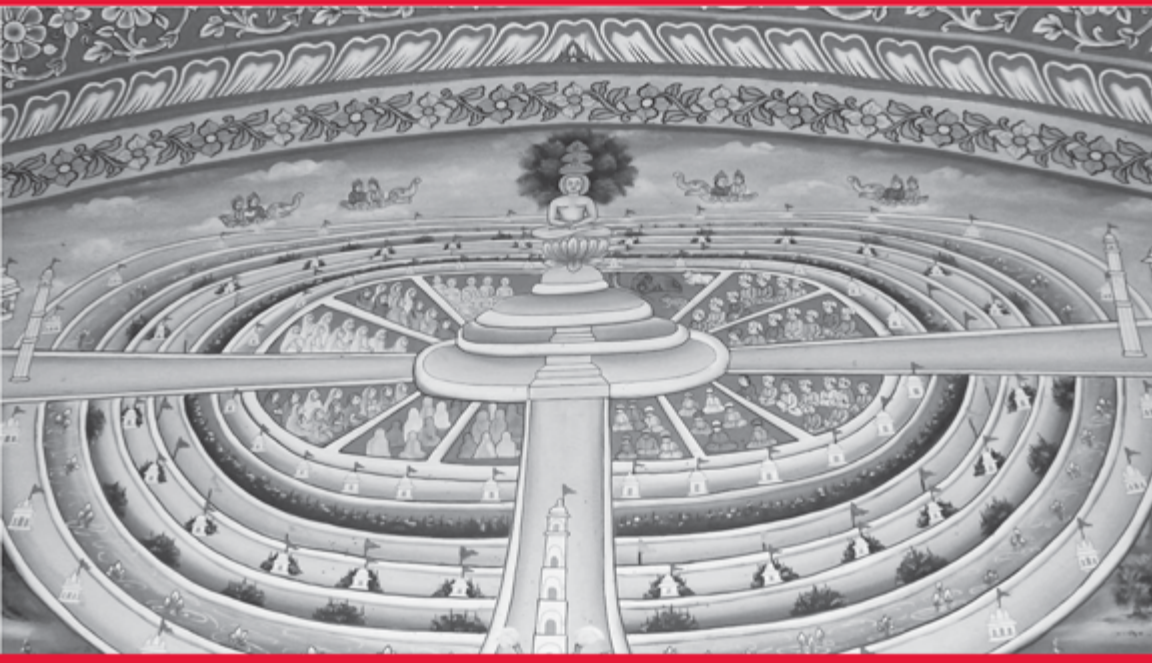
O'REILLY®

In Search of Database Nirvana

The Challenges of Delivering Hybrid Transaction/Analytical Processing



Rohit Jain

Strata+ Hadoop WORLD

Make Data Work

strataconf.com

Presented by O'Reilly and Cloudera, Strata + Hadoop World helps you put big data, cutting-edge data science, and new business fundamentals to work.

- Learn new business applications of data technologies
- Develop new skills through trainings and in-depth tutorials
- Connect with an international community of thousands who work with data

Job # D2044

In Search of Database Nirvana

The Challenges of Delivering Hybrid Transaction/Analytical Processing

Rohit Jain

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

In Search of Database Nirvana
by Rohit Jain

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editor: Marie Beaugureau
Production Editor: Kristen Brown
Copyeditor: Octal Publishing, Inc.
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

August 2016: First Edition

Revision History for the First Edition
2016-08-01: First Release

978-1-491-95903-9

[LSI]

Table of Contents

In Search of Database Nirvana

The Swinging Database Pendulum

It often seems like the IT industry sways back and forth on technology decisions.

About a decade ago, new web-scale companies were gathering more data than ever before and needed new levels of scale and performance from their data systems. There were Relational Database Management Systems (RDBMSs) that could scale on Massively-Parallel Processing (MPP) architectures, such as the following:

- NonStop SQL/MX for Online Transaction Processing (OLTP) or operational workloads
- Teradata and HP Neoview for Business Intelligence (BI)/Enterprise Data Warehouse (EDW) workloads
- Vertica, Aster Data, Netezza, Greenplum, and others, for analytics workloads

However, these proprietary databases shared some unfavorable characteristics:

- They were not cheap, both in terms of software and specialized hardware.
- They did not offer schema flexibility, important for growing companies facing dynamic changes.
- They could not scale elastically to meet the high volume and velocity of big data.

- They did not handle semistructured and unstructured data very well. (Yes, you could stick that data into an XML, BLOB, or CLOB column, but very little was offered to process it easily without using complex syntax. Add-on capabilities had vendor tie-ins and minimal flexibility.)
- They had not evolved User-Defined Functions (UDFs) beyond scalar functions, which limited parallel processing of user code facilitated later by Map/Reduce.
- They took a long time addressing reliability issues, where Mean Time Between Failure (MTBF) in certain cases grew so high that it became cheaper to run Hadoop on large numbers of high-end servers on Amazon Web Services (AWS). By 2008, this cost difference became substantial.

Most of all, these systems were too elaborate and complex to deploy and manage for the modest needs of these web-scale companies. Transactional support, joins, metadata support for predefined columns and data types, optimized access paths, and a number of other capabilities that RDBMSs offered were not necessary for these companies' big data use cases. Much of the volume of data was transitionary in nature, perhaps accessed at most a few times, and a traditional EDW approach to store that data would have been cost prohibitive. So these companies began to turn to NoSQL databases to overcome the limitations of RDBMSs and avoid the high price tag of proprietary systems.

The pendulum swung to *polyglot programming* and *persistence*, as people believed that these practices made it possible for them to use the best tool for the task. Hadoop and NoSQL solutions experienced incredible growth. For simplicity and performance, NoSQL solutions supported data models that avoided transactions and joins, instead storing related structured data as a JSON document. The volume and velocity of data had increased dramatically due to the Internet of Things (IoT), machine-generated log data, and the like. NoSQL technologies accommodated the data streaming in at very high ingest rates.

As the popularity of NoSQL and Hadoop grew, more applications began to move to these environments, with increasingly varied use cases. And as web-scale startups matured, their operational workload needs increased, and classic RDBMS capabilities became more relevant. Additionally, large enterprises that had not faced the same

challenges as the web-scale startups also saw a need to take advantage of this new technology, but wanted to use SQL. Here are some of their motivations for using SQL:

- It made development easier because SQL skills were prevalent in enterprises.
- There were existing tools and an application ecosystem around SQL.
- Transaction support was useful in certain cases in spite of its overhead.
- There was often the need to do joins, and a SQL engine could do them more efficiently.
- There was a lot SQL could do that enterprise developers now had to code in their application or MapReduce jobs.
- There was merit in the rigor of predefining columns in many cases where that is in fact possible, with data type and check enforcements to maintain data quality.
- It promoted uniform metadata management and enforcement across applications.

So, we began seeing a resurgence of SQL and RDBMS capabilities, along with NoSQL capabilities, to offer the best of both the worlds. The terms *Not Only SQL* (instead of No SQL) and *NewSQL* came into vogue. A slew of SQL-on-Hadoop implementations were introduced, mostly for BI and analytics. These were spearheaded by Hive, Stinger/Tez, and Impala, with a number of other open source and proprietary solutions following. NoSQL databases also began offering SQL-like capabilities. New SQL engines running on NoSQL or HDFS structures evolved to bring back those RDBMS capabilities, while still offering a flexible development environment, including graph database capabilities, document stores, text search, column stores, key-value stores, and wide column stores. With the advent of Spark, by 2014 companies began abandoning the adoption of Hadoop and deploying a very different application development paradigm that blended programming models, algorithmic and function libraries, streaming, and SQL, facilitated by in-memory computing on immutable data.

The pendulum was swinging back. The polyglot trend was losing some of its charm. There were simply too many languages, inter-

faces, APIs, and data structures to deal with. People spent too much time gluing different technologies together to make things work. It required too much training and skill building to develop and manage such complex environments. There was too much data movement from one structure to another to run operational, reporting, and analytics workloads against the same data (which resulted in duplication of data, latency, and operational complexity). There were too few tools to access the data with these varied interfaces. And there was no single technology able to address all use cases.

Increasingly, the ability to run transactional/operational, BI, and analytic workloads against the same data without having to move it, transform it, duplicate it, or deal with latency has become more and more desirable.

Companies are now looking for one query engine to address all of their varied needs—*the ultimate database nirvana*. 451 Research uses the terms *convergence* or *converged data platform*. The terms *multi-model* or *unified* are also used to represent this concept. But the term coined by IT research and advisory company, Gartner, *Hybrid Transaction/Analytical Processing (HTAP)*, perhaps comes closest to describing this goal.

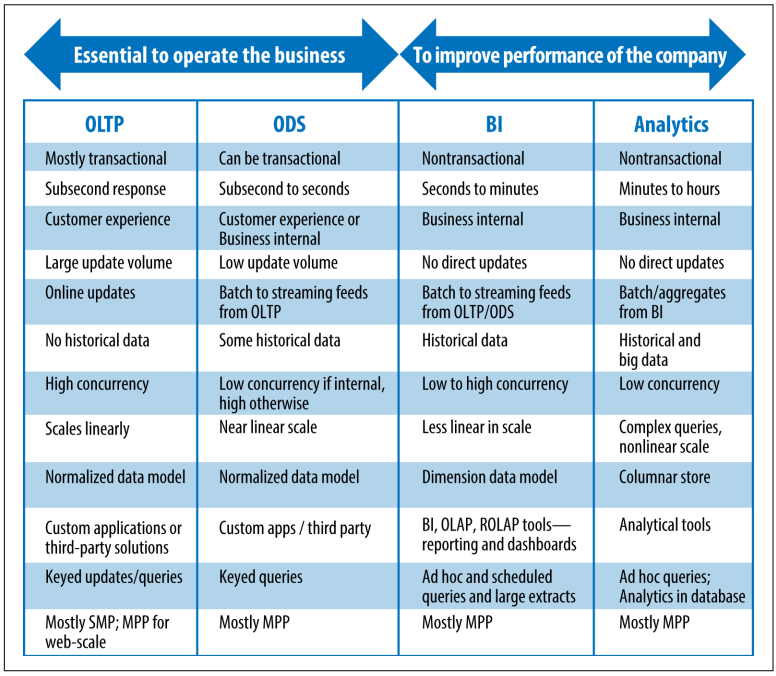
But can such a nirvana be achieved? This report discusses the challenges one faces on the path to HTAP systems, such as the following:

- Handling both operational and analytical workloads
- Supporting multiple storage engines, each serving a different need
- Delivering high levels of performance for operational and analytical workloads using the same data model
- Delivering a database engine that can meet the enterprise operational capabilities needed to support operational and analytical applications

Before we discuss these points, though, let's first understand the differences between *operational* and *analytical* workloads and also review the distinctions between a query engine and a storage engine. With that background, we can begin to see why building an HTAP database is such a feat.

HTAP Workloads: Operational versus Analytical

People might define operational versus analytical workloads a bit differently, but the characteristics described in Figure 1-1 will suffice for the purposes of this report. Although the term HTAP refers to transactional and analytical workloads, throughout this report we will refer to operational workloads (which include transactional workloads) versus BI and analytic workloads.



OLTP	ODS	BI	Analytics
Mostly transactional	Can be transactional	Nontransactional	Nontransactional
Subsecond response	Subsecond to seconds	Seconds to minutes	Minutes to hours
Customer experience	Customer experience or Business internal	Business internal	Business internal
Large update volume	Low update volume	No direct updates	No direct updates
Online updates	Batch to streaming feeds from OLTP	Batch to streaming feeds from OLTP/ODS	Batch/aggregates from BI
No historical data	Some historical data	Historical data	Historical and big data
High concurrency	Low concurrency if internal, high otherwise	Low to high concurrency	Low concurrency
Scales linearly	Near linear scale	Less linear in scale	Complex queries, nonlinear scale
Normalized data model	Normalized data model	Dimension data model	Columnar store
Custom applications or third-party solutions	Custom apps / third party	BI, OLAP, ROLAP tools—reporting and dashboards	Analytical tools
Keyed updates/queries	Keyed queries	Ad hoc and scheduled queries and large extracts	Ad hoc queries; Analytics in database
Mostly SMP; MPP for web-scale	Mostly MPP	Mostly MPP	Mostly MPP

Figure 1-1. Different types and characteristics of operational and analytical workloads

OLTP and Operational Data Stores (ODS) are operational workloads. They are low latency, very high volume, high concurrency workloads that are used to operate a business, such as taking and fulfilling orders, making shipments, billing customers, collecting payments, and so on. On the other hand, BI/EDW and analytics workloads are considered analytical workloads. They are relatively higher latency, lower volume, and lower concurrency workloads that are used to improve the performance of a company, by analyzing

operational, historical, and external (big) data, to make strategic decisions, or take actions, to improve the quality of products, customer experience, and so forth.

An HTAP query engine must be able to serve everything, from simple, short transactional queries to complex, long-running analytical ones, delivering to the service-level objectives for all these workloads.

Query versus Storage Engine

Query engines and storage engines are distinct. (However, note that this distinction is lost with RDBMSs, because the storage engine is proprietary and provided by the same vendor as the query engine is. One exception is MySQL, which can connect to various storage engines.)

Let's assume that SQL is the predominant API people use for a query engine. (We know there are other APIs to support other data models. You can map some of those APIs to SQL. And you can extend SQL to support APIs that cannot be easily mapped.) With that assumption, a query engine has to do the following:

- Allow clients to connect to it so that it can serve the SQL queries these clients submit.
- Distribute these connections across the cluster to minimize queueing, to balance load, and potentially even localize data access.
- Compile the query. This involves parsing the query, normalizing it, binding it, optimizing it, and generating an optimal plan that can be run by the execution engine. This can be pretty extensive depending on the breadth and depth of SQL the engine supports.
- Execute the query. This is the execution engine that runs the query plan. It is also the component that interacts with the storage engine in order to access the data.
- Return the results of the query to the client.

Meanwhile, a storage engine must provide at least some of the following:

- A storage structure, such as HBase, text files, sequence files, ORC files, Parquet, Avro, and JSON to support key-value, Bigtable, document, text search, graph, and relational data models
- Partitioning for scale-out
- Automatic data repartitioning for load balancing
- Projection, to select a set of columns
- Selection, to select a set of rows based on predicates
- Caching of data for writes and reads
- Clustering by key for keyed access
- Fast access paths or filtering mechanisms
- Transactional support/write ahead or audit logging
- Replication
- Compression and encryption

It could also provide the following:

- Mixed workload support
- Bulk data ingest/extract
- Indexing
- Colocation or node locality
- Data governance
- Security
- Disaster recovery
- Backup, archive, and restore functions
- Multitemperature data support

Some of this functionality could be in the storage engine, some in the query engine, and some shared between the two. For example, both query and storage engines need to collaborate to provide high levels of concurrency and consistency.

These lists are not meant to be exhaustive. They illustrate the complexities of the negotiations between the query and storage engines.

Now that we've defined the different types of workloads and the different roles of query engines and storage engines, for the purposes

of this report, we can dig in to the challenges of building a system that supports all workloads and many data models at once.

Challenge: A Single Query Engine for All Workloads

It is difficult enough for a query engine to support single operational, BI, or analytical workloads (as evidenced by the fact that there are different proprietary platforms supporting each). But for a query engine to serve all those workloads means it must support a wider variety of requirements than has been possible in the past. So, we are traversing new ground, one that is full of obstacles. Let's explore some of those challenges.

Data Structure—Key Support, Clustering, Partitioning

To handle all these different types of workloads, a query engine must first and foremost determine what kind of workload it is processing. Suppose that it is a single-row access. A single-row access could mean scanning all the rows in a very large table, if the structure does not have keyed access or any mechanism to reduce the scan. The query engine would need to know the key structure for the table to assess if the predicate(s) provided cover the entire key or just part of the key. If the predicate(s) cover the entire unique key, the engine knows this is a single-row access and the storage engine supporting direct keyed access can retrieve it very fast.

A Point about Sharding

People often talk about *sharding* as an alternative to *partitioning*. Sharding is the separation of data across multiple clusters based on some logical entity, such as region, customer ID, and so on. Often the application is burdened with specifying this separation and the mechanism for it. If you need to access data across these shards, this requires federation capabilities, usually above the query engine layer.

Partitioning is the spreading of data across multiple files across a cluster to balance large amounts of data across disks or nodes, and also to achieve parallel access to the data to reduce overall execution time for queries. You can have multiple partitions per disk, and the separation of data is managed by specifying a hash, range, or

combination of the two, on key columns of a table. Most query and storage engines support this capability, relatively transparently to the application.

You should never use sharding as a substitute for partitioning. That would be a very expensive alternative from the perspective of scale, performance, and operational manageability. In fact, you can view them as complementary in helping applications scale. How to use sharding and partitioning is an application architecture and design decision.

Applications need to be shard-aware. It is possible that you could scale by sharding data across servers or clusters, and some query engines might facilitate that. But scaling parallel queries across shards is a much more limiting and inefficient architecture than using a single parallel query engine to process partitioned data across an MPP cluster.

If each shard has a large amount of data that can span a decent-size cluster, you are much better off using partitioning and executing a query in parallel against that shard. However, messaging, repartitioning, and broadcasting data across these shards to do joins is very complex and inefficient. But if there is no reason for queries to join data across shards, or if cross-shard processing is rare, certainly there is a place for partitioned shards across clusters. The focus in this report on partitioning.

In many ways the same challenges exist for query engines trying to use other query engines, such as PostrgreSQL or Derby SQL, where essentially the query engine becomes a data federation engine (discussed later in this report) across shards.

Statistics

Statistics are necessary when query engines are trying to generate query plans or understand whether a workload is operational or analytical. In the single-row-access scenario described earlier, if the predicate(s) used in the query only cover some of the columns in the key, the engine must figure out whether the predicate(s) cover the leading columns of the key, or any of the key columns. Let us assume that leading columns of the key have equality predicates specified on them. Then, the query engine needs to know how many rows would qualify, and how the data that it needs to access is spread across the nodes. Based on the partitioning scheme—that is,

how data is spread across nodes and disks within those nodes—the query engine would need to determine whether it should generate a serial plan or a parallel plan, or whether it can rely on the storage engine to very efficiently determine that and access and retrieve just the right number of rows. For this, it needs some idea as to how many rows will qualify.

The only way for the query engine to know the number of rows that will qualify, so as to generate an efficient query plan, is to gather statistics on the data ahead of time to determine the cardinality of the data that would qualify. If multiple key columns are involved, most likely the cardinality of the combination of these columns is much smaller than the product of their individual cardinalities. So the query engine must have multicolumn statistics for key columns. Various statistics could be gathered. But at the least it needs to know the unique entry counts, and the lowest and highest, or second lowest and second highest, values for the column(s).

Skew is another factor to take into account. Skew becomes relevant when data is spread across a large number of nodes and there is a chance that a large amount of data could end up being processed by just a few nodes, overwhelming those nodes and affecting all of the workloads running on the cluster (given that most would need those nodes to run), whereas other nodes are waiting on these few nodes to finish executing the query. If the only types of workloads the query engine has to handle are OLTP or operational ones, the chances are it does not need to process large amounts of data and therefore does not need to worry about skew in the data, other than at the data partitioning layer, which can be controlled via the choice of a good partitioning key. But if it's also processing BI and analytics workloads, skew could become an important factor. Skew also depends on the amount of parallelism being utilized to execute a query.

For situations in which skew is a factor, the database cannot completely rely on the typical *equal-width histograms* that most databases tend to collect. In equal-width histograms, statistics are collected with the range of values divided into equal intervals, based on the lowest and highest values found and the unique entry count calculated. However, if there is a skew, it is difficult to know which value has a skew because it would fall into a specific interval that has many other values in its range. So, the query engine has to either

collect some more information to understand skew or use *equal-height histograms*.

Equal height histograms have the same number of rows in each interval. So if there is a skewed value, it will probably span a larger number of intervals. Of course, determining the right interval row size and therefore number of intervals, the adjustments needed to highlight skewed values versus nonskewed values (where not all intervals might end up having the same size) while minimizing the number of intervals without losing skew information is not easy to do. In fact, these histograms are a lot more difficult to compute and lead to a number of operational challenges. Generally, sampling is needed in order to collect these statistics fast, because the data must be sorted in order to put them into these interval buckets. You need to devise strategies to incrementally update these statistics and when to update them. These come with their own challenges.

Predicates on Nonleading Key Columns or Nonkey Columns

Things begin getting really tricky when the predicates are not on the leading columns of the key but are nonetheless on some of the columns of the key. What could make this more complex is an IN list against these columns with OR predicates, or even NOT IN conditions. A capability called Multidimensional Access Method (or MDAM) provides efficient access capabilities when leading key column values are not known. In this case, the multicolumn cardinality of leading column(s) with no predicates needs to be known in order to determine if such a method will be faster in accessing the data than a full table scan. If there are intermediate key columns with no predicates, their cardinalities are essential, as well. So, multikey column considerations are almost a must if these are not operational queries with efficient keys designed for their access.

Then, there are predicates on nonkey columns. The cardinality of these is relevant because it provides an idea as to the reduction in size of the resulting number of rows that need to be processed at upper layers of the query—such as joins and aggregates.

All of the above keyed and nonkeyed access cardinalities help determine join strategies and degree of parallelism.

If the storage engine is a columnar storage engine, the kind of compression used (dictionary, run length, and so on) becomes important because it affects scan performance. Also, the sequence in which these predicates should be evaluated becomes important in that case because you want to reduce as many rows as early as possible, so you want to begin with predicates on columns that give you the largest reduction first. Here too, clustered access versus a full scan versus efficient mechanisms to reduce scans of column values—which might be provided by the storage engine—are relevant. As are statistics.

Indexes and Materialized Views

Then, there is the entire area of indexing. What kinds of indexes are supported by the storage engine or created by the query engine on top of the storage engine? Indexes offer alternate access paths to the data that could be more efficient. There are indexes designed for index-only scans to avoid accessing the base table by having all relevant columns in the index.

There are also materialized views. Materialized views are relevant for more complex workloads for which you want to prematerialize joins or aggregates for efficient access. This is highly complex because now you need to figure out if the query can actually be serviced by a materialized view. This is called *materialized view query rewrite*.

Some databases call indexes and materialized views by different names, such as *projections*, but ultimately the goal is the same—to determine what the available alternate access paths are for efficient keyed or clustered access to avoid large, full-table scans.

Of course, as soon as you add indexes, a database now needs to maintain them in parallel. Otherwise, the total response time will increase by the number of indexes it must maintain on an update. It has to provide transactional support for indexes to remain consistent with the base tables. There might be considerations such as colocation of the index with the base table. The database must handle unique constraints. One example in BI and analytics environments (as well as some other scenarios) is that bulk loads might require an efficient mechanism to update the index and ensure that it is consistent.

Indexes are used more for operational workloads and much less so for BI and analytical workloads. On the other hand, materialized

views, which are materialized joins and/or aggregations of data in the base table, and similar to indexes in providing quick access, are primarily used for BI and analytical workloads. The increasing need to support operational dashboards might be changing that somewhat. If materialized view maintenance needs to be synchronous with updates, they too can be a large burden on updates or bulk loads. If materialized views are maintained asynchronously, the impact is not as severe, assuming that audit logs or versioning can be used to refresh them. Some databases support user-defined materialized views to provide more flexibility to the user and not burden operational updates. The query engine should be able to automatically rewrite queries to take advantage of any of these materialized views when feasible.

Storage engines also use other techniques like Bloom filters and hash tables to speed access. The query engine needs to be aware of all the alternative access paths made available by the storage engine to get at the data. It also needs to know how to exploit them or implement them itself in order to deliver high performance for operational and analytical workloads.

Degree of Parallelism

All right, so now we know how we are going to scan a particular table, we have an estimate of rows that will be returned by the storage engine from these scans, and we understand how the data is spread across partitions. We can now consider both serial and parallel execution strategies, and balance the potentially faster response time of parallel strategies against the overhead of parallelism.

Yes, parallelism does not come for free. You need to involve more processes across multiple nodes, and each process will consume memory, compete for resources in its node, and that node is subject to failure. You also must provide each process with the execution plan, for which they must then do some setup to execute. Finally, each process must forward results to a single node that then has to collate all the data.

All of this results in potentially more messaging between processes, increases skew potential, and so on.

The optimizer needs to weigh the cost of processing those rows by using a number of potential serial and parallel plans and assess

which will be most efficient, given the aforementioned overhead considerations.

To offer really high concurrency for all workloads (including large EDW workloads that can have a very large number of concurrent queries being executed in seconds or subseconds), the optimizer needs to assess the degree of parallelism needed for each query. To execute a query most efficiently in terms of response time and resources used, the query engine should base each operation's degree of parallelism on the cardinality of rows that operation needs to process. Scans that filter rows, joins, and aggregates can often lead to substantial reduction in data. It makes no sense to use, say, 100 nodes to execute an operation when 5 nodes are sufficient to do so. Not only that, as soon as the maximum degree of parallelism required by the query—based on the cardinality of the data it will process—is known, the query can be allocated to run on a segment, or subset of the nodes, in the cluster. If the cluster were divided into a number of equal segments, it can be very efficiently used by allocating queries to run in those segments, or a combination of segments, thereby dramatically increasing concurrency. This yields the twin benefits of using system resources very efficiently while gaining more resiliency by reducing the degree of parallelism. This is illustrated in Figure 1-2.

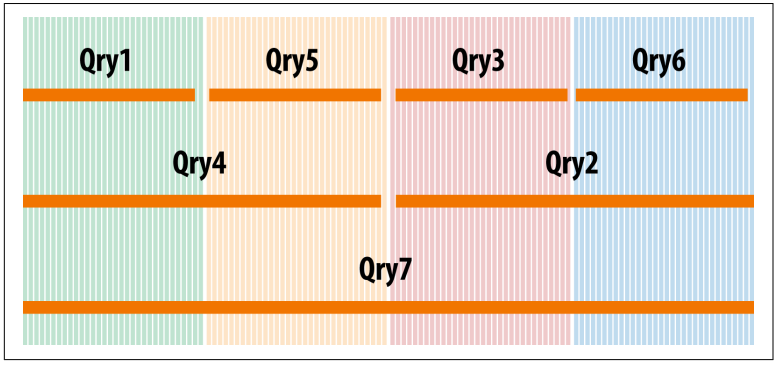


Figure 1-2. Nodes used based on degree of parallelism needed by query. Each node is shown by a vertical line (128 nodes total) and each color band denotes a segment of 32 nodes. Properly allocating queries can increase concurrency, efficiency, and resiliency while reducing the degree of parallelism.

As the cluster is expanded and newer technology is used for the added nodes, with potentially more resource capacity than existing nodes on the cluster, this segmentation can help use that capacity more efficiently by allocating more queries to the newer segment.

Reducing the Search Space

The options discussed so far provide optimizers a large number of potentially good query plans. There are various technologies such as Cascades, used by NonStop SQL (and now part of Apache Trafodion) and Microsoft SQL Server, that are great for optimizers but have the disadvantage of having this very large search space of query plans to evaluate. For long-running queries, spending extra time to find a better plan by trawling through more of that search space can have dramatic payoffs. But for operational queries, the returns of finding a better plan diminish very fast, and compile-time spent looking for a better plan becomes an issue, because most operational queries need to be processed within seconds or even subseconds.

One way to address this compile-time issue for operational queries is to provide query plan caching. These techniques cannot be naive string matching mechanisms alone, even after literals or parameters have been excluded. Table definitions could change since the last time the plan was executed. A cached plan might need to be invalidated in those cases. Schema context for the table could change, not obvious in the query text. A plan handling skewed values could be substantially different from a plan on values that are not skewed. So, sophisticated query plan caching mechanisms are needed to reduce the time it takes to compile while avoiding a stale or inefficient plan. The query plan cache needs to be actively managed to remove least recently used plans from cache to accommodate frequently used ones.

The optimizer can be a cost-based optimizer, but it must be rules driven, with the ability to add heuristics and rules very efficiently and easily as the optimizer evolves to handle different workloads. For instance, it should be able to recognize patterns. A star join is not likely in an operational query. But for BI queries, it could detect such a join. If it does, it can use specialized indexes designed for that purpose, or it could decide to do a cross product of the dimension tables (something optimizers otherwise avoid), before doing a nested join to the fact table, instead of scanning the entire fact table and doing repeated hash joins against the dimension tables.

Join Type

That brings us to join types. For operational workloads, a database needs to support nested joins and probe cache for nested joins. A probe cache for nested joins is where the optimizer understands that access to the inner table will have enough repetition due to the unsorted nature of the rows coming from the outer table, so that caching those results would really help with the join.

For BI and analytics workloads, a merge or hybrid hash join would most likely be more efficient. A nested join can be useful for such workloads some of the times. However, nested join performance tends to degrade rapidly as the amount of data to be joined grows.

Because a wrong choice can have a severe impact on query performance, you need to add a premium to the cost and not choose a plan purely on cost. Meaning, if there is a nested join with a slightly lower cost than a hash join, you don't want to choose it, because the downside risk of it being a bad choice is huge, whereas the upside might not be all that better. This is because cardinality estimations are just that: estimations. If you chose a nested join or serial plan and the number of rows qualifying at run time are equal to or lower than compile time estimations, then that would turn out to be a good plan. However, if the actual number of rows qualifying at run time is much higher than estimated, a nested or serial plan might not be just bad, it can be devastating. So, a large enough risk premium can be assigned to nested joins and serial plans, so that hash joins and parallel plans are favored, to avoid the risk of a very bad plan. This premium can be adjusted, because different workloads respond differently to costing, especially when considering the balance between operational queries and BI or analytics queries.

For BI and analytics queries, if the data being processed by a hash join or a sort is large, detecting memory pressure and overflowing gracefully to disk is important. Operational queries, however, generally don't have to deal with large amounts of data to the point that this is an issue.

Data Flow and Access

The architecture for a query engine needs to handle large parallel data flows with complex operations for BI and analytics workloads as well as quick direct access for operational workloads.

For BI and analytics queries for which larger amounts of data are likely to be processed, the query execution architecture should be able to parallelize at multiple levels. The first level is *partitioned parallelism*, so that multiple processes for an operation such as join or aggregation are executed in parallel. Second is at the operator level, or *operator parallelism*. That is, scans, multiple joins, aggregations, and other operations being performed to execute the query should be running concurrently. The query should not be executing just one operation at a time, perhaps materializing the results on disk in between as MapReduce does.

All processes should be executing simultaneously with data flowing through these operations from scans to joins to other joins and aggregates. That brings us to the third kind of parallelism, which is *pipeline parallelism*. To allow one operator in a query plan (say, a join) to consume rows as they are produced by another operator (say, another join or a scan), a set of up and down interprocess message queues, or intraprocess memory queues, are needed to keep a constant data flow between these operators (see Figure 1-3).

Operator-Level Degree of Parallelism

Figure 1-3 also illustrates how the optimizer needs to figure out the degree of parallelism required for each operator, based on the cardinality of rows it estimates that operator will have to process at that execution step. This is illustrated by one scan with two degrees of parallelism, the other scan and GROUP BY with three degrees of parallelism, and the join with four degrees of parallelism. The right degree of parallelism can then be used for each operator when executing the query. This leads to much more efficient use of system resources than using the entire cluster for every operation. This was also discussed in another context in "Degree of Parallelism" on page 13, where this information is used to determine the degree of parallelism needed by the entire query, as illustrated in Figure 1-2.

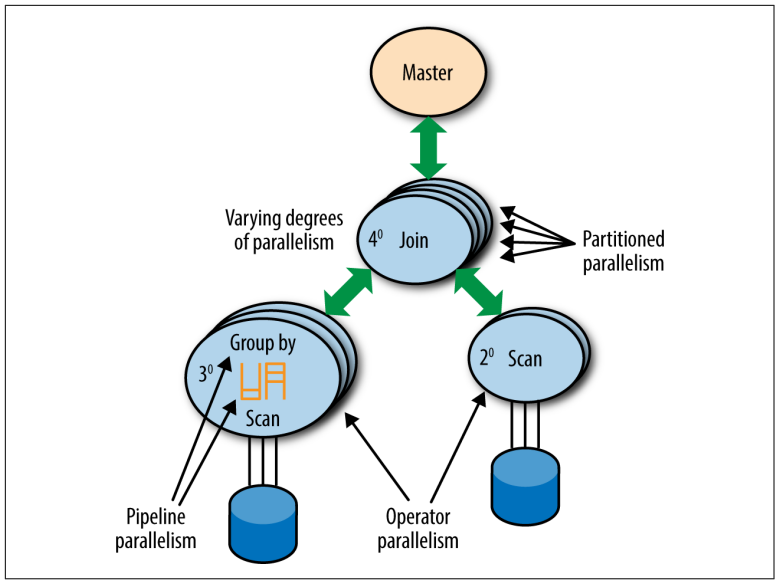


Figure 1-3. Exploiting different levels of parallelism

But for OLTP and operational queries, this data flow architecture (Figure 1-4) can be a huge overhead. If you are accessing a single row, or just a few rows, you don't need the queues and complex data flows. In such a case, you can have optimizations to reduce the path length and quickly just access and return the relevant row(s).

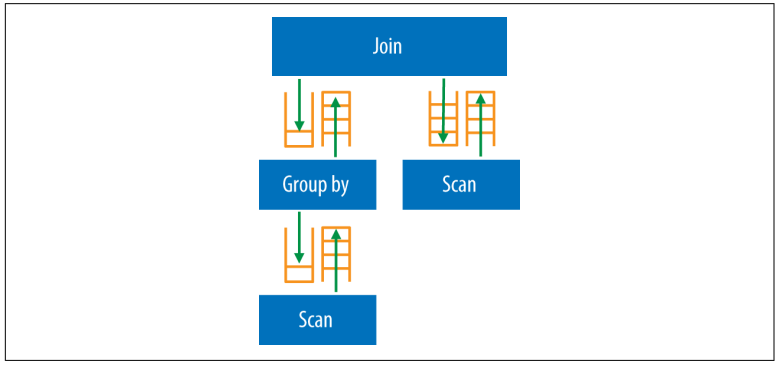


Figure 1-4. Data flow architecture

While you are optimizing for OLTP queries with fast paths, for BI and analytics queries you need to consider prefetching blocks of data, provided the storage engine supports this, while the query engine is busy processing the previous block of data. So the nature

of processing varies widely for the kind of workloads the query engine is processing, and it must accommodate all of these variants. Figures 1-5 through 1-8 illustrate how these processing scenarios can vary from a single row or single partition access serial plan or parallel multiple direct partition access for an operational query, to multitiered parallel processing of BI and analytics queries to facilitate complex aggregations and joins.

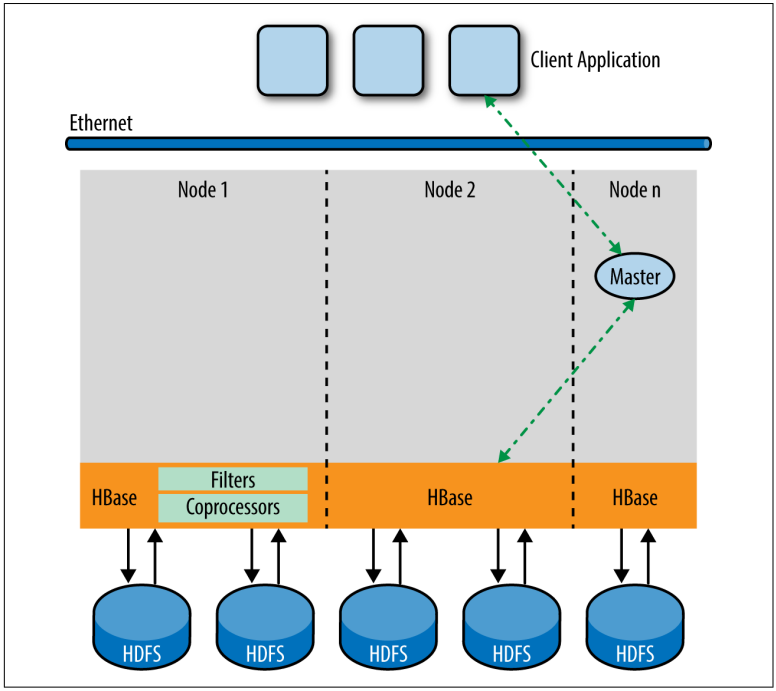


Figure 1-5. Serial plan for reads and writes of single rows or a set of rows clustered on key columns, residing in a single partition. An example of this is when a single row is being inserted, deleted, or updated for a customer, or all the data being accessed for a customer, for a specific transaction date, resides in the same partition.

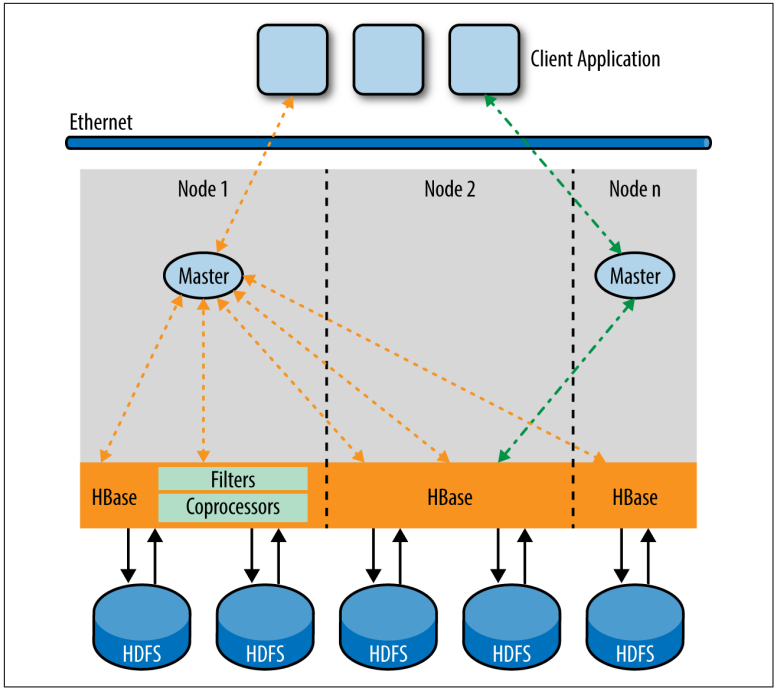


Figure 1-6. Serial or parallel plan, based on costing, where the Master directly accesses rows across multiple partitions. This occurs when few rows are expected to be processed by the Master, or parallel aggregations or joins are not required or beneficial. An example of this could be when a customer's data that needs to be accessed is spread across partitions based on transaction date.

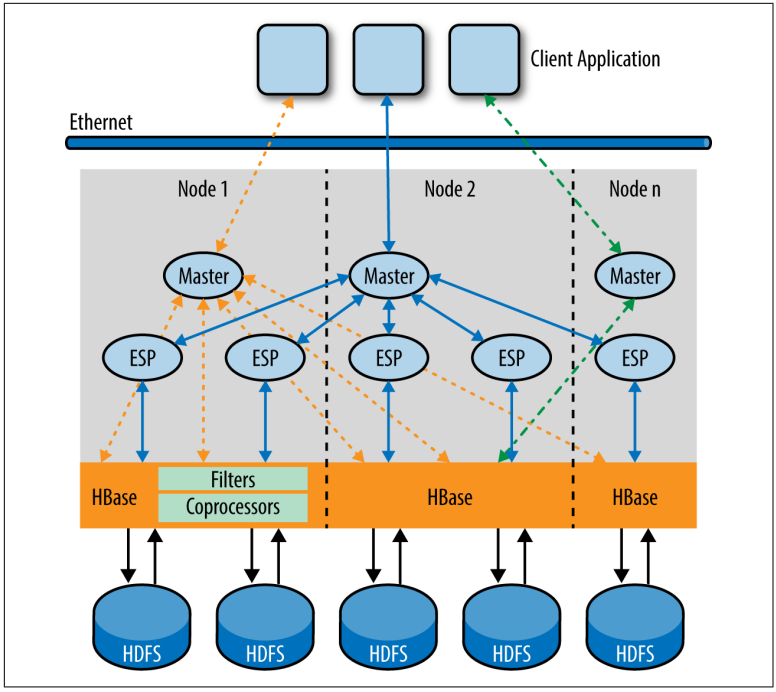


Figure 1-7. Parallel plan where a large amount of data needs to be processed, and parallel aggregation or collocated join done by parallel Executor Server Processes would be a lot faster than doing it all in the Master.

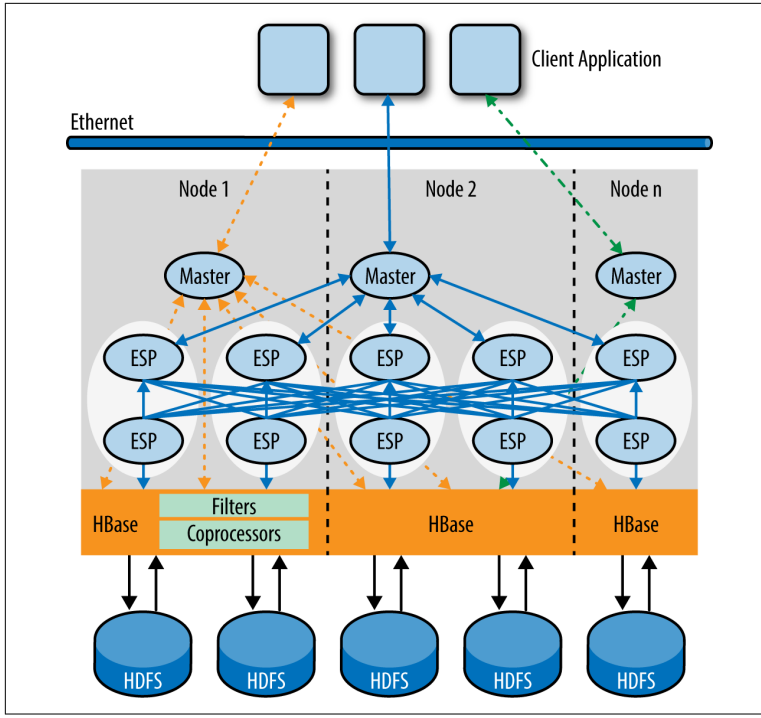


Figure 1-8. Parallel plan where a large amount of data needs to be processed, and either multiple joins, or joins requiring repartitioning or broadcasting of data, would be required.

Mixed Workload

One of the biggest challenges for HTAP is the ability to handle mixed workloads; that is, both OLTP queries and the BI and analytics queries running concurrently on the same cluster, nodes, disks, and tables. Workload management capabilities in the query engine can categorize queries by data source, user, role, and so on and allow users to prioritize workloads and allocate a higher percentage of CPU, memory, and I/O resources to certain workloads over others. Or, short OLTP workloads can be prioritized over BI and analytics workloads. Various levels of sophistication can be used to manage this at the query level.

However, storage engine optimization is required, as well. The storage engine should automatically reduce the priority of longer running queries and suspend execution on a query when a higher priority query needs to be serviced, and then go back to running the

longer running query. This is called *antistarvation*, because you don't want to starve out higher priority queries from running, or even same or lower priority queries from running, while a single query hogs all the resources. An alternate way to address this might be to direct update workloads to the primary partition for a specific row being updated and query workloads to its replicates if the storage engine can facilitate this without loss of consistency.

Streaming

More and more applications need incoming streams of data processed in real time, necessitating the application of functions, aggregations, and trigger actions across a stream of data, often time-series data, over row count or time-based windows. This is very different from processing statistical or user-defined functions, sophistical algorithms, aggregates, and even Online Analytical Processing (OLAP) window functions over data persisted in a table on disk or memory. Even though Jennifer Widom had proposed new SQL syntax to handle streams in 2008, there is no standard SQL syntax to process streaming data. Query engines have to be able to deal with this new data processing paradigm.

Feature Support

Last but not least is the list of features you need to support for operational and analytical workloads. These features range from referential integrity, stored procedures, triggers, various levels of transactional isolation and consistency, for operational workloads; to materialized views; fast/bulk Extract, Transform, Load (ETL) capabilities; and OLAP, time series, statistical, data mining, and other functions for BI and analytics workloads.

Features common to both types of workloads are many. Some of the capabilities a query engine needs to support are scalar and table mapping UDFs, inner, left, right, and full outer joins, un-nesting of subqueries, converting correlated subqueries to joins, predicate push down, sort avoidance strategies, constant folding, recursive union, and so on.

This is not close to an exhaustive list, but supporting all these capabilities for these different workloads takes a huge investment of resources.

Challenge: Supporting Multiple Storage Engines

It is not a revelation that row-optimized stores work well for OLTP and operational workloads, whereas column stores work well for BI and analytics workloads. Write-heavy workloads benefit from writing out rows in row-wise format. For Hadoop, there is HBase for low latency workloads, and column-wise ORC Files or Parquet for BI and analytics workloads. There is a promise—especially by in-memory database vendors—that column-wise structures work well for all kinds of workloads. In reality, there is often a performance compromise when the same data structure is used for workloads with very different access patterns. Cloudera promises a storage engine in Kudu that will meet the requirements of a set of mixed workloads, but it clearly states that this compromises performance versus HBase for low latency and Parquet for BI and analytics workloads.

The NoSQL revolution has demonstrated the need to support multiple data models in order to service the varied needs of workloads today. But to reduce data duplication and movement, the ideal would be if multiple storage engines could be supported by a single query engine. Data could reside in the data structure/storage engine ideal for the pattern of data access needed, and the query engine can transparently support all of these. To this end, Hive, Impala, Drill, Spark, Presto, and other SQL-on-Hadoop solutions support multiple data structures. However, these query engines don't yet support OLTP or operational workloads.

The issue is not just around the inability of storage engines to service multiple kinds of workloads. It is also difficult for query engines to be able to support multiple storage engines, even if it is understood that some data transformation and movement might be necessary from one storage engine to another to support mixed workloads. There are numerous challenges. We will discuss some of them next.

Statistics

As was discussed earlier, statistics are needed to support any query workload in order to generate a good query plan or to even understand whether the workload is operational or analytical. So, when a

new storage engine is added, you need to add support for statistics gathering. Does the storage engine have an efficient mechanism to sample rows? Does it have an easy way to query new data for incremental stats computations? Does it already gather count and other metrics that can be used for fast stats collection? Does it gather counts on records deleted, changed, or inserted, so that the query engine can determine when to schedule another update of its statistics? You need to answer a number of questions to set up the gathering of statistics to work efficiently with the storage engine.

Key Structure

For operational workloads, keyed access is important for subsecond response times. Single-row access requires access via a key. Ideally multicolumn keys are supported, because invariably transactional or fact tables have multicolumn keys. If multicolumn keys are not supported, the query engine needs to map the multiple primary key columns to the single storage engine key. Also, short operational queries often require clustered access to retrieve a small number of rows. Range access over clustered keys helps operational workloads meet Service Level Agreements (SLAs). The query engine needs to understand what keyed access options are available for the storage engine in order to appropriately exploit them for the most efficient access. It needs to optimize this access for each storage engine it supports.

Partitioning

How the storage engine partitions data across disks and nodes is also very important for the query engine to understand. Does it support hash and/or range partitioning, or a combination of these? How is this partitioning determined? Does the query engine need to salt data so that the load is balanced across partitions in order to avoid bottlenecks? If it does, how can it add a salt key, say as the leftmost column of the table key, and still avoid table scans? Does the storage engine handle repartitioning or rebalancing of partitions as the cluster is expanded or contracted, or does the query engine need to worry about doing that? This can get very complex because users might need full read and write access while this repartitioning of data is occurring. How the data is spread across partitions is very important if the query engine is going to have parallel processes working on data from these partitions. It needs to try to localize

access and reduce repartitioning of data (also called *shuffle*), or sending data across nodes, as much as possible. Without understanding the key and partitioning structures of the storage engine, it is impossible to efficiently process data for a query. Operational queries might not often encounter some of these challenges of processing data in parallel. But how data is partitioned can influence whether fast operational queries can be serviced by access to a single partition, disk, or node, versus having to access multiple partitions to service a query with stringent service level objectives.

Data Type Support

The query engine needs to understand what data types are supported by the storage engine, and what constraints the storage engine enforces on those types, so that the query engine can enforce any constraints that the storage engine doesn't. For example, in some cases, it is better for the query engine to store the data in its own encoded data type if the storage engine just supports string arrays. Or, if the query engine enforces CHECK constraints, again, can it subcontract that to the storage engine, or does it need to enforce them? Certainly, referential constraints enforcement is not something a storage engine will likely take on.

Does the storage engine provide full character set support so that UTF-8 support between the query and storage engine is sufficient for the query engine to support all character sets? What about collations?

Does the storage engine provide compression or encryption support, or will the query engine need to provide that support at its level? The query engine will most likely be the one issuing the table creation DDL statements and needs to know which options for block level and key compression, and other table structure options need to be exposed by it to the user. In some cases, a storage engine table might be mapped to a query engine's view of that table.

Projection and Selection

Most storage engines support projection, wherein only the desired columns are returned by the storage engine. However, if that is not the case, the query engine must unpack the data and do the projection. Similarly, predicate evaluation provided by the storage engine needs to be understood. What predicates will the storage engine be

able to evaluate (=, <, >, <>, between, LIKE)? Can it evaluate predicates on multiple columns, or multicolumn predicates ((a,b) > (5,1))? Can it handle an IN list? How long an IN list? Generally, can it handle a combination of ORs and ANDs, and to what level of complexity? Does it automatically determine the most efficient sequence in which to apply the predicates, such as evaluating predicates on a column that yields the highest reduction in cardinality first, especially if the storage engine is a column store? Can it handle predicates involving multiple columns of the same table; for instance, c1 > c2? Besides literals, can it handle string expressions (`STR(c1, 3, 2)='10'`), or date and time expressions? How complex can these expressions be? How does the storage engine handle default or missing values (NULLs)? The query engine needs to understand all of these aspects of a storage engine in order to determine what predicates it can push to the storage engine and which ones it needs to implement itself.

As memory prices have gone down, it has become possible to configure large amounts of memory per node. The promise of nonvolatile memory is around the corner. Not only are storage engines designing efficient structures in memory for access to data for both transactional and analytical workloads, but they are implementing techniques to make operations such as projection and selection more efficient. They are taking advantage of L1, L2, and L3 CPU cache to process data at even faster speeds than main memory, implementing vectorization to process sets of rows/column values at a time, and coming up with other innovative ways, such as minimizing or eliminating the cost of serialization/deserialization, to push data processing speeds to the ultimate while utilizing the hardware to the utmost. This moves the bottleneck from disk to main memory, and ultimately to the CPU. Efficient use of the CPU now becomes imperative. The query engine must be able to exploit these storage engine efficiencies and be frugal in the use of CPU itself.

Extensibility

Some storage engines can run user-defined code, such as coprocessors in HBase, or before and after triggers, in order to implement more functionality on the storage engine/server side to reduce the amount of message traffic and improve efficiencies.

For example, the optimizer might be smart enough to do eager aggregation if it can push that aggregation to the storage engine.

This is especially effective if the data is clustered on the GROUP BY columns or if the number of groupings are expected to be small. If the storage engine does not support this, the query engine could use coprocessors to do the same. Complex predicate evaluations with expressions and functions, collocated joins and index maintenance, security enforcement, some ANSI Trigger actions, and even transactional support can be performed at this level.

Security Enforcement

Handling security is another point of contact between the query engine and the storage engine. The storage engine might have its own underlying security implementation. If it is well integrated with the SQL model, the query engine can use it. Otherwise, the query engine must administer privileges to schemas, tables, columns, and even rows in the case of fine-grained access control. It might need to integrate with, and even map, its authorization framework to that of the storage engine. If there are other considerations related to security, for example Hadoop security solutions such as Sentry or Ranger managing these objects, the query engine needs to integrate with that security framework. Depending on support for security logging and other Security Information and Event Management (SIEM) capabilities, the query engine must be able to integrate with those storage engine and platform capabilities as well, or provide its own. With HTAP, this is all made more complicated by the fact that there might be multiple storage engines that need to be supported.

Transaction Management

We can assume that a storage engine will provide some level of replication for high availability, backup and restore, and some level of multi–data center support. But transactional support can be a different thing altogether. There are the Atomic, Consistent, Isolated, Durable (ACID) and Basic Availability, Soft-state, Eventual consistency (BASE) models of transaction management. Depending on the support provided by the storage engine, the query engine might need to provide full multirow, multitable, and multistatement transaction support. It also needs to provide online backup, transactionally consistent recovery, often to a point-in-time, and multi–data center support with active-active synchronous updates as well as eventual consistency. It needs to be able to integrate this implementation with write-ahead logging and other mechanisms of the stor-

age engine so as to take advantage of the underlying replication and other high-availability capabilities of the engine. These can be very different, as is already the case with Cassandra, HBase, or Hive, for example; each has its own transactional model.

Also, going back to extensibility of storage engine capabilities, such as coprocessors in HBase, the query engine needs to implement this transactional capability as close to and integrated with the storage engine as possible to be efficient and distributed. Otherwise, it will end up with a more generalized implementation, but not a scalable and efficient one.

Metadata Support

The query engine needs to add metadata support for the storage engine tables being supported. There are potential mappings (such as catalog, name, location, as well as data type mappings); extended options specific to the storage engine (such as compression options); supporting multiple column families—for example, mapping data partition options to underlying storage engine structures (e.g., HBase regions); and so on. When someone changes the storage engine table externally, how is this metadata updated, or how is it marked as invalid until someone fixes the discrepancy?

In fact, if there is access to the storage engine external to the query engine, there is a whole can of worms to deal with, such as how can transactional consistency and data integrity be guaranteed? What should the query engine do if the data in a column is not consistent with the data type it is supposed to have, or contains invalid data, perhaps because it has been updated by some other means? Also, do you allow secondary indexes, views, constraints, materialized views, and so on to be created on such a table? Not only does metadata support need to be provided for this support, but the issue of inconsistency potentially caused by external access has to be addressed.

There can be differences in metadata support needed for operational workloads versus BI and analytic workloads. For example, operational features such as referential integrity, triggers, and stored procedures need metadata support that is not needed for BI workloads, whereas metadata support for materialized views is not often needed for operational workloads.

Performance, Scale, and Concurrency Considerations

There are various performance, scaling, and concurrency considerations for each different storage engine with which a query engine needs to integrate in order to support HTAP workloads. For instance, is there a bulk-load mechanism available and what are its implications on transactional consistency and availability of the table during such loads? Are rowset inserts available so that large amounts of inserts can be performed efficiently? Are bulk-extract capabilities available, with sets of rows being returned in each buffer, instead of just a row at a time interface? Are faster scan options available, such as snapshot scans? Is the ability to request prefetching of data in large blocks available, when the query engine knows that a large amount of data is expected to be scanned, perhaps for an extract or for large complex queries? How can the query engine parallelize updates and access to the storage engine? If the query engine supports executor processes to parallelize queries, how best should it access the storage engine files/partitions and regions? What levels of concurrent access can the storage engine support? How best can the query engine utilize what the storage engine supports and compensate for what it does not? This is tough to determine, and more difficult to implement.

Error Handling

Error handling, too, is different for each storage engine that a query engine might need to support for HTAP. What are the error handling mechanisms available in the storage engine and how are these errors logged? Does the storage engine log errors or does the query engine need to interpret each type of error and log it, as well as provide some meaningful error message and remediation guidance to the user?

Other Operational Aspects

Although error handling is certainly one operational aspect, there are other considerations as well. For example, in the case of HBase there are issues with compaction, or splitting. These have significant impact on performance and user workloads, let alone transactional workloads. How will these be managed by the query engine to provide the best customer experience?

Suffice it to say that supporting a storage engine is not a trivial task. Sure, you can take shortcuts; support, say, InputFormat or OutputFormat; or even instrument a UDF or some connector technology to make the query engine talk to the storage engine. But to really integrate with a storage engine and exploit it to its best capabilities and provide the most efficient parallel implementation, it takes a lot of work and effort. If you need to support multiple storage engines to reach the database nirvana of one query engine on top of multiple storage engines, know that your task is cut out for you.

Just as the industry came together to create standard interfaces to query engines such as Java Database Connectivity (JDBC) and Open Database Connectivity (ODBC), it would be nice if disparate storage engines could provide a more common interface to query engines. Query engines could quickly and with less effort identify the capabilities of the storage engine via such an interface, integrate with it, and exploit its full capabilities. Maybe the industry will reach a point of maturity at which this could happen. For now, the APIs for the various storage engines and their capabilities are very diverse.

Although some of these challenges look very similar to the challenges that data federation engines must deal with, the difference is that data federation engines are interfacing with query engines using the standard JDBC/ODBC interfaces; in other words, subcontracting the query out to another query engine. This level of abstraction is very different from the much deeper level of integration required between a query engine and a storage engine geared toward a much higher degree of efficiency and performance. Just imagine splitting a proprietary RDBMS such as Oracle into a query and storage engine. That level of integration is quite a different than a federation engine interfacing with an Oracle instance.

Challenge: Same Data Model for All Workloads

As with many other terms in the IT industry, the words "data model" are overloaded. In the RDBMS context, a data model is the level of normalization that you implement. Data for operational workloads is normalized from first normal form to sixth normal form. For historical data that serves BI and analytics workloads, you need a different data model, such as a star schema or an even more elaborate snowflake schema, with dimension tables, fact tables, and so forth. The data model is very different to accommodate the vary-

ing access patterns seen in OLTP and operational queries, which are focused on specific entities such as customers, orders, suppliers, and products versus a broader, more historical perspective for BI or analytics queries.

Does HTAP render such delineation of data models unnecessary? I have seen claims to this effect by some in-memory database providers but have not seen any benchmark—artificial (TPC) or real-world—that shows that BI and analytics workloads work well on a third normal form implementation. Nor have I seen anything that indicates that an OLTP/operational system performs well with a star schema, in-memory or otherwise. Benchmarks aside, performance is determined by the kinds of workloads a customer is running, and these workloads have no semblance to TPC benchmarks, or even a mix of these, if one existed.

So, it is one thing to say that you understand that there is some level of data duplication, transformation, aggregation, and movement from one data structure, data model, and storage engine to another, and you want the query engine to make it all look seamless. That can be difficult to do but might be achievable. However, it is quite another thing to say that you don't want any data duplication, transformation, aggregation, or movement, of data, and you don't want any compromise on performance for any of the workloads. With Kudu, Cloudera is hoping there are a number of folks who do want that single copy of data but are willing to sacrifice both the high-end operational and analytical performance to achieve that. That is, they would be willing to compromise performance, for design, development, and operational simplicity. The jury is still out on this. But it would be interesting if the IT industry were to swing back the pendulum in that direction after having aggressively pursued a NoSQL polyglot persistence agenda, where performance at scale and concurrency was not just a requirement, it was the *only* requirement.

But, as alluded to earlier, there are different definitions of data models now. There are key-value, ordered key-value, Bigtable, document, full-text search, and graph data models. And to achieve the nirvana of a single storage engine, people keep stretching these data models to their limits by trying to have them do unnatural things for which they were not designed. Thus, there are entire parent-child relations implemented in documents, claiming that none of the RDBMS capabilities are needed anymore, and that document stores can meet all workload requirements. Or, is it now the time for graph

databases? Can we stuff all kinds of context in edges and vertices to essentially support the variety of workloads? In reality, even though each of these data models do solve certain problems, none of them on their own can support all of the varied workloads that an enterprise needs to deal with today.

Leveraging document capabilities, text search where necessary, graph structure where hierarchical or network-type relationships need to be accessed efficiently, and key-value–based models that support semistructured or unstructured data very well, along with a fully structured relational database implementation, would certainly give you the best of all the worlds. Although some query engines are certainly trying to support as many of these as possible, as we have seen, the challenge is in the efficient integration of these storage engines with the query engine. Add to that the challenge of blending the standard SQL API (which is actually not that standard; there are proprietary implementations that vary widely) with nonstandard APIs for these other data models.

Either way you look at the term "data model," HTAP has a number of challenges to overcome if it is to support multiple, if not all, data models, servicing all enterprise workloads, with the same query engine.

Challenge: Enterprise-Caliber Capabilities

As larger enterprises began expanding their BI workloads toward more analytics and then advanced analytics, such as data mining and machine learning, they initially complemented their BI and analytical workloads running on proprietary databases and platforms, with additional workloads running on Hadoop but focused on big data. But many enterprises followed by offloading some of these workloads from their proprietary databases onto Hadoop. Some companies are now offering the capability to run full RDBMS OLTP and operational workloads on Hadoop as well, thereby expanding the types of workloads toward the hybrid transactional and analytical model. But to really host all workloads, including mission-critical workloads, an HTAP database engine needs to provide enterprise-caliber capabilities, which we discuss in the following sections.

High Availability

Availability is an important metric for any database. You can get around 99.99 percent availability with HBase, for example. Four nines, as they are known, means that you can have about 52.56 minutes of unscheduled downtime per year. Many mission-critical applications strive for five nines, or 5.26 minutes of downtime per year. Of course, the cost of that increase in availability can be substantial. The question is this: what high-availability characteristics can the query and storage engine combined provide?

Typically, high availability is very difficult for databases to implement, with the need to address the following questions:

- Can you upgrade the underlying OS, Hadoop distribution, storage or query engine to a new version of software while the data is available not only for reads, but also for writes, with no downtime—in other words, support rolling upgrades?
- Can you redistribute the data across new nodes or disks that have been added to the cluster, or consolidate them on to fewer nodes or disks, completely online with no downtime? Related to this is the ability to repartition the data for whatever reason. Can you do that online? Online could mean just read access or both read and write access to the data during the operation.
- Can you make online DDL changes to the database, such as changing the data type of a column, with no impact on reads and writes? Addition and deletion of nonkey columns have been relatively easy for databases to do.
- Can you create and drop secondary indexes online?
- Is there support for online backups, both full and incremental?

Which of the preceding capabilities your applications need will depend on the mix of operational and analytical workloads you have as well as how important high availability is for those workloads.

Security

Security implementations between operational and analytical workloads can be very different. For operational workloads, generally security is managed at the application level. The application interfaces with the user and manages all access to the database. On the other hand, BI and analytics workloads could have end users work-

ing with reporting and analytical tools to directly access the database. In such cases, there is a possibility that authorization is pushed to the database and the query and storage engine need to manage that security.

Integration into SIEM systems could be pertinent for analytical workloads, as well. But certainly many operational workloads require a higher degree of security controls and visibility.

Manageability

One of the most important aspects of a database is the ability to manage it and its workloads. As you can see in Figure 1-9, manageability entails a long list of things, and perhaps can only be partially implemented.

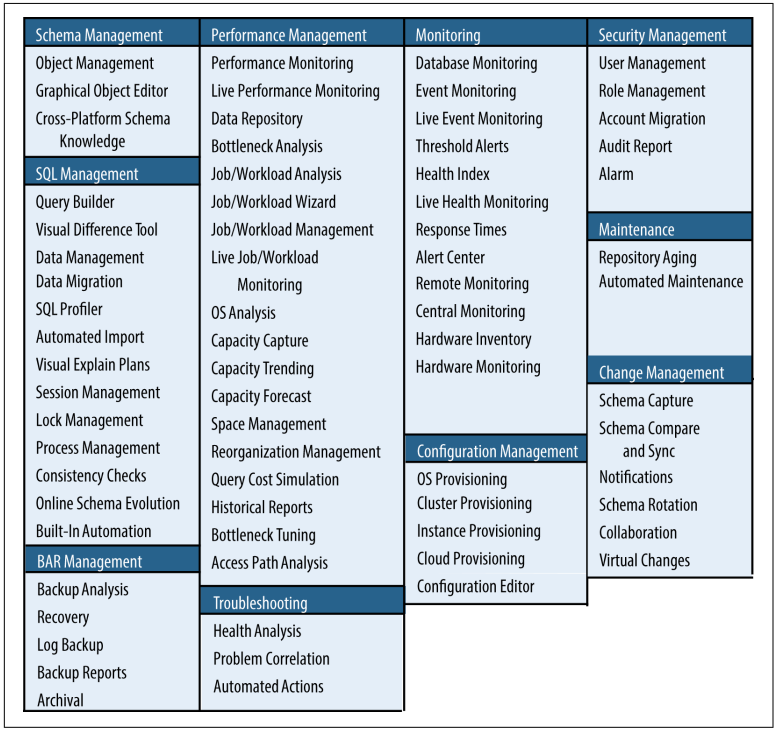


Figure 1-9. Database management tasks

Given the mixed nature of HTAP workloads, some management tasks become increasingly challenging, especially workload management. Accounting for an OLTP or operational workload is generally

done toward a "transaction" or interaction. The idea is to assess service level objectives in terms of transactions per second. Because the latency of such transactions can be so small, tracking performance at a per-statement, or even transaction level, would be very expensive. Thus, you need to accumulate the statistics at some interval and average it out. On the BI and analytics side, you generally do this assessment at a report or query level. These queries are generally longer, and capturing metrics on them and monitoring and managing based on query-level statistics could work just fine.

Managing mixed workloads to SLAs can be very challenging, based on priority and/or resource allocation. The ad hoc nature of analytics does not blend well with operational workloads that require consistent subsecond response times, even at peak loads.

If the query engine is integrating with different storage engines, even getting all the metrics for a query can be challenging. Although the query engine can track some metrics, such as time taken by the storage engine to service a request, it might not have visibility to what resources are being consumed by the underlying storage engine. How can you collect the vital CPU, memory, and I/O metrics, as well as numerous other metrics like queue length, memory swaps, and so on when that information is split between the query engine and the storage engine and there is nothing to tie that information together? For example, if you want to get the breakdown of the query resource usage by operation (for every step in the query plan—scan, join, aggregation, etc.) and by table accessed, especially when these operations are executed in parallel and the tables are partitioned, this becomes a very difficult job, unless the implementation has put in instrumentation and hooks to gather that end-to-end information. How do you find out why you have a skew or bottleneck, for example?

This does not purport to be an exhaustive list of challenges related to enterprise-level operational capabilities that are needed if these varied type of workloads are going to run on a single platform. There is the integration with YARN or Mesos, for example, which would take a lot more deliberation about how to manage resources for different application workloads across a cluster. Hopefully, though, it gives you a good sense for the challenges at hand.

Assessing HTAP Options

Although this report covered the details of the challenges for a query engine to support workloads that span the spectrum from OLTP, to operational, to BI, to analytics, you also can use it as a guide to assess a database engine, or combination of query and storage engines, geared toward meeting your workload requirements, whether they are transactional, analytical, or a mix. The considerations for such an assessment would mirror the four areas covered as challenges:

- What are the capabilities of the query engine that would meet your workload needs?
- What are the capabilities of the storage engines that would meet your workload needs? How well does the query engine integrate with those storage engines?
- What data models are important for your applications? Which storage engines support those models? Does a single query engine support those storage engines?
- What are the enterprise caliber capabilities that are important to you? How do the query and storage engines meet those requirements?

Capabilities of the Query Engine

The considerations to assess the capabilities of a query engine of course depends on what kinds of workloads you want to run. But because this report is about supporting a mixed HTAP workload, here are some of the considerations that are relevant:

Data structure—key support, clustering, partitioning

- How does the query engine utilize keyed access provided by the storage engine?
- Does the query engine support multicolumn keys even if the storage engine supports just a single key value?
- Does the query engine support access to a set of data where predicates on leading key columns are provided, as long as the storage engine supports clustering of data by key and supports such partial key access?

- How does the query engine handle predicates that are not on the leading columns of the key but on other columns of the key?

Statistics

- Does the query engine maintain statistics for the data?
- Can the query engine gather cardinality for multiple key or join columns, besides that for each column?
- Do these statistics provide the query engine information about data skew?
- How long does it take to update statistics for a very large table?
- Can the query engine incrementally update these statistics when new data is added or old data is aged?

Predicates on nonleading or nonkey columns

- Does the query engine have a way of efficiently accessing pertinent rows from a table, even if there are no predicates on the leading column(s) of a key or index, or does this always result in a full table scan?
- How does the query engine determine that it is efficient to use skip scan or MDAM instead of a full table scan?
- How does the query engine use statistics on key columns, multikey or join columns, and nonkey columns to come up with an efficient plan with the right data access, join, aggregate, and degree of parallelism strategy?
- Does the query engine support a columnar storage engine?
- Does the query engine access columns in sequence of their predicate cardinalities so as to gain maximum reduction in qualifying rows up front, when accessing a columnar storage engine?

Indexes and materialized views

- What kinds of indexes are supported by the engine and how can they be utilized?
- Can the indexes be unique?

- Are the indexes always consistent with the base table?
- Are index-only scans supported?
- What impact do the indexes have on updates, especially as you add more indexes?
- How are the indexes kept updated through bulk loads?
- Are materialized views supported?
- Can materialized views be synchronously and asynchronously maintained?
- What is the overhead of maintaining materialized views?
- Does the query engine automatically rewrite queries to use materialized views when it can?
- Are user-defined materialized views supported for query rewrite?

Degree of parallelism

- How does the query engine access data that is partitioned across nodes and disks on nodes?
- Does the query engine rely on the storage engine for that, or does it provide a parallel infrastructure to access these partitions in parallel?
- If the query engine considers serial and parallel plans, how does it determine the degree of parallelism needed?
- Does the query engine use only the number of nodes needed for a query based on that degree of parallelism?

Reducing the search space

- What optimizer technology does the query engine use?
- Can it generate good plans for large complex BI queries as well as fast compiles for short operational queries?
- What query plan caching techniques are used for operational queries?
- How is the query plan cache managed?

- How can the optimizer evolve with exposure to varied workloads?
- Can the optimizer detect query patterns?

Join type

- What are the types of joins supported?
- How are joins used for different workloads?
- What is the impact of using the wrong join type and how is that impact avoided?

Data flow and access

- How does the query engine handle large parallel data flows for complex analytical queries and at the same time provide quick direct access to data for operational workloads?
- What other efficiencies, such as prefetching data, are implemented for analytical workloads, and for operational workloads?

Mixed workload

- Can you prioritize workloads for execution?
- What criteria can you use for such prioritization?
- Can these workloads at different service levels be allocated different percentages of resources?
- Does the priority of queries decrease as they use more resources?
- Are there antistarvation mechanisms or a way to switch to a higher priority query before resuming a lower priority one?

Streaming

- Can the query engine handle streaming data directly?
- What functionality is supported against this streaming data such as row- and/or time-based windowing capabilities?

- What syntax or API is used to process streaming data? Would this lock you in to this query engine?

Feature support

- What capabilities and features are provided by the database for operational, analytical, and all other workloads?

Integration Between the Query and Storage Engines

The considerations to assess the integration between the query and storage engines begins with understanding what capabilities you need a storage engine to provide. Then, you need to assess how well the query engine exploits and expands on those capabilities, and how well it integrates with those storage engines. Here are certain points to consider, which will help you determine not only if they are supported, but also at what level are they being supported: by the query engine or storage engine, or a combination of the two:

Statistics

- What statistics on the data does the storage engine maintain?
- Can the query engine use these statistics for faster histogram generation?
- Does the storage engine support sampling to avoid full-table scans to compute statistics?
- Does the storage engine provide a way to access data changes since the last collection of statistics, for incremental updates of statistics?
- Does the storage engine maintain update counters for the query engine to schedule a refresh of the statistics?

Key structure

- Does the storage engine support key access?
- If it is not a multicolumn key, does the query engine map it to a multicolumn key?

- Can it be used for range access on leading columns of the key?

Partitioning

- How does the storage engine partition data across disks and nodes? Does it support hash and/or range partitioning, or a combination of these?
- Does the query engine need to salt data so that the load is balanced across partitions to avoid bottlenecks?
- If it does, how can it add a salt key as the leftmost column of the table key and still avoid table scans?
- Does the storage engine handle repartitioning of partitions as the cluster is expanded or contracted, or does the query engine do that?
- Is there full read/write access to the data as it is rebalanced?
- How does the query engine localize data access and avoid shuffling data between nodes?

Data type support

- What data types do the query and storage engines support and how do they map?
- Can value constraints be enforced on those types?
- Which engine enforces referential constraints?
- What character sets are supported?
- Are collations supported?
- What kinds of compression are provided?
- Is encryption supported?

Projection and selection

- Is projection done by the storage or query engine? What predicates are evaluated by the query and storage engines?
- Where are multicolumn predicates, IN lists, and multiple predicates with ORs and ANDs, evaluated?
- How long can IN lists be?

- Does the storage engine evaluate predicates in sequence of their filtering effectiveness?
- How about predicates comparing different columns of the same table?
- Where are complex expressions in predicates, potentially with functions, evaluated?
- How does the storage engine handle default or missing values?
- Are techniques like vectorization, CPU L1, L2, L3 cache, reduced serialization overhead, used for high performance?

Extensibility

- Does the storage engine support server side pushdown of operations, such as coprocessors in HBase, or before and after triggers in Cassandra?
- How does the query engine use these?

Security enforcement

- What are the security frameworks for the query and storage engines and how do they map relative to ANSI SQL security enforcement?
- Does the query engine integrate with the underlying Hadoop Kerberos security model?
- Does the query engine integrate with security frameworks like Sentry or Ranger?
- How does the query engine integrate with security logging, and SIEM capabilities of the underlying storage engine and platform security?

Transaction management

- Are replication for high availability, backup and restore, and multi-data center support provided completely by the storage engine, or is the query engine involved with ensuring consistency and integrity across all operations?

- What level of ACID or BASE transactional support has been implemented?
- How is transactional support integrated between the query and storage engines, such as write-ahead logs, and use of coprocessors? How well does it scale—is the transactional workload completely distributed across multiple transaction managers?
- Is multi–datacenter support provided?
- Is this active-active single or multiple master replication?
- What is the overhead of transactions on throughput and system resources?
- Is online backup and point-in-time recovery provided?

Metadata support

- How does the storage engine metadata (e.g., table names, location, partitioning, columns, data types) get mapped to the query engine metadata?
- How are storage engine specific options (e.g., compression, encryption, column families) managed by the query engine?
- Does the query engine provide transactional support, secondary indexes, views, constraints, materialized views, and so on for an external table?
- If changes to external tables can be made outside of the query engine, how does the query engine deal with those changes and the discrepancies that could result from them?

Performance, scale, and concurrency considerations

- If bulk load is available for the storage engine how does the query engine guarantee transactional consistency across loads?
- Does the storage engine accommodate rowset inserts and selects to process large number of rows at a time?
- What types of fast-scanning options are provided by the storage engine—snapshot scans, prefetching, and so on?
- Does the storage engine provide an easy way for the query engine to integrate for parallel operations?

- What level of concurrency and mixed workload capability can the storage engine support?

Error handling

- How are storage and query engine errors logged?
- How does the query engine map errors from the storage engine to meaningful error messages and resolution options?

Other operational aspects

- How are storage engine–specific operational aspects such as compaction or splitting handled by the query engine to minimize operational and performance impact?

Data Model Support

Here are the considerations to assess the data model support:

Operational versus analytical data models

- How well is the normalized data model supported for operational workloads?
- How well are the star and snowflake data models supported for analytical workloads?

NoSQL data models

- What storage engine data models are supported by the query engine—key-value, ordered key-value, Bigtable, document, full-text search, graph, and relational?
- How well are the storage engine APIs covered by the query engine API?
- How well does the query engine map and/or extend its API to support the storage engine API?

Enterprise-Caliber Capabilities

Security was covered earlier, but here are the other considerations to assess enterprise-caliber capabilities:

High availability

- What percentage of uptime is provided (99.99%–99.999%)?
- Can you upgrade the underlying OS online (with data available for reads and writes)?
- Can you upgrade the underlying file system online (e.g., Hadoop Distributed File System)?
- Can you upgrade the underlying storage engine online?
- Can you upgrade the query engine online?
- Can you redistribute data to accommodate node and/or disk expansions and contractions online?
- Can the table definition be changed online; for example, all column data type changes, and adding, dropping, renaming columns?
- Can secondary indexes be created and dropped online?
- Are online backups supported—both full and incremental?

Manageability

- What required management capabilities are supported (see Figure 1-9 for a list)?
- Is operational performance reported in transactions per second and analytical performance by query?
- What is the overhead of gathering metrics on operational workloads as opposed to analytical workloads?
- Is the interval of statistics collection configurable to reduce this overhead?
- Can workloads be managed to Service Level Objectives, based on priority and/or resource allocation, especially high priority operational workloads against lower priority analytical workloads?

- Is there end-to-end visibility of transaction and query metrics from the application, to the query engine, to the storage engine?
- Does it provide metric breakdown down to the operation (for every step of the query plan) level for a query?
- Does it provide metrics for table access across all workloads down to the partition level?
- Does it provide enough information to find out where the skew or bottlenecks are?
- How is it integrated with YARN or Mesos?

Conclusion

This report has attempted to do a modest job of highlighting at least some of the challenges of having a single query engine service both operational and analytical needs. That said, no query engine necessarily has to deliver on all the requirements of HTAP, and one certainly could meet the mixed workload requirements of many customers without doing so. The report also attempted to explain what you should look for and where you might need to compromise as you try and achieve the "nirvana" of a single database to handle all of your workloads, from operational to analytical.

About the Author

Rohit Jain is cofounder and CTO at Esgyn, an open source database company driving the vision of a Converged Big Data Platform. Rohit provided the vision behind Apache Trafodion, an enterprise-class MPP SQL Database for Big Data, donated to the Apache Software Foundation by HP in 2015. EsgynDB, Powered by Apache Trafodion, is delivering the promise of a Converged Big Data Platform with a vision of any data, any size, and any workload. A veteran database technologist over the past 28 years, Rohit has worked for Tandem, Compaq, and Hewlett-Packard in application and database development. His experience spans online transaction processing, operational data stores, data marts, enterprise data warehouses, business intelligence, and advanced analytics on distributed massively parallel systems.