InfiniDB®
Performance Tuning Guide

Release: 4.6
Document Version: 4.6-1

Contents

1 Introduction

Welcome to the InfiniDB Performance Tuning Guide. When viewed from the perspective of a developer or DBA experienced with row-based databases there are some InfiniDB operations that map to traditional database operations and some that do not have a direct corollary. In addition, there are some underlying operations common to traditional row-based DBMS systems that do not exist within InfiniDB (e.g. InfiniDB does not execute a full table scan).

The purpose of this guide is to help you tune InfiniDB, which is a column-oriented RDBMS that allows for significant parallelization and scalability against large volumes of data with significantly reduced I/O requirements.

Operations available in the InfiniDB Analytic Database have been optimized for scanning, joining, and aggregating significant amounts of data in a multi-threaded and optionally distributed system with excellent performance characteristics. Significant amounts of data as used here can range from 100’s of millions of rows to 100’s of billions of rows, and additionally from 100’s of gigabytes to 100’s of terabytes of data. InfiniDB has been designed to support scaling well beyond these numbers. InfiniDB also provides excellent performance at smaller scale.

1.1 Audience

This guide is intended for a number of different roles, including:

- Database Administrators
- Application and Web Developers
- Data Architects
- System and Web Administrators

1.2 List of Documentation

The InfiniDB Database Platform documentation consists of several guides intended for different audiences. The documentation is described in the following table:

Document	Description
InfiniDB Administrator’s Guide	Provides detailed steps for maintaining InfiniDB.
InfiniDB Apache Hadoop™ Configuration Guide	Installation and Administration of an InfiniDB for Apache Hadoop system.
InfiniDB Concepts Guide	Introduction to the InfiniDB analytic database.
InfiniDB Minimum Recommended Technical Specifications	Lists the minimum recommended hardware and software specifications for implementing InfiniDB.
InfiniDB Installation Guide	Contains a summary of steps needed to perform an install of InfiniDB.
InfiniDB Multiple UM Configuration Guide	Provides information for configuring multiple User Modules.
InfiniDB SQL Syntax Guide	Provides syntax native to InfiniDB.

InfiniDB Windows Installation and Administrator's Guide	Provides information for installing and maintaining InfiniDB for Windows.

1.3 Obtaining documentation

These guides reside on our http://www.infinidb.co website. Contact support@infinidb.co for any additional assistance.

1.4 Documentation feedback

We encourage feedback, comments, and suggestions so that we can improve our documentation. Send comments to support@infinidb.co along with the document name, version, comments, and page numbers.

1.5 Additional resources

If you need help installing, tuning, or querying your data with InfiniDB, you can contact support@infinidb.co.

2 Performance Tuning Overview

2.1 InfiniDB Architectural Goals

There are a number of core architectural goals that have contributed to design decisions associated with performance. InfiniDB goals include:

1. Significantly reduce I/O costs for queries, both for memory accesses as well as storage accesses.
    - Fundamentally reduce the I/O required for most queries against big data.
    - Eliminate random access to blocks (pages) of data.
    - Implement a global data buffer cache that enables partial caching or full caching of large data sets.
2. Enable database operations to execute in parallel with minimal synchronization.
    - Eliminate synchronization around reading blocks of data.
    - Minimize thread synchronization issues through a number or mechanisms including queues feeding and receiving from significant thread pools.
3. Define units of work for the parallel database operations that:
    - Work well for different storage systems.
    - Allow a Performance Module to routinely run at or near 100% of CPU utilization.
    - Eliminate dedicating threads to queries, i.e. small queries execute quickly even under load.
4. Minimize data shipping costs, i.e. minimize piping data between local or distributed threads.
    - Send the operation to the data rather than shipping data where possible.
    - Set up operations to avoid shipping costs for the largest table where possible.
5. Minimize the tuning requirements and deliver performance for ad-hoc analysis of detailed data

Note that, at this point in time, minimal elapsed time for a single row lookup is not among the primary goals. The performance for a single row lookup from a billion rows may still be accomplished in a fraction of a second, but may be less efficient than a traditional index/ table combination.

2.2 Join Acceleration via InfiniDB

InfiniDB accomplishes joins internal to the InfiniDB engine rather than via traditional MySQL joins. These joins:

1. Are optimized for millions or billions of rows via hash join operations
2. Are multi-threaded and distributed across a scalable thread pool.
3. Stream a large fact table in one pass against an arbitrary number of dimension tables without materializing intermediate results.
4. Feed into aggregation operations that may significantly reduce data transmission costs.

Performance tuning guidelines within this document apply to joins and other operations executed solely within the InfiniDB Analytic Database.

2.3 Syntax Note - Not all syntax currently supported

InfiniDB does not yet have coverage for all available syntax with full performance capabilities. Please refer the syntax guide for supported syntax. As new features are added, these extensions to the syntax will aggressively leverage the multi-threaded and scalable performance as seen for other operations.

Additional syntax is available now through alternate configuration, i.e. configuring InfiniDB as a standard storage engine. However, this comes at a cost of eliminating multi-threaded/distributed join and aggregation operations.

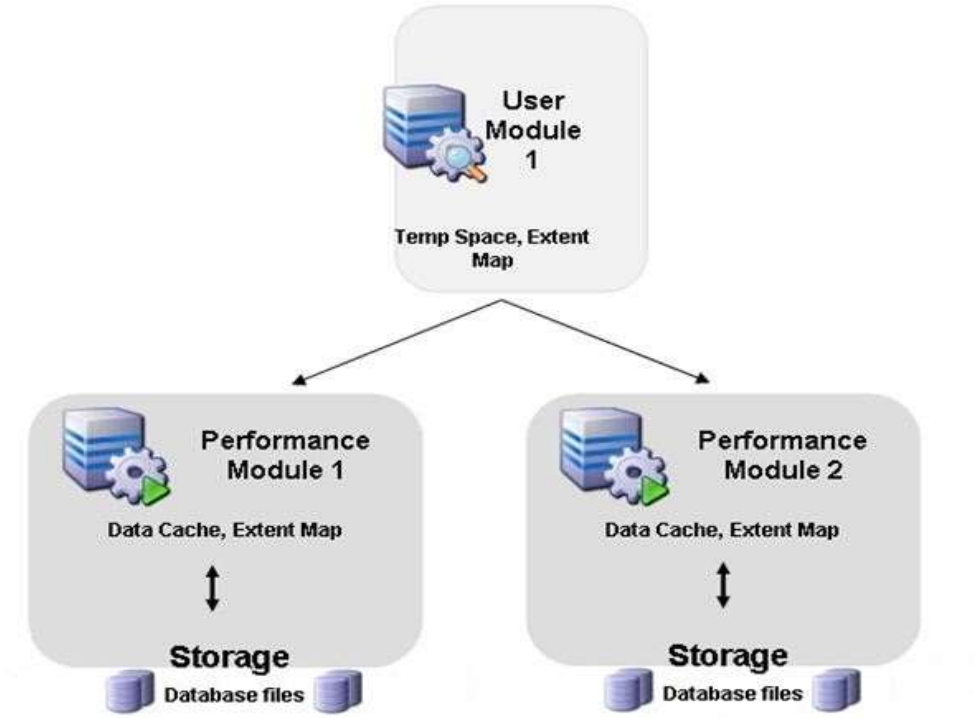
2.4 The InfiniDB Primary Components

InfiniDB provides a modular architecture that consists of three main components, all of which work together to comprise an InfiniDB instance. These components include:

- **User Module (UM)**: The User Module is responsible for breaking down SQL requests and distributing the various parts to one or more Performance Modules that actually retrieve requested data from either memory caches or disk. State for any single query is maintained on a single User Module.
- **Performance Module (PM)**: The Performance Module executes granular units of work received from a User Module in a multi-threaded manner. InfiniDB additionally allows distribution of the work across many Performance Modules.
- **Storage**: InfiniDB can use either local storage or shared storage (e.g. SAN) to store data. A user can have only a single server act as an InfiniDB server and have everything configured and running on that one server, or they can scale out with multiple servers.

In addition, InfiniDB persists meta-data about each column in a shared object referred to as the Extent Map. The Extent Map is referenced by the User Module when determining what operations to issue against which data, and further referenced by the Performance Module if needed to read blocks from disk. Each column is made up of one or more files, and each file can contain multiple Extents (generally contiguous allocations of data) that are also tracked within the Extent Map.

A UM understands a query plan, while the PM(s) understand data blocks and operations. The Extent Map enables this abstraction.



Simple representation of a two Performance Module installation.

2.5 *InfiniDB Analytic Database Editions*

InfiniDB is a column-oriented database purpose built to load and query big data and is available in both open source and commercial editions. All editions of InfiniDB include the following features important for performance (the full feature set description is available in other documentation):

- **Column-oriented architecture**: InfiniDB stores data by column rather than by row, allowing any scan operation to ignore columns not part of the scan, and select column operations to ignore columns not referenced in the query.
- **Multi-threaded design**: InfiniDB distributes work in support of queries at a very granular level and structures most queries to allow these operations to execute with minimal or no synchronization. The work is automatically mapped and distributed to all Performance Modules with the results reduced/returned to the calling user module.
- **Automatic vertical and horizontal partitioning**: In addition to the significant reduction in I/O based on column storage, InfiniDB creates small Extents (think partitions) for most columns and stores some meta-data data to allow for Extent elimination to occur under some circumstances.
- **Flexible concurrency**: InfiniDB allows for concurrent access to the data, allowing large queries to consume all available Performance Module resources for small moments of time, while still allowing smaller queries to execute without waiting for the larger query to finish.
- **Massive parallel processing (MPP) capable**: InfiniDB can use multiple commodity hardware machines to achieve near linear increases in overall performance. Multiple Performance Modules will allow for distribution of granular work across all threads on all available Performance Modules.

- **Distributed shared-nothing data cache**: In a multiple-node InfiniDB configuration, data is distributed among the various nodes and their data caches. No node shares data with the other, however all are accessed in the InfiniDB MPP architecture when data is read to satisfy queries. In essence then, InfiniDB creates one large logical data cache that is accessed in a distributed fashion in parallel by all participating nodes. This allows InfiniDB to literally cache large databases when enough nodes are present.

3 InfiniDB Distributed Processing Model

The InfiniDB job issue process can be related to a Map/Reduce metaphor. The UM issues operations to a (potentially distributed) thread pool that independently execute database operations (filter, aggregate, join, etc.) against the data and then return a (potentially) reduced data set to the UM for any additional operations.

However, there are some differences between InfiniDB and Map/Reduce. The jobs (referred to as primitive operations) are already mapped to SQL syntax and there is no external API available, so this is not a toolkit or a software call. In addition, there are differences in the job scheduling processes.

3.1 Work Granularity of User Module to Performance Module Requests

Requests for database operations are issued by the User Module to one or more Performance Modules to execute scans against ranges of data blocks containing 8 million rows known as Extents. If the data is not already cached, the request may incur a read from storage in support of that operation. To maximize multi-block read capabilities of the system, all blocks within one Extent are mapped and issued to a target PM. In addition, any other queries that also involve data from that same Extent will also be sent to the same PM. Of course, if the number of Performance Modules increases, then the re-mapping will quickly occur to allow new request to include the any newly added PMs. Even for non-distributed systems, execution of operations at this granularity enhances spatial and temporal locality of block touches, i.e. adjacent blocks are read together and operated on within a narrow range of time

Note that for any partially filled Extents within larger tables, and for the partial (and only) Extent for smaller tables, operations are only requested up to a high water mark (HWM) within each Extent.

The default and tested configuration for an Extent is 8 million rows. This allocation size on disk varies between 8 and 64MB of data on disk per Extent per column. Note that the value is set at system installation time, is global for a given InfiniDB instance, and cannot be changed after startup. Behavior for a new instance may be set by a Calpont.xml parameter provided separate from this document, but would require additional validation and verification.

3.2 Work Granularity within a Performance Module (Batch Primitive)

Database operations within a Performance Modules are resolved in a much more granular method, allowing individual threads to operate independently on individual blocks of data within an Extent. The smallest operation is also known as a Primitive, and indeed the software process running on each PM responsible for Primitive Processing is PrimProc. A Batch Primitive is the execution of one or more database Primitives against all of the rows stored within a bounded set of blocks.

A block for InfiniDB is 8192 bytes of data, supporting between 1024 and 8192 column-rows for fixed size storage, and a variable number of varchar columns. Note that varchar(7) and smaller is mapped to a fixed length field and stored as a fixed length field.

A bit of abstraction is useful here when discussing which specific blocks are included within a Batch Primitive under different circumstances. A Batch Primitive actually operates against a small, fixed range of rows (8k rows) for one column that is stored in 1-8 blocks, depending on the data size. In addition, multiple individual primitives within a Batch Primitive can operate on multiple columns for that small, fixed size range of rows. So, the smallest possible primitive could be satisfied by a single 8k block of data, while others may read from more than 100 individual blocks to satisfy a query projecting many columns.

3.3 The Batch Primitive Step (BPS)

A Batch Primitive Step (BPS) is a query execution plan step indicating the point at which operations are issued from the UM to one or more PMs, with potential follow-up executed on the UM. Individual threads within all available PMs execute the requested Batch Primitives against the assigned range of rows. Effectively, there is a global thread pool available to process Batch Primitives, whether there is one server or many.

A query will be satisfied by one or more Batch Primitive Steps. A Batch Primitive Step can execute, depending on query requirements, some, or all of the following:

- **Single Column Scan**: Scan one or more Extents for a given column based on a single column predicate, including: =, <>, in (list), between, isnull, etc.
  See "Tuning First Scan Operations" for additional details.

- **Additional Single Column Filters**: Project additional column(s) for any rows found by a previous scan and apply additional single column predicates as needed. Access of blocks is based on row identifier, going directly to the block(s).
  See "Tuning Additional Column Reads"19 for additional details.

- **Table Level Filters**: Project additional columns as required for any table level filters such as: column1 < column2, or more advanced functions and expressions. Access of blocks is again based on row identifier, going directly to the block(s).

- **Project Join Columns for Joins**: Project additional join column(s) as needed for any join operations. Access of blocks is again based on row identifier, going directly to the block(s).
  See "InfiniDB Multi-Join Tuning" 23for additional details.

- **Execute Multi-Join:** Apply one or more hash join operation against projected join column(s) and use that value to probe a previously built hash map. Build out tuples as need to satisfy inner or outer join requirements. *Note: Depending on the size of the previously built hash map, the actual join behavior may be executed either on the server running PM processes, or the server running UM processes. In either case, the Batch Primitive Step is functionally identical.*
  *See "*InfiniDB Multi-Join Tuning*"* and "Memory Management" for additional details.

- **Cross-Table Level Filters:** Project additional columns from the range of rows for the Primitive Step as needed for any cross-table level filters such as: table1.column1 < table2.column2, or more advanced functions and expressions. Access of blocks is again based on row identifier, going directly to the block(s). When a pre-requisite join operation takes place on the UM, then this operation will also take place on the UM, otherwise it will occur on the PM.

- **Aggregation/Distinct Operation Part 1:** Apply any local group by, distinct, or aggregation operation against the set of joined rows assigned to a given Batch Primitive. Part 1 of This process is handled by Performance Modules

- **Aggregation/Distinct Operation Part 2:** Apply any final group by, distinct, or aggregation operation against the set of joined rows assigned to a given Batch Primitive. This processing is handled by the User Module.
  See “Memory Management”28 for additional details on aggregation.

Effectively, a Batch Primitive executes table oriented SQL commands against column data files by applying filters as early as possible, and deferring projection of additional columns as late as possible with the goal of minimizing I/O. In addition, the Batch Primitive executes group/aggregate/distinct operations to reduce bytes being returned to the User Module.

4 The Extent Map

An Extent is a logical block of space that exists within a physical file for a given column, and is anywhere from 8-64MB in size. By default, each Extent supports 8M rows, with smaller data types consuming less space on disk.

The Extent Map can be considered a catalog of all data persisted on storage, with one entry for each Extent. Each catalog entry contains a logical identifier for each range of blocks that is part of a row identifier, a high water mark (HWM) for each partially filled Extent, as well as a spot to hold a minimum and maximum value for each Extent for most data types. Currently, character data types greater than 8 bytes and varchar data types greater than 7 bytes do not populate the min and max values. All other data types including date, decimal, integer, and smaller strings allow population of the min and max values.

4.1 Population of the Extent Map by a Scan

The Extent Map is automatically populated through any DML or bulk load capability, and no special statistics gathering is required. These min and max values are persisted on disk so that the information is available after a system restart.

The minimum and maximum values stored for each extent for a column are used to eliminate I/O under specific circumstances and can be shown with the select calgetstats(); function which exposes information about the most recently executed statement for a session. For this example we will look at the values reported for `BlocksTouched` and `PartitionBlocksEliminated`.

- `BlocksTouched` - number of 8kb data blocks touched in support of the query.
- `PartitionBlocksEliminated` - number of blocks avoided based only on min/max comparisons.

The following query is run against a scale factor 1000 Star Schema Benchmark (SSB) data set containing 6 billion rows. The data was loaded one month at a time based on `lo_orderdate` field, and no sorting was executed. The query analyzes a specific date range, comparing the price to the cost for 77 million records (perhaps to exclude heavily discounted items), and then calculates an average of about 60 million discrete calculated values to determine a net revenue.

```
select  lo_discount, avg(lo_extendedprice - lo_supplycost), count(*)
    from  lineorder
   where  lo_orderdate between 19940101 and 19940131
          and lo_supplycost < lo_extendedprice * .5
group by  1 order by 2;
```

```
+-------------+-------------------------------------+----------+
| lo_discount | avg(lo_extendedprice - lo_supplycost) | count(*) |
+-------------+-------------------------------------+----------+
|        3.00 |                      3806802.786682 |  5510648 |
|        5.00 |                      3807880.542760 |  5511323 |
|        6.00 |                      3808025.905069 |  5511063 |
|        9.00 |                      3808060.552264 |  5503826 |
|        7.00 |                      3808450.935915 |  5506420 |
|       10.00 |                      3808529.811736 |  5508111 |
|        1.00 |                      3808842.678177 |  5509157 |
```

```
|        0.00 |                        3809090.349685 |  5510113 |
|        2.00 |                        3809309.979333 |  5507300 |
|        4.00 |                        3809375.700146 |  5506796 |
|        8.00 |                        3810136.417883 |  5509236 |
+-------------+---------------------------------------+----------+
11 rows in set (3.20 sec)

+-----------------------------------------------------------------------------------------------
-----------------------------------+
| calgettats()
|
+-----------------------------------------------------------------------------------------------
-----------------------------------+
| Query Stats: MaxMemPct-0; NumTempFiles-0; TempFileSpace-0MB; ApproxPhyI/O-0; CacheI/O-
456180; BlocksTouched-456180; PartitionBlocksEliminated-2890042; MsgBytesIn-4MB; MsgBytesOut-
0MB; Mode-Distributed| 1256230262 897059 |
+-----------------------------------------------------------------------------------------------
-----------------------------------+
1 row in set (0.00 sec)
```

4.2 Partition Block Elimination for Other Columns

Partition Block Elimination based on min/max values can occur for other columns as well. When we modify the query to use a different date field (`lo_commitdate`) that is generally related to the order date we find very similar reductions in both blocks touched and elapsed time. For most data sets that include one or more ascending key values or other data patterns, corresponding benefits are possible.

```
mysql>   select  lo_discount, avg(lo_extendedprice - lo_supplycost), count(*)
    ->     from  lineorder
    ->    where  lo_commitdate between 19940101 and 19940131
    ->   and  lo_supplycost < lo_extendedprice * .5
    -> group by  1 order by  2 ;

+-------------+---------------------------------------+----------+
| lo_discount | avg(lo_extendedprice - lo_supplycost) | count(*) |
+-------------+---------------------------------------+----------+
|        1.00 |                        3807548.123096 |  5509631 |
|        2.00 |                        3807764.108009 |  5506100 |
|        3.00 |                        3808203.553947 |  5508280 |
|       10.00 |                        3808530.528851 |  5508275 |
|        0.00 |                        3808644.366889 |  5508041 |
|        8.00 |                        3808727.418191 |  5507120 |
|        6.00 |                        3809173.101862 |  5507031 |
|        4.00 |                        3809189.422454 |  5508216 |
|        9.00 |                        3809475.131073 |  5507767 |
|        5.00 |                        3809746.089397 |  5511763 |
|        7.00 |                        3809870.163028 |  5510417 |
+-------------+---------------------------------------+----------+
11 rows in set (4.08 sec)

mysql>
mysql> select calgetstats();
+-----------------------------------------------------------------------------------------------
---------------------------------------+
| calgettats()
|
+-----------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------
---------------------------------------+
```

```
| Query Stats: MaxMemPct-0; NumTempFiles-0; TempFileSpace-0MB; ApproxPhyI/O-0; CacheI/O-
1561569; BlocksTouched-1561569; PartitionBlocksEliminated-2779450; MsgBytesIn-14MB;
MsgBytesOut-0MB; Mode-Distributed| 1256231405 276729 |
+--------------------------------------------------------------------------------------------
--------------------------------------+
1 row in set (0.00 sec)
```

4.3 Column Storage + Batch Primitive + Extent Map features for I/O Elimination

The InfiniDB column storage automatically allows for I/O reduction for any columns not included in the query. In addition, the aggressive promotion of any operations that filter data within the Batch Primitive Step can automatically eliminate significant I/O for columns that are referenced following a filter. Finally, InfiniDB can leverage the min and max values stored for each column as the query is analyzed prior to issuing Batch Primitives, potentially reducing I/O for the initial scan as well.

These different I/O elimination techniques can be shown graphically. Given a simple table with 5 columns and 100 million rows spread across 13 Extents per column (12 full, and one half full), a query that selects columns a, b, and c from a table based on filters for columns a and b may end up touching just 5 of 65 Extents to satisfy the query.

- Column Storage Optimization
  - Blocks for columns d and e are ignored because those columns were not referenced in the query. Extents eliminated are highlighted in yellow in the following diagram.

    Extent
- Extent Map Optimization
  - Extent Map min and max values for the filter on column a happen to eliminate Extents 1 through 9. Extents Eliminated highlighted in green below.

    Extent
  - The filter on column b eliminates Extents 9 through 11, leaving only Extents 12 and 13 to be scanned. Extents eliminated highlighted in blue below.

    Extent
- Batch Primitive Optimization
  - Assuming actual execution of filters against columns a and column b excluded any rows within Extent 13 and narrowed the data set to a few rows within Extent 12, only Extent 12 for column c would need to be referenced, and potentially a small subset of blocks as well. Extents eliminated for column c highlighted in orange below.

    Extent

Extent #	column a	column b	column c	column d	column e
1	Extent	Extent	Extent	Extent	Extent
2	Extent	Extent	Extent	Extent	Extent
3	Extent	Extent	Extent	Extent	Extent
4	Extent	Extent	Extent	Extent	Extent
5	Extent	Extent	Extent	Extent	Extent
6	Extent	Extent	Extent	Extent	Extent
7	Extent	Extent	Extent	Extent	Extent
8	Extent	Extent	Extent	Extent	Extent
9	Extent	Extent	Extent	Extent	Extent
10	Extent	Extent	Extent	Extent	Extent
11	Extent	Extent	Extent	Extent	Extent
12	Extent	Extent	Extent	Extent	Extent
13	Extent	Extent	Extent	Extent	Extent

Extent representation for 100 million rows, 5 columns.

5 Tuning of Physical I/O

Before examining how best to tune physical I/O in InfiniDB, it is wise to first define a few terms that will be used throughout this section:

- **Extent:** An Extent is a generally contiguous allocation of space within a storage system. Each column will add an Extent as needed as the table grows. The Extents are actually a variable size, between 8 and 64 MB for most cases and store 8M rows.
- **DBRoot:** a mount point made available to the InfiniDB instance as it is created.
- **Multi-Block Read:** Combining many individual block reads from storage into one read operation. Most storage subsystems will deliver data blocks at varying sustained rates (bandwidth), depending primarily on the size of the requested read operation. Disk performance can significantly degrade when individual random blocks are requested, as each may require a mechanical disk head movement to take place is support of each block.

As rows are added to a given table within the database, all of the columns for the table will grow in size by adding additional Extents as needed. The set of first Extents for each column for a given table is written to one DBRoot. The set of second Extents for the same table will be written to the next DBRoot in a round-robin manner. Other tables will start in different DBRoots as well. This distribution of data is designed to maximize the storage resources that are available to satisfy a wide variety of concurrent column file read conditions.

The above happens automatically as data is loaded based on the defined Extent size, and indeed there are no additional storage parameters associated with any table create operations.

5.1 Tuning First Scan Operations

Tuning I/O requires an understanding of the sequence of operations executed by a Batch Primitive Step. There can be one or more columns referenced for a given BPS, with 0 or more filters. In all cases, 1 column will be read first, and an entire Extent will be read in support of this first operation. Additional column blocks may be read, depending on the selectivity of previous filters. However, for additional columns the row identifier is available that can allow for individual block accesses, if indicated.

A ‘First Scan’ operation will require every block within an Extent, and ‘Additional Column Reads’ can be accomplished either by scanning all blocks (maximizing multi-block read) or by reading individual blocks (minimizing the total number of blocks under some circumstances).

Tuning for the First Scan operation is generally driven by two competing goals and the primary unit of work, the Extent. The goals are:

1. Allow as many threads as possible to process against the unit of work (the Extent) in parallel. Reads are asynchronous in nature, so threads are not stalled while waiting for I/O.
2. Leverage Multi-Block read to get the most out of your storage configuration.

As InfiniDB is expected to run in many different server/storage configurations, tweaking these parameters may influence performance for any given installation. The default should give good performance across a wide variety of configurations.

The parameter within the Calpont.xml configuration file that controls this behavior is:
`ColScanReadAheadBlocks`
and is defaulted to 512 (blocks of data).

Example usage: an 8 byte data type contains 8192 blocks of data to be read in support of a scan operation. For a storage subsystem that maximizes throughput when reading at least 128 blocks at a time, values of 128, 256, 512, or larger for the `ColScanReadAheadBlocks` parameter would all enable best-case performance from the storage subsystem. For a server with 8 cores available to execute parallel reads, a value of 1024, 512, 256, or smaller would all allow (up to) all 8 cores to read in parallel.

It is likely that for many server/disk configurations there may be multiple valid settings that deliver similar results, because they satisfy both requirements:

1. Allowing sufficient threads to read in parallel.
2. Sufficiently leveraging Multi-Block Reads to maximize bandwidth.

5.2 Tuning Additional Column Reads

Physical I/O performance for additional column reads is dependent on the specific cardinality of filters, or filter chains, applied to previous columns. If only one row is required from an Extent, it would be faster for any storage system to read only that block. If some, most, or all blocks are required, it may be faster to leverage Multi-Block Reads even if some individual blocks are not referenced in support of the query.

There are two read methods available to additional column read operations:

1. Read individual blocks based on the already known address.
2. Use Multi-Block Reads, reading `ColScanReadAheadBlocks` blocks at a time.

The choice of these two operations is made within the PM as reads are required. Effectively, statistics are gathered from previous reads for the same column and compared against a parameter:

The parameter within the Calpont.xml file that controls this behavior is:
`PrefetchThreshold`
and is defaulted to 5 (percent of blocks used by the query).

The default setting of 5 indicates that if more than 5% of previous blocks read were actually needed to satisfy the query; then continue to use Multi-Block Reads. If the percentage of blocks required to satisfy the query drops below 5%, then subsequent reads will begin using individual block lookups.

Note that parameter is related to the number of blocks required by a filter, not the number of rows. A predicate returning 5% of the rows may require greater than 90% of the blocks, Of course, if the data values are highly clustered, the number of blocks may be as low as 5-10%.

The actual behavior is related to a complex interaction among a number of items:

1. The specific filter or filter chains used for a query.
2. The frequency of values for columns used by a filter.
3. The actual distribution of those values within blocks.
4. Currently blocks within the data buffer cache at that moment in time.
5. The storage system relative cost to read individual blocks versus total blocks.
6. The impact of concurrent queries on the storage system.

This parameter typically has no impact on most queries. This is based on a number of InfiniDB features and behaviors:

1. The significant reduction in I/O based on multiple optimizations may prevent reaching a storage system bottleneck under any circumstance.
2. The reduced I/O requirements for many queries may allow more queries to be satisfied from cache. In addition, a global scalable data cache is available.
3. In general, most analytic queries will touch many blocks within an Extent and not be near the boundary condition.

6 Concurrency and Queries

For InfiniDB, concurrency management is handled by managing the rate at which requests for Batch Primitive operations are issued from the UM to the PM, rather than assigning fewer or more threads to given queries, or by assigning thread priorities.

The process flow is governed by a parameter:
`MaxOutstandingRequests`
And the default value is 5 (Extents).

A previously mentioned, a Batch Primitive request issued from the UM is a request to execute specific operations against a range of rows within 1 Extent (1 Extent for each column included). The process follows this general feedback loop,

- The UM issues up to MaxOutstandingRequests number of Batch Primitives.
- The PM processes blocks of data and returns individual responses.
- The UM receives the individual block responses from all outstanding Batch Primitives to determine when to issue the next Batch Primitive for the next Extent.

For example with a parameter setting of 5 (Extents), Batch Primitive requests will be issued to process 5 Extents. Once block responses are returned, the UM will issue one additional Batch Primitive against a new Extent to keep the aggregate number of outstanding requests above the parameter setting. Any combination of returned blocks that meets 1 Extent in size will cause another Batch Primitive for the next Extent to be issued.

The defined goal based on the parameter setting is to keep the equivalent of 5 Extents being actively processed at any given point in time with the above parameter setting.

Effectively, this allows for large queries to use all available resources when not otherwise being consumed, and for small queries to execute with minimal delay. The exact time to execute an individual Batch Primitive for 1 Extent (for each column) can be a small fraction of a second, up to a second or so when reads from disk for many dozens of columns may be involved.

To minimize delays for smaller queries while under significant load, it is recommended that the value be set to a smaller integer value.

To preference larger queries, the value of `MaxOutstandingRequests` can be raised slightly. There can be a small amount of time (perhaps a second) for the system to reach a steady state that allows the PM to run at or near full CPU utilization. For most cases, the default value will keep all PMs busy most of the time, and additional requests in the queue will not change elapsed time (beyond reducing the ramp-up time).

The `MaxOutstandingRequests` should generally scale with the number of Performance Modules to ensure that any large queries run in isolation can quickly ramp up to use all available resources. Under concurrent workload scenarios, queues are filled in support of multiple queries, such that full system utilization is generally achieved regardless of this parameter setting.

Based on experiences testing and benchmarking the system, reasonable starting values for different scale-out configurations that allow quick ramp-up for large queries run in isolation are:

PMs	MaxOutstandingRequests
1-2	3
3-4	5
5-6	7
7-8	9
9 or more	PM count + 1

7 InfiniDB Multi-Join Tuning

The InfiniDB join processing model does not follow nested loop operations, but instead executes hash join operations. Nested loop operations are likely faster for joining a smaller number of rows (if supported by an index) because any cost to build a hash structure is eliminated. Alternatively, hash joins generally perform well for joining thousands of rows or more, and perform considerably faster than nested loop operations for millions or billions of rows.

The InfiniDB engine has a number of methods of structuring a hash join operation that preferentially sequence operations to not require creating a hash structure for the largest table involved in a join operation. Instead the largest table (fact table) will be streamed past all required filter, join, and aggregation operations such that the fact table need not be materialized. Structuring the query to avoiding materialization of the fact table is accomplished automatically by InfiniDB optimizations.

The hash joins executed by InfiniDB are run within the scope of a Batch Primitive Step managed by the UM (and therefore executed by Batch Primitive operations run by the PM). A Batch Primitive Step is a step in the execution plan for a query and can be represented as a node in a join graph. The actual execution can occur slightly differently depending on a couple of parameter settings and the cardinality of each of the smaller table(s) being joined.

7.1 Simple Two Table Join Details

For a given two table join operation one of the tables will be determined to be the large table and the other table will be the smaller table. This estimation is based on the number of blocks in the table, and an estimation of the cardinality of the predicate(s). This join will be accomplished with two Batch Primitive Steps (BPS). There will be a BPS that scans the smaller table, applying any available filters as it goes and returns the data to the UM. This can be notated as BPS(small).

The following conditional behavior occurs to minimize network or inter-process communication required to accomplish large joins (minimize data shipping costs). Once the data is returned to the UM the size of the data set will be determined. This measured value will be compared against a parameter:

```
PmMaxMemorySmallSide
```

with a default value of 64M (size in MB) with a max of 4G.

The default is sufficient for something like 1 million rows on the small side to be joined against billions (or trillions) of rows on the large side. Actual cardinality depends on join data type as well as additional small side columns included in the query. If the size of the data set is smaller than the `PmMaxMemorySmallSide`, the data will be sent to the PM(s) to allow creation of a distributed hash map. Otherwise, it will be created on the UM.

Once the small side hash map is instantiated, either on the UM or the PM(s), then the BPS(large) will be issued for the largest table. For cases where the small side hash map has been shipped to the PMs, the join operation and aggregation operation will happen in a fully distributed manner. If the small side hash map is on the UM, then the join and any aggregation will happen on the UM. In all cases column and table filters are applied identically in a fully multi-threaded and distributed manner on the PM(s).

Effectively a two table join is executed by two Batch Primitive Steps:

```
BPS(small) -> BPS(large)
```

Using data warehousing terminology, this simple query translates to:

```
BPS(dimension) -> BPS(fact)
```

7.2 Multi-Join Details

The above processing model is also used to support joining multiple small side tables with one large table in a single BPS.

For one large table joined to two smaller tables, the operations would be:

```
BPS(small_1)
            \
BPS(small_2)   ----> BPS(large)
```

This is accomplished with the two small side steps occurring prior to processing of the large side. Note that this processing model is identical whether the small side hash map is instantiated on either the UM or the PM. The above graph of steps can in fact be executed regardless of the size of each object. Both joins can be run on the PM(s), both on the UM, or split up in any combination depending entirely on the exact run-time cardinality/size found for each of the objects

Indeed this join processing model can handle an arbitrary number of join operations with 1 BPS for each small table feeding into the join operation, and 1 BPS for the largest table.

For a star schema data model, many queries without self joins can be satisfied in exactly this manner:

```
BPS(dimension_1)     \
BPS(dimension_2)      \
BPS(dimension_3)        > BPS(fact)
  … through …         /
BPS(dimension_20)    /
```

This allows for fully multi-threaded and optionally distributed processing with minimal synchronization between threads. In addition, returning rows or aggregations from the fact table is deferred until after all possible filters have been applied.

Other join combinations are accomplished with different combinations of chained join operators and multi-join operators. For example, snowflake schemas will impose additional join operations that are pre-requisites to building hash maps for the BPS(fact) table. Both inner and outer join operations are supported by these chained and multi-join operators.

7.3 Join Optimizations

The InfiniDB optimizer uses statistics to determine the largest table within a global join tree, and then sequences the operations to enable that table as the streaming table (minimizing data transport cost). Additional sub-trees not yet set by the global largest table can have a local large side/small side determination made as well. Where the actual cardinalities are close, the selection of table may not have any performance impact. For data sets with a large fact table and smaller dimension tables, the chance of a sub-optimal choice is significantly smaller. To allow for tuning queries when the estimation done by the optimizer is sub-optimal, a hint is available to set the largest table in the join operation.

The `INFINIDB_ORDERED` hint will indicate that the first table in the From clause should be treated as the global largest table, and joins will be sequenced to elimination materialization of join results for that table where possible.

The hint sets the global largest table as the final table to be joined. The tables joined to this global largest table within the query will then dictate which tables (or join sub-trees) feed that final operation, not the ordering of tables in the `From` clause. Recall that InfiniDB executes multiple join operations in one BPS. Therefore, given a join of 1 large to n smaller tables, the sequencing of joins among the small tables will not, under most circumstances, change the I/O characteristics of the query.

For example, the hint as used here will cause the optimizer to treat the region table as the largest table within the join tree, regardless of actual cardinality.

```
select /*! INFINIDB_ORDERED */ r_regionkey
from region r, customer c, nation n
where r.r_regionkey = n.n_regionkey
 and n.n_nationkey = c.c_nationkey;
```

Again, as there is no nested-loop operation, the hint does not specify a driving table or impose a sequence of tables to be joined one at a time. These concepts do not exist within InfiniDB, at least in their traditional usage.

7.4 Disk Based Joins

InfiniDB performs in-memory joins on the UM node and when a join operation exceeds the memory allocated on the UM for query joins, the query is aborted with an error code. The option is available to have such queries using disk for storing intermediate join data when the memory required for the join exceeds the UM memory limit. While such query performance will be slower than the joins that completely fit in memory and still bound by the temp space availability – it increases the probability of completion of those joins that have large cardinality and memory needs.

- Currently excluded from disk based join processing are:
  - Aggregation, including count(distinct)
  - DML

There are new configuration variables that allow you to manage this capability at the instance level. These variables reside in `HashJoin` element in the `Calpont.xml` configuration file (residing in the `etc` directory for your InfiniDB installation):

- `AllowDiskBasedJoin` – Controls the option to use disk Based joins or not. Valid values are Y (enabled) or N (disabled). By default, this option is disabled.
- `TempFileCompression` – Controls whether the disk join files are compressed or non-compressed. Valid values are Y (use compressed files) or N (use non-compressed files).
- `TempFilePath` – The directory path used for the disk joins. By default, this path is the `tmp` directory for your InfiniDB installation (i.e., `/usr/local/Calpont/tmp`). Files (named `infinidb-join-data*`) in this directory will be created and cleaned on an as needed basis. The entire directory is removed and recreated by ExeMgr at startup.)

  Note: When using disk based joins, it is strongly recommended that the `TempFilePath` reside on its own partition because the partition may fill up as queries are executed.

In addition to the above instance wide variables, a session level variable exists to limit the amount of memory used by a user before the join is switched over to a disk based join. It can be set either at the instance level or session level (refer to the InfiniDB SQL Syntax Guide for session level modification):

- `infinidb_um_mem_limit` - Memory limit in MB per user (i.e. switch to disk based join if this limit is exceeded). By default, this limit is not set (value of 0).

7.5 Key Tuning Parameter: PmMaxMemorySmallSide

The `PmMaxMemorySmallSide` parameter sets an upper limit (max of 4G) on the size of any single small side hash map sent to the PM and determines whether a join is executed in a fully distributed manner.

7.5.1 General tuning guidelines for a single server installation:

Set the `PmMaxMemorySmallSide` large enough to support your largest expected small side join, up to available memory. Even within a single server, the data shipping cost come into play and executing the filter earlier (at the PM) reduces intra-server data transmission costs.

7.5.2 General tuning guidelines for a multiple PM installation:

Tuning the `PmMaxMemorySmallSide` for a multiple PM configuration is related to the amount of available memory on the server, the amount of memory desired to be available for a data buffer cache, the number of simultaneous join operations expected to take place concurrently, and the size of those join operations.

The expectation should be that:

> (# of concurrent PM small side hash maps) * (average size of each PM hash map)
> should be less than
> (total server memory) – (size of the data buffer cache)

Note that no memory is explicitly consumed by setting this parameter; the parameter only limits the size of the largest possible hash map instantiated on the PM(s).

A reasonable setting for a server with 16GB memory, with modest concurrency but join cardinality for the second largest table of up to 10 million might be: set `PmMaxMemorySmallSide` to 512M and setting the data buffer cache to 14 GB.

For significant concurrency, the value of 512M may still be valid, depending on the average size of each PM hash map. However, if memory utilization on the server closely approaches 100%, or exceeds that value and begins to swap, then a smaller setting for `PmMaxMemorySmallSide` would likely reduce memory on the server.

8 Memory Management

8.1 Key Tuning Parameter: NumBlocksPct and TotalUmMemory

The amount of memory dedicated to the data buffer cache within each Performance Module process is set by this Calpont.xml parameter:

PM Memory Parameter:
`NumBlocksPct`

the default varies based on the offering (Single server will default to 50 and Multi-Server installation will default to 70). A value of 70 will allow the data buffer cache to grow to consume 70% of available memory on the server.

Setting for the `NumBlocksPct` should be more conservative when the User Module and the Performance Module are running on the same server (single server configuration) as the User Module may have significant needs for temporary space to manage temporary data sets. For a single server installation, it is recommended that the combined `NumBlocksPct` and `TotalUmMemory` not exceed 75% of the total memory available on the server.

The `TotalUmMemory` parameter constrains the maximum amount of memory available for managing intermediate results for joins, aggregations, and set operations on the User Module. This does not dedicate any memory but instead constrains the maximum amount of memory that may be consumed in support of a single join hash map. This can generally be raised if needed to allow the occasional small side hash map that may include hundreds of millions of rows with minimal impact otherwise.

With PM processes partitioned from UM processes in different servers, it may be entirely reasonable to set the cache to 95% of total PM memory if the average small side join cardinality is under 100k or so. The PMs may then vary between 95 and 97% memory utilization for many different concurrency scenarios.

Note that InfiniDB can support joining billions or trillions of large side rows to any number of small side hash maps and the size of the large size is not limited in any way by any parameter setting. The PM/`TotalUmMemory` parameters apply only to the smaller side maps.

9 Scalability

9.1 Scalability – Data Size and Performance

Once a given system has been configured and any problem queries have been optimized to run efficiently, additional performance can be achieved through scaling the Performance Modules. In addition, scaling the system can allow for analyzing larger data volumes with consistent performance.

9.2 Scalability – User Module

Under conditions where the CPU utilization increases on the User Module, additional User Modules may be added to distribute that load. As one of the InfiniDB goals is to distribute work as much as possible, many queries can have as much a 99% of CPU cycles occur on the PMs.

10 Tuning Tools and Utilities

10.1 Query Summary Statistics – calgetstats

A `select calgetstats()` command can be run following a query to present summary results of a previous operation. The example query here is a 4 table join run against a Star Schema Benchmark data set with 5.99 billion rows in the fact table based on three filters, two of which are expressed through joins. Final aggregation is against about 20 million rows and in this example an 8 PM configuration was used.

A breakdown of the results will follow the example output.

```
         select  d_year, lo_tax, p_size, s_region,  count(*)
           from  dateinfo, part, supplier, lineorder
          where  s_suppkey = lo_suppkey
            and  d_datekey = lo_orderdate
            and  p_partkey = lo_partkey
            and  lo_orderdate between 19980101 and 19981231
            and  s_nation = 'BRAZIL'
            and  p_size <> 23
         group by  1,2,3,4
         order by  1,2,3,4;

+--------+--------+--------+----------+----------+
| d_year | lo_tax | p_size | s_region | count(*) |
+--------+--------+--------+----------+----------+
|   1998 |   0.00 |      1 | AMERICA  |    47607 |
|   1998 |   0.00 |      2 | AMERICA  |    47194 |
... results abbreviated ...
|   1998 |   8.00 |     48 | AMERICA  |    47846 |
|   1998 |   8.00 |     49 | AMERICA  |    47394 |
|   1998 |   8.00 |     50 | AMERICA  |    47030 |
+--------+--------+--------+----------+----------+
441 rows in set (3.66 sec)

mysql> select calgetstats();
+-------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------
----------------------------------------+
| calgettats()
|
+-------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------
----------------------------------------+
| Query Stats: MaxMemPct-4; NumTempFiles-0; TempFileSpace-0MB; ApproxPhyI/O-0; CacheI/O-
1363254; BlocksTouched-1363254; PartitionBlocksEliminated-2637824; MsgBytesIn-811MB;
MsgBytesOut-0MB; Mode-Distributed| 1256555629 961957 |
+-------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------
----------------------------------------+
1 row in set (0.02 sec)
```

The output from the `calgetstats()` function are as follows:

- **MaxMemPct-4**; This field shows memory utilization for the User Module (UM) in support of any UM join, group by, aggregation, distinct, or other operation.
- **NumTempFiles-0**; This field shows any temporary file utilization for the User Module (UM) in support of any UM join, group by, aggregation, distinct, or other operation.
- **TempFileSpace-0MB**; This field shows the size of any temporary file utilization for the User Module (UM) in support of any UM join, group by, aggregation, distinct, or other operation.
- **ApproxPhyI/O-0**; This field shows any reads from storage for the query. Under some circumstances it may vary slightly from actual, typically less than 1/10th of 1 percent.
- **CacheI/O-1363254**; This field shows block touches required for the query, reduced by the number of discrete Physical I/O reads requested.
- **BlocksTouched-1363254**; This field shows block touches required for the query.
- **PartitionBlocksEliminated-2637824**; This field shows blocks that were avoided based on the partition elimination that took place based on the Extent Map min/ max values. Note that this does not report I/O reduction from column storage or I/O reduction from deferring column reads until after filters have been applied.
- **MsgBytesIn-811MB**; A measure of process to process data movement.
- **MsgBytesOut-0MB**; A measure of process to process data movement.
- **Mode-Distributed**; An indicator whether the query join processing was handled within InfiniDB, or by traditional MySQL join functionality. This should be Distributed for best performance against big data.

10.2 Query Detail Statistics - calgettrace

Additional information at a more detailed level can be presented by enabling a SQL tracing functionality in InfiniDB. The steps to use SQL tracing are as follows:

1. Enable a new trace, which is done by issuing the following command: `select calsettrace(1);`
2. Run the query
3. View the results of the trace by issuing the following command: `select calgettrace();`

For example, after enabling a new trace, a query such as the following can be run and analyzed:

```
  select 'BRAZIL',  d_year, lo_tax, p_size, s_region,  count(*)
    from  dateinfo, part, supplier, lineorder
   where  s_suppkey = lo_suppkey
     and  d_datekey = lo_orderdate
     and  p_partkey = lo_partkey
     and  lo_orderdate between 19980101 and 19981231
     and  s_nation = 'BRAZIL'
     and  p_size <> 23
group by  1,2,3,4,5
order by  1,2,3,4,5;
```

```
mysql> select calgettrace();
+-------------------------------------------------------------------------------------------
-----------------------------------------------------+
| calgettrace()
|
+-------------------------------------------------------------------------------------------
-----------------------------------------------------+
Desc Mode Table       TableOID ReferencedOIDs       PIO     LIO     PBE     Elapsed Rows
```

```
BPS   PM   part         21557    (p_partkey,p_size)    1370   1377   0      0.409   1371884
BPS   PM   dateinfo     21602    (d_datekey,d_year)    12     10     0      0.030   2556
DSS   PM   supplier     21572    (s_nation)            0      1      -      0.000   1
BPS   PM   supplier     21572    (s_nation,s_region,   2449   3564   0      0.305   40078
                                  s_suppkey)
HJS   PM   supplier-    21572    -                     -      -      -      -----   -
           supplier
BPS   PM   lineorder    21537    (lo_orderdate,      131711 131095 266240 1.739   48510
                                  lo_partkey,lo_suppkey,lo_tax)
HJS   PM   lineorder-   21537    -                   -      -      -      -----   -
           dateinfo
HJS   PM   lineorder-   21537    -                   -      -      -      -----   -
           supplier
HJS   PM   lineorder-   21537    -                   -      -      -      -----   -
           part
TAS   UM   -            -        -                   -      -      -      1.395   441
TNS   UM   -            -        -                   -      -      -      0.000   441
+---------------------------------------------------------------------------------------------------
--------------------------------------------------------+
1 row in set (0.02 sec)
```

The output from the calgettrace() includes the following headings:

- Desc – Operation being executed.
- Mode – Whether executed within the UM or the PM.
- Table – Table for which columns may be scanned/projected.
- TableOID – ObjectID for the table being scanned.
- ReferencedOIDs – ObjectIDs for the columns required by the query.
- PIO – Physical I/O (reads from storage) executed for the query.
- LIO – Logical I/O executed for the query, also known as Blocks Touched.
- PBE – Partition Blocks Eliminated identifies blocks eliminated by Extent Map min/max.
- Elapsed – Elapsed time for a give step.
- Rows – Intermediate rows returned

The trace output will report on the sequence of operations as well as their cost in block touches, elapsed time, and row cardinality. The initial filter applied will be on the supplier table.

```
DSS   PM   supplier    21572    (s_nation)            0      1      -      0.000   1
```

The filter for = 'BRAZIL' references a variable length field stored within a dictionary structure that may allow for sharing a single string value across many rows. This is accomplished in a discrete step labeled Dictionary Signature Step (DSS).

```
BPS   PM   supplier    21572    (s_nation,s_region,   2449   3564   0      0.305   40078
                                 s_suppkey)
```

The BPS reading other columns from supplier for subsequent join.

```
HJS   PM   lineorder-  21537    -                     -      -      -      -----   -
           supplier
```

A Hash Join Step (HJS) that associates the DSS filter with the supplier projection. Note that this is actually a part of the previous BPS and will always report a time of 0. The Mode field indicates a PM join was executed here.

```
BPS  PM   dateinfo    21602    (d_datekey,d_year)   12     10     0      0.030   2556
```

The BPS reading dateinfo table columns for subsequent join.

```
BPS  PM   part        21557    (p_partkey,p_size)   1370   1377   0      0.409   1371884
```

The BPS reading part table columns for subsequent join.

```
BPS  PM   lineorder   21537    (lo_orderdate,      131711 131095 266240  1.739   48510
                                lo_partkey,lo_suppkey,lo_tax)
```

The BPS that scans and projects from lineorder, and executes join operations against supplier, dateinfo, and part. Output cardinality is 48,510 rows after all filters and joins.

```
HJS  PM   lineorder-  21537    -                    -      -      -      -----   -
          dateinfo
```

The HJS is an indicator for the hash join operations that take place within the previous BPS. The PM (or UM) value in the Mode field indicates whether the hash maps were created on the UM or the PM. Elapsed time reported here will always be zero as the hash joins operations actually take place within the previous BPS.

```
TAS  UM   -           -        -                    -      -      -      1.395   441
```

Tuple Aggregation Step (TAS) reports the elapsed time between the UM receiving the first intermediate aggregation results from the feeding BPS and completing the required aggregation. For most aggregation scenarios, this time will correspond closely with the previous BPS operation. The Mode reported will always be UM for this operation, but the underlying functionality is always accomplished in a 2 phase operation, with local aggregation done first on the PM(s), and a final aggregation on the UM.

```
TNS  UM   -           -        -                    -      -      -      0.000   441
```

The Tuple Annex Step (TNS) is the last step if there are any constant columns to be filled in (‘,e’. BRAZIL in the above example) and certain LIMIT, order by and final distinct cases.

Note, as a number of operations happen in parallel, the sum of individual elapsed times will exceed the query elapsed time. In this specific example above, the BPS step consuming 1.739 and the Tuple Aggregation Step (TAS) step consuming 1.395 seconds are actually created as a pipelined operation, with the BPS feeding partially aggregated data to the TAS. Therefore the elapsed time to execute both the BPS(lineorder) and the following TAS is closer to the 1.739 seconds reported.

10.3 Flush Cache – calflushcache

InfiniDB offers a development/testing utility to allow for simpler testing of Physical I/O that will flush data from the data buffer cache across all PMs. This is accomplished by issuing the following command:

```
select calflushcache();
```

Note that this is intended as a development/testing utility only. There are no known benefits from flushing the cache and increasing PIO, instead queries will generally run longer.

10.4 Read or Change Parameter – configxml.sh

A utility is provided that allows for getconfig / setconfig to read or set values within the Calpont.xml parameter file.

Usage: `/usr/local/Calpont/bin/configxml.sh {setconfig|getconfig} section variable set-value`

Example syntax to update some parameters mentioned within this document:

```
/usr/local/Calpont/bin/configxml.sh setconfig HashJoin PmMaxMemorySmallSide 640M
/usr/local/Calpont/bin/configxml.sh setconfig JobList MaxOutstandingRequests 7
/usr/local/Calpont/bin/configxml.sh setconfig DBBC NumBlocksPct 90
```

10.5 Display Extent Map - editem

A utility called `editem` can be used to determine whether a given column has been populated. The utility requires an objectid available from either a trace output, or from the system catalog.

Warning: The editem utility can be safely used to display values for inspection. Other options for editem can be destructive in nature and should only be used as instructed by InfiniDB support, or in an isolated test environment.

Example select from system catalog:

```
select columnname, objectid from calpontsys.syscolumn where tablename =
'lineorder' and columnname = 'lo_commitdate';
```

Editem help provides a list of available commands by using the –h flag as all other supplied utilities do (e.g. `/usr/local/Calpont/bin/editem –h`)

The editem -o <objected> command lists min and max values for a given column. Note that missing min values are represented by the maximum possible value for that data type, and that missing max values are represented by the minimum possible value for that data type.

11 Column Data Storage Differences

There can be some benefits and trade-offs for column storage detailed below, including any tactics available to avoid trade-offs and maximize column storage capabilities.

- **Insert Trade-Offs** - Individual row inserts can have different performance profile with column storage. Insertion of 1 row into a given row-based system may touch only 1 table block, plus some number of index blocks. Insertion into a column storage DBMS will touch at least one block per column, but no additional cost for index blocks. Note that this costs changes with bulk load (cpimport), load data infile, or bulk insert operations. For insert of thousands of rows, the relative block touches converge at a few thousand rows, with column storage becoming more efficient beyond a few thousand when compared with table + index for row DBMS.
    - **InfiniDB Tactic** – Use cpimport. With InfiniDB's cpimport bulk load utility the blocks required for inserts with column storage (without indexes) is smaller than for row based DBMS. In addition, elimination of indexes allows for generally faster load performance, with better consistency in load time for big data. For InfiniDB, loading a batch of rows is the same operation whether the table is empty or contains 10 billion rows.
- **Delete Trade-Offs** - Individual row deletes can have different performance profile with column storage. Deletion of 1 row from a given row-based system may touch only 1 table block, plus some number of index blocks. Deletion from a column storage DBMS will touch at least one block per column, but no additional cost for index blocks. For deletions of thousands of rows from a given range of rows, the relative block touches converge, with column storage becoming more efficient when compared with table + index for row DBMS.

Performance for delete operations can vary for any DBMS, depending on distribution of row to be deleted among blocks. For example, deletion of 1000 rows from a row based DBMS storing 100 rows per 8k block can touch anywhere between 10 (perfect distribution) and 1000 blocks (worst case distribution). For column storage, assuming a 10 column table, with each column storing between 1024 and 8192 rows per block, the best case scenario may be 10 blocks (perfect distribution), but the worst case would be 10,000 blocks (worst case scenario).

- **InfiniDB Tactic** – Delete in batches. For example, deletion of rows based on an in-clause with 100 keys pointing to a million rows will run significantly faster than 100 discrete delete statements.

- **Update Benefits** – Updates for column storage will have a significant advantage across most scenarios. The best case for a row DBMS update of 1000 rows is 100 blocks (assuming 100 rows per column). The best case scenario for a column DBMS is 1 block. Note that the performance benefits increase if indexes must be updated as well.
    - **InfiniDB Tactic** – already optimized such that updates will generally run significantly faster.

12 Data Load Rates and Near-Real-Time Loading

Data load rate can vary depending on any batching opportunities as well as the underlying storage capabilities, the table definition, data types, and values. However as a general rule of thumb, the load rate from the `cpimport` bulk load utility can be hundreds of thousands to millions of rows per second. Load rate for load data infile can be thousands of rows per second. Load rate for individual inserts is slower, some 10's of rows per second. In all cases the data is made available in a model that allows for consistent read behavior.

The `cpimport` bulk load capability can load data a thousand rows at a time, millions at a time, or more. There is some modest startup time (about a second) for the import that suggests that peak rate occurs when loading 100 thousand or more rows at a time. That load rate should be consistent through subsequent loads, as the fundamental operations don't change.

When loading data in any manner, the inserted rows can cause additional Extents to be added to the table, incurring a (typically) 5-20 second delay while ensuring that the Extents are laid out in a generally contiguous manner within storage. This ensures best possible performance for subsequent select operations by maximizing multi-block read benefits.

12.1 Process Prioritization

By default, the import and query processes have the same priority. This priority may be changed (only in Linux) by adding entries into the InfiniDB configuration file Calpont.XML. This may come in useful if concurrent query and imports are executing with regularity and import speed has been reduced as a result. These entries are as follows:

```
<ExeMgr1>
<Priority>###</Priority>
</ExeMgr1>

<PrimitiveServers>
<Priority>###</Priority>
</PrimitiveServers>

<WriteEngine>
<Priority>###</Priority>
</WriteEngine>
```

where `###` has the following values:

- 1-40 - where 1 is the lowest priority and 40 is the highest priority. The three processes above have a default priority of 21.

Changing the ExeMgr1 and PrimitiveServers priorities will modify query priority and changing the WriteEngine priority will modify import priority.

13 Performance Rules of Thumb for InfiniDB

There are a number of rules of thumb that experienced tuners may feel comfortable with based on working with row-based DBMS system. A number of these traditional rules of thumb change significantly with InfiniDB, and indeed may present opportunities for thousands of novel approaches to bloom to better handle data warehousing complexity.

- *Indexes are useful when querying for less than 5% of the data*. This rule of thumb changes with column storage that never scans a table (no full table scan). For example, for a table with 20 similar columns the rule of thumb would change to less than .25% of the data as the cost to scan 1 column is already 5% of the cost to scan the table. Because of the reduced utility of indexes, the random I/O behavior, and the significant cost to maintain, indexes are not implemented with InfiniDB at this time.
    - New InfiniDB paradigm – leverage the power of multi-threaded/distributed operations with more efficient I/O.

- *Don't include infrequently accessed columns when modeling a large table*. For a row based DBMS systems, any additional column created as part of the table can slow down **every** query that includes significant number of rows (and can't be satisfied by a covering index or other duplication/materialization of the data).
    - New InfiniDB paradigm – because columns not referenced in a query are ignored, additional data can be made available for infrequent analysis with only a cost for additional storage and load.

- *Column data types not really important*. For many row based DBMS systems, especially those that store every column in a variable size data type, there may be no performance difference based on data types. If the data only contains a single character value, changing a char(2) to a char(1) for a row-based dbms may change the cost to scan a non-indexed column by 1% or less. For a column storage system, a column scan would require half as many blocks, improving efficiency, and accelerating lookups by column.
    - New InfiniDB paradigm – Instead of tuning indexes to speed lookup, column data types can be tuned to speed lookup.

- *Redundant data is always bad*. Redundant data absolutely creates problems related both to update/consistency logic, as well as increasing costs for all queries that touch the table. However, for a data warehouse scenario implemented with a write-once approach to loading the data, the update/consistency problem may disappear. For a row-based storage the additional I/O costs still remain for every query touching the table.
    - New InfiniDB paradigm – An additional column may provide a new access path to the data, whether it is the leading portion of a field, or a concatenation of multiple fields.

- *Cache must be larger than the Fact table (or active partitions) to be useful*. A full table scan or a full scan of an active partition of the table, for most proprietary row-based dbms must fit completely in memory to allow a second scan to be satisfied from cache.

- New InfiniDB paradigm – Because column storage eliminates columns not referenced by the query, and accesses columns individually, benefits from cache can still occur even when sized at a fraction of the table size.

- *Reads from disk are 20-40 times slower than from cache*. While the actual ratio varies depending on a large number of factors, there is generally a significant drop-off in performance when reading from disk. A 20-40 x ratio implies that 95% to 97.5% of query cost is directly related to reads from disk.
    - New InfiniDB paradigm – About a 2-10 improvement from cache. Because of InfiniDB I/O efficiencies (column storage, partition block elimination, deferring I/O until after filters), the absolute I/O cost of most queries goes down significantly. As a result, the relative cost may be between 2 and 10 times faster from cache.

- *Partitioning is critical for data warehouse performance*. Actually this one is still true. Traditional proprietary DBMS systems allow the DBA to declare a 1, or 2, level partitioning strategy to reduce block touches for queries using appropriate filters against those 1 or 2 columns.
    - New InfiniDB paradigm – Automatic partitioning in place based on column storage. Automatic partitioning in place for all columns that show an ascending pattern or other clustering of data based on the load methodology. Note that this is not limited to 1 or 2 columns, but is available for all columns with appropriate characteristics.
    - Note that the InfiniDB partition is not enforced. If the data is loaded randomly, then there may be no partition elimination benefits at all.

- *Joins require the right indexes*. Any nested loop join operation will perform better when any lookup within the loop is supported by an index. For many cases, a join without an index will have very poor performance. In addition, if you flipped the nested loop operation, a different index would be required.
    - New InfiniDB paradigm – Leverage Hash Join capabilities to eliminate requirements for indexes and, in addition, remove significant row-by-row processing costs associated with nested loop processing.

- *Conditional Performance Expectations*. When tuned properly, and the queries stay within the boundaries defined by the tuning, row based DBMS systems can be tuned for those queries. However, other queries that don't use the index, or are not solved by 1 or 2 explicitly declared partition columns may have very poor performance, sometimes 2 orders of magnitude worse performance. Indexes have very different performance characteristics when they are fully cached versus when they are re-loaded again and again to satisfy random access. Join order can significantly change the cost of any nested loop operations.
    - New InfiniDB paradigm – A much more consistent performance experience should be available. For a table with 1,2, 4 or 8 byte columns across a billion records will show a some differences between the scan rates, but not orders of magnitude. Join ordering is handled automatically to minimize data shipping costs.

- *"Select *" is bad if you don't need all columns.*
    - InfiniDB same-old paradigm – Definitely still bad if you don't need all columns.