables that offer a fast approach for dumping data randomly across the AMPs that bypasses hash partitioning, well-suited for temporary staging tables.

teradata.

Work Flow Self-Regulation

A shared-nothing parallel database has a special challenge when it comes to knowing how much new work it can accept, and how to identify congestion that is starting to build up inside one or more of the parallel units. With the optimizer attempting to apply multiple dimensions of parallelism to each query that it sees, it is easy to reach very high resource utilization within a Teradata system, even with just a handful of active queries.

Designed for stress, the NewSQL Engine is able to function with large numbers of users, a very diverse mix of work, and a fully-loaded system. Being able to keep on functioning full throttle under conditions of extreme stress relies on internal techniques that were built inside the database to automatically and transparently manage the flow of work, while the system stays up and productive.

Even though the data placement conventions in use with the NewSQL Engine lend themselves to even placement of the data across AMPs, the data is not always accessed by queries in a perfectly even way. During the execution of a multi-step query, there will be occasions when some AMPs require more resources for certain steps than do other AMPs. For example, if a query from an airline company site is executing a join based on airport codes, you can expect whichever AMP is performing the join for rows with Atlanta (ATL) to need more resources than does the AMP that is joining rows with Anchorage (ANC).

Does the database keep pushing work down to the AMPs even when some AMPs can't handle more? At what point does the database put on the brakes and stop sending new work messages to the AMPs? Is a central coordinator task necessary to poll the work levels of each AMP, then mandate when more work can be sent down or when it's time to put on the brakes? A central coordinator task is something to avoid in a shared-nothing parallel database because it becomes a non-parallel operation which can become a bottleneck as activity, database size, or number of units of parallelism increases.

AMP-Level Control

The NewSQL Engine manages the flow of work that enters the system in a highly-decentralized manner, in keeping with its shared-nothing architecture. There is no centralized coordinator. There is no message-passing between AMPs to determine if it's time to hold back new requests. Rather, each AMP evaluates its own ability to take on more work, and temporarily pushes back when it experiences a heavier load than it can efficiently process. And when an AMP does have to push back, it does that for the briefest moments of time, often measured in milliseconds.

This bottom-up control over the flow of work was fundamental to the original architecture of the database as designed. All-AMP step messages come down to the AMPs, and each AMP will decide whether to begin working on it, put it on hold, or ignore it. This AMP-level mindfulness is the cornerstone of the database's ability to accept impromptu swings of very high and very low demand, and gracefully and unobtrusively manage what-ever comes its way.

Individual AMPs have two techniques at their disposal when they experience stress:

1. Queuing up arriving messages
2. Turning away new messages

Both techniques provide a short-lived breather for the AMP, allowing it to focus on finishing off the tasks at hand before taking on new work.

Queuing Up Arriving Messages

Each AMP contains a set of structures that aid in performing database work. Among those structures are a defined number of AMP worker tasks (AWTs). Each AMP has by default 80 AWTs.

AMP Worker Tasks

AWTs are the tasks inside of each AMP that get the database work done. This database work may be initiated by the internal database software routines, such as dead-lock detection or other background tasks. Or the work may originate from a user-submitted query.

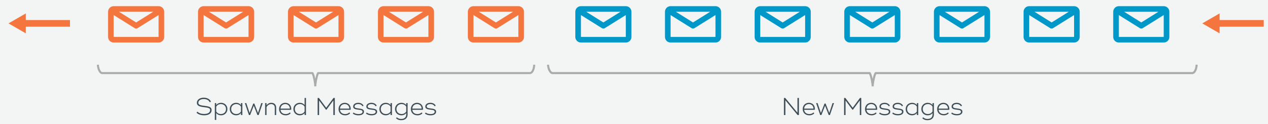


Figure 14. The work message queue is ordered by work type (descending).

These pre-allocated AWTs are assigned to each AMP at startup and, like taxi cabs queued up for fares at the airport, they wait for work to arrive, do the work, and come back for more work.

Because of their stateless condition, AWTs respond quickly to a variety of database execution needs. There is a fixed number of AWTs on each AMP. For a task to start running it must acquire an available AWTs. Having an upper limit on the number of AWTs per AMP keeps the number of activities performing database work within each AMP at a reasonable level. AWTs play the role of both expeditor and governor.

As part of the optimization process, a query is broken into one or many AMP execution steps. An AMP step may be simple, such as read one row using a unique primary index or apply a table level lock. Or an AMP step may be a very large block of work, such as scanning a table, applying selection criteria on the rows read, redistributing the rows that are selected, and sorting the redistributed rows.

The Message Queue

When all AMP worker tasks on an AMP are busy servicing other query steps, arriving work messages are placed in a message queue that resides in the AMP's memory. This is a holding area until an AWT frees up and can service the message.

This queue is sequenced first by message work type, which is a category indicating the importance of the work message. Within work type the queue is sequenced by the priority of the request the message is coming from.

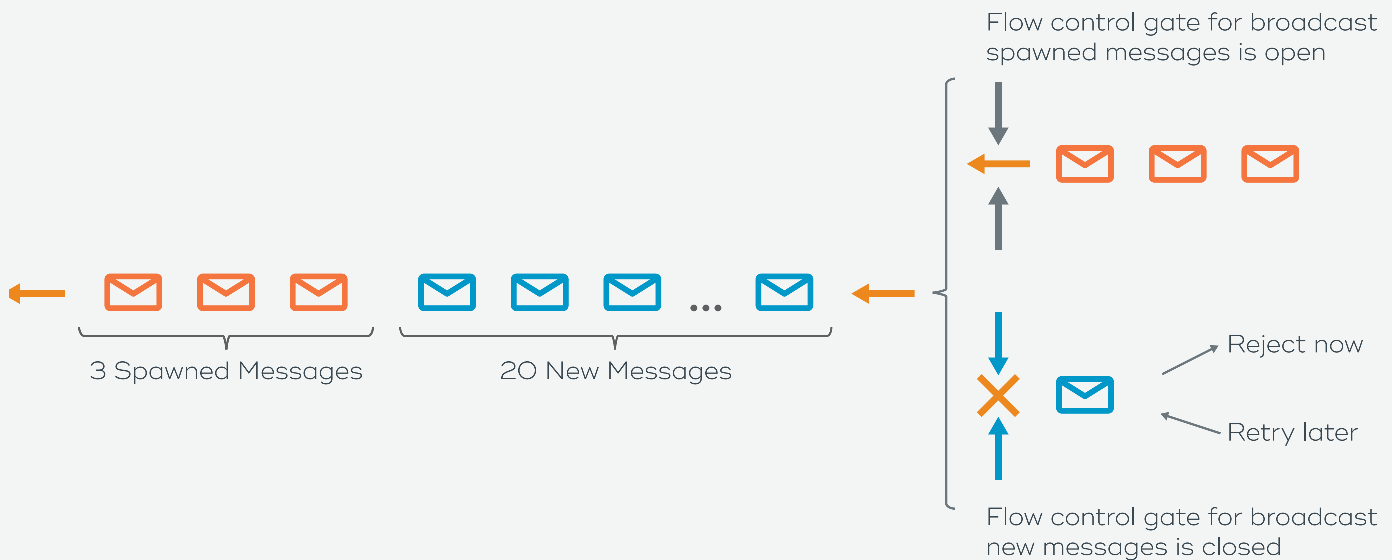


Figure 15. Flow control gates close when a threshold of messages is reached.

teradata.

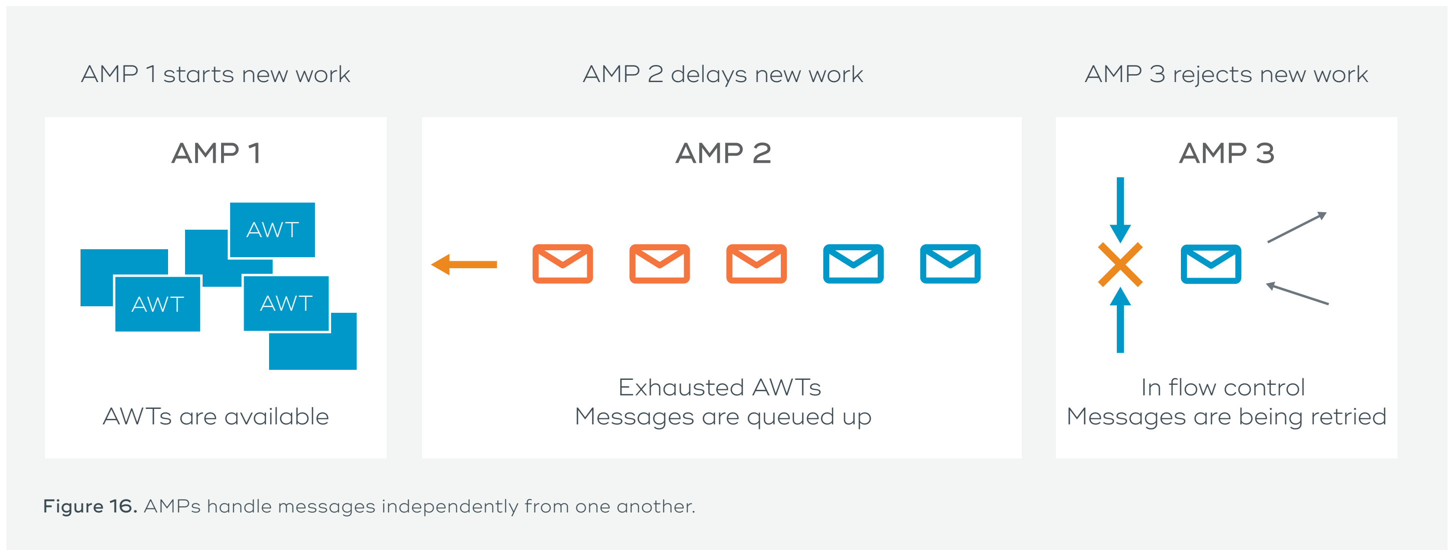


Figure 16. AMPs handle messages independently from one another.

Messages that start new work are always placed at the end of the message queue, adhering to one of the underlying Teradata philosophies of: It's more important to complete work that is already active than it is to start new work (see Figure 14).

It is normal during busy processing times to have some AMPs run out of AWTs and begin to queue arriving messages. This condition is usually temporary, and in most cases is transparent to the user submitting queries.

Messages representing a new query step are broadcast to all participating AMPs by the dispatcher. In such a case, some AMPs may provide an AWT immediately, while other AMPs may have to queue the message. Some AMPs may dequeue their message and start working on the step sooner than others. This is typical behavior on a busy system where each AMP is managing its own flow of work.

Once a message has either acquired an AWT or been accepted onto the message queue across each AMP in the dynamic BYNET group, then it is assumed that each AMP will eventually process it, even if some AMPs take longer than others. The sync point for the parallel processing of each step is at step completion, when each AMP signals across the completion semaphore that it has completed its part. The BYNET channels set up for this purpose are discussed more fully in the BYNET section of this paper.

Turning Away New Messages

The message queue is the first level of push-back on an AMP. However, the message queue cannot endlessly expand in cases where more and more messages are landing on the queue and fewer and fewer are being released to run. The NewSQL Engine's flow control mechanisms go a step further than queuing up work messages.

In order to conserve memory, the length of the message queue is controlled. When the number of messages queued reaches a threshold, the AMP will turn to a second mechanism of relief—sending newly-arriving messages back to their source.

Messages have senders and receivers. In the case of a new step being sent to the AMPs, the dispatcher is the sender and some number of AMPs are the receivers. However, in cases such as spawned work,[6] one AMP is the sender and another AMP or AMPs are the receivers.

Flow Control Gates

Each AMP has flow control gates that monitor and manage messages arriving from senders. There are separate flow control gates for each different message work type.[7] New work messages will have their own flow control gates, as will spawned work messages. The flow control gates keep a count of the active AWTs of that work type as well as how many messages are queued up waiting for an AWT.

teradata.

Once the queue of messages of a certain work type grows to a specified length, new messages of that type are no longer accepted and that AMP is said to be in a state of flow control, as shown in Figure 15. The flow control gate will temporarily close, pulling in the welcome mat, and arriving messages will be returned to the sender. The sender, often the dispatcher module within the PE, continues to retry the message, until that message can be received on that AMP's message queue.

Getting Back to Normal

Because the NewSQL Engine is a message-passing database, there are many different types of message queues within the system architecture. All of these different queues (sometimes referred to as "mailboxes") have limits set on the number of messages they can accommodate at any point in time. Any of them, including the work message queue already discussed, can go into a state of flow control if the system is temporarily overwhelmed by work. Every AMP handles these decisions independently of every other AMP as illustrated in Figure 16.

Because the acceptance and rejection of work messages happens at the lowest level, in the AMP, there are no layers to go through when the AMP is able to get back to normal message delivery and processing. The impact of

turning on and turning off the flow of messages is kept local—only the AMP hit by an over-abundance of messages at that point in time throttles back temporarily.

Riding the Wave of Full Usage

Teradata was designed as a throughput engine, able to exploit parallelism in order to maximize resource usage of each request when only a few queries are active, while at the same time able to continue churning out answer sets in high demand situations. To protect overall system health under extreme usage conditions, highly-decentralized internal controls were put into the foundation, as discussed in this section.

The original architecture related to flow control and AMP worker tasks has needed very little improvement or even tweaking over the years. 80 AWTs per AMP is still the default setting for new Teradata systems. Message work types, the work message queue, and retry logic all work the same as they always did.

There have been a few extensions in regards to AMP worker tasks that have emerged over time, including:

- Setting up reserve pools of AWTs exclusively for use by tactical queries, protecting high priority work from being impacted when there is a shortage of AWTs.
- Automatic reserve pools of AWTs just for load utilities that become available when the number of AWTs per AMP is increased to a very high level, intended to reduce resource contention between queries and load jobs for enterprise platforms with especially high concurrency.

Workload Management

The second section in this whitepaper called attention to the multifaceted parallelism available for queries on the NewSQL Engine. The subsequent section discussed how the optimizer uses those parallel opportunities in smart ways to improve performance on a query by query basis. And the previous section illustrated internal AMP-level controls to keep high levels of user demand and an over-abundance of parallelism from bringing the system to its knees.

In addition to those automatic controls at the AMP level, Teradata has always has had some type of system-level workload management, mainly priority differences, that are used by the internal database routines.

The Original Four Priorities

One of the challenges faced by the original architects of Teradata Database was how to support maximum levels of resource usage on the platform, and still get critical pieces of internal database code to run quickly when it needed to. For example, if there is a rollback taking place due to an aborted transaction, it benefits the entire system if the reversal of updates to clean up the failure can be executed quickly.

teradata.

It was also important to ensure that background tasks running inside the database didn't lag too far behind. If city streets are so congested with automobile traffic that the weekly garbage truck can't get through and is delayed for weeks at a time, a health crisis could arise.

The solution the original architects found was a simple priority scheme that applied priorities to all tasks running on the system. This rudimentary approach offered four priority buckets, each with a greater weight than the one that came before: L for Low, M for Medium, H for High and R for rush. The default priority was medium, and indeed most work ran at medium, and was considered equally-important to other medium priority work that was active.

However, database routines and even small pieces of code could assign themselves one of the other three priorities, based on the importance of the work. Developers, for example, decided to give all END TRANSACTION activity the rush priority, because finishing almost-completed work at top speed frees up valuable resources sooner, and was seen as critical within the database. In addition, if the administrator wanted to give a favored user a higher priority, all that was involved was manually adding one of the priority identifiers into the user's account string.

Background tasks discussed in the section about space management were designed to use priorities as well. Some of these tasks, like the task that deletes transient journal rows that are no longer needed, were designed to start out at the low priority, but increase their priority over time if the system was so busy that they were not able to get their work accomplished. This approach kept such tasks in the background most of the time, except when their need to complete becomes critical.

Impact of Mixed Workloads

The simple approach to priorities was all the internal database tasks required. And early users of the database were satisfied running all their queries at the default medium priority. But requirements shifted over time as Teradata users began to supplement their traditional decision support queries with new types of more varied workloads.

In the late 1990's, a few Teradata sites began to issue direct look-up queries against entities like their Inventory tables or their Customer databases, at the same time as their standard decision support queries were running. Call centers started using data in their Teradata Database to validate customer accounts and recent interactions. Tactical queries and online applications blossomed, at the same time as more sites turned to continuous loading to supplement their batch windows, giving their end users more timely access to recent activity. Service level goals reared their head. Stronger, more flexible workload management was required.

Evolution of Workload Management

While the internal management of the flow of work has changed little, the capabilities within system-level workload management have expanded dramatically over the years. As the first step beyond the original four priorities, Teradata engineering developed a more extensive priority scheduler composed of multiple resource partitions and performance groups, and the flexibility of assigning your own customized weighting values. These custom weightings and additional enhancements make it easier to match controls to business workloads and priorities than the original capabilities designed more for controlling internal system work.

Additional workload management features and options that have evolved over the years include:

- Concurrency control mechanisms, called throttles, that can be placed at multiple levels and tailored to specific types of queries or users.
- An improved and more effective priority scheduler to accompany the Linux SLES 11 operating system that can protect short, critical work more effectively from more resource-intensive lower-priority jobs.
- Rules to reject queries that are poorly-written or that are inappropriate to run at certain times of the day.
- Ability to automatically change workload settings by time of day or system conditions.
- Ability to automatically reduce the priority of a running query which exceeds the threshold of resources consumed for its current priority.

- Two complete collections of workload management features referred to as Teradata Active System Management and Teradata Integrated Workload Management.
- A user-friendly front-end GUI called Viewpoint Workload Designer that supports ease of setup and tuning.

Workload management in Teradata has proven to be rapidly expanding area, indispensable to customers that are running a wide variety of work on their Teradata platform. While internal background tasks and subsets of the database code continue to run at the four different priority levels initially defined for them, many Teradata sites have discovered that their end users experiences are better and they can get more work through the system when taking advantage of the wider workload management choices today. And many do just that.

Conclusion

Foundations are important. If you remember the fairy tale of the three little pigs, the first little pig built his house of straw, the second little pig built his house of sticks, and the third little pig built his house of bricks. When the big bad wolf came along, he was able to blow down the houses built of straw and sticks, but not the house built of bricks. So only the third little pig survived to live happily ever after.

Teradata Database, now the Teradata Vantage NewSQL Engine, is a survivor. Its ability to grow in new directions and continue to sustain its core competencies is a direct result of its strong, tried-and-true foundation.

An interesting pattern has emerged over the years as the NewSQL Engine has matured, a pattern that underscores the unusual adaptability of the database: Logical components and their physical implementation have become more and more disassociated.

- In Version1 Release 1 each AMP was a physical node that actually owned its own disk drives and directly managed how data was located on its disks. Today an AMP is a software virtual processor that co-exists on an SMP with other such virtual processors, all of whom share the node resources. Yet each AMP maintains its shared-nothing characteristics, same as in the first release.
- The YNet was a proprietary, physical interconnect, supported with bits of code on each AMP. But today, what was the YNet has evolved into the BYNET. Due to recent virtualization techniques, the BYNET is free to run on whatever off-the-shelf interconnect hardware offers the most benefit. Although there have been many enhancements over time, all of the original reliability, coordination, and performance capabilities designed into the YNet remain intact.
- The master index, cylinder index and data block file system structures originally were put in place to point to actual physical locations of rows on an AMP's disks. Today the underlying storage is completely managed by TVS, yet the same index structures remain in place to keep the logical associations in order. Even though new features have been added over the years, the essential building blocks of the file system are the same.

The natural evolution towards the virtualization of key database functionality is significant because it broadens the usefulness of the NewSQL Engine. For much of its history, Teradata database software has run on purpose-built hardware, where the underlying platform has been optimized to support high throughput, critical SLAs, and solid reliability. While those benefits remain well-suited for enterprise platforms, this virtualization opens the door for the NewSQL Engine to participate in more portable, less demanding solutions. Public or private cloud architectures, as well as as-a-service offerings, can now enjoy the core NewSQL Engine capabilities as described in this white paper.

This white paper attempts to familiarize you with a few of the features that make up important building blocks of the NewSQL Engine, so you can see for yourself the elegance and the durability of the architecture. This paper points out recent enhancements that have grown out of this original foundation, building on it rather than replacing it.

These foundational components have such a widespread consequence that they simply cannot be tacked on as an afterthought. The database must be born with them.

Endnotes

1. Table duplication is a join choice where the smaller table is duplicated in its entirety to all other AMPs working on that step.
2. The parsing engine is the virtual processing unit that communicates with clients and with AMPs. It performs session management, parsing, optimization and enforces workload management rules. There may be one or more parsing engines per node and each can support 120 sessions at a time.
3. A spool file is an intermediate answer set that temporarily holds results from one query step that feed into a subsequent step. Users have limits on how much spool space their queries may use, and a query that exceeds its spool limit will be aborted.
4. A row-ID is the unique identifier of the row within the logical file system structure. It includes the hash-ID, and in addition, detail to help locate the row, such as the partition number and the hash uniqueness value. Row-ID and hash-ID are often used synonymously.
5. FSG refers to the file system sub-system and it's manipulations of segments of data. FSG stands for "File Segment." FSG cache is the primary and original cache on the NewSQL Engine.
6. Spawned work takes place when a query step requires more than one AWT to get its work done, such as is the case during row redistribution, where one AMP is required to read and redistribute the rows to other AMPs, and a second AWT is required to receive rows being sent to it from other AMPs.
7. A work type is given to each arriving work message, in order to reflect the importance of the work to be performed. "New" messages and "spawned" messages use different work types, for example.

About Teradata

With all the investments made in analytics, it's time to stop buying into partial solutions that overpromise and underdeliver. It's time to invest in answers. Only Teradata leverages all of the data, all of the time, so you can analyze anything, deploy anywhere, and deliver analytics that matter most to your business. And we do it on-premises, in the cloud, or anywhere in between. We call this pervasive data intelligence. It's the answer to the complexity, cost and inadequacy of today's analytics. And how we transform how businesses work and people live through the power of data.

Get the answer at **Teradata.com**.

17095 Via Del Campo, San Diego, CA 92127 Teradata.com

Teradata and the Teradata logo are registered trademarks of Teradata Corporation and/or its affiliates in the U.S. and worldwide. Teradata continually improves products as new technologies and components become available. Teradata, therefore, reserves the right to change specifications without prior notice. All features, functions and operations described herein may not be marketed in all parts of the world. Consult your Teradata representative or Teradata.com for more information.

© 2019 Teradata Corporation All Rights Reserved. Produced in U.S.A. 03.19 EB3053

teradata.