Hybrid Row-Column Partitioning in Teradata®

Mohammed Al-Kateb
Teradata Labs
601 N Nash Street
El Segundo, CA
mohammed.al-kateb
@teradata.com

Paul Sinclair
Teradata Labs
601 N Nash Street
El Segundo, CA
paul.sinclair
@teradata.com

Grace Au
Teradata Labs
601 N Nash Street
El Segundo, CA
grace.au
@teradata.com

Carrie Ballinger
Teradata Labs
601 N Nash Street
El Segundo, CA
carrie.ballinger
@teradata.com

ABSTRACT

Data partitioning is an indispensable ingredient of database systems due to the performance improvement it can bring to any given mixed workload. Data can be partitioned horizontally or vertically. While some commercial proprietary and open source database systems have one flavor or mixed flavors of these partitioning forms, Teradata Database offers a *unique* hybrid row-column store solution that seamlessly combines both of these partitioning schemes. The *key* feature of this hybrid solution is that either row, column, or combined partitions are all stored and handled in the same way internally by the underlying file system storage layer. In this paper, we present the main characteristics and explain the implementation approach of Teradata's row-column store. We also discuss query optimization techniques applicable specifically to partitioned tables. Furthermore, we present a performance study that demonstrates how different partitioning options impact the performance of various queries.

1. INTRODUCTION

Data partitioning is a principal factor in query optimization and processing [17]. It allows access to a subset of data if and when possible, which can improve the overall performance considerably by reducing I/O cost, boosting system throughput, increasing query parallelism, maximizing locality of joins and aggregations [29], and giving the opportunity for finer locking granularity [15].

Data can generally be partitioned by row or by column. Row partitioning divides a table horizontally. Each row partition clusters together a subset of the rows. With row partitioning, only a subset of rows is accessed for queries that specify a single value or a range of values on the partitioning column(s). Consider the example in Figure 1 in which a Sales table is partitioned by row on the transaction date. With row partitioning, the following query that retrieves "ItemNo" for the date of "05-29-2011" needs to access and read only one row partition instead of the whole entire table.

```
SELECT ItemNo
FROM   Sales
WHERE  TxnDate = '05-29-2011'
```

TxnNo	TxnDate	ItemNo	Quantity	
100	05-29-2011	756	1	One row partition
100	05-29-2011	124	3	
290	05-30-2011	437	1	One row partition
450	05-30-2011	110	1	
530	05-30-2011	815	2	

Figure 1: Example of Row Partitioning

Column partitioning divides a table vertically into disjoint sets of columns [26]. Each column or group of columns in a table becomes a partition containing the column partition values of that column partition. With column partitioning, a query needs to access only the column partition(s) that contain the columns referenced in the query. Consider the example in Figure 2. In this example, the Sales table is partitioned by column such that each column is placed in a separate column partition. The query that retrieves "ItemNo" for items sold on "05-29-2011" needs to access only 2 columns of the table.

It is evident that both row and column partitioning can improve query performance in different ways. If the two partitioning forms can be combined together, the improvement on query performance can be substantial. Consider the example with a mix of row and column partitioning as shown in Figure 3. With this hybrid partitioning, retrieving "ItemNo" for items sold on "05-29-2011" requires access to only two columns of only two rows.

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/4.0/. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org.
Proceedings of the VLDB Endowment, Vol. 9, No. 13
Copyright 2016 VLDB Endowment 2150-8097/16/09.

TxnNo	TxnDate	ItemNo	Quantity
100	05-29-2011	756	1
100	05-29-2011	124	3
290	05-30-2011	437	1
450	05-30-2011	110	1
530	05-30-2011	815	2

One column partition | One column partition | One column partition | One column partition

Figure 2: Example of Column Partitioning

TxnNo	TxnDate	ItemNo	Quantity
100	05-29-2011	756	1
100	05-29-2011	124	3
290	05-30-2011	437	1
450	05-30-2011	110	1
530	05-30-2011	815	2

One row partition (rows 1–2); One row partition (rows 3–5)

One column partition | One column partition | One column partition | One column partition

Figure 3: Example of Row-Column Partitioning

In this paper, we present Teradata's *hybrid row-column store* that allows for row and column partitioning and a mixed storage of both. First, we describe the Teradata Database parallel architecture and its main components. Then we discuss the key features and explain the implementation approach of data partitioning in Teradata. We also discuss several query optimization techniques applicable to partitioned tables. Lastly, we show the results of a performance study conducted to examine the impact of different partitioning choices.

Teradata Database is a shared-nothing architecture [25] that can be deployed to **M**assively **P**arallel **P**rocessing (MPP) systems[1] [7]. The architecture contains two types of multi-threaded virtual processing units: **P**arsing **E**ngines (PE) and **A**ccess **M**odule **P**rocessors (AMP). A PE executes the database software and communicates between client systems and AMPs. Each AMP owns part of the data on the physical disk space and manages the database interactions between PEs and virtual disks. The communication between PEs and AMPs is carried through a virtual layer of an interprocessor network known as BYNET.

In Teradata, *all tables are partitioned* by nature through a multitier partitioning mechanism. The first tier of partitioning is implicit. It establishes the distribution of data across AMPs. It is determined based on whether a table has a **P**rimary **I**ndex (PI), **P**rimary **A**MP Index (PA), or **no** **P**rimary **I**ndex (NoPI). A table with either a PI or PA is partitioned over AMPs by being hash-distributed on the value of PI/PA columns. A NoPI table is partitioned randomly. The middle tier is defined explicitly using the PARTITION BY clause and applies to rows distributed to an AMP. Rows can be partitioned by row or by column or both. There can be multiple levels of row partitioning but at most one level of column partitioning. All partitions are stored and handled by the underlying file system in the same way. The file system is not row-based or column-based and is agnostic to the partitioning scheme. The last tier of partitioning determines whether data is further partitioned based on rowhash, and it applies only to PI tables.

We discuss various optimization techniques that pertain to partitioned tables. Teradata's query optimizer considers different optimizations that are applicable over partitioned tables for single-table access and join queries. Some optimization techniques such as partition elimination are common for different kind of partitioning, while other techniques such as late materialization are specific for tables partitioned by column.

We present a performance study conducted based on the TPC-H benchmark [28]. The purpose of the experiments is to analyze the trade-offs and examine the impact of different partitioning options. Performance metrics include I/O count, CPU time, and elapsed time. Metrics are reported for nonpartitioned, row partitioned, column partitioned, and row-column partitioned tables. Queries used in experiments include simple full-table scans, aggregation queries, join queries, and rollup queries. The results show that some partitioning forms can result in significantly smaller table sizes and can improve the performance of some queries considerably.

The remainder of this paper is organized as follows. Section 2 gives an overview of Teradata Parallel Database. Section 3 explains data partitioning in Teradata. Section 4 discusses different query optimization techniques over partitioned tables. Section 5 presents performance study. Section 6 discusses related work. Finally, Section 7 concludes the paper.

2. TERADATA PARALLEL DATABASE

Teradata is a parallel database [6] with a shared-nothing architecture that inherently enables horizontal scalability. The architecture has four main components: PE, AMP, VDisk, and BYNET (see Figure 4).

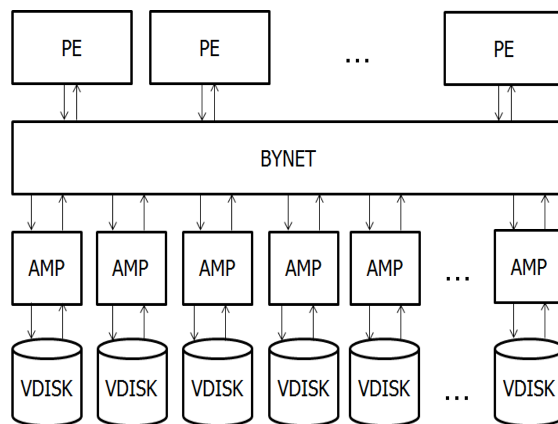


Figure 4: Teradata Shared-Nothing Architecture

A PE is a multithreaded virtual processor responsible for controlling sessions, validating and enforcing security rules, parsing requests, optimizing queries, and dispatching processing steps to AMPs. An AMP is also a multithreaded virtual processor. It executes database operations (e.g., locking, journaling, updates, and retrieves) on a portion of data on virtual disks (VDISKs). A VDISK is a physical space

[1]MPP systems consist of one or more **S**ymmetric **M**ulti **P**rocessing (SMP) systems.

of the storage allocated and assigned by the virtual storage subsystem to each AMP. The virtual storage subsystem automatically puts hot data on faster storage resources (including in memory) and cold data on slower storage resources. PEs and AMPs exchange messages via BYNET, which is a logical layer of communication. In multinode systems, BYNET communication across nodes is done by means of a physical network such as InfiBand or Ethernet. In a single-node system, BYNET is just a virtual layer. BYNET implements bidirectional, multicast, and point-to-point communications between processes.

The building blocks of parallelism in Teradata Database are PE and AMP. A PE is the unit of parallelism at the session level. It handles multiple sessions that run concurrently within a node and across nodes. It consists of Syntaxer, Resolver, Security, Query Rewrite, Query Optimization, Steps Generation, and Dispatcher subsystems.

Syntaxer parses SQL text, builds a skeleton tree, and reports syntax errors. Resolver retrieves dictionary information, annotates the skeleton tree, and reports semantic errors. Security subsystem verifies access rights and performs requested access logging. Query Rewrite subsystem applies rewrite rules and generates semantically-equivalent queries. Query optimizer optimizes queries and parallelizes steps. Step Generation subsystem builds execution steps and sends steps to Dispatcher. The Dispatcher collects all messages of a request and dispatches steps to AMPs.

A PE does not access database storage directly. It receives requests from client applications, resolves and optimizes requests, and generates steps to execute requests. Then it dispatches steps to AMPs, receives response messages back from AMPs, processes response, and returns the final response to client system.

An AMP is the unit of parallelism for data processing and is not associated to a specific session. Its functions include accounting, journaling, locking, and data conversion.

An AMP is a collection of worker tasks, which are threads that process database requests. A worker task performs the actual work requested by a particular step such as sorting, aggregation, and joins. It picks up a request from a queue of requests according to their priorities, services the request, and then waits for another request to arrive from PEs. AMPs can be grouped to form AMP clusters which are vital for fault tolerances.

With PEs and AMPs, parallel processing takes place at different levels. A PE manages multiple sessions at the same time.[2] A step is parallelized across all the AMPs with one worker task (i.e., thread) per AMP working concurrently on behalf of the query. Multiple steps with no inter-dependency are also parallelized. The system automatically manages the number of parallel steps that run concurrently for a query to avoid worker task exhaustion and concurrency conflicts.[3]

Parallel processing is driven by the database parallel-aware query optimizer. Query optimization in Teradata is rule-based and cost-based. Rule-based optimizations are normally in the form of query rewrites and are performed by the Query Rewrite subsystem [11]. Examples of query rewrites include projection pushdown, predicate pushdown, join elimination, outer-to-inner join conversion, set operation branch elimination, and view folding. Cost-based optimizations are done by the Optimizer subsystem and involve cardinality estimation, selectivity estimation, indexes selection, derived statistics, and data redistribution and duplication. Examples of cost-based optimizations include single-table access path selection, join indexes planning, local vs. global aggregations, and single-AMP vs. all-AMP steps.

Teradata supports UDT, LOB, JSON, and Period data types. It also supports temporal query processing [4] and geospatial databases.

[2]Up to 128 sessions per PE.

[3]Default is 80 worker tasks per AMP.

3. HYBRID DATA PARTITIONING

There are generally two forms of data partitioning. Row partitioning breaks up a table horizontally based on expressions defined on partitioning columns of interest. Column partitioning divides a table vertically by grouping one or more columns together. Teradata Database natively supports row and column partitioning and a hybrid of both.

3.1 Example

Figure 5 shows an example of how a table can be partitioned on an AMP in the context of the following simple aggregation query that calculates the average on column F in table T for rows that have the value of column B between 4 and 7.

```
SELECT avg(F)
FROM   T
WHERE  B between 4 and 7
```

There are different ways to access data to answer this query depending on partitioning of data:

- With no partitioning (option 1), the whole table is accessed.
- With row partitioning on column B (option 2), all columns of only 3 rows are accessed.
- If the table is partitioned by column such that each column is in a separate column partition (option 3), then all rows of only 2 columns are accessed.
- With a mix of row and column partitioning (option 4), only 2 columns of only 3 rows need to be accessed to answer the query.

3.2 Rowid-based File System

Hybrid data partitioning in Teradata Database is achieved primarily by means of its file system, which is agnostic to the specifics of partitioning scheme. The file system is not row-based or column-based. It is *rowid*-based. A *rowid* is a fixed-length 16-byte key value that uniquely identifies a row (or part of a row) in a table. It is used by the file system to position to a specific row using an optimized 2-level B* tree [8]. Rows (or parts of rows) are always maintained on AMP in the order of their *rowids*.

The *rowid* of a row is constructed when the row is inserted in a table. The generation and structure of the *rowid* depends on whether the table has a Primary Index (PI tables), Primary Amp Index (PA tables), or neither (NoPI tables).

For a PI table[4], the *rowid* is generated based on hash value calculated on the primary index columns. It consists of 8-byte internal partition number, 4-byte row hash, and

[4]PI tables can be defined with a UPI (Unique Primary Index) or with a NUPI (Non-Unique Primary Index).

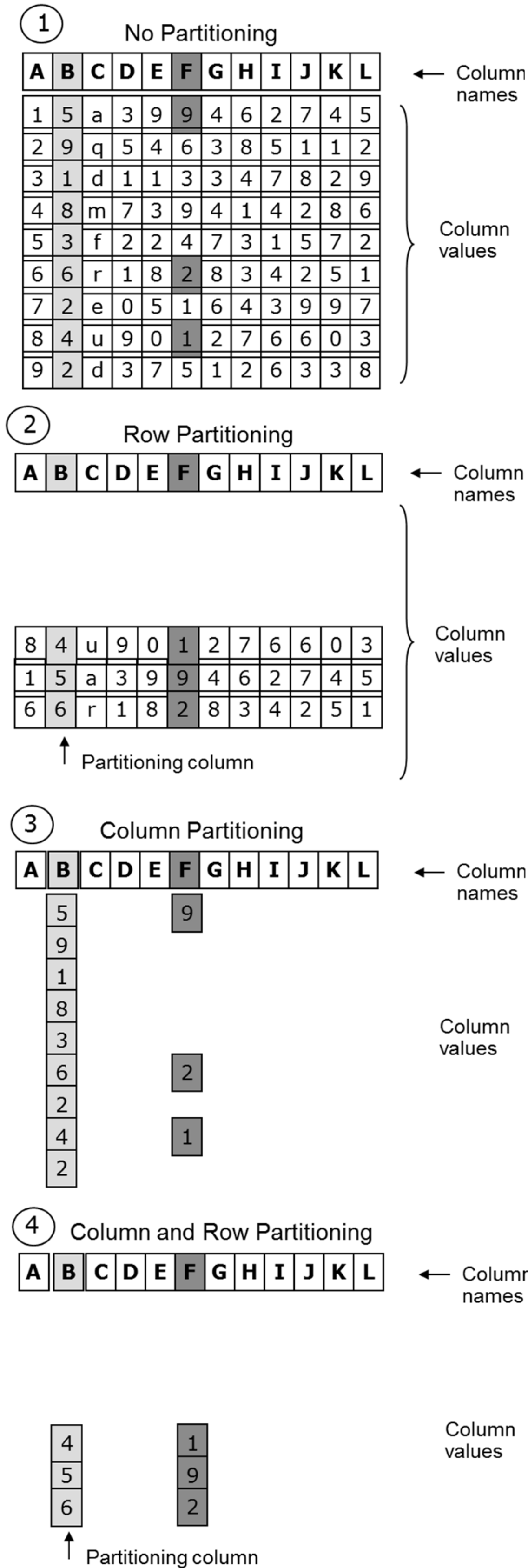


Figure 5: Partitioning Options in Teradata's Hybrid Store

4-byte uniqueness. The internal partition number may be compressed on disk to 0 bits if there is no actual partitioning on the table or to 16 bits for 2-byte partitioned tables. The high-order 20 bits of the row hash define the hash bucket. The hash bucket, in turn, determines the AMP to which the row is distributed. Within each internal partition and row hash, uniqueness is a sequence that starts at the value 1 and is incremented by 1 for each row added with the same partition number and row hash. Figure 6 illustrates the structure of *rowid* for PI tables.

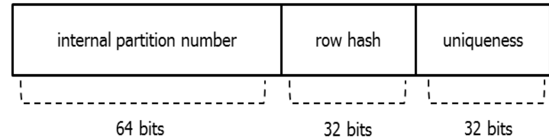


Figure 6: *rowid* of PI Tables

For PA and NoPI tables, *rowid* also has 8-byte internal partition number. But unlike PI tables, there is no row hash. Therefore, the high-order 20 bits define the hash bucket and the uniqueness uses the remaining 44 bits. The hash bucket assigned to a row of a PA and NoPI table is chosen as the first hash bucket owned by the AMP that receives the row. That 44 bits allows for a total of $\sim 17^{12}$ unique values. If the uniqueness is exhausted, the next hash bucket owned by the AMP can be used and the uniqueness is reset back to 1. Figure 7 illustrates the structure of *rowid* for PA and NoPI tables.

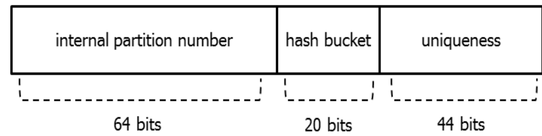


Figure 7: *rowid* of PA and NoPI Tables

Table 1 further illustrates the difference between PI, PA, and NoPI tables in terms of row distribution and ordering. Rows of PI and PA tables are hash-distributed to AMPs. Rows of NoPI tables are randomly distributed to AMPs. Once rows land on an AMP, they are always kept in the order of their *rowid*. Since the partition number is the first part of a *rowid*, rows that belong to the same partition are stored together. Within each partition, rows of a PI table are ordered by their row hash and rows of PA and NoPI tables are simply assigned the next available uniqueness and inserted in that order.

Table 1: Rows Distribution and Ordering

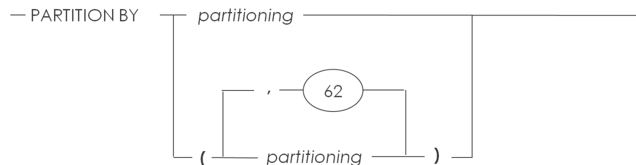
	PI Tables	PA Tables	NoPI Tables
Hash-distribution to AMP	✓	✓	X
Row-hash Ordering on AMP	✓	X	X

3.3 PARTITION BY Clause

Partitioning is effectively done at different levels. The distribution of rows to AMPs establishes the level 0 of partitioning and is implied from table definition. The middle

level specifies additional levels (1 to n) of partitioning by row, column, or both. There can be multiple levels of row partitioning but only one level of column partitioning. The partitioning is the same for each AMP. There is an additional implied level of partitioning based on the row hash at the lowest level $n + 1$. This implicit partitioning applies only to PI tables.

The middle level partitions a table by row, column, or both explicitly using the PARTITION BY clause to specify levels 1 to n of partitioning. PARTITION BY clause has the following syntax, which can be used when a table is created or altered.



where 62 is the maximum number of additional partitioning expressions.

3.4 Partitioning Expressions

Row partitioning can be specified by RANGE_N and/or CASE_N expressions using the following syntax, respectively.

```
RANGE_N(test-value BETWEEN
        range [, range]...)

CASE_N(conditional-expr1,
       ...,
       conditional-exprn)
```

Conditions are evaluated left to right until a condition results in true, unknown, or all conditions have been evaluated. If the last evaluated condition is true, the partitioning expression function returns the number of the corresponding condition, with numbering starting at 1. For range expressions, ranges must be ascending and non-overlapping.

The following is an example of defining partitions using RANGE_N that defines 84 partitions (1 to 84) based on month.

```
PARTITION BY(
             RANGE_N(
             order_date BETWEEN
             DATE '2014-01-01' AND
             DATE '2020-12-31'
             EACH INTERVAL '1' MONTH)
            )
```

An example of defining partitions using CASE_N is as follows. This example defines 6 partitions (1 to 6). The UNKNOWN partition is used assuming that the value of v can be NULL.

```
PARTITION BY(
             CASE_N(v<=0,
                    v=1 ,
                    v=2 ,
                    v=3 ,
                    v>=4,
                    UNKNOWN)
            )
```

A partitioning expression may have multiple column references and a column may be referenced in multiple partitioning expressions. Nevertheless, in practice, partitioning expressions are normally more useful when they have a single reference to a column that is not referenced in the other partitioning expressions of the PARTITION BY clause.

A table is defined as column partitioned using the following syntax.

```
COLUMN [[NO] AUTO COMPRESS]
       [[ALL BUT] (column_groupings)]
```

A column partitioned table can be defined with autocompression enabled or disabled. Without columns grouping, each column forms a separate column partition. For example, the following partitioning creates a column partitioned table with autocompression enabled and with each column in a separate column partition.

```
PARTITION BY(
             COLUMN
            )
```

The system offers the flexibility of altering tables to add, drop, or modify partitions.

3.5 Teradata Columnar

The column partitioning feature in Teradata Database is known as Teradata Columnar. This feature allows grouping columns of a table into disjoint sets of columns. Each column partition is assigned a partition number. A **c**olumn **p**artition (CP) can be single-column or multicolumn.

The physical format in which a CP is stored can be either COLUMN or ROW format. COLUMN format means CP values are packed in *containers*. In other words, each container stores a series of column partition values of a column partition. ROW format means each column partition value goes in its own *subrow*. A subrow is similar to a regular table row, but it is a subset of the columns in that row. The format of each CP can be specified when a CP table is created or altered. If the format is not specified explicitly, the system determines which format to use. The baseline assumption is that narrow CPs use COLUMN format and wide CPs use ROW format.

3.5.1 Column Format

Using COLUMN format, CP values from multiple logical rows (i.e., table rows) are packed together into a physical row. This format is particularly useful if many CP values can be packed into container. Values can be packed into container only if they have the same internal partition number and hash value (PI)/hash bucket (PA/NoPI).

COLUMN format enables row header compression and autocompression. With row header compression, it is possible to store one row header for a container instead of storing a row header for each CP value. Only the first CP value of a container row has a *rowid* stored in the row header. With autocompression, data is automatically compressed by the system as CP values (which can be multicolumn) are inserted into a container. Initially, CP values are appended without any autocompression until a container is full. Then the form of autocompression is determined for the container and the container is compressed. Compression is one of the powerful attributes of column partitioning to take advantage of [1]. Compression techniques implemented include null compression, run-length compression, value-list (dictionary) compression, and trim compression.

3.5.2 Row Format

ROW format means that each CP value is stored into a physical row with regular row format. A series of subrows with increasing *rowid*s represent a CP. This format makes row header compression and autocompression ineffective. ROW format is usually preferable over COLUMN format for PI tables since only a few values can be packed into a container. For ROW format, each CP value has a *rowid* stored with it in the row header of a physical row containing the CP value. This provides direct *rowid* access to CP values.

3.5.3 CP with PI, PA, and NoPI

A CP table can be defined with a PI, PA, or NoPI. For CP PI tables, one common scenario is to partition the table into two partitions. One partition is for frequently used (*hot*) columns and the other is for rarely used (*cold*) columns. A CP PI table may also be partitioned into three CPs in order to put columns referenced frequently in predicates in one CP, columns frequently used for projection in another CP, and other columns in a third CP. ROW format is appropriate for this kind of wide CPs.

CP PI tables provide most of the advantages of a traditional PI table (e.g., single-AMP access and local joins and aggregations) while it reduces the I/O for a wide range of queries. Autocompression is not effective in this case. There is also no row header compression for CP PI tables. In fact, there is row header expansion. If a CP PI table is partitioned into two partitions with ROW format, the number of row headers doubles. For wide rows, this overhead may be negligible. But as more CPs are added, the increase in row headers may become excessive, especially if PI values are roughly unique.

A CP with a PA is similar to PI tables in terms of row redistribution. Rows are distributed to AMPs based on hash value of PA columns. But once a row lands on an AMP, it is appended to a partition/hash bucket.

Both CP PI and CP PA tables can do a single-AMP access when the value of the PI/PA columns is specified. They can also be performant for aggregations queries with GROUP BY on PI/PA columns columns. They also allow for efficient join processing with dynamic hash join or product join when there is an equality join on PI/PA columns and the other table has the same PA/PI columns. If the other table does not have the same PA/PI columns, that table can be redistributed to AMPs instead of being duplicated on all AMPs.

A CP NoPI table does not have the aforementioned advantages of CP PA/PI tables. However, it provides faster data loading which can be further enhanced using block-level distribution. CP NoPI tables can also be useful when there is no good choice of a PI or PA column(s) in the table.

3.5.4 Rowid for CP tables

For CP tables, there is a logical and physical *rowid*. Logical *rowid* is a system-wide unique value that identifies and corresponds to a logical row (i.e., table row) in the CP table. Each CP value has the logical *rowid* of the corresponding logical row. The CP number in logical *rowid* is always 1.

The physical *rowid* is the actual *rowid* of the physical row stored in the file system. It is similar to the logical *rowid* except that it has the actual CP number. In other words, in order to position to a specific CP value, physical *rowid* can be derived from its logical counterpart by modifying the CP number in a logical *rowid* to be the actual CP number of that CP value.

3.6 Multilevel Partitioning

A table can be defined with multilevel partitioning. At any given partitioning level, the corresponding partitioning expression determines the partition number and defines how data is partitioned at that level. The subsequent partitioning expression defines how each of these partitions is subpartitioned.

The following is an example of multilevel table defined with column partitioning at the first level and row partitioning using RANGE_N at the second level.

```
CREATE TABLE Sales(TxnNo    INTEGER,
                   TxnDate  DATE,
                   ItemNo   INTEGER,
                   Quantity INTEGER)
NO PRIMARY INDEX,
PARTITION BY(
COLUMN,
RANGE_N(TxnDate BETWEEN
        DATE '2011-01-01' AND DATE '2011-12-31'
        EACH INTERVAL '1' DAY))
```

To add a new level of partitioning n, the following property must be preserved.

$$\prod_{i=1}^{n} d_i < \left(2^{63} - 1\right)$$

where d_i is the number of partitions defined at level i. 2^{63} is maximum number of partitions that can be represented using 8-byte internal partition number. If the product of the number of partitions at all levels is less than 2^{16}, the *rowid* of the partitioned table can be compressed to use 2 bytes for the internal partition number.

Multilevel partitioning has a direct impact on query performance because it typically defines a large number of partitions. If there is a large number of small *nonempty* partitions per AMP, the performance may be degraded. The order of partitioning levels is also influential. Hence, it is generally better to put a partitioning level that is more likely to get partition elimination for queries at a higher level.

Multilevel partitioning can also result in overpartitioning, which can have a negative impact on performance. Overpartitioning occurs in the presence of fine granularity in partitioning expressions. This can result in a very large number of partitions that have a small number of data blocks per AMP. For queries with coarse granularity conditions on the partitioning columns, full-table scan may be more performant over partitioning in this case.

3.7 Combined Partition Number

It is irrelevant for the file system whether a table has single level, multilevel, row, column, or hybrid partitioning. All the file system deals with as far as partitioning is concerned is the partition number in the *rowid* of a row. This number is referred to as the ***c****ombined* **p**artition **n**umber (**CPN**) due to the fact that it can represent a partition *with any format* and *at any level*. The main property of CPN is that if rows are maintained in the order of CPN, it would be the same order resulting from ordering on the value of the first partitioning expression, then on the value of second partitioning expression, etc.

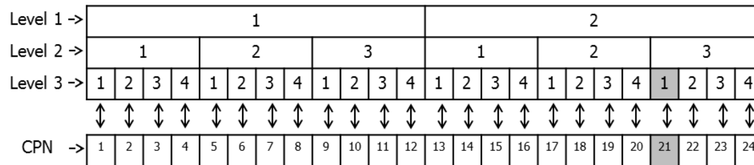


Figure 8: An Illustration of Combined Partition Number

The CPN of partition p at level n is calculated as follows:

$$\left(\sum_{i=1}^{n-1}\left((p_i - 1) \times \prod_{j=i+1}^{n} d_j\right)\right) + p_n$$

where p_i is the number assigned to the partition at level i and d_i is the number of partitions defined at level i.

Consider an example with 2 partitions for the first level, 3 partitions for the second level, and 4 partitions for the third level. The CPN of partition (2,3,1) highlighted in Figure 8 is calculated as $(1 \times 12) + (2 \times 4) + 1 = 21$.

As mentioned above in Section 3.2, rows are stored in the order of *rowid*. Since the internal partition number comes first in a *rowid* (recall Figure 6 and Figure 7), all physical rows within the same combined partition are stored together in the same or adjacent data blocks.

An empty combined partition is not assigned any data blocks and it takes no space in the system. However, it may be required to access and read one data block to determine that a partition does not actually have any rows.

4. QUERY OPTIMIZATION

Data partitioning gives the opportunity for partition-based optimizations. In this section, we discuss optimization techniques implemented in Teradata's query optimizer. Some optimizations are applicable to any form of partitioning while others pertain specifically to either row or column partitioning.

The basic optimization over partitioned tables is **S**tatic **P**artition **E**limination (SPE), which is widely implemented in database systems [13, 18, 20]. It determines the set of partitions that need to be accessed in order to answer a specific query with the smallest overall cost. SPE applies to row and column partitioning. For row partitions, SPE is based on a single-point, range, or in-list conditions specified on the partitioning column(s). For column partitions, it is based on columns referenced in query. SPE is applied to each level independently and the result is combined into a single partition elimination list to further reduce the size of data that need to be scanned. For an index access that uses a secondary index, SPE can be applied to *rowids* of the index rows if there are equality, range, or in-list conditions on the partitioning columns. SPE can also be useful for other DML operations such as deletes by enabling full partition deletes.

4.1 Optimizations for Row-level Partitioning

This section presents optimization techniques that are applicable to row partitions.

4.1.1 Dynamic Partition Elimination

If the range or list of values that defines the partitions of interest are not explicitly specified, but can be indirectly implied via a join condition with another table, Dynamic Partition Elimination (DPE) is considered. DPE is an optimization technique that determines the relevant partition(s) to join at run time when value of the join column(s) in the other table is known.

4.1.2 Partition-aware Merge Join

If the primary index column(s) is not the same or is not part of the row partitioning column(s), rows with same PI values can be scattered across multiple partitions. Even though rows within each partition are ordered by hash of the PI, traditional merge join with the PI of such row-partitioned table would require the table to be materialized first so that the rows can be in one sorted order of the PI hash. To address this cases, Teradata supports partitioning-aware flavors of merge join.

- *Sliding-window merge join:* This variation does not require the extra spooling and sorting. Sliding-window merge join manages the join with multiple partitions at a time. A partitioned read that allocates one file context per partition is used to read across the multiple partitions so that join processing returns rows in one sorted order of the PI hash. An additional pass of sliding-window join is done if the number of partitions to join with exceeds the maximum number of file contexts. While sliding-window joins can be done (with SPE or DPE) in one or very few passes and can be effective with a small number of partitions, they may not be favorable when there is a large number of small combined partitions.

- *Rowkey-based merge join:* Teradata's optimizer also supports another partition-to-partition flavor of merge join referred to as *rowkey-based merge join*. This technique is applicable when there is equality binding conditions on PI and partitioning columns of two row-partitioned tables that have the same PI and partitioning expression(s). Rowkey refers to the internal partition number and the rowhash part of a *rowid*. Rowkey-based merge join can still be used if one of the tables is not bound on PI and partitioning columns. In this case, that table is spooled and sorted into a partitioned spool based on the PI and partitioning expression of the other table.

4.2 Optimizations for Column-level Partitioning

In this section, we discuss optimization techniques on CP tables.

4.2.1 Late Materialization

This optimization technique is inherent in Teradata Columnar implementation. A column value is not materialized until it is needed.

Consider the following simple query:

```
SELECT Col1, Col2
FROM   CPTbl
WHERE  Col3 > 20 and Col4 = 10
```

Let's say the two predicates are evaluated in the order they are specified in and assume the CPTbl is read using the equivalence of a full-table read. First, the CP containing Col3 is scanned for the first column value that qualifies the predicate "Col3 > 20". A logical *rowid* is computed based on position of the qualified value. Col4 of the qualified logical row is then read to evaluate the predicate "Col4 = 10". This means Col4 values of those rows that do not satisfy "Col3 > 20" are not read. Similarly, only those Col1 values and Col2 values that correspond to the rows that satisfy both predicates are read.

4.2.2 Predicate Ordering

When there are multiple predicates involving multiple CP, the optimizer uses a *one-lookahead* algorithm to find a predicates evaluation order that yield the smallest CPU + IO. The CPU cost of a predicate CP depends on the complexity of the associated predicate while the IO cost depends on column size and compression ratio. Both depend on selectivity of the previous predicate CP which determines the number of values to read in a subsequent CP.

4.2.3 Single-table Access Path

Optimizer supports different *access methods*. One example is scanning each predicate CP independently to produce a bitmap of the qualified rows and then construct the conjunction or disjunction of the multiple bitmaps. The final bitmap serves as an index to the qualified rows.

4.2.4 CP Joins

A logical join step involving a CP table can be decomposed into semantically-equivalent multiple physical join steps. We explain the CP join technique using the following table definitions and example query.

Table *cpt1* is a CP table with single column partitioning (i.e., each column is in a column partition), specified with PARTITION BY COLUMN.

```
CREATE TABLE cpt1(a1 INTEGER,
                  b1 INTEGER,
                  c1 INTEGER,
                  d1 INTEGER,
                  e1 INTEGER)
NO PRIMARY INDEX
PARTITION BY COLUMN
```

Table *t2* is a regular table with a primary index on *a2* and a secondary index on *c2*. Rows in *t2* are distributed to different AMPs based on *a2* values (unlike *cpt1*, whose rows are distributed randomly). The secondary index on *c2* defines an access path to the base table.

```
CREATE SET TABLE t2(a2 INTEGER,
                    b2 INTEGER,
                    c2 INTEGER,
                    d2 INTEGER,
                    e2 INTEGER)
PRIMARY INDEX (a2),
INDEX (c2)
```

The following example query projects all columns with a join condition on *c1* from *cpt1* and *c2* from *t2*:

```
SELECT *
FROM   cpt1, t2
WHERE  c1=c2;
```

The above query can be executed with one of the following methods:

- **1-Step CP Join:** The baseline is to join *cpt1* and *t2* in *one* step. Because the join condition is not on primary index, in preparation of the join step, one table must be duplicated to all AMPs or both tables need to be redistributed based on the hash of the join column. Assuming both table are redistributed, the join plan looks as show in Figure 9. All columns are spooled from *cpt1* and *t2* into *spool1* and *spool2*, respectively.[5] Then both spools are redistributed by the hash code of the join column and joined with a join condition of "c1=c2".

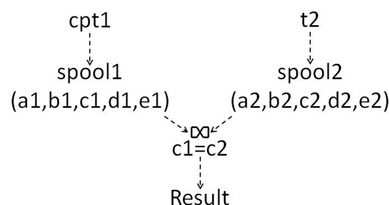


Figure 9: 1-Step Join

- **2-Step CP Join:** The downside of the 1-step CP join is in the cost of spooling and redistributing *all* column partitions (for columns *a1*, *b1*, *c1*, *d1*, and *e1*) from the CP table, although only one column partition (for column *c1*) need to be consumed in the join step. This cost becomes more significant as the number and size of column partitions increase. This overhead can be avoided by breaking down the join step into *two* steps as illustrated in Figure 10. Initially, only CPs containing join columns are spooled from *cpt1* into *spool1*. In the first join step, *spool1* and *spool2* are joined with a join condition of "c1=c2" producing a rowid *spool3* that contains the *rowid* of qualifying rows from *cpt1*. Then in the second join step, the rowid *spool3* is joined back to the remaining four column partitions directly from the CP table using a rowid join.[6]

- **3-Step CP Join:** The first join step of the 2-step CP join can be decomposed further into two join steps resulting in a sequence of *three* join steps as shown in Figure 11. Table *t2* has an index on *c2*. In Teradata, this index is physically stored as an *index subtable* that contains the index value along with corresponding *rowids*. With 3-step CP join, the first join step is to join *spool1* to the index subtable with a join condition of "c1=c2". In Teradata terminology, this join method is referred to as *nested join*. The outcome of the nested join is a rowid *spool2*. The second join step joins *spool2* back to *t1* to retrieve qualifying rows in their entirety using

[5] Spools are intermediate/buffer tables.

[6] The rowid join uses the *rowid* from the spool to locate matching row from the base table directly in the file system.

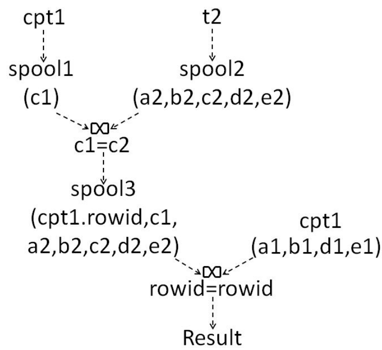


Figure 10: 2-Step Join

a rowid join. The outcome of this rowid join is yet another rowid *spool3*. The third join step joins *spool3* back to the rest of column partitions from the CP base table using a rowid join.

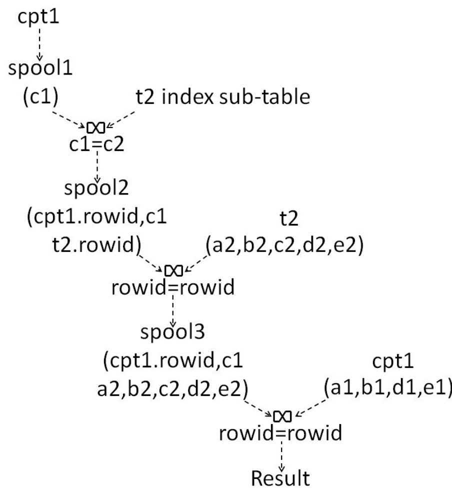


Figure 11: 3-Step Join

5. EXPERIMENTS

This section presents a performance study on partitioned tables. The main purpose of the experiments is to observe the impact of different column options. The performance impact of different optimization techniques such as DPE and CP joins is an interesting point to examine but it is beyond the scope of this paper.

5.1 Experiments Setup

We ran experiments on Teradata 6650 Enterprise Data Warehouse (EDW) platform. The system has 3 nodes (+1 standby) and each node has 42 AMPs. We used the TPC-H benchmark [28] database of 1 TB size. Experiments were executed against Orders and Lineitems tables using five different variations of table design as shown in Table 2.

Experiments results are reported for I/O counts, CPU time, and elapsed times. The values of these metrics were captured by Terdata's DBQL (DataBase Query Log). I/O count is the total logical I/O. CPU time is the total (i.e., sum) CPU time of all AMPs and is measured in seconds. Elapsed time is the duration between the start time of a query and its first response time and is also measured in seconds.

Table 2: Design of Tables Used in Experiments

PI	Regular PI table with no partitioning
RPPI	Partitioned primary index table with 84 monthly row partitions
CP	NoPI Single-column CP table with COLUMN format and autocompression
CRP	Multilevel partitioned table with column partitioning at the first level and row partitioning (by month) at the second level
RCP	Multilevel partitioned table with row partitioning (by month) at the first level and column partitioning at the second level

5.2 Experiments Results

5.2.1 Table Size

Figures 12 and 13 show the total size of Lineitems and Orders tables, respectively. Results show that column partitioned tables (with and without row partitioning) are about 50% smaller in size on average. This result is attributed to autocompression and row header compression that help reduce the space required to store the data.

The small additional reduction in the size of CRP and RCP tables is because row partitioning sorts and stores data based on the date column of Lineitems and Orders tables, which makes that column benefits from run-length compression. Run-length is a compression technique that compresses the same values that appear consecutively in a CP container. It is most effective for CP tables if CP values are ordered by the column(s) of the CP.

The size of RPPI tables is somewhat larger than the size of PI tables. This is because the row header of RPPI tables has extra bytes to store internal partition number. Since PI tables are not partitioned, the internal partition number in row header is compressed to 0 bytes.

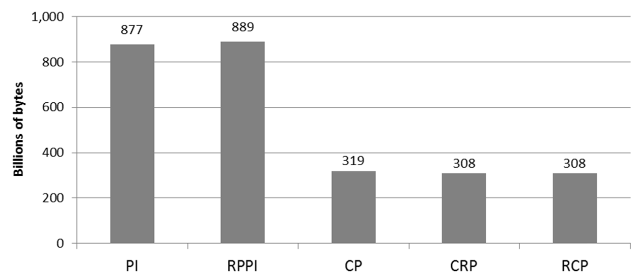


Figure 12: Size of Lineitems Table

5.2.2 Full-Table Scan

This test uses a simple query that retrieves all the rows of all columns of the Lineitems. Table 3 shows the result of this experiment. While CP tables have smaller I/O count, they have significantly larger CPU time (and elapsed time accordingly). The additional CPU consumption comes from the process of bringing all CPs together to reconstruct each row in the result set.

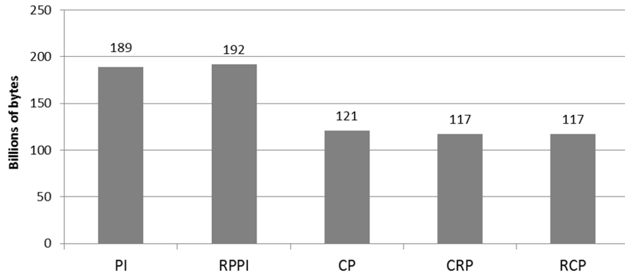


Figure 13: Size of Orders Table

This test suggests that these kind of queries are not suited to column partitioning because there is no CP elimination. This is particularly true on CPU-bound systems. On an I/O-bound system, however, the decreased I/O may be beneficial to reduce the overall elapsed time.

Table 3: Full-Table Scan Comparison

Table	I/O Count	CPU Time	Elapsed Time
PI	32,771,499	26,417	562
RPPI	32,957,752	26,476	558
CP	24,205,741	46,736	812
CRP	24,082,499	46,574	856
RCP	24,494,340	45,718	844

5.2.3 Aggregation

In this experiment, we ran an aggregation query on the Lineitems table with 2 GROUP BY columns. The query has three variations. In the first variation, only 3 of the 16 columns in the table are accessed. In the second variation, 9 columns are accessed. In the third variation, 15 columns are accessed.

Figure 14 shows elapsed time. Results show that for PI and RPPI tables, the elapsed time is nearly the same regardless the number of columns accessed. This is expected because with these table formats, the whole row is always accessed. For CP, CRP, and RCP tables, however, there is longer elapsed time as more columns are accessed.

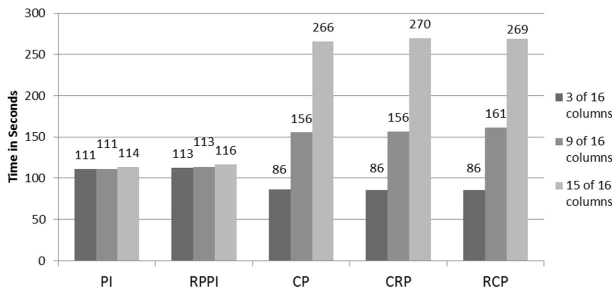


Figure 14: Elapsed Time for Aggregation

Tables 4, 5, and 6 list the breakdown of performance metric for all variations of the aggregation query.

5.2.4 Joins

In this test, the Orders and Lineitems tables are joined on the PI column of the PI and RPPI tables. Results in Figure

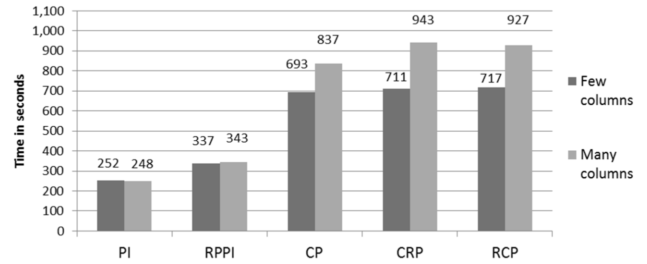


Figure 15: Elapsed Time for Join

Table 4: First Variation of Aggregation Query

Table	I/O Count	CPU Time	Elapsed Time
PI	13,578,067	2,597	111
RPPI	13,764,320	2,629	113
CP	173,833	5,941	86
CRP	180,117	5,900	86
RCP	250,996	5,953	86

15 demonstrate that this kind of query performs better on tables with a PI. Since the CP tables in the experiments are NoPI tables, they are not suitable for this query. Adding a PI/PA on the join column can help CP tables perform better in this case.

Tables 7 show the breakdown of performance metrics for join query.

One interesting observation is that the I/O count is about the same for PI table as well as for CP, CRP, and RCP tables, whereas the CPU is almost 4 times higher for the CP tables. The overhead in CPU time is due to row redistribution and sorting required on NoPI tables prior to perform joins.

The high I/O count for the RPPI case is due to the presence of too many row partitions that does not honor sliding-window merge join directly between the Orders and Items tables. In this case, both tables are spooled, the spools sorted on the join columns, and a merge join is done between the spools.

5.2.5 Rollup Operation

This experiment runs a complex rollup query against the Lineitems table. The query includes many functions in the SELECT clause, such as KURTOSIS and SKEW functions. The query is executed with 4 different variations. First variation “Few columns, all rows” references 4 columns and all rows qualify. Second variation “Few columns, 1 month” is similar to the first one but only one month out of 7 years of data is requested. The third variation “Many columns, all rows” accesses 14 out of the 16 columns in the table with all rows qualify. The last variation “Many columns, 1 month” is similar to the third one but only one month of out of 7 years of data is requested.

Table 8 understandably shows that with only 4 columns accessed, CP, CRP, and RCP tables overall perform better than PI and RPPI tables. As concluded in full-table scan comparison, PI and RPPI tables incur the overhead of reading all the columns even if they are not used in the query.

Table 9 shows that query that references few columns for one month runs significantly faster with CRP and RCP tables because of column and row partition eliminations.

Table 5: Second Variation of Aggregation Query

Table	I/O Count	CPU Time	Elapsed Time
PI	13,578,067	2,787	111
RPPI	13,764,320	2,827	113
CP	1,388,846	10,679	156
CRP	1,632,828	10,560	156
RCP	1,858,108	10,143	161

Table 6: Third Variation of Aggregation Query

Table	I/O Count	CPU Time	Elapsed Time
PI	13,578,067	5,096	114
RPPI	13,764,320	5,116	116
CP	2,697,405	18,491	266
CRP	2,541,000	18,458	270
RCP	2,924,652	18,465	269

RPPI table also runs fast due to row partition elimination.

Results of the other two variations of the rollup query in Table 10 and Table 11 can be explained similarly. In all these variations, the increase in CPU time for CP tables is due to the overhead of reconstructing table row in the result set.

6. RELATED WORK

All major database proprietary support row partitioning, including IBM DB2 [13], Microsoft SQL Server [18], and Oracle [20]. Some other commercial systems offer column-oriented solutions [2]. Sybase IQ [27] is one of the first commercially available columnar relational database management systems. Vertica [16] is an MPP analytical database engine with columnar storage features. It is the commercialization of the C-Store project [26]. SAP HANA [10] is an in-memory database that is particularly optimized for column-based storage. Infobright [14] is yet another database system with columnar architecture, which comes with advanced compression capabilities in its Brighthouse [24] data warehousing analytical platform. Some open-source database systems like Druid [9] and MonetDB [19] are even built as column-store platforms. PostgreSQL [22] also has columnar store extension. While some of the aforementioned systems have a mix of row and column solutions (e.g., SAP HANA), Teradata extends a native and unique spectrum for data partitioning starting with traditional row-partitioned tables from one end to a true columnar database on the other end.

A great deal and long history of work on query optimization that pertains to row-partitioned tables have been proposed and presented. In a recent work from academia, Herodotou et al. in [12] introduced partition-aware multi-way join techniques over partitioned tables. The proposed algorithms were implemented in the query optimizer of PostgreSQL as a proof of concept. The recent work from industry in [5] introduced an algebraic representation of partitioned tables and operations applicable to them, which can be used for SPE and DPE in a unified framework. Main optimization techniques discussed in [5] address multilevel partitioning and they are primarily based on the DPE paradigm. While Teradata employs some of the optimization techniques discussed in [12] and [5] like SPE and DPE, it has some distinctive partitioning-based optimizations such as multistep CP joins.

Table 7: Join Query

Table	I/O Count	CPU Time	Elapsed Time
PI	25,709,671	8,804	252
RPPI	39,411,626	15,900	337
CP	24,419,299	35,757	693
CRP	25,147,819	37,440	711
RCP	25,341,401	37,457	717

Table 8: Rollup Query with Few Columns and All Rows

Table	I/O Count	CPU Time	Elapsed Time
PI	13,578,001	1,141	111
RPPI	13,764,254	1,153	113
CP	1,371,560	2,890	42
CRP	754,132	2,858	42
RCP	876,867	2,775	42

7. CONCLUSIONS & FUTURE WORK

In this paper, we introduced a unique hybrid row-column store solution implemented in the Teradata Parallel Database. We explained how hybrid partitioning is achieved seamlessly using the underlying file system. We discussed various optimization techniques that takes advantage of different partitioning alternatives. Finally, we presented the result of a performance study the shows the impact of different partitioning options.

Part of our plan for future work is to examine other performance attributes of different partitioning combinations. This includes deeper study on tables with PA and CP tables with ROW format. We also plan to enhance SPE, DPE, and CP joins with new optimizations. Another avenue for future work is designer tools for picking the best partitioning for tables used in a workload, which is an important issue for physical database design [3, 17] that becomes even more challenging in shared-nothing architecture [21, 23, 30].

8. ACKNOWLEDGMENTS

We thank Ramesh Bhashyam, Steve Cohen, Sanjay Nair, Donald Pederson, Ranjan Priyadarshi, and all members of Teradata Labs who worked on column-row store feature for their contributions.

9. REFERENCES

[1] D. Abadi, S. Madden, and M. Ferreira. Integrating Compression and Execution in Column-oriented Database Systems. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD '06, pages 671–682, 2006.

[2] D. J. Abadi, P. A. Boncz, and S. Harizopoulos. Column-oriented Database Systems. *Proc. VLDB Endow.*, 2(2):1664–1665, Aug. 2009.

[3] S. Agrawal, V. Narasayya, and B. Yang. Integrating Vertical and Horizontal Partitioning into Automated Physical Database Design. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, SIGMOD '04, pages 359–370. ACM, 2004.

[4] M. Al-Kateb, A. Ghazal, A. Crolotte, R. Bhashyam, J. Chimanchode, and S. P. Pakala. Temporal Query Processing in Teradata. In *Proceedings of the 16th International Conference on Extending Database Technology*, EDBT '13, pages 573–578. ACM, 2013.

Table 9: Rollup Query with Few Columns for One Month

Table	I/O Count	CPU Time	Elapsed Time
PI	13,578,001	945	111
RPPI	172,663	21	2
CP	1,090,431	1,963	30
CRP	10,006	56	1
RCP	10,552	57	1

Table 10: Rollup Query with Many Columns and All Rows

Table	I/O Count	CPU Time	Elapsed Time
PI	13,578,001	1,421	111
RPPI	13,764,254	1,438	113
CP	4,089,172	5,143	75
CRP	3,434,071	5,045	75
RCP	3,783,586	4,959	74

Table 11: Rollup Query with Many Columns for One Month

Table	I/O Count	CPU Time	Elapsed Time
PI	13,578,001	1,110	111
RPPI	178,355	26	2
CP	3,256,328	2,114	32
CRP	47,492	88	2
RCP	48,472	88	2

[5] L. Antova, A. El-Helw, M. A. Soliman, Z. Gu, M. Petropoulos, and F. Waas. Optimizing Queries over Partitioned Tables in MPP Systems. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 373–384. ACM, 2014.

[6] C. Ballinger and R. Fryer. Born to Be Parallel: Why Parallel Origins Give Teradata An Enduring Performance Edge. *IEEE Data Eng. Bull.*, 20(2):3–12, 1997.

[7] J. Catozzi and S. Rabinovici. Operating System Extensions for The Teradata Parallel VLDB. In *Proceedings of the 27th International Conference on Very Large Data Bases*, VLDB '01, pages 679–682, 2001.

[8] D. Comer. Ubiquitous B-Tree. *ACM Comput. Surv.*, 11(2):121–137, June 1979.

[9] Druid. Druid. `http://druid.io/`.

[10] F. Färber, N. May, W. Lehner, P. Große, I. Müller, H. Rauhe, and J. Dees. The SAP HANA database – an architecture overview. *IEEE Data Eng. Bull.*, 35(1):28–33, 2012.

[11] A. Ghazal, D. Y. Seid, A. Crolotte, and B. McKenna. Exploiting Interactions Among Query Rewrite Rules in The Teradata DBMS. In *Database and Expert Systems Applications, 19th International Conference, DEXA 2008, Turin, Italy, September 1-5, 2008. Proceedings*, pages 596–609, 2008.

[12] H. Herodotou, N. Borisov, and S. Babu. Query Optimization Techniques for Partitioned Tables. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, pages 49–60. ACM, 2011.

[13] IBM. DB2 Partitioned Tables. `http://www.ibmbigdatahub.com/blog/creating-and-using-partitioned-tables`.

[14] Infobright. Infobright. `https://infobright.com/`.

[15] E. P. Jones, D. J. Abadi, and S. Madden. Low Overhead Concurrency Control for Partitioned Main Memory Databases. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 603–614, 2010.

[16] A. Lamb, M. Fuller, R. Varadarajan, N. Tran, B. Vandiver, L. Doshi, and C. Bear. The Vertica Analytic Database: C-store 7 Years Later. *Proc. VLDB Endow.*, 5(12):1790–1801, Aug. 2012.

[17] S. T. March and D. G. Serverance. The Determination of Efficient Record Segmentations and Blocking Factors for Shared Data Files. *ACM Trans. Database Syst.*, 2(3):279–296, Sept. 1977.

[18] Microsoft. Partitioned Table and Index Strategies Using SQL Server 2008. `https://technet.microsoft.com/en-us/library/dd578580(v=sql.100).aspx`.

[19] MonetDB. MonetDB. `https://www.monetdb.org/Home`.

[20] Oracle. Oracle Database VLDB and Partitioning Guide. `https://docs.oracle.com/cd/E18283_01/server.112/e16541.pdf`.

[21] A. Pavlo, C. Curino, and S. Zdonik. Skew-aware Automatic Database Partitioning in Shared-nothing, Parallel OLTP Systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 61–72, 2012.

[22] PostgreSQL. Postgre SQL. `http://www.postgresql.org/`.

[23] J. Rao, C. Zhang, N. Megiddo, and G. Lohman. Automating Physical Database Design in A Parallel Database. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, SIGMOD '02, pages 558–569. ACM, 2002.

[24] D. Ślęzak, J. Wróblewski, V. Eastwood, and P. Synak. Brighthouse: An Analytic Data Warehouse for Ad-hoc Queries. *Proc. VLDB Endow.*, 1(2):1337–1345, Aug. 2008.

[25] M. Stonebraker. The Case for Shared Nothing. *IEEE Database Eng. Bull.*, 9(1):4–9, 1986.

[26] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. J. O'Neil, P. E. O'Neil, A. Rasin, N. Tran, and S. B. Zdonik. C-Store: A Column-oriented DBMS. In *Proceedings of the 31st International Conference on Very Large Data Bases, Trondheim, Norway, August 30 - September 2, 2005*, pages 553–564, 2005.

[27] Sybase. Sybase IQ. `http://go.sap.com/product/data-mgmt/sybase-iq-big-data-management.html`.

[28] TPC. TPC-H Benchmark. `http://www.tpc.org/tpch/spec/tpch2.14.4.pdf`.

[29] E. Zamanian, C. Binnig, and A. Salama. Locality-aware Partitioning in Parallel Database Systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 17–30, 2015.

[30] D. C. Zilio, A. Jhingran, and S. Padmanabhan. Partitioning Key Selection for A Shared-nothing Parallel Database System. In *IBM Research Report RC*, 1994.