Apache Hive: From MapReduce to Enterprise-grade Big Data Warehousing

Jesús Camacho-Rodríguez, Ashutosh Chauhan, Alan Gates, Eugene Koifman, Owen O'Malley, Vineet Garg, Zoltan Haindrich, Sergey Shelukhin, Prasanth Jayachandran, Siddharth Seth, Deepak Jaiswal, Slim Bouguerra, Nishant Bangarwa, Sankar Hariappan, Anishek Agarwal, Jason Dere, Daniel Dai, Thejas Nair, Nita Dembla, Gopal Vijayaraghavan, Günther Hagleitner
Hortonworks Inc.

ABSTRACT

Apache Hive is an open-source relational database system for analytic big-data workloads. In this paper we describe the key innovations on the journey from batch tool to fully fledged enterprise data warehousing system. We present a hybrid architecture that combines traditional MPP techniques with more recent big data and cloud concepts to achieve the scale and performance required by today's analytic applications. We explore the system by detailing enhancements along four main axis: Transactions, optimizer, runtime, and federation. We then provide experimental results to demonstrate the performance of the system for typical workloads and conclude with a look at the community roadmap.

Keywords Databases; Data Warehouses; Hadoop; Hive

ACM Reference Format:
Jesús Camacho-Rodríguez, Ashutosh Chauhan, Alan Gates, Eugene Koifman, Owen O'Malley, Vineet Garg, Zoltan Haindrich, Sergey Shelukhin, Prasanth Jayachandran, Siddharth Seth, Deepak Jaiswal, Slim Bouguerra, Nishant Bangarwa, Sankar Hariappan, Anishek Agarwal, Jason Dere, Daniel Dai, Thejas Nair, Nita Dembla, Gopal Vijayaraghavan, Günther Hagleitner. 2019. Apache Hive: From MapReduce to Enterprise-grade Big Data Warehousing. In *2019 International Conference on Management of Data (SIGMOD '19), June 30-July 5, 2019, Amsterdam, Netherlands.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3299869.3314045

1 INTRODUCTION

When Hive was first introduced over 10 years ago [55], the motivation of the authors was to expose a SQL-like interface on top of Hadoop MapReduce to abstract the users from dealing with low level implementation details for their parallel batch processing jobs. Hive focused mainly on Extract-Transform-Load (ETL) or batch reporting workloads that consisted of (*i*) reading huge amounts of data, (*ii*) executing transformations over that data (e.g., data wrangling, consolidation, aggregation) and finally (*iii*) loading the output into other systems that were used for further analysis.

As Hadoop became a ubiquitous platform for inexpensive data storage with HDFS, developers focused on increasing the range of workloads that could be executed efficiently within the platform. YARN [56], a resource management framework for Hadoop, was introduced, and shortly afterwards, data processing engines (other than MapReduce) such as Spark [14, 59] or Flink [5, 26] were enabled to run on Hadoop directly by supporting YARN.

Users also increasingly focused on migrating their data warehousing workloads from other systems to Hadoop. These workloads included interactive and ad-hoc reporting, dashboarding and other business intelligence use cases. There was a common requirement that made these workloads a challenge on Hadoop: They required a *low-latency SQL engine*. Multiple efforts to achieve this were started in parallel and new SQL MPP systems compatible with YARN such as Impala [8, 42] and Presto [48] emerged.

Instead of implementing a new system, the Hive community concluded that the current implementation of the project provided a good foundation to support these workloads. Hive had been designed for large-scale reliable computation in Hadoop, and it already provided SQL compatibility (alas, limited) and connectivity to other data management systems. However, Hive needed to evolve and undergo major renovation to satisfy the requirements of these new use cases, adopting common data warehousing techniques that had been extensively studied over the years.

Previous works presented to the research community about Hive focused on (*i*) its initial architecture and implementation on top of HDFS and MapReduce [55], and (*ii*) improvements to address multiple performance shortcomings in the original system, including the introduction of an optimized columnar file format, physical optimizations to reduce the

SIGMOD '19, June 30-July 5, 2019, Amsterdam, Netherlands

ACM ISBN 978-1-4503-5643-5/19/06…$15.00
https://doi.org/10.1145/3299869.3314045

number of MapReduce phases in query plans, and a vectorized execution model to improve runtime efficiency [39]. Instead, this paper describes the significant novelties introduced in Hive after the last article was presented. In particular, it focuses on the renovation work done to improve the system along four different axes:

SQL and ACID support (Section 3). SQL compliance is a key requirement for data warehouses. Thus, SQL support in Hive has been extended, including correlated subqueries, integrity constraints, and extended OLAP operations, among others. In turn, warehouse users need support to *insert*, *update*, *delete*, and *merge* their individual records on demand. Hive provides ACID guarantees with Snapshot Isolation using a transaction manager built on top of the Hive Metastore.

Optimization techniques (Section 4). Query optimization is especially relevant for data management systems that use declarative querying languages such as SQL. Instead of implementing its own optimizer from scratch, Hive chooses to integrate with Calcite [3, 19] and bring its optimization capabilities to the system. In addition, Hive includes other optimization techniques commonly used in data warehousing environments such as query reoptimization, query result caching, and materialized view rewriting.

Runtime latency (Section 5). To cover a wider variety of use cases including interactive queries, it is critical to improve latency. Hive support for optimized columnar data storage and vectorization of operators were presented in a previous work [39]. In addition to those improvements, Hive has since moved from MapReduce to Tez [15, 50], a YARN compatible runtime that provides more flexibility to implement arbitrary data processing applications than MapReduce. Furthermore, Hive includes LLAP, an additional layer of persistent long-running executors that provides data caching, facilities runtime optimizations, and avoids YARN containers allocation overhead at start-up.

Federation capabilities (Section 6). One of the most important features of Hive is the capacity to provide a unified SQL layer on top of many specialized data management systems that have emerged over the last few years. Thanks to its Calcite integration and storage handler improvements, Hive can seamlessly push computation and read data from these systems. In turn, the implementation is easily extensible to support other systems in the future.

The rest of the paper is organized as follows. Section 2 provides background on Hive's architecture and main components. Sections 3-6 describe our main contributions to improve the system along the axes discussed above. Section 7 presents an experimental evaluation of Hive. Section 8 discusses the impact of the new features, while Section 9 briefly describes the roadmap for the project. Finally, Section 10 summarizes our conclusions.

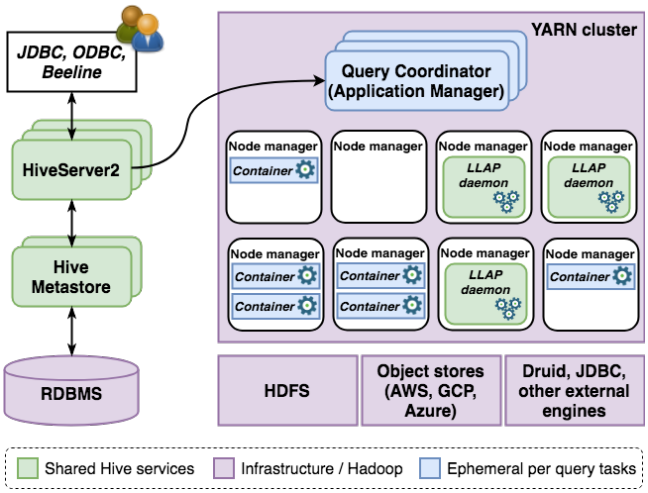


Figure 1: Apache Hive architecture.

2 SYSTEM ARCHITECTURE

In this section we briefly introduce Hive's architecture. Figure 1 depicts the main components in the system.

Data storage. Data in Hive can be stored using any of the supported file formats in any file system compatible with Hadoop. As of today, the most common file formats are ORC [10] and Parquet [11]. In turn, compatible file systems include HDFS which is the most commonly used distributed file system implementation, and all the major commercial cloud object stores such as AWS S3 and Azure Blob Storage. In addition, Hive can also read and write data to other standalone processing systems, such as Druid [4, 58] or HBase [6], which we discuss in more detail in Section 6.

Data catalog. Hive stores all information about its data sources using the Hive Metastore (or HMS, in short). In a nutshell, HMS is a catalog for all data queryable by Hive. It uses a RDBMS to persist the information, and it relies on DataNucleus [30], a Java object-relational mapping implementation, to simplify the support of multiple RDBMS at the backend. For calls that require low latency, HMS may bypass DataNucleus and query the RDMBS directly. The HMS API supports multiple programming languages and the service is implemented using Thrift [16], a software framework that provides an interface definition language, code generation engine, and binary communication protocol implementation.

Exchangeable data processing runtime. Hive has become one of the most popular SQL engines on top of Hadoop and it has gradually moved away from MapReduce to support more flexible processing runtimes compatible with YARN [50]. While MapReduce is still supported, currently the most popular runtime for Hive is Tez [15, 50]. Tez provides more flexibility than MapReduce by modeling data processing as DAGs with vertices representing application logic and edges

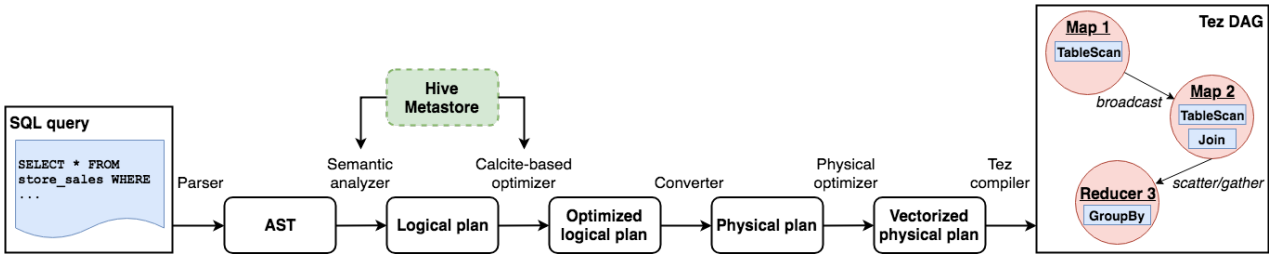


Figure 2: Query preparation stages in HiveServer2.

representing data transfer, similar to other systems such as Dryad [40] or Hyracks [22]. In addition, Tez is compatible with LLAP, the persistent execution and cache layer presented in Section 5.

Query server. HiveServer2 (or HS2, in short) allows users to execute SQL queries in Hive. HS2 supports local and remote JDBC and ODBC connections; Hive distribution includes a JDBC thin client called Beeline.

Figure 2 depicts the stages that a SQL query goes through in HS2 to become an executable plan. Once a user submits a query to HS2, the query is handled by the driver, which parses the statement and generates a Calcite [3, 19] logical plan from its AST. The Calcite plan is then optimized. Note that HS2 accesses information about the data sources in HMS for validation and optimization purposes. Subsequently, the plan is converted into a physical plan, potentially introducing additional operators for data partitioning, sorting, etc. HS2 executes additional optimizations on the physical plan DAG, and if all operators and expressions in the plan are supported, a vectorized plan [39] may be generated from it. The physical plan is passed to the task compiler, which breaks the operator tree into a DAG of executable tasks. Hive implements an individual task compiler for each supported processing runtime, i.e., Tez, Spark, and MapReduce. After the tasks are generated, the driver submits them to the runtime application manager in YARN, which handles the execution. For each task, the physical operators within that task are first initialized and then they process the input data in a pipelined fashion. After execution finishes, the driver fetches the results for the query and returns them to the user.

3 SQL AND ACID SUPPORT

Standard SQL and ACID transactions are critical requirements in enterprise data warehouses. In this section, we present Hive's broaden SQL support. Additionally, we describe the improvements made to Hive in order to provide ACID guarantees on top of Hadoop.

3.1 SQL support

To provide a replacement for traditional data warehouses, Hive needed to be extended to support more features from standard SQL. Hive uses a nested data model supporting all major atomic SQL data types as well as non-atomic types such as `STRUCT`, `ARRAY` and `MAP`. Besides, each new Hive release has increased its support for important constructs that are part of the SQL specification. For instance, there is extended support for correlated subqueries, i.e., subqueries that reference columns from the outer query, advanced OLAP operations such as grouping sets or window functions, set operations, and integrity constraints, among others. On the other hand, Hive has preserved multiple features of its original query language that were valuable for its user base. One of the most popular features is being able to specify the physical storage layout at table creation time using a `PARTITIONED BY` columns clause. In a nutshell, the clause lets a user partition a table horizontally. Then Hive stores the data for each set of partition values in a different directory in the file system. To illustrate the idea, consider the following table definition and the corresponding physical layout depicted in Figure 3:

```
1 CREATE TABLE store_sales (
2   sold_date_sk INT, item_sk INT, customer_sk INT, store_sk INT,
3   quantity INT, list_price DECIMAL(7,2), sales_price DECIMAL(7,2)
4 ) PARTITIONED BY (sold_date_sk INT);
```

The advantage of using the `PARTITIONED BY` clause is that Hive will be able to skip scanning full partitions easily for queries that filter on those values.

3.2 ACID implementation

Initially, Hive only had support to insert and drop full partitions from a table [55]. Although the lack of row level operations was acceptable for ETL workloads, as Hive evolved to support many traditional data warehousing workloads, there was an increasing requirement for full DML support and ACID transactions. Hence, Hive now includes support to execute `INSERT`, `UPDATE`, `DELETE`, and `MERGE` statements. It provides ACID guarantees via Snapshot Isolation [24] on read and well defined semantics in case of failure using a transaction manager built on top of the HMS. Currently transactions can only span a single statement; we plan to support multi-statement transactions in the near future. However, it

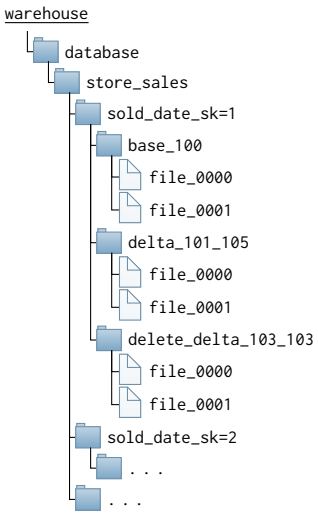


Figure 3: Physical layout for partitioned table.

is possible to write to multiple tables within a single transaction using Hive multi-insert statements [55].

The main challenges to overcome in supporting row level operations in Hive were (*i*) the lack of a transaction manager in the system, and (*ii*) the lack of file updates support in the underlying file system. In the following, we provide more details about the implementation of ACID in Hive and how these issues were addressed.

Transaction and lock management. Hive stores transaction and locking information state in HMS. It uses a global transaction identifier or *TxnId*, i.e., a monotonically increasing value generated by the Metastore, for each transaction run in the system. In turn, each *TxnId* maps into one or multiple write identifiers or *WriteIds*. A *WriteId* is a monotonically increasing value generated by the Metastore as well, but within a table scope. The *WriteId* is stored with each record that is written by a transaction; all records written by the same transaction to the same table share the same *WriteId*. In turn, files sharing the same *WriteId* are identified uniquely using a *FileId*, while each record within a file is identified uniquely by a *RowId* field. Note that the combination of *WriteId*, *FileId* and *RowId* identifies uniquely each record in a table. A delete operation in Hive is modeled as an insert of a labeled record that points to the unique identifier of the record being deleted.

To achieve Snapshot Isolation, HS2 obtains a *logical snapshot* of the data that needs to read when a query is executed. The snapshot is represented by a transaction list comprising the highest allocated *TxnId* at that moment, i.e., the high watermark, and the set of open and aborted transactions below it. For each table that the query needs to read, HS2 first generates the *WriteId* list from the transaction list by contacting HMS; the *WriteId* list is similar to the transaction list but within the scope of a single table. Each scan operation in the plan is bound to a *WriteId* list during compilation. The readers in that scan will skip rows whose *WriteId* (*i*) is higher than the high watermark, or (*ii*) is part of the set of open and aborted transactions. The reason to keep both global and per-table identifiers is that the readers for each table keep a smaller state, which becomes critical for performance when there are a large number of open transactions in the system.

For partitioned tables the lock granularity is a partition, while the full table needs to locked for unpartitioned tables. HS2 only needs to obtain exclusive locks for operations that disrupt readers and writers, such as a `DROP PARTITION` or `DROP TABLE` statements. All other common operations just acquire shared locks. Updates and deletes use optimistic conflict resolution by tracking their write sets and resolving the conflict at commit time, letting the first commit win.

Data and file layout. Hive stores data for each table and partition in a different directory (recall Figure 3). Similar to [45], we use different stores or directories within each table or partition to support concurrent read and write operations: *base* and *delta*, which in turn may contain one or multiple files. The files in the base store contain all valid records up to a certain *WriteId*. For instance, folder `base_100` contains all records up to *WriteId* 100. On the other hand, a delta directory contains files with records within a *WriteId* range. Hive keeps separate delta directories for inserted and deleted records; update operations are split into delete and insert operations. An insert or delete transaction creates a delta directory with records bound to a single *WriteId*, e.g., `delta_101_101` or `delete_delta_102_102`. Delta directories containing more than one *WriteId* are created as part of the compaction process (explained below).

As mentioned previously, a table scan in a query has a *WriteId* list associated to it. The readers in the scan discard full directories as well as individual records that are not valid based on the current snapshot. When deletes are present in the delta files, records in the base and insert delta files need to be anti-joined with the delete deltas that apply to their *WriteId* range. Since delta files with deleted records are usually small, they can be kept in-memory most times, accelerating the merging phase.

Compaction. Compaction is the process in Hive that merges files in delta directories with other files in delta directories (referred to as *minor compaction*), or files in delta directories with files in base directories (referred to as *major compaction*). The crucial reasons to run compaction periodically are (*i*) decreasing the number of directories and files in tables, which could otherwise affect file system performance, (*ii*) reducing the readers effort to merge files at query execution time, and (*iii*) shorten the set of open and aborted TxnIds and *WriteIds* associated with each snapshot, i.e., major compaction deletes

history, increasing the *TxnId* under which all records in the tables are known to be valid.

Compaction is triggered automatically by HS2 when certain thresholds are surpassed, e.g., number of delta files in a table or ratio of records in delta files to base files. Finally, note that compaction does not need any locks over the table. In fact, the cleaning phase is separated for the merging phase so that any ongoing query can complete its execution before files are deleted from the system.

4 QUERY OPTIMIZATION

While the support for optimization in the initial versions of Hive was limited, it is evident that the development of its execution internals is not sufficient to guarantee efficient performance. Thus, currently the project includes many of the complex techniques typically used in relational database systems. This section describes the most important optimization features that help the system generate better plans and deliver improvements to query execution, including its fundamental integration with Apache Calcite.

4.1 Rule and cost-based optimizer

Initially, Apache Hive executed multiple rewritings to improve performance while parsing the input SQL statement. In addition, it contained a rule-based optimizer that applied simple transformations to the physical plan generated from the query. For instance, the goal of many of these optimizations was trying to minimize the cost of data shuffling, a critical operation in the MapReduce engine. There were other optimizations to push down filter predicates, project unused columns, and prune partitions. While this was effective for some queries, working with the physical plan representation made implementing complex rewritings such as join reordering, predicate simplification and propagation, or materialized view-based rewriting, overly complex.

For that reason, a new plan representation and optimizer powered by Apache Calcite [3, 19] were introduced. Calcite is a modular and extensible query optimizer with built-in elements that can be combined in different ways to build your own optimization logic. These include diverse rewriting rules, planners, and cost models.

Calcite provides two different planner engines: (*i*) a *cost-based planner*, which triggers rewriting rules with the goal of reducing the overall expression cost, and (*ii*) an *exhaustive planner*, which triggers rules exhaustively until it generates an expression that is no longer modified by any rules. Transformation rules work indistinctly with both planners.

Hive implements *multi-stage optimization* similar to other query optimizers [52], where each optimization stage uses a planner and a set of rewriting rules. This allows Hive to reduce the overall optimization time by guiding the search for different query plans. Some of the Calcite rules enabled in Apache Hive are join reordering, multiple operators reordering and elimination, constant folding and propagation, and constraint-based transformations.

Statistics. Table statistics are stored in the HMS and provided to Calcite at planning time. These include the table cardinality, and number of distinct values, minimum and maximum value for each column. The statistics are stored such that they can be combined in an additive fashion, i.e., future inserts as well as data across multiple partitions can add onto existing statistics. The range and cardinality are trivially mergeable. For the number of distinct values, HMS uses a bit array representation based on HyperLogLog++ [38] which can be combined without loss of approximation accuracy.

4.2 Query reoptimization

Hive supports query reoptimization when certain errors are thrown during execution. In particular, it implements two independent reoptimization strategies.

The first strategy, *overlay*, changes certain configuration parameters for all query reexecutions. For instance, a user may choose to force all joins in query reexecutions to use a certain algorithm, e.g., *hash partitioning with sort-merge*. This may be useful when certain configuration values are known to make query execution more robust.

The second strategy, *reoptimize*, relies on statistics captured at runtime. While a query is being planned, the optimizer estimates the size of the intermediate results in the plan based on statistics that are retrieved from HMS. If those estimates are not accurate, the optimizer may make planning mistakes, e.g., wrong join algorithm selection or memory allocation. This in turn may lead to poor performance and execution errors. Hive captures runtime statistics for each operator in the plan. If any of the aforementioned problems is detected during query execution, the query is reoptimized using the runtime statistics and executed again.

4.3 Query results cache

Transactional consistency of the warehouse allows Hive to reuse the results of a previously executed query by using the internal transactional state of the participating tables. The query cache provides scalability advantages when dealing with BI tools which generate repetitive identical queries.

Each HS2 instance keeps its own *query cache* component, which in turn keeps a map from the query AST representation to an entry containing the results location and information of the snapshot of the data over which the query was answered. This component is also responsible for expunging stale entries and cleaning up resources used by those entries.

During query compilation, HS2 checks its cache using the input query AST in a preliminary step. The unqualified table references in the query are resolved before the AST is used to prove the cache, since depending on the current

database at query execution time, two queries with the same text may access tables from different databases. If there is a cache hit and the tables used by the query do not contain new or modified data, then the query plan will consist on a single task that will fetch the results from the cached location. If the entry does not exist, then the query is run as usual and results generated for the query are saved to the cache if the query meets some conditions; for instance, the query cannot contain non-deterministic functions (*rand*), runtime constant functions (*current_date*, *current_timestamp*), etc.

The query cache has a pending entry mode, which protects against a thundering herd of identical queries when data is updated and a cache miss is observed by several of them at the same time. The cache will be refilled by the first query that comes in. Besides, that query may potentially receive more cluster capacity since it will serve results to every other concurrent identical query suffering a cache miss.

4.4 Materialized views and rewriting

Traditionally, one of the most powerful techniques used to accelerate query processing in data warehouses is the precomputation of relevant materialized views [28, 31, 35–37].

Apache Hive supports materialized views and automatic query rewriting based on those materializations. In particular, materialized views are just *semantically enriched* tables. Therefore, they can be stored natively by Hive or in other supported systems (see Section 6), and they can seamlessly exploit features such as LLAP acceleration (described in Section 5.1). The optimizer relies in Calcite to automatically produce full and partially contained rewritings on Select-Project-Join-Aggregate (SPJA) query expressions (see Figure 4). The rewriting algorithm exploits integrity constraints information declared in Hive, e.g., *primary key*, *foreign key*, *unique key*, and *not null*, to produce additional valid transformations. The algorithm is encapsulated within a rule and it is triggered by the cost-based optimizer, which is responsible to decide whether a rewriting should be used to answer a query or not. Note that if multiple rewritings are applicable to different parts of a query, the optimizer may end up selecting more than one view substitutions.

Materialized view maintenance. When data in the sources tables used by a materialized view changes, e.g., new data is inserted or existing data is modified, we will need to refresh the contents of the materialized view to keep it up-to-date with those changes. Currently, the rebuild operation for a materialized view needs to be triggered manually by the user using a `REBUILD` statement.

By default, Hive attempts to rebuild a materialized view incrementally [32, 34], falling back to full rebuild if it is not possible. Current implementation only supports incremental rebuild when there were `INSERT` operations over the source tables, while `UPDATE` and `DELETE` operations will force a full rebuild of the materialized view.

(a)

```
CREATE MATERIALIZED VIEW mat_view AS
SELECT d_year, d_moy, d_dom, SUM(ss_sales_price) AS sum_sales
FROM store_sales, date_dim
WHERE ss_sold_date_sk = d_date_sk AND d_year > 2017
GROUP BY d_year, d_moy, d_dom;
```

(b)

q_1:

```
SELECT SUM(ss_sales_price) AS sum_sales
FROM store_sales, date_dim
WHERE ss_sold_date_sk = d_date_sk AND
  d_year = 2018 AND d_moy IN (1,2,3);
```

q_1':

```
SELECT SUM(sum_sales)
FROM mat_view
WHERE d_year = 2018 AND d_moy IN (1,2,3);
```

(c)

q_2:

```
SELECT d_year, d_moy, SUM(ss_sales_price) AS sum_sales
FROM store_sales, date_dim
WHERE ss_sold_date_sk = d_date_sk AND d_year > 2016
GROUP BY d_year, d_moy;
```

q_2':

```
SELECT d_year, d_moy, SUM(sum_sales)
FROM (
  SELECT d_year, d_moy, SUM(sum_sales) AS sum_sales
  FROM mat_view
  GROUP BY d_year, d_moy
  UNION ALL
  SELECT d_year, d_moy, SUM(ss_sales_price) AS sum_sales
  FROM store_sales, date_dim
  WHERE ss_sold_date_sk = d_date_sk AND
    d_year > 2016 AND d_year <= 2017
  GROUP BY d_year, d_moy
) subq
GROUP BY d_year, d_moy;
```

Figure 4: Materialized view definition (a) and sample fully (b) and partially (c) contained rewritings.

One interesting aspect is that the incremental maintenance relies on the rewriting algorithm itself. Since a query is associated with a snapshot of the data, the materialized view definition is enriched with filter conditions on the *WriteId* column value of each table scanned (recall Section 3.2). Those filters conditions reflect the snapshot of the data when the materialized view was created or lastly refreshed. When the maintenance operation is triggered, the rewriting algorithm may produce a partially contained rewriting that reads the materialized view and the new data from the source tables. This rewritten plan is in turn transformed into (*i*) an `INSERT` operation if it is a SPJ materialized view, or (*ii*) a `MERGE` operation if it is a SPJA materialized view.

Materialized view lifecycle. By default, once the materialized view contents are stale, the materialized view will not be used for query rewriting.

However, in some occasions it may be fine to accept rewriting on stale data while updating the materialized views in micro batches. For those cases, Hive lets users combine a

rebuild operation run periodically, e.g., every 5 minutes, and define a window for data staleness allowed in the materialized view definition using a table property [1], e.g., 10 minutes.

4.5 Shared work optimization

Hive is capable of identifying overlapping subexpressions within the execution plan of a given query, computing them only once and reusing their results. Instead of triggering transformations to find equivalent subexpressions within the plan, the *shared work optimizer* only merges *equal* parts of a plan, similar to other reuse-based approaches presented in prior works [25, 41]. It applies the reutilization algorithm just before execution: it starts merging scan operations over the same tables, then it continues merging plan operators until a difference is found. The decision on the data transfer strategy for the new shared edge coming out of the merged expression is left to the underlying engine, i.e., Apache Tez. The advantage of a reuse-based approach is that it can accelerate execution for queries computing the same subexpression more than once, without introducing much optimization overhead. Nonetheless, since the shared work optimizer does not explore the complete search space of equivalent plans, Hive may fail to detect existing reutilization opportunities.

4.6 Dynamic semijoin reduction

Semijoin reduction is a traditional technique used to reduce the size of intermediate results during query execution [17, 20]. The optimization is specially useful for star schema databases with one or many dimension tables. Queries on those databases usually join the fact table and dimension tables, which are filtered with predicates on one or multiple columns. However, those columns are not used in the join condition, and thus, a filter over the fact table cannot be created statically. The following SQL query shows an example of such a join between the *store_sales* and *store_returns* fact tables and the *item* dimension table:

```
1 SELECT ss_customer_sk, SUM(ss_sales_price) AS sum_sales
2 FROM store_sales, store_returns, item
3 WHERE ss_item_sk = sr_item_sk AND
4   ss_ticket_number = sr_ticket_number AND
5   ss_item_sk = i_item_sk AND
6   i_category = 'Sports'
7 GROUP BY ss_customer_sk
8 ORDER BY sum_sales DESC;
```

By applying semijoin reduction, Hive evaluates the subexpression that is filtered (in the example above, the filter on the *item* table), and subsequently, the values produced by the expression are used to skip reading records from the rest of tables.

The semijoin reducers are introduced by the optimizer and they are pushed into the scan operators in the plan. Depending on the data layout, Hive implements two variants of the optimization.

Dynamic partition pruning. It is applied if the table reduced by the semijoin is partitioned by the join column. Once the filtering subexpression is evaluated, Hive uses the values produced to skip reading unneeded partitions dynamically, while the query is running. Recall that each table partition value is stored in a different folder in the file system, hence skipping them is straightforward.

Index semijoin. If the table reduced by the semijoin is not partitioned by the join column, Hive may use the values generated by the filtering subexpression to (*i*) create a range filter condition with the minimum and maximum values, and (*ii*) create a Bloom filter with the values produced by the subexpression. Hive populates the semijoin reducer with these two filters, which may be used to avoid scanning entire row groups at runtime, e.g., if data is stored in ORC files [39].

5 QUERY EXECUTION

Improvements in the query execution internals such as the transition from MapReduce to Apache Tez [50] and the implementation of columnar-based storage formats and vectorized operators [39] reduced query latency in Hive by orders of magnitude. However, to improve execution runtime further, Hive needed additional enhancements to overcome limitations inherent to its initial architecture that was tailored towards long running queries. Among others, (*i*) execution required YARN containers allocation at start-up, which quickly became a critical bottleneck for low latency queries, (*ii*) Just-In-Time (JIT) compiler optimizations were not as effective as possible because containers were simply killed after query execution, and (*iii*) Hive could not exploit data sharing and caching possibilities within and among queries, leading to unnecessary IO overhead.

5.1 LLAP: Live Long and Process

Live Long and Process, also known as LLAP, is an optional layer that provides persistent multi-threaded query executors and multi-tenant in-memory cache to deliver faster SQL processing at large scale in Hive. LLAP does not replace the existing execution runtime used by Hive, such as Tez, but rather enhances it. In particular, execution is scheduled and monitored by Hive query coordinators transparently over both LLAP nodes as well as regular containers.

The data I/O, caching, and query fragment execution capabilities of LLAP are encapsulated within *daemons*. Daemons are setup to run continuously in the worker nodes in the cluster, facilitating JIT optimization, while avoiding any start-up overhead. YARN is used for coarse-grained resource allocation and scheduling. It reserves memory and CPU for the daemons and handles restarts and relocation. The daemons are stateless: each contains a number of executors to run

[1] Table properties allow users in Hive to tag table and materialized view definitions with metadata key-value pairs.

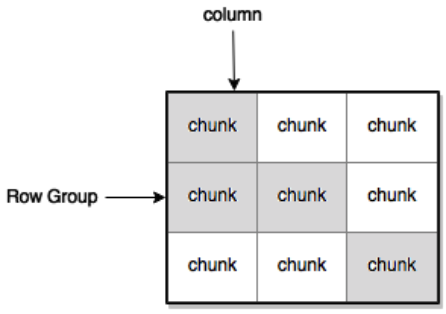


Figure 5: Cache addressing in LLAP.

several query fragments in parallel and a local work queue. Failure and recovery is simplified because any node can still be used to process any fragment of the input data if a LLAP daemon fails.

I/O elevator. The daemon uses separate threads to off-load data I/O and decompression, which we refer to as *I/O elevator*. Data is read in batches and transformed into an internal run-length encoded (RLE) columnar format ready for vectorized processing. A column batch is moved into execution phase as soon as it is read, which allows previous batches to be processed while the following batches are being prepared.

Transformation from underlying file format into LLAP internal data format is accomplished using plugins that are specific to each format. Currently LLAP supports translation from ORC [10], Parquet [11], and text file formats.

The I/O elevator can push down projections, sargable predicates [2] and Bloom filters to the file reader, if they are provided. For instance, ORC files can take advantage of these structures to skip reading entire column and row groups.

Data caching. LLAP features an off-heap cache as its primary buffer pool for holding data coming into the I/O elevator. For each file, the cache is addressed by the I/O elevator along two dimensions: row and column groups. A set of rows and columns form a row-column *chunk* (refer to Figure 5). The I/O elevator reassembles the selected projection and evaluates the predicates to extract chunks which can be reconstituted into vector batches for the operator pipeline. In case of cache misses, the cache is repopulated with the missing chunks before the reconstitution is performed. The result is the incremental filling of the cache as a user navigates the dataset along denormalized dimensions, which is a common pattern that the cache optimizes for.

LLAP caches metadata and data from the input files. To maintain the validity of the cache in the presence of file updates, LLAP uses a unique identifier assigned to every file stored in HDFS together with information about the file length. This is similar to the ETag fields present in blob stores such as AWS S3 or Azure Blob Storage.

[2]Predicates that can be evaluated using an index seek.

The metadata, including index information, is cached even for data that was never in the cache. In particular, the first scan populates the metadata in bulk, which is used to determine the row groups that have to be loaded and to evaluate predicates before determining cache misses. The advantage of this approach is that LLAP will not load chunks that are effectively unnecessary for a given query, and hence will avoid trashing the cache.

The cache is incrementally mutable, i.e., the addition of new data to the table does not result in a complete cache invalidation. Specifically, the cache acquires a file only when a query addresses that file, which transfers the control of the visibility back to the query transactional state. Since ACID implementation handles transactions by adjusting visibility at the file level (recall Section 3.2), the cache turns into an MVCC view of the data servicing multiple concurrent queries possibly in different transactional states.

The eviction policy for the data in the cache is exchangeable. Currently a simple LRFU (Least Recently/Frequently Used) replacement policy that is tuned for analytic workloads with frequent full and partial scan operations is used by default. The unit of data for eviction is the chunk. This choice represents a compromise between low-overhead processing and storage efficiency.

Query fragment execution. LLAP daemons execute arbitrary query plan *fragments* containing operations such as filters, projections, data transformations, joins, partial aggregates, and sorting. They allow parallel execution for multiple fragments from different queries and sessions. The operations in the fragments are vectorized and they run directly on the internal RLE format. Using the same format for I/O, cache, and execution minimizes the work needed for interaction among all of them.

As mentioned above, LLAP does not contain its own execution logic. Instead, its executors can essentially replicate YARN containers functionality. However, for stability and security reasons, only Hive code and statically specified user-defined functions (UDFs) are accepted in LLAP.

5.2 Workload management improvements

The workload manager controls the access to LLAP resources for each query executed by Hive. An administrator can create *resource plans*, i.e., self-contained resource-sharing configurations, to improve execution predictability and cluster sharing by concurrent queries running on LLAP. These factors are critical in multi-tenant environments. Though multiple resource plans can be defined in the system, at a given time only one of them can be active for a given deployment. Resource plans are persisted by Hive in HMS.

A resource plan consists of (*i*) one or more pool of resources, with a maximum amount of resources and number of concurrent queries per pool, (*ii*) mappings, which root

incoming queries to pools based on specific query properties, such as user, group, or application, and (*iii*) triggers which initiate an action, such as killing queries in a pool or moving queries from one pool to another, based on query metrics that are collected at runtime. Though queries get guaranteed fractions of the cluster resources as defined in the pools, the workload manager tries to prevent that the cluster is underutilized. In particular, a query may be assigned idle resources from a pool that it has not been assigned to, until a subsequent query that maps to that pool claims them.

To provide an example, consider the following resource plan definition for a production cluster:

```
1  CREATE RESOURCE PLAN daytime;
2  CREATE POOL daytime.bi
3    WITH alloc_fraction=0.8, query_parallelism=5;
4  CREATE POOL daytime.etl
5    WITH alloc_fraction=0.2, query_parallelism=20;
6  CREATE RULE downgrade IN daytime
7    WHEN total_runtime > 3000 THEN MOVE etl;
8  ADD RULE downgrade TO bi;
9  CREATE APPLICATION MAPPING visualization_app IN daytime TO bi;
10 ALTER PLAN daytime SET DEFAULT POOL = etl;
11 ALTER RESOURCE PLAN daytime ENABLE ACTIVATE;
```

Line 1 creates the resource plan *daytime*. Lines 2-3 create a pool *bi* with 80% of the LLAP resources in the cluster. Those resources may be used to execute up to 5 queries concurrently. Similarly, lines 4-5 create a pool *etl* with the rest of resources that may be used to execute up to 20 queries concurrently. Lines 6-8 create a rule that moves a query from the *bi* to the *etl* pool of resources when the query has run for more than 3 seconds. Note that the previous operation can be executed because query fragments are easier to preempt compared to containers. Line 9 creates a mapping for an application called *interactive_bi*, i.e., all queries fired by *interactive_bi* will initially take resources from the *bi* pool. In turn, line 10 sets the default pool to *etl* for the rest of queries in the system. Finally, line 11 enables and activates the resource plan in the cluster.

6 FEDERATED WAREHOUSE SYSTEM

Over the last decade, there has been a growing proliferation of specialized data management systems [54] which have become popular because they achieve better cost-effective performance for their specific use case than traditional RDBMSs.

In addition to its native processing capabilities, Hive can act as a *mediator* [23, 27, 57] as it is designed to support querying over multiple independent data management systems. The benefits of unifying access to these systems through Hive are multifold. Application developers can choose a blend of multiple systems to achieve the desired performance and functionality, yet they need to code only against a single interface. Thus, applications become independent of the underlying data systems, which allows for more flexibility in changing systems later. Hive itself can be used to implement data movement and transformations between different systems, alleviating the need for third-party tools. Besides, Hive as a mediator can globally enforce access control and capture audit trails via Ranger [12] or Sentry [13], and also help with compliance requirements via Atlas [2].

6.1 Storage handlers

To interact with other engines in a modular and extensible fashion, Hive includes a *storage handler* interface that needs to be implemented for each of them. A storage handler consists of: (*i*) an *input format*, which describes how to read data from the external engine, including how to split the work to increase parallelism, (*ii*) an *output format*, which describes how to write data to the external engine, (*iii*) a *SerDe* (serializer and deserializer) which describes how to transform data from Hive internal representation into the external engine representation and vice versa, and (*iv*) a *Metastore hook*, which defines notification methods invoked as part of the transactions against HMS, e.g., when a new table backed by the external system is created or when new rows are inserted into such table. The minimum implementation of a usable storage handler to read data from the external system contains at least an input format and deserializer.

Once the storage handler interface has been implemented, querying the external system from Hive is straightforward and all the complexity is hidden from user behind the storage handler implementation. For instance, Apache Druid [4, 58] is an open-source data store designed for business intelligence (OLAP) queries on event data that is widely used to power user-facing analytic applications; Hive provides a Druid storage handler so it can take advantage of its efficiency for the execution of interactive queries. To start querying Druid from Hive, the only action necessary is to register or create Druid data sources from Hive. First, if a data source already exists in Druid, we can map a Hive external table to it with a simple statement:

```
1 CREATE EXTERNAL TABLE druid_table_1
2 STORED BY 'org.apache.hadoop.hive.druid.DruidStorageHandler'
3 TBLPROPERTIES ('druid.datasource' = 'my_druid_source');
```

Observe that we do not need to specify column names or types for the data source, since they are automatically inferred from Druid metadata. In turn, we can create a data source in Druid from Hive with a simple statement as follows:

```
1 CREATE EXTERNAL TABLE druid_table_2 (
2   __time TIMESTAMP, dim1 VARCHAR(20), m1 FLOAT)
3 STORED BY 'org.apache.hadoop.hive.druid.DruidStorageHandler';
```

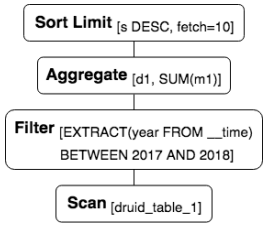
Once the Druid sources are available in Hive as external tables, we can execute any allowed table operations on them.

6.2 Pushing computation using Calcite

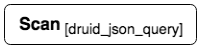
One of the most powerful features of Hive is the possibility to leverage Calcite adapters [19] to push complex computation

```
SELECT d1, SUM(m1) AS s
FROM druid_table_1
WHERE EXTRACT(year FROM __time)
  BETWEEN 2017 AND 2018
GROUP BY d1
ORDER BY s DESC
LIMIT 10;
```

(a) SQL query over Druid table



(b) Plan before optimization




```
{
  "queryType": "groupBy",
  "dataSource": "my_druid_source",
  "granularity": "all",
  "dimension": "d1",
  "aggregations":
    [ { "type": "floatSum", "name": "s", "fieldName": "m1" } ],
  "limitSpec": {
    "limit": 10,
    "columns":
      [ {"dimension": "s", "direction": "descending" } ]
  },
  "intervals":
    [ "2017-01-01T00:00:00.000/2019-01-01T00:00:00.000" ]
}
```

(c) Plan after optimization

Figure 6: Query federation example in Hive.

to supported systems and generate queries in the languages supported by those systems.

Continuing with the Druid example, the most common way of querying Druid is through a REST API over HTTP using queries expressed in JSON [3]. Once a user has declared a table that is stored in Druid, Hive can transparently generate Druid JSON queries from the input SQL queries. In particular, the optimizer applies rules that match a sequence of operators in the plan and generate a new equivalent sequence with more operations executed in Druid. Once we have completed the optimization phase, the subset of operators that needs to be executed by Druid is translated by Calcite into a valid JSON query that is attached to the scan operator that will read from Druid. Note that for a storage handler to support Calcite automatically generated queries, its input format needs to include logic to send the query to the external system (possibly splitting the query into multiple sub-queries that can be executed in parallel) and read back the query results.

As of today, Hive can push operations to Druid and multiple engines with JDBC support[4] using Calcite. Figure 6 depicts a query executed over a table stored in Druid, and the corresponding plan and JSON query generated by Calcite.

[3]http://druid.io/docs/latest/querying/querying.html

[4]Calcite can generate SQL queries from operator expressions using a large number of different dialects.

7 PERFORMANCE EVALUATION

To study the impact of our work, we evaluate Hive over time using different standard benchmarks. In this section, we present a summary of results that highlight the impact of the improvements presented throughout this paper.

Experimental setup. The experiments presented below were run on a cluster of 10 nodes connected by a 10 Gigabit Ethernet network. Each node has a 8-core 2.40GHz Intel Xeon CPU E5-2630 v3 and 256GB RAM, and has two 6TB disks for HDFS and YARN storage.

7.1 Comparison with previous versions

We have conducted experiments using TPC-DS queries on a 10TB scale data set. The data was stored in ACID tables in HDFS using the ORC file format, and the fact tables were partitioned by day. All the information needed for reproducibility of these results is publicly available [5]. We compared two different versions of Hive [7] with each other: (*i*) Hive v1.2, released in September 2015, running on top of Tez v0.5, and (*ii*) the latest Hive v3.1, released in November 2018, using Tez v0.9 with LLAP enabled. Figure 7 shows the response times (perceived by the user) for both Hive versions; note the logarithmic *y* axis. For each query, we report the average over three runs with warm cache. First, observe that only 50 queries could be executed in Hive v1.2; the response time values for the queries that could not be executed are omitted in the figure. The reason is that Hive v1.2 lacked support for set operations such as `EXCEPT` or `INTERSECT`, correlated scalar subqueries with non-equi join conditions, interval notation, and order by unselected columns, among other SQL features. For those 50 queries, Hive v3.1 significantly outperforms the preceding version. In particular, Hive v3.1 is faster by an average of 4.6x and by up to a maximum of 45.5x (refer to *q58*). Queries that improved more than 15x are emphasized in the figure. More importantly, Hive v3.1 can execute the full set of 99 TPC-DS queries thanks to its SQL support improvements. The performance difference between both versions is so significant that the aggregated response time for all queries executed by Hive v3.1 is still 15% lower than the time for 50 queries in Hive v1.2. New optimization features such as shared work optimizer make a big difference on their own; for example, *q88* is 2.7x faster when it is enabled.

7.2 LLAP acceleration

In order to illustrate the benefits of LLAP over query execution using Tez containers uniquely, we run all the 99 TPC-DS queries in Hive v3.1 using the same configuration but with LLAP enabled/disabled. Table 1 shows the aggregated time for all the queries in the experiment; we can observe that

[5]http://github.com/hortonworks/hive-testbench/tree/hdp3

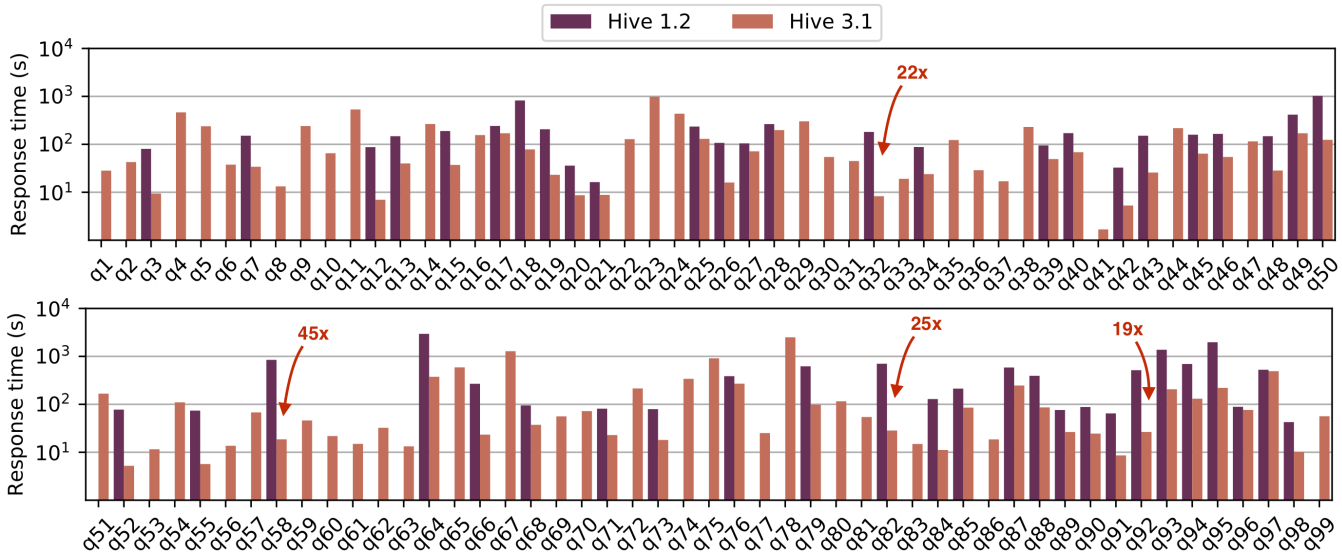


Figure 7: Comparison of query response times among different Hive versions.

Execution mode	Total response time (s)
Container (without LLAP)	41576
LLAP	15540

Table 1: Response time improvement using LLAP.

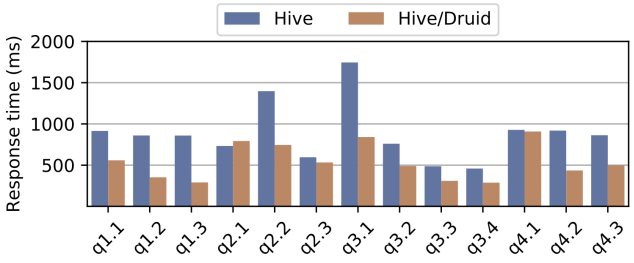


Figure 8: Comparison of query response times between native Hive and federation to Druid.

LLAP on its own reduces the workload response time dramatically by 2.7x.

7.3 Query federation to Druid

In the following, we illustrate the significant performance benefits that can be derived from the combination of using Hive's materialized views and federation capabilities. For this experiment, we use the Star-Schema Benchmark (SSB) [47] on a 1TB scale data set. The SSB benchmark is based on TPC-H and it is meant to simulate the process of iteratively and interactively querying a data warehouse to play what-if scenarios, drill down and better understand trends. It consists of a single fact table and 4 dimension tables, and the workload contains 13 queries that join, aggregate, and place fairly tight dimensional filters over different sets of tables.

For the experiment, we create a materialized view that denormalizes the database schema [6]. The materialization is stored in Hive. Then we run the queries in the benchmark which are automatically rewritten by the optimizer to be answered from the materialized view. Subsequently, we store the materialized view in Druid v0.12 and repeat the same steps. Figure 8 depicts response times for each variant. Observe that Hive/Druid is 1.6x faster than execution over the materialized view stored natively in Hive. The reason is that Hive pushes most of the query computation to Druid using Calcite, and thus it can benefit from Druid providing lower latency for those queries.

8 DISCUSSION

In this section, we discuss the impact of the improvements presented in this work, and multiple challenges faced and lessons learned while implementing these new features.

The decision on the features that should be added to the project over time is mainly influenced by shortcomings faced by users, either reported directly to the Hive community or through the companies commercializing the project.

For instance, enterprise users asked for the implementation of ACID guarantees in order to offload workloads from data warehouses introduced in their organizations long before Hadoop. Besides, new regulations coming into effect recently, such as the European GDPR that gives individuals the right to request erasure of personal data, have only emphasized the importance of this new feature. Not surprisingly, ACID support has quickly become a key differentiator to choose Hive over other SQL engines on top of

[6] https://hortonworks.com/blog/sub-second-analytics-hive-druid/

Hadoop. However, bringing ACID capabilities to Hive was not straightforward and the initial implementation had to go through major changes because it introduced a reading latency penalty, compared to non-ACID tables, that was unacceptable for users. This was due to a detail in the original design that we did not foresee having such a large impact on performance in production: Using a single delta type of file to store inserted, updated, and deleted records. If a workload consisted of a large number of writing operations or compaction was not run frequently enough, the file readers had to perform a sort-merge across a large number of base and delta files to consolidate the data, potentially creating memory pressure. Further, filter pushdown to skip reading entire row groups could not be applied to those delta files. The second implementation of ACID, described in this paper, was rolled out with Hive v3.1. It solves the issues described above and performance is at par with non-ACID tables.

Another common and frequent request by Hive users over time was to improve its runtime latency. LLAP development started 4 years ago to address issues that were inherent to the architecture of the system, such as the lack of data caching or long running executors. The feedback from initial deployments was quickly incorporated to the system. For instance, predictability was a huge concern in production due to cluster resource contention among user queries, which lead to the implementation of the workload manager. Currently it is extremely rewarding to see companies deploying LLAP to serve queries over TBs of data in multi-tenant clusters, achieving average latency in the order of seconds [7].

Another important decision was the integration with Calcite for query optimization. Representing queries at the right abstraction level was critical to implementing advanced optimization algorithms in Hive, which in turn brought huge performance benefits to the system. Besides, the integration was leveraged to easily generate queries for federation to other systems. Since nowadays it is common for organizations to use a plethora of data management systems, this new feature has been received with lots of excitement by Hive users.

9 ROADMAP

Continuous improvements to Hive. The community will continue working on the areas discussed in this paper to make Hive a better warehousing solution. For instance, based on the work that we have already completed to support ACID capabilities, we plan to implement multi-statement transactions. In addition, we shall continue improving the optimization techniques used in Hive. Statistics captured at runtime for query reoptimization are already persisted in HMS, which will allow us to feedback that information into the optimizer to obtain more accurate estimates, similar to other systems [53, 60]. Materialized views work is still ongoing and one of the most requested features is the implementation of an advisor or recommender [1, 61] for Hive. Improvements to LLAP performance and stability as well as the implementation of new connectors to other specialized systems, e.g., Kafka [9, 43], are in progress too.

Standalone Metastore. HMS has become a critical part in Hadoop as it is used by other data processing engines such as Spark or Presto to provide a central repository with information about all data sources in each system. Hence, there is a growing interest and ongoing work to spin-off the Metastore from Hive and to develop it as a standalone project. Each of these systems will have their own catalog in the Metastore and it will be easier to access data across different engines.

Containerized Hive in the cloud. Cloud-based data warehousing solutions such as Azure SQL DW [18], Redshift [33, 49], BigQuery [21, 46] and Snowflake [29, 51] have been gaining popularity over the last few years. Besides, there is an increasing interest in systems such as Kubernetes [44] to provide container orchestration for applications deployed indistinctly on-premises or in the cloud. Hive's modular architecture makes it easy to isolate its components (HS2, HMS, LLAP) and run them in containers. Ongoing effort is focused on finalizing the work to enable the deployment of Hive in commercial clouds using Kubernetes.

10 CONCLUSIONS

Apache Hive's early success stemmed from the ability to exploit parallelism for batch operations with a well-known interface. It made data load and management simple, handling node, software and hardware failures gracefully without expensive repair or recovery times.

In this paper we show how the community expanded the utility of the system from ETL tool to fully-fledged enterprise-grade data warehouse. We described the addition of a transactional system that is well suited for the data modifications required in star schema databases. We showed the main runtime improvements necessary to bring query latency and concurrency into the realm of interactive operation. We also described cost-based optimization techniques necessary to handle today's view hierarchies and big-data operations. Finally, we showed how Hive can be used today as a relational front-end to multiple storage and data systems. All of this happened without ever compromising on the original characteristics of the system that made it popular.

Apache Hive architecture and design principles have proven to be powerful in today's analytic landscape. We believe it will continue to thrive in new deployment and storage environments as they emerge, as it is showing today with containerization and cloud.

[7] https://azure.microsoft.com/en-us/blog/hdinsight-interactive-query-performance-benchmarks-and-integration-with-power-bi-direct-query/

ACKNOWLEDGMENTS

We would like to thank the Apache Hive community, contributors and users, who build, maintain, use, test, write about, and continue to push the project forward. We would also like to thank Oliver Draese for his feedback on the original draft of this paper.

REFERENCES

[1] Sanjay Agrawal, Surajit Chaudhuri, and Vivek R. Narasayya. 2000. Automated Selection of Materialized Views and Indexes in SQL Databases. In *PVLDB*.

[2] Apache Atlas 2018. Apache Atlas: Data Governance and Metadata framework for Hadoop. http://atlas.apache.org/.

[3] Apache Calcite 2018. Apache Calcite: Dynamic data management framework. http://calcite.apache.org/.

[4] Apache Druid 2018. Apache Druid: Interactive analytics at scale. http://druid.io/.

[5] Apache Flink 2018. Apache Flink: Stateful Computations over Data Streams. http://flink.apache.org/.

[6] Apache HBase 2018. Apache HBase. http://hbase.apache.org/.

[7] Apache Hive 2018. Apache Hive. http://hive.apache.org/.

[8] Apache Impala 2018. Apache Impala. http://impala.apache.org/.

[9] Apache Kafka 2018. Apache Kafka: A distributed streaming platform. http://kafka.apache.org/.

[10] Apache ORC 2018. Apache ORC: High-Performance Columnar Storage for Hadoop. http://orc.apache.org/.

[11] Apache Parquet 2018. Apache Parquet. http://parquet.apache.org/.

[12] Apache Ranger 2018. Apache Ranger: Framework to enable, monitor and manage comprehensive data security across the Hadoop platform. http://ranger.apache.org/.

[13] Apache Sentry 2018. Apache Sentry: System for enforcing fine grained role based authorization to data and metadata stored on a Hadoop cluster. http://sentry.apache.org/.

[14] Apache Spark 2018. Apache Spark: Unified Analytics Engine for Big Data. http://spark.apache.org/.

[15] Apache Tez 2018. Apache Tez. http://tez.apache.org/.

[16] Apache Thrift 2018. Apache Thrift. http://thrift.apache.org/.

[17] Peter M. G. Apers, Alan R. Hevner, and S. Bing Yao. 1983. Optimization Algorithms for Distributed Queries. *IEEE Trans. Software Eng.* 9, 1 (1983), 57–68.

[18] Azure SQL DW 2018. Azure SQL Data Warehouse. http://azure.microsoft.com/en-us/services/sql-data-warehouse/.

[19] Edmon Begoli, Jesús Camacho-Rodríguez, Julian Hyde, Michael J. Mior, and Daniel Lemire. 2018. Apache Calcite: A Foundational Framework for Optimized Query Processing Over Heterogeneous Data Sources. In *SIGMOD*.

[20] Philip A. Bernstein, Nathan Goodman, Eugene Wong, Christopher L. Reeve, and James B. Rothnie Jr. 1981. Query Processing in a System for Distributed Databases (SDD-1). *ACM Trans. Database Syst.* 6, 4 (1981), 602–625.

[21] BigQuery 2018. BigQuery: Analytics Data Warehouse. http://cloud.google.com/bigquery/.

[22] Vinayak R. Borkar, Michael J. Carey, Raman Grover, Nicola Onose, and Rares Vernica. 2011. Hyracks: A flexible and extensible foundation for data-intensive computing. In *ICDE*.

[23] Francesca Bugiotti, Damian Bursztyn, Alin Deutsch, Ioana Ileana, and Ioana Manolescu. 2015. Invisible Glue: Scalable Self-Tunning Multi-Stores. In *CIDR*.

[24] Michael J. Cahill, Uwe Röhm, and Alan David Fekete. 2008. Serializable isolation for snapshot databases. In *SIGMOD*.

[25] Jesús Camacho-Rodríguez, Dario Colazzo, Melanie Herschel, Ioana Manolescu, and Soudip Roy Chowdhury. 2016. Reuse-based Optimization for Pig Latin. In *CIKM*.

[26] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache Flink™: Stream and Batch Processing in a Single Engine. *IEEE Data Eng. Bull.* 38, 4 (2015), 28–38.

[27] Michael J. Carey, Laura M. Haas, Peter M. Schwarz, Manish Arya, William F. Cody, Ronald Fagin, Myron Flickner, Allen Luniewski, Wayne Niblack, Dragutin Petkovic, Joachim Thomas, John H. Williams, and Edward L. Wimmers. 1995. Towards Heterogeneous Multimedia Information Systems: The Garlic Approach. In *RIDE-DOM Workshop*.

[28] Surajit Chaudhuri, Ravi Krishnamurthy, Spyros Potamianos, and Kyuseok Shim. 1995. Optimizing Queries with Materialized Views. In *ICDE*.

[29] Benoît Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, Allison W. Lee, Ashish Motivala, Abdul Q. Munir, Steven Pelley, Peter Povinec, Greg Rahn, Spyridon Triantafyllis, and Philipp Unterbrunner. 2016. The Snowflake Elastic Data Warehouse. In *SIGMOD*.

[30] DataNucleus 2018. DataNucleus: JDO/JPA/REST Persistence of Java Objects. http://www.datanucleus.org/.

[31] Jonathan Goldstein and Per-Åke Larson. 2001. Optimizing Queries Using Materialized Views: A practical, scalable solution. In *SIGMOD*.

[32] Timothy Griffin and Leonid Libkin. 1995. Incremental Maintenance of Views with Duplicates. In *SIGMOD*.

[33] Anurag Gupta, Deepak Agarwal, Derek Tan, Jakub Kulesza, Rahul Pathak, Stefano Stefani, and Vidhya Srinivasan. 2015. Amazon Redshift and the Case for Simpler Data Warehouses. In *SIGMOD*.

[34] Ashish Gupta and Inderpal Singh Mumick. 1995. Maintenance of Materialized Views: Problems, Techniques, and Applications. *IEEE Data Eng. Bull.* 18, 2 (1995), 3–18.

[35] Himanshu Gupta. 1997. Selection of Views to Materialize in a Data Warehouse. In *ICDT*.

[36] Himanshu Gupta, Venky Harinarayan, Anand Rajaraman, and Jeffrey D. Ullman. 1997. Index Selection for OLAP. In *ICDE*.

[37] Venky Harinarayan, Anand Rajaraman, and Jeffrey D. Ullman. 1996. Implementing Data Cubes Efficiently. In *SIGMOD*.

[38] Stefan Heule, Marc Nunkesser, and Alexander Hall. 2013. HyperLogLog in practice: algorithmic engineering of a state of the art cardinality estimation algorithm. In *EDBT*.

[39] Yin Huai, Ashutosh Chauhan, Alan Gates, Günther Hagleitner, Eric N. Hanson, Owen O'Malley, Jitendra Pandey, Yuan Yuan, Rubao Lee, and Xiaodong Zhang. 2014. Major technical advancements in apache hive. In *SIGMOD*.

[40] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. 2007. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys*.

[41] Alekh Jindal, Shi Qiao, Hiren Patel, Zhicheng Yin, Jieming Di, Malay Bag, Marc Friedman, Yifung Lin, Konstantinos Karanasos, and Sriram Rao. 2018. Computation Reuse in Analytics Job Service at Microsoft. In *SIGMOD*.

[42] Marcel Kornacker, Alexander Behm, Victor Bittorf, Taras Bobrovytsky, Casey Ching, Alan Choi, Justin Erickson, Martin Grund, Daniel Hecht, Matthew Jacobs, Ishaan Joshi, Lenni Kuff, Dileep Kumar, Alex Leblang, Nong Li, Ippokratis Pandis, Henry Robinson, David Rorke, Silvius Rus, John Russell, Dimitris Tsirogiannis, Skye Wanderman-Milne, and Michael Yoder. 2015. Impala: A Modern, Open-Source SQL Engine for Hadoop. In *CIDR*.

[43] Jay Kreps, Neha Narkhede, and Jun Rao. 2011. Kafka : a Distributed Messaging System for Log Processing. In *NetDB*.

[44] Kubernetes 2018. Kubernetes: Production-Grade Container Orchestration. http://kubernetes.io/.

[45] Per-Åke Larson, Cipri Clinciu, Campbell Fraser, Eric N. Hanson, Mostafa Mokhtar, Michal Nowakiewicz, Vassilis Papadimos, Susan L. Price, Srikumar Rangarajan, Remus Rusanu, and Mayukh Saubhasik. 2013. Enhancements to SQL server column stores. In *SIGMOD*.

[46] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. 2010. Dremel: Interactive Analysis of Web-Scale Datasets. In *PVLDB*.

[47] Patrick E. O'Neil, Elizabeth J. O'Neil, Xuedong Chen, and Stephen Revilak. 2009. The Star Schema Benchmark and Augmented Fact Table Indexing. In *TPCTC*.

[48] Presto 2018. Presto: Distributed SQL query engine for big data. http://prestodb.io/.

[49] Redshift 2018. Amazon Redshift: Amazon Web Services. http://aws.amazon.com/redshift/.

[50] Bikas Saha, Hitesh Shah, Siddharth Seth, Gopal Vijayaraghavan, Arun C. Murthy, and Carlo Curino. 2015. Apache Tez: A Unifying Framework for Modeling and Building Data Processing Applications. In *SIGMOD*.

[51] Snowflake 2018. Snowflake: The Enterprise Data Warehouse Built in the Cloud. http://www.snowflake.com/.

[52] Mohamed A. Soliman, Lyublena Antova, Venkatesh Raghavan, Amr El-Helw, Zhongxian Gu, Entong Shen, George C. Caragea, Carlos Garcia-Alvarado, Foyzur Rahman, Michalis Petropoulos, Florian Waas, Sivaramakrishnan Narayanan, Konstantinos Krikellas, and Rhonda Baldwin. 2014. Orca: a modular query optimizer architecture for big data. In *SIGMOD*.

[53] Michael Stillger, Guy M. Lohman, Volker Markl, and Mokhtar Kandil. 2001. LEO - DB2's LEarning Optimizer. In *PVLDB*.

[54] Michael Stonebraker and Ugur Çetintemel. 2005. "One Size Fits All": An Idea Whose Time Has Come and Gone. In *ICDE*.

[55] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Ning Zhang, Suresh Anthony, Hao Liu, and Raghotham Murthy. 2010. Hive - a petabyte scale data warehouse using Hadoop. In *ICDE*.

[56] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. 2013. Apache Hadoop YARN: yet another resource negotiator. In *SOCC*.

[57] Gio Wiederhold. 1992. Mediators in the Architecture of Future Information Systems. *IEEE Computer* 25, 3 (1992), 38–49.

[58] Fangjin Yang, Eric Tschetter, Xavier Léauté, Nelson Ray, Gian Merlino, and Deep Ganguli. 2014. Druid: a real-time analytical data store. In *SIGMOD*.

[59] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster Computing with Working Sets. In *USENIX HotCloud*.

[60] Mohamed Zaït, Sunil Chakkappen, Suratna Budalakoti, Satyanarayana R. Valluri, Ramarajan Krishnamachari, and Alan Wood. 2017. Adaptive Statistics in Oracle 12c. In *PVLDB*.

[61] Daniel C. Zilio, Jun Rao, Sam Lightstone, Guy M. Lohman, Adam J. Storm, Christian Garcia-Arellano, and Scott Fadden. 2004. DB2 Design Advisor: Integrated Automatic Physical Database Design. In *PVLDB*.