Major Technical Advancements in Apache Hive

Yin Huai[1] Ashutosh Chauhan[2] Alan Gates[2] Gunther Hagleitner[2] Eric N. Hanson[3]
Owen O'Malley[2] Jitendra Pandey[2] Yuan Yuan[1] Rubao Lee[1] Xiaodong Zhang[1]

[1]The Ohio State University [2]Hortonworks Inc. [3]Microsoft

[1]{huai, yuanyu, liru, zhang}@cse.ohio-state.edu
[2]{ashutosh, gates, ghagleitner, owen, jitendra}@hortonworks.com
[3]ehans@microsoft.com

ABSTRACT

Apache Hive is a widely used data warehouse system for Apache Hadoop, and has been adopted by many organizations for various big data analytics applications. Closely working with many users and organizations, we have identified several shortcomings of Hive in its file formats, query planning, and query execution, which are key factors determining the performance of Hive. In order to make Hive continuously satisfy the requests and requirements of processing increasingly high volumes data in a scalable and efficient way, we have set two goals related to storage and runtime performance in our efforts on advancing Hive. First, we aim to maximize the effective storage capacity and to accelerate data accesses to the data warehouse by updating the existing file formats. Second, we aim to significantly improve cluster resource utilization and runtime performance of Hive by developing a highly optimized query planner and a highly efficient query execution engine. In this paper, we present a community-based effort on technical advancements in Hive. Our performance evaluation shows that these advancements provide significant improvements on storage efficiency and query execution performance. This paper also shows how academic research lays a foundation for Hive to improve its daily operations.

Categories and Subject Descriptors

H.2 [**Database Management**]: Systems

Keywords

Databases; Data Warehouse; Hadoop; Hive; MapReduce

1. INTRODUCTION

Apache Hive is a data warehouse system for Apache Hadoop [1]. It has been widely used in organizations to manage and process large volumes of data, such as eBay, Facebook, LinkedIn, Spotify, Taobao, Tencent, and Yahoo!. As an open source project, Hive has a strong technical development community working with widely located and diverse users and organizations. In recent years, more than 100 developers have made technical efforts to improve Hive on more than 3000 issues. With its rapid development pace, Hive has been significantly updated by new innovations and research since the original Hive paper [45] was published four years ago. We will present its major technical advancements in this paper.

Hive was originally designed as a translation layer on top of Hadoop MapReduce. It exposes its own dialect of SQL to users and translates data manipulation statements (queries) to a directed acyclic graph (DAG) of MapReduce jobs. With an SQL interface, users do not need to write tedious and sometimes difficult MapReduce programs to manipulate data stored in Hadoop Distributed Filesystem (HDFS).

This highly abstracted SQL interface significantly improves the productivity of data management in Hadoop and accelerates the adoption of Hive. The efficiency and productivity of Hive are largely affected by how its data warehouse layer is designed, implemented, and optimized to best utilize the underlying data processing engine (e.g. Hadoop MapReduce) and HDFS. In order to make Hive continuously satisfy requirements of processing increasingly high volumes of data in a scalable and efficient way, we must improve both data storage as well as query execution aspect of Hive. First, Hive should be able to store datasets managed by it in an efficient way which guarantees both storage efficiency as well as fast data access. Second, Hive should be able to generate highly optimized query plans and execute them using a query execution model that utilizes hardware resources well.

Closely working with many users and organizations, the Hive development community has identified several shortcomings in its file formats, query planning, and query execution, which largely determine performance of queries submitted to Hive. In this paper, we present a community-based effort on addressing these several shortcomings with strong support and scientific basis from several academic research results. The main contributions of this paper are:

1. A new file format, Optimized Record Columnar File (ORC File), has been added to Hive which provides high storage and data access efficiency with low overhead.
2. The query planning component has been updated with insightful analyses in complex queries, significantly reducing unnecessary operations in query plans.
3. A vectorized query execution model has been introduced to Hive, which significantly improves the efficiency of query execution at runtime by better utilizing Modern CPUs.

The rest of this paper is organized as follows. Section 2 provides an overview of Hive's Architecture. Section 3 presents shortcomings we have identified and the rationale of the related advancements. Section 4 introduces the advancement on the file format component. Section 5 introduces the advancement on the query

SIGMOD'14, June 22–27, 2014, Snowbird, UT, USA.

http://dx.doi.org/10.1145/2588555.2595630.

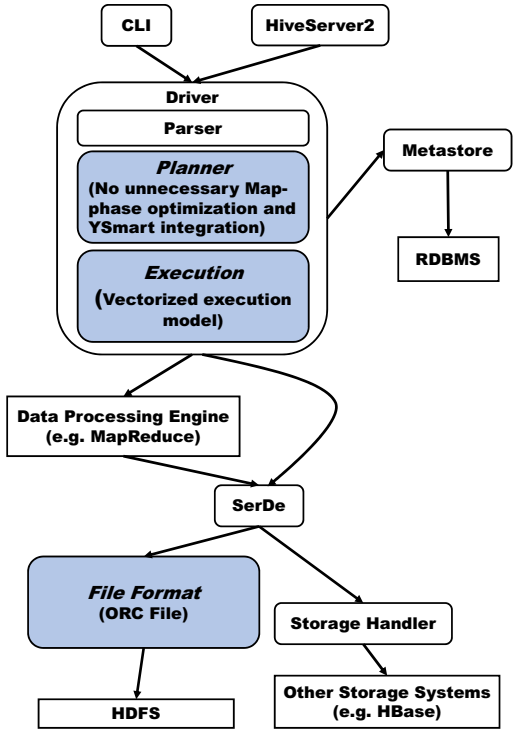


Figure 1: The architecture of Hive. Rounded rectangles are components in Hive. Shaded rounded rectangles are advanced components, and they also show major advancements that will be introduced in this paper.

planning component. Section 6 introduces the advancement on the query execution component. Section 7 reports results of our evaluation on these advancements. Section 8 surveys related work. Section 9 is the conclusion.

2. HIVE ARCHITECTURE

Figure 1 shows the architecture of Hive. Hive exposes two interfaces to users to submit their statements. These interfaces are *Command Line Interface (CLI)* and *HiveServer2* [2]. Through these two interfaces, a statement will be submitted to the *Driver*. The Driver first parses the statement and then passes the Abstract Syntax Tree (AST) corresponding to this statement to the *Planner*. The Planner then chooses a specific planner implementation to analyze different types of statements. During the process of analyzing a submitted statement, the Driver needs to contact the *Metastore* to retrieve needed metadata from a Relational Database Management System (RDBMS), e.g. PostgreSQL.

Queries used for data retrieval and processing are analyzed by the *Query Planner*. Hive translates queries to executable jobs for an underlying data processing engine that is currently Hadoop MapReduce [1]. For a submitted query, the query planner walks the AST of this query and assembles the operator tree to represent data operations of this query. An operator in Hive represents a specific data operation. For example, a `FilterOperator` is used to evaluate predicates on its input records. Because a query submitted to Hive will be evaluated in a distributed environment, the query planner will also figure out if an operator requires its input records to be partitioned in a certain way. If so, it then inserts a boundary represented by one or multiple `ReduceSinkOperators` (`RSOps`) before this operator to indicate that the child operator of these `RSOps` need rows from a re-partitioned dataset. For example, for a group-by clause `GROUP BY key`, a `RSOp` will be used to tell the underlying MapReduce engine to group rows having the same value of `key`. After an operator tree is generated, the query planner applies a set of optimizations to the operator tree. Then, the entire operator tree will be passed to the task compiler, which breaks the operator tree to multiple stages represented by executable tasks. For example, the MapReduce task compiler generates a DAG of Map/Reduce tasks assembled in MapReduce jobs based on an operator tree. In the end of query planning, another phase of optimizations are applied to generated tasks.

After the query planner has generated MapReduce jobs, the Driver will submit those jobs to the underlying MapReduce engine to evaluate the submitted query. In the execution of a Map/Reduce task, operators inside this task are first initialized and then they will process rows fetched by the MapReduce engine in a pipelined fashion. To read/write a table with a specific file format, Hive assigns the corresponding file reader/writer to tasks reading/writing this table. For a file format, a serialization-deserialization library (called SerDe in the rest of this paper) is used to serialize and deserialize data. After all MapReduce jobs have finished, the Driver will fetch the results of the query to the user who submitted the query.

Besides processing data directly stored in HDFS, Hive can also process data stored in other storage systems, e.g. HBase [4]. For those systems, a corresponding *Storage Handler* is needed. For example, the HBase storage handler is used when a query needs to read or write data from or to HBase.

3. IDENTIFIED SHORTCOMINGS

For increasingly diverse organizations, Hadoop has become the de facto place for data storage and processing [46]. Meanwhile, Hive has become one of primary gateways to manipulate, retrieve, and process data stored and managed in Hadoop and its supported systems. With increasing adoption of Hive in production systems, the data volume managed by Hive has increased many folds. At the same time, query workloads on Hive have become much more diverse. From the storage perspective, users expect Hive to efficiently store high volumes of data in a given storage space. This helps users achieve a cost-effective deployment of HDFS. From the query execution perspective, Hive already delivers a high throughput by utilizing Hadoop MapReduce at runtime. However, with new users coming on board, there is an increasing expectation for Hive to provide faster response time. Another motivating factor to deliver a low latency response is a better integration of Hive with business intelligence tools. Users of such tools expect an interactive response time. To meet these two demands, the storage efficiency and query execution performance of Hive must be further improved. In the rest of this section, we will revisit those critical components determining the storage efficiency and query execution performance, and identify shortcomings in these components.

In Hive, the storage efficiency is determined by SerDes and file formats. Hive originally used two simple file formats provided by Hadoop, which are `TextFile` and `SequenceFile`. A file stored with `TextFile` contains plain text data. While, a file stored with `SequenceFile` is a flat file consisting of binary key/value pairs. Because these two formats are data-type-agnostic, Hive uses SerDes to serialize every row into plain text or binary sequences when writing data to HDFS, and to deserialize files in these two formats back to rows with their original schema when reading data from HDFS. Because these two formats store rows in a row-by-

[1]Starting from Hive 0.13, a query can also be translated to a job that is executable in Apache Tez [3]. Without loss of generality, we will mainly use MapReduce in our paper since it is the original data processing engine used by Hive.

row manner, rows have more entropy, which make them hard to be densely compressed. In Hive 0.4, Record Columnar File (RCFile) [27] was introduced. Because RCFile is a columnar file format, it achieved certain improvement on storage efficiency. However, RCFile is still data-type-agnostic and its corresponding SerDe serializes a single row at a time. Under this structure, data-type specific compression schemes cannot be effectively used. Thus, **the first key shortcoming of Hive on storage efficiency is that data-type-agnostic file formats and one-row-at-a-time serialization prevent data values being efficiently compressed**.

The query execution performance is largely determined by the file format component, the query planning component (the query planner), and the query execution component containing implementation of operators. Although RCFile is a columnar file format, its SerDe does not decompose a complex data type (e.g. `Map`). Thus, when a query needs to access a field of a complex data type, all fields of this type have to be read, which introduces inefficiency on data reading. Also, RCFile was mainly designed for sequential data scan. It does not have any index and it does not take advantages of semantic information provided by queries to skip unnecessary data. Thus, **the second key shortcoming of the file format component is that data reading efficiency is limited by the lack of indexes and non-decomposed columns with complex data types**.

The query planner in the original Hive translates every operation specified in this query to an operator. When an operation requires its input datasets to be partitioned in a certain way [2], Hive will insert `RSOps` as the boundary between a Map phase and a Reduce phase. Then, the query planner will break the entire operator tree to MapReduce jobs based on these boundaries. During query planning and optimization, the planner only focuses on a single data operation or a single MapReduce job at a time. This query planning approach can significantly degrade the query execution performance by introducing unnecessary and time consuming operations. For example, the original query planner was not aware of correlations between major operations in a query [31]. Thus, **the third key shortcoming of the query planner is that the query translation approach ignores relationships between data operations, and thus introduces unnecessary operations hurting the performance of query execution**.

The query execution component of Hive was heavily influenced by the working model of Hadoop MapReduce. In a Map or a Reduce task, the MapReduce engine fetches a key/value pair and then forwards it to the Map or Reduce function at a time. For example, in the case of word count, every Map task processes a line of a text file at a time. Hive inherited this working model and it processes rows with a one-row-at-a-time way. However, this working model does not fit the architecture of modern CPUs and introduces high interpretation overhead, under-utilized parallelism, low cache performance, and high function call overhead [19] [35] [50]. Thus, **the fourth key shortcoming of the query execution component is that the runtime execution efficiency is limited by the one-row-at-a-time execution model**.

4. FILE FORMAT

To address the shortcoming of the storage and data access efficiency, we have designed and implemented an improved file format called Optimized Record Columnar File (ORC File) [3] which has several significant improvements over RCFile. In ORC File, we have de-emphasized its SerDe and made the ORC file writer data type aware. With this change, we are able to add various type-specific data encoding schemes to store data efficiently. To efficiently access data stored with ORC File, we have introduced different kinds of indexes that do not exist in RCFile. Those indexes are critical to help the ORC reader find needed data and skip unnecessary data. Also, because the ORC writer is aware of the data type of a value, it can decompose a column with a complex data type (e.g. `Map`) to multiple child columns, which cannot be done by RCFile. Besides these three major improvements, we also have introduced several practical improvements in ORC File, e.g. a larger default stripe size and a memory manager to bound the memory footprint of an ORC writer, aiming to overcome inconvenience and inefficiency we have observed through years' operation of RCFile.

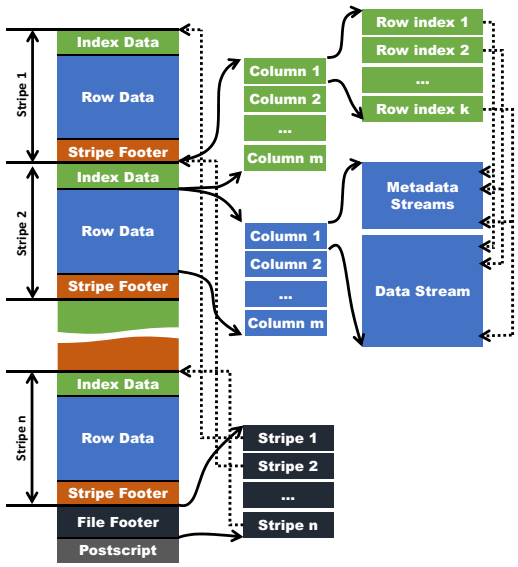


Figure 2: The structure of an ORC file. Round-dotted lines represent position pointers.

In this section, we give an overview of ORC File and its improvements over RCFile. First, we introduce the way that ORC File organizes and stores data values of a table (table placement method). Then, we introduce indexes and compression schemes in ORC File. Finally, we introduce the memory manager in ORC File, which is a critical and yet often ignored component. This memory manager bounds the total memory footprint of a task writing ORC files. It can prevent the task from failing caused by running out of memory. Figure 2 shows the structure of an ORC file. We will use it to explain the design of ORC File.

4.1 The Table Placement Method

When a table is stored by a file format, the table placement method of this file format describes the way that data values of this table are organized and stored in underlying filesystems. Based on the definition in [28], the table placement method of ORC File shares the basic structure with that of RCFile. For a table stored in an ORC file, it is first horizontally partitioned to multiple *stripes*. Then, in a stripe, data values are stored in a column by column way. All columns in a stripe are stored in the same file. Also, to be adaptive to different query patterns, especially ad hoc queries, ORC File does not put columns into column groups.

From the perspective of the table placement method, ORC File has three improvements over RCFile. First, it provides a larger de-

[2] In the rest of this paper, this kind of data operations are called major operations and an operator evaluating a major operation is called a major operator.

[3] In the rest of this paper, we use *ORC File* to refer to the file format we introduced and we use *an ORC file* or *ORC files* to refer to one or multiple files stored in HDFS with the format of ORC File.

Table 1: The approach to decomposing a column with a complex data type.

Data type	Child columns
`Array`	A single child column containing array elements.
`Map`	Two child columns, the key field and the value field.
`Struct`	Every field is a child column.
`Union`	Every field is a child column.

fault stripe size (256 MB). While, the default stripe size in RCFile size is 4 MB [4]. With a larger stripe, ORC File can achieve more efficient data reading operations than RCFile because the reader of ORC File can better utilize the sequential bandwidth of hard disks and read less unnecessary data from its underlying filesystem.

The second improvement is that for a column with a complex data type (e.g. `Map`), unlike RCFile, the writer of ORC File decomposes this column to multiple child columns. Table 1 summarizes complex data types in Hive and how they are decomposed. After the process of column decomposition, columns in a table form a tree structure and only leaf nodes (leaf columns) are storing data values of this table. Actual data of a leaf column are stored in a data stream. To facilitate the reader of an ORC file, the metadata of a column are also stored in metadata streams. In the column tree, internal columns (represented by internal nodes in the tree structure) are used to record metadata, e.g. the length of an array. So, those internal columns will not have data streams. Figure 3 shows an example table (defined in Figure 3(a)) and the column tree of it. The schema of the table defines a row with a `Struct` type which is the root column (Column id is 0). Because `col1`, `col2`, `col4`, and `col9` are four fields of a row, they appear as child columns of the root columns. With the decomposition shown in Table 1, this table is decomposed to six leaf columns. When reading a stripe, those leaf columns are read in a column by column way. With the help from those internal columns, we can reconstruct a row from those leaf columns. Also, for a column with a complex data type, the ORC reader is able to only read needed child columns [5].

The last improvement of ORC File over RCFile from the perspective of a table placement method is that users of ORC File can have the choice to align stripe boundaries in an ORC file with the HDFS block boundaries. For an ORC file, the stripe is usually smaller than the HDFS block size used for this ORC file. Without this alignment, it is possible that a stripe is stored in two HDFS blocks, which means this stripe may be stored in different machines. So, reading a stripe stored in two HDFS blocks likely involves remote data reading operations. When this alignment is enabled and the remaining space of a HDFS block cannot fit a stripe, the writer of an ORC file will pad data to the end of this HDFS block. The next stripe will start at the beginning of next HDFS block. With this alignment, a stripe will be always stored in a single HDFS block.

4.2 Indexes

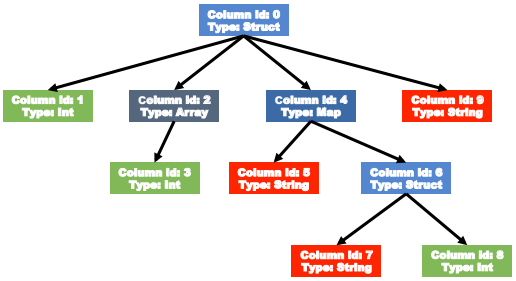
To efficiently read data from HDFS, we have added indexes to ORC File. Because the speed of loading data into Hive and storage efficiency are important to a file format in Hive, in the design of ORC File, we decided to only use sparse indexes. There are two kinds of sparse indexes that are data statistics and position pointers.

[4]To be consistent with ORC File, we use the term stripe at here. Actually, a stripe in RCFile is called a row group.

[5]As of Hive 0.13, Hive has not taken advantage of this ORC File feature. Future releases of Hive will be able to only read needed child columns of a column with a complex data type.

```
CREATE TABLE tbl (
  col1 Int,
  col2 Array<Int>,
  col4 Map<String,
           Struct<col7:String,
                  col8:Int>>,
  col9 String
)
```

(a) The schema of the table



(b) The column tree after column decomposition.

Figure 3: An example table and how columns in it are decomposed.

Data Statistics.

Data statistics are used by the ORC reader to not read unnecessary data from the HDFS. These statistics are created when the ORC writer is creating an ORC file. Representative statistics are the number of values, the minimum value, the maximum value, the sum, and the length (for text types and binary type). For complex data types, `Array`, `Map`, `Struct` and `Union`, their child columns also record data statistics.

In ORC File, data statistics have three levels. First, columns in an ORC file have file level statistics which are recorded at the end of this file. These statistics are used in query optimizations, and they are also used to answer simple aggregation queries. Second, ORC File stores stripe level statistics for every column. ORC readers use these statistics to analyze which stripes are needed to evaluate a query. Those unneeded stripes will not be read from HDFS. Third, to further reduce the amount of unnecessary data read from HDFS, besides stripe level statistics, we have fine-grained statistics inside a stripe, which are called index group level statistics. We logically break data values of a column to multiple index groups with a fixed number of values. The default number of values in an index group is 10000. Data statistics are recorded for every index group. With data statistics for every index group, the query processing engine of Hive can push certain predicates to the reader of an ORC file and an ORC reader will not read unnecessary index groups from HDFS. The number of values in an index group is configurable. An index group containing a small number of values can provide more fine-grained statistics about a column. However, the size of data statistics will increase. Thus, users should tradeoff the size and the granularity of data statistic when determining the number of values in an index group.

Position Pointers.

When reading an ORC file, the reader needs to know two kinds of positions to perform efficient data reading operations. First, because a column in a stripe has multiple logical index groups, the reader of an ORC file needs to know the starting points of every index group in metadata streams and data streams. In Figure 2, round-dotted lines pointing to the metadata stream and data stream represent this kind of position pointers. Second, an ORC file can contain multiple stripes and a HDFS block of this ORC file can

contain multiple stripes. To efficiently locate the starting point of a stripe, position pointers of stripes are needed. Those pointers are stored in the file footer of an ORC file (round-dotted lines pointing to starting points of stripes in Figure 2).

4.3 Compression

ORC File uses a two-level compression scheme. A stream is first encoded by a stream type specific data encoding scheme. Then, an optional general-purpose data compression scheme can be used to further compress this stream.

For a column, it is stored in one or multiple streams. Based on the type of a stream, we can divide streams to four primitive types. Based on its type, a stream has its own data encoding scheme. These four types of primitive streams are introduced as follows.

- **Byte Stream**: A byte stream basically stores a sequence of bytes and it does not encode data.
- **Run Length Byte Stream**: A run length byte stream stores a sequence of bytes. For a sequence of identical bytes, it stores the repeated byte and the occurrences.
- **Integer Stream**: An integer stream stores a sequence of integers. It can encode these integers with run length encoding and delta encoding. The specific encoding schemes used for a sub-sequence of integers are determined based on the pattern of it.
- **Bit Field Stream**: A bit field stream is used to store a sequence of boolean values. In this stream, a bit represents a boolean value. Under the cover, a bit field stream is backed by a run length byte stream.

Due to the page limit, we are unable to explain what streams are used for every column type. We will present how an `Int` column and a `String` column are stored as two examples. Interested readers may refer to [5] for details. For an `Int` column, one bit field stream and one integer stream are used. The bit field stream is used to record if a value is null. The integer stream is used to record integer values of this `Int` column. For a `String` column, the ORC writer will first check if using dictionary encoding can store data efficiently by evaluating if the ratio of the number of distinct entries in the dictionary to the number of encoded values is not greater than a configurable threshold (the default threshold is 0.8). If so, the ORC writer will use dictionary encoding scheme, and this column will be stored in one bit field stream, one byte stream, and two integer streams. Like an `Int` column, the bit field stream is used to record if a value is null. The byte stream is used to store the dictionary. One integer stream is used to store the length of each entry in the dictionary. The second integer stream is used to store values of this column. If the ratio of the number of distinct entries in the dictionary to the number of encoded values is greater than the threshold, the ORC writer will know that there are many distinct values and using dictionary encoding cannot efficiently store the data. Thus, it will automatically store this column without the dictionary encoding. Instead of storing the dictionary and storing values as indexes to the dictionary, the ORC writer will use a byte stream to store values of this `String` column and use an integer stream to store the length of each value.

In ORC File, besides those stream type specific schemes, users can further ask the writer of an ORC file to compress streams with a general-purpose codec among ZLIB, Snappy and LZO. For a stream, the general-purpose codec will compress this stream to multiple small compression units. In the current implementation, the default size of a compression unit is 256 KB.

4.4 Memory Manager

When the writer of an ORC file writes data, it buffers the entire stripe in memory. Thus, the memory footprint of an ORC writer is the size of a stripe. Because the default size of a stripe is large, when there are lots of writers concurrently writing to multiple ORC files in a single Map or Reduce task (for example, when a user uses dynamic partitioning, and partitioning columns have lots of distinct values), this task can run out of memory. To bound the memory consumption of those concurrent writers, we add a memory manager in ORC File. In a Map or Reduce task, the memory manager sets a threshold which bounds the maximum amount of memory that can be used by writers in this task [6]. Then, every new writer registers to this memory manager with its stripe size (registered stripe size). When the total amount of memory used by writers (the total registered stripe sizes) exceeds the memory threshold, the memory manager will scale down the actual stripe sizes used in those writers with a ratio of the memory threshold to the total registered stripe sizes. When a writer is closed, the memory manager will subtract the registered stripe size of this writer from the total registered stripe size. If the total registered stripe size is under the threshold, the actual stripe sizes of all writers will be set back to their original stripe sizes. With this control mechanism, the memory footprint of active writers of ORC files in a Map or Reduce task is bounded.

5. QUERY PLANNING

To address the shortcoming of the query planner, we first have identified three major issues caused by the the original query translation approach of Hive, introducing unnecessary operations and data movements, and significantly degrading the performance of a query. These issues are summarized as follows.

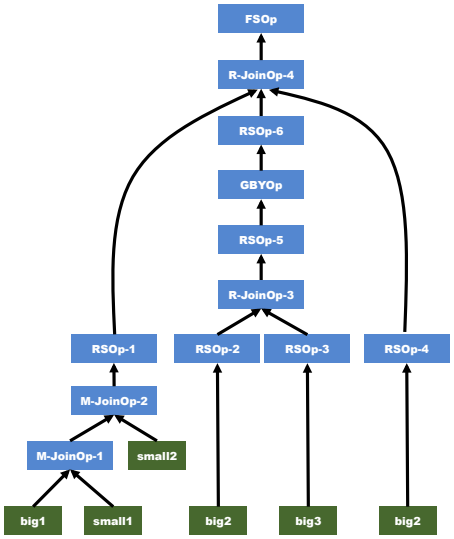
- **Unecessary Map phases.** Because a MapReduce job can only shuffle data once, it is common that a query will be executed by multiple MapReduce jobs. In this case, the Map phase loading intermediate results is used merely to load data back from HDFS for data shuffling. Thus, if a MapReduce job generating intermediate results does not have a Reduce phase, it introduces an unnecessary Map phase to load its outputs back from HDFS.
- **Unnecessary data loading.** For a query, a table can be used by multiple operations. If these operations are executed in the same Map phase, Hive can load the table once. However, if these operations are in different MapReduce jobs, this table will be loaded multiple times, which introduce extra I/O operations to the query evaluation.
- **Unnecessary data re-partitioning.** Originally, Hive generates a MapReduce job for every major operation (a data operation requiring its input datasets to be partitioned in a certain way). In a complex query, the input datasets of a major operation may be already partitioned in a proper way by its previous major operations. For this case, we call these two operations are correlated. The original Hive ignores correlations between major operations and thus, can introduce unnecessary data re-partitioning, which results in unnecessary MapReduce jobs and poor query evaluation performance.

For unnecessary Map phases, we have analyzed when Hive will generate Map-only jobs and added an optimization to merge a Map-only job to its child job (Section 5.1). For unnecessary data loading and re-partitioning, we have introduced a Correlation Optimizer to eliminate unnecessary MapReduce jobs (Section 5.1). In this sec-

[6]The default threshold is half of the total memory allocated to this task.

```
SELECT big1.key, small1.value1, small2.value1,
       big2.value1, sq1.total
FROM big1
JOIN small1 ON (big1.sKey1 = small1.key)
JOIN small2 ON (big1.sKey2 = small2.key)
JOIN (SELECT key,
             avg(big3.value1) AS avg,
             sum(big3.value2) AS total
      FROM big2 JOIN big3 ON (big2.key = big3.key)
      GROUP BY big2.key) sq1 ON (big1.key = sq1.key)
JOIN big2 ON (sq1.key = big2.key)
WHERE big2.value1 > sq1.avg;
```

(a) Query



(b) Operator tree

Figure 4: The running example used in Section 5. For an arrow connecting two operators, it starts from the parent operator and ends at the child operator.

tion, we use a running example shown in Figure 4(a) to illustrate optimizations that we will introduce in the rest of this section.

5.1 Eliminating Unnecessary Map phases

In Hive, a Map-only job is generated when the query planner converts a MapReduce job for a Reduce Join to a Map Join. In a Reduce Join, input tables are shuffled and they are joined in Reduce tasks. On the other hand, in a Map Join, two tables are joined in Map tasks. There are several join schemes for a Map Join. One representative example is, for a two way join, to build a hashtable for the smaller table and load it in every Map task reading the larger table for a hash join.

Because we convert a Reduce Join to a Map Join after MapReduce jobs have been assembled, we have a Map-only job for every Map Join at first. Consequently, we would introduce unnecessary operations and elapsed time to the evaluation of the submitted query. To eliminate those unnecessary Map phases, every time when we convert a Reduce Join to a Map Join, we try to merge the generated Map-only job to its child job if the total size of small tables used to build hash tables in the merged job is under a configurable threshold. This threshold is used to prevent a Map task loading a partition of the big table running out of memory.

In our example shown in Figure 4, `small1` and `small2` are two small tables, and `big1` is a large table. At first, Hive generates regular Reduce Joins for Joins involving `small1` and `small2`. Then, Hive automatically converts these two Reduce Joins to Map Joins, which are shown as `M-JoinOp-1` and `M-JoinOp-2` in Figure 4(b). With the optimization introduced in this subsection, these two Map Joins are merged into the same Map phase and will be executed in a pipelined fashion. The results of these two Map Joins will be emitted to the shuffling phase for the Reduce Join `R-JoinOp-4`.

5.2 Correlation Optimizer

To eliminate unnecessary data loading and re-partitioning, we have introduced a Correlation Optimizer into Hive. This optimizer is based on the idea of correlation-aware query optimizations proposed in YSmart [31]. The main idea of this optimizer is to analyze if a major operation really needs to re-partition its input datasets and then to eliminate unnecessary MapReduce jobs through removing boundaries between a Map phase and a Reduce phase. The optimized plan will have less number of shuffling phases. Also, in the optimized plan, a table originally used in those MapReduce jobs are used in the same MapReduce job and Hive can automatically load the common table once instead of multiple times in the original plan.

In this section, we first introduce correlations exploited by this optimizer. Then, we introduce how we have implemented this optimizer. We will also cover new challenges we have overcome and how the design of the Correlation Optimizer is different from the original YSmart.

5.2.1 Correlations

In the implementation of the Correlation Optimizer, we consider two kinds of correlations in a query, input correlation and job flow correlation. An input correlation means that a table is used by multiple operations originally executed in different MapReduce jobs. A job flow correlation means that when a major operator depends on another major operator, these two major operators require their input datasets to be partitioned in the same way. Interested readers may refer to the original paper of YSmart [31] for details about correlations.

5.2.2 Implementation

During the development of the Correlation Optimizer, we found that the biggest challenge was not to identify optimization opportunities in a query, but to make the optimized plan executable. Except for scanning a table once for operations appearing in the same Map phase, we found that Hive did not have any support for multi-query optimizations. Moreover, the push-based model inside a Reduce task requires an intrinsic coordination mechanism between major operators executed in the same Reduce task to make those major operations work in the same pace. To make the optimized plan work, we needed to extend the query execution layer with the development of the optimizer.

Correlation Detection.

In Hive, every query has one or multiple terminal operators which are the last operators in the operator tree. Those terminal operators are called `FileSinkOperators` (`FSops`). To give an easy explanation, if an operator `A` is on another operator `B`'s path to a `FSOp`, `A` is the downstream of `B` and `B` is the upstream of `A`.

For a given operator tree like the one shown in Figure 4(b), the Correlation Optimizer starts to visit operators in the tree from those `FSOps` in a depth-first way. The tree walker stops at every `RSOp`. Then, a correlation detector starts to find a correlation from this `RSOp` and its siblings by finding the furthest correlated upstream `RSOps` in a recursive way. If we can find any correlated upstream `RSOp`, we find a correlation. Currently, there are three conditions to determine if an upstream `RSOp` and a downstream `RSOp` are correlated, which are (1) emitted rows from these two `RSOps` are

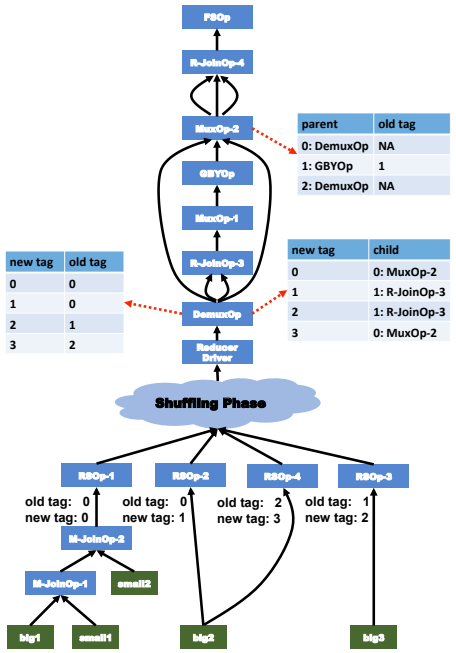


Figure 5: The optimized operator tree of the running example shown in Figure 4.

sorted in the same way; (2) emitted rows from these two RSOps are partitioned in the same way; and (3) these RSOps do not have any conflict on the number reducers. Interested readers may refer to our implementation [6] and design document [7] for details.

For the example shown in Figure 4(b), the Correlation Optimizer can find one correlation containing six RSOps, which are RSOp-1 to RSOp-6.

Operator Tree Transformation.

After the Correlation Optimizer finds all correlations in an operator tree, it starts to transform the operator tree.

There are two kinds of RSOps in a correlation. First, we have a list of bottom layer RSOps which are necessary ones used to emit rows to the shuffling phase. Originally, because the MapReduce engine considers the input data of the Reduce phase as a single data stream, each bottom layer RSOp was assigned a tag which is used to identify the source of a row at the Reduce phase. For a correlation, we reassign tags to those bottom layer RSOps and keep the mapping from new tags to old tags. For example, in Figure 5, RS1, RS2, RS3 and RS4 are bottom layer RSOps and their old tags are 0, 0, 1, and 2, respectively. After the transformation, their new tags are 0, 1, 2, and 3, respectively.

Second, we have a set of unnecessary RSOps which can be removed from the operator tree. Because the Reduce phase of the optimized plan can have multiple major operators consuming input rows from the shuffling phase, we added DemuxOperator (DemuxOp) in the beginning of the Reduce phase (after the Reducer Driver, which is the entry point of a Reduce phase) to reassign rows to their original tags and dispatch rows to different major operators based on new tags. For example, two mappings tables of the DemuxOp in Figure 5 show how DemuxOp reassigns tags and dispatches rows. Then, we remove all unnecessary RSOps in this correlation inside the Reduce phase. Finally, for every GroupByOperator (GBYOp) and JoinOp, we add a MuxOperator (MuxOp) as the single parent. For a GBYOp, its parent MuxOp is used to inform when this GBYOp should buffer input rows and emit output rows. For a parent MuxOp of a JoinOp, besides what a MuxOp does for a GBYOp, its also needs to assign original tags to rows passed to this JoinOp. For example, the MuxOp-2 shown in Figure 5 basically forwards rows from the DemuxOp and it needs to assign tags for rows from the GBYOp.

Operator Coordination.

Hive inherits the push-based data processing model in a Map and a Reduce task from the MapReduce engine. Because of this execution model, simply removing unnecessary RSOps does not make the plan executable. For example, an operator in a Reduce phase generated by the Correlation Optimizer can have two JoinOps and one is at the upstream of another one. By simply removing the downstream RSOp, this JoinOp will not know when to start buffer its input rows and when to generate output rows. To make the optimized plan work, an operator coordination mechanism is needed.

Currently, this coordination mechanism is implemented in the DemuxOp and MuxOp. When a new row is sent to the Reducer Driver, it checks if it needs to start a new group of rows by checking values of those key columns. If a new group of rows is coming, it first sends the signal of ending the existing row group to the DemuxOp. Then, the DemuxOp will propagate this signal to the operator tree. When a MuxOp gets this ending group signal, it will check if all of its parent operators have sent this signal to it. If so, it will ask its child to generate results and send this signal to its child. After the signal of ending group has been propagated through the entire operator tree in the Reduce phase, the Reducer Driver then will send a signal of starting a new row group to the DemuxOp. This signal will also be propagated through the entire operator tree. Finally, the Reducer Driver will forward the new row to DemuxOp. Interested readers may refer to our design document [7] for more details.

6. QUERY EXECUTION

To address the shortcoming of the runtime execution efficiency, we need to design a new execution model that takes advantage of characteristics of modern CPUs. The original Hive heavily relies on using lazy deserialization to reduce the amount of data being deserialized. The data is read as byte arrays and the column values are deserialized only when being accessed. Lazy deserialization does help in saving some deserialization costs, but introduces virtualized calls to deserialization routines in the inner loop of execution, which slows down the execution pipelines.

As demonstrated in [19], the effective execution speed of modern CPUs greatly depends on parallelism. Modern CPUs have multi-staged pipelines with large number of stages, and superscalar architectures contain more than one such pipelines. To fully utilize the parallelism in such a pipelined architecture, it is important to avoid unnecessary branches in the instructions. Also, the higher the data independence among the instructions being executed at different stages of a pipeline, the more parallelism can be achieved. Moreover, processing rows in a one-row-at-a-time model also causes poor cache performance [35] [50]. To address all of these issues, we have designed and implemented the vectorized execution model in Hive. In this section, we provide an overview of the vectorized query execution model in Hive. Interested readers can also refer to [8] [9] for details.

6.1 Vectorized Query Execution

In the vectorized execution model, a dataset is represented as batches of rows. In a row batch, data values of a column are represented as a column vector. The number of rows in the batch is configurable and should be chosen to fit the entire batch in the pro-

```
class VectorizedRowBatch {
    boolean selecetedInUse;
    int[] selected;
    int size;
    ColumnVector[] columns;
}
```

Figure 6: A snippet of a row batch.

```
class LongColumnVector extends ColumnVector {
   long[] vector
}
```

Figure 7: A column vector for `long` encapsulates an array of `long`s. `LongColumnVector` is used to represent all varieties of integers, boolean and timestamp data types..

```
class LongColumnAddLongScalarExpression {
  int inputColumn;
  int outputColumn;
  long scalar;
  void evaluate(VectorizedRowBatch batch) {
    long [] inVector = ((LongColumnVector)
      batch.columns[inputColumn]).vector;
    long [] outVector = ((LongColumnVector)
      batch.columns[outputColumn]).vector;
    if (batch.selectedInUse) {
      for (int j = 0; j < batch.size; j++) {
        int i = batch.selected[j];
        outVector[i] = inVector[i] + scalar;
      }
    } else {
      for (int i = 0; i < batch.size; i++) {
        outVector[i] = inVector[i] + scalar;
      }
    }
  }
}
```

Figure 8: Expression to add a long column with a constant. The array `selected[]` in the batch is used to keep track of valid rows without a branch instruction.

cessor cache. By default, this number is set to 1024, which was carefully chosen to minimize overhead and typically allows one row batch to fit in the processor cache. The query execution progresses by applying each expression on the entire column vector. Figure 6 shows a snippet of a row batch and Figure 7 shows a snippet of a column vector.

In row mode of execution (one-row-at-a-time), each row traverses the whole operator tree before the next row is processed. While, in vectorized execution, a whole row batch is processed through the operator tree. The expressions have been re-implemented as vectorized expressions that work directly on column vectors and produce output in column vectors.

Vectorized query execution doesn't support lazy deserialization and assumes that data reader will provide deserialized data. This will not introduce much overhead that lazy deserialization wanted to avoid because filters and projections are pushed down to the data reader, which will read only relevant data.

6.2 Vectorized Expressions

The biggest chunk of this work has been to implement vectorized expressions for each row mode expression. Vectorized expressions process column vectors in a tight loop and minimize branching and method calls from within the loop. Specialized expressions are implemented for each data type. For example, there are different expressions for adding two long columns or a long column plus a double column. Similarly, different expressions have been implemented that add a column to a constant. Figure 8 shows the snippet of a vector expression that adds a long column to a constant.

The inner loop in Figure 8 is a very tight loop with no branches or method calls. The iterations are completely independent of each other and can be parallelized in the superscalar instruction pipelines of modern CPUs. If the batch fits into the L1 cache, cache misses will be minimal. Additionally, there is a strong locality of reference in the instruction cache.

There are more optimizations that have been implemented into the expressions. The column vectors contain a no-null flag that is set, by the data reader, to true if it is known that the column doesn't contain any null in the batch. The expressions avoid a null check in the inner loop if the no-null flag is set. If this flag is not set, a null check is necessary. There is also an is-repeating flag to indicate if all values of the column are the same. The expressions further optimize for this case by doing computation in constant time if possible for the whole column vector, rather than time proportional to the vector size. This can allow the benefits of run-length encoding to extend to faster query execution.

Vectorized expressions for `AND`, `OR` and comparison expressions have two sets of implementations, one that produces a boolean output column, and other that does not produce an output but achieves in-place filtering by manipulating the `selected` array. The array of `selected` is populated with the index of the rows that are passed by the expression. Subsequent expressions only work on rows that are selected by the previous expressions.

6.3 Vectorized Expression Templates

Many vectorized expressions have similar code for different data types. These expressions are generated from pre-defined templates by variable substitution. For example, there is a template for all comparison operations among numeric types that generate a boolean output column. The expressions are generated at build time and compiled into the Hive jar. There is no dynamic code generation. Dynamic code generation can achieve more optimized code but that work has been left for the future.

6.4 Vectorization Optimizer

Vectorization has been added as a rule based optimization in the query planner. The planner first generates a non-vectorized plan and then vectorization optimization is invoked if configured. The vectorization optimization first validates the plan to ensure vectorization is applicable to the operators and expressions used in the plan. If validation succeeds, the optimizer iterates through all the operators and expressions, and replaces each expression tree with corresponding vectorized expressions, as applicable.

6.5 Vectorized Reader

Vectorized query execution works with any file format that can provide data input as vectorized row batch (Figure 6). The columnar file formats fit more naturally with vectorized query execution because column vectors in the row batch can be more efficiently read from the underlying data layout. The reader must deserialize the data and should also populate various flags (e.g. no-null flag) to enable optimizations as mentioned in Section 6.2.

7. PERFORMANCE EVALUATION

To show the effectiveness of those advancements introduced in Section 4, Section 5, and Section 6, we report results of our experiments in this section. Our experiments were designed to show benefits from each advancement separately. We will first introduce the setup of our experiments and then report results.

7.1 Setup

We conducted our experiments in a cluster with 11 nodes launched in Amazon EC2 . The instance type was `m1.xlarge` which has 4 cores, 15 GB memory, and 4 disks. The operating system used for these nodes was Ubuntu Server 12.04.3 LTS 64-bit. The Hadoop version was 1.2.1. The Hive version was 0.13-SNAPSHOT built on

Table 2: Sizes of datasets (GB) stored by Text, RCFile, RCFile with Snappy compression, ORC File, and ORC File with Snappy compression.

	SS-DB	TPC-H	TPC-DS
Text	248.35	323.84	279.87
RCFile	128.23	269.00	159.69
RCFile Snappy	55.15	118.33	105.28
ORC File	53.51	168.96	102.24
ORC File Snappy	39.20	86.67	94.05

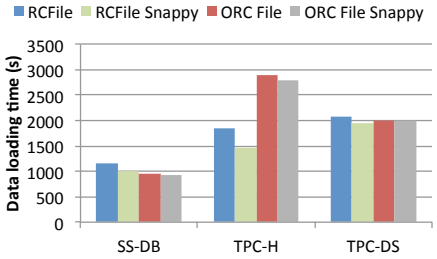


Figure 9: Elapsed times of loading datasets to file formats of RCFile, RCFile with Snappy compression, ORC File, and ORC File with Snappy compression.

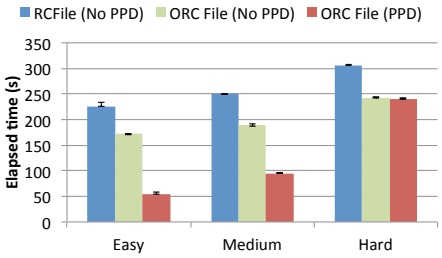
Nov. 28th, 2013 for experiments of file formats and query execution and Hive 0.14-SNAPSHOT built on Mar. 7th, 2014 for experiments of query planning. This Hadoop cluster had 1 master node running NameNode and JobTracker, and 10 slave nodes running DataNode and TaskTracker.

In our experiments, we used queries from three benchmarks, which are SS-DB [22], TPC-H [10], and TPC-DS [11]. SS-DB is a standard science DBMS benchmark and it is used to simulate applications that manipulate array-oriented data. We used a `large` scale factor to generate the SS-DB dataset. This dataset contains 1000 images which are divided to 50 cycles (20 images in each cycle). In our experiments, we only used one cycle of images. TPC-H and TPC-DS are two standard decision support benchmarks. We used the scale factor of 300 for both of them. In our experiments, all tables in TPC-H and TPC-DS were loaded into Hive. For every query used in our experiments, we tested it five times. For elapsed times, we will report medians, and 25% and 50% percentiles will be reported as error bars. To eliminate the impact from OS buffer caches, before every run of a query, we freed all cached objects, and then freed OS page cache, dentries and inodes.

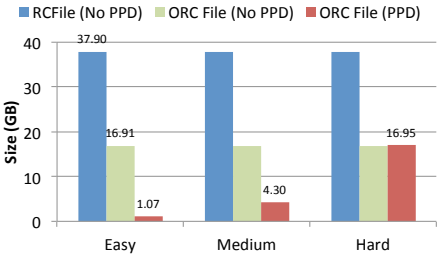
For Hadoop, we set that the Reduce phase starts after the entire Map phase has finished. For a slave node, it can run three concurrent Map tasks or Reduce tasks. The HDFS block size was set to 512 MB. Due to the page limit, we cannot introduce all configuration properties used in Hadoop and Hive, and show queries used in our experiments. Interested readers may refer to https://github.com/yhuai/hive-benchmarks for details.

7.2 File Format

In this section, we show the storage efficiency and data reading efficiency of ORC File. To show the storage efficiency, we compared the sizes of datasets of SS-DB, TPC-H and TPC-DS stored by RCFile and ORC. For each file format, we also stored datasets with and without using Snappy compression (referred to as RCFile Snappy and ORC File Snappy). Table 2 shows the sizes of those datasets stored with RCFile, RCFile Snappy, ORC File, and ORC File Snappy. The sizes of datasets stored in plain text are also provided in Table 2 (referred to as Text) as references. As we can seen,



(a) Elapsed times



(b) Amounts of data read from HDFS

Figure 10: Elapsed times and sizes of data read from HDFS of SS-DB query 1.easy, 1.medium, and 1.hard. `No PPD` means that predicates were not pushed down to the ORC reader level. While, `PPD` means that predicates were pushed down to the ORC reader level and ORC File used indexes to determine what data to read.

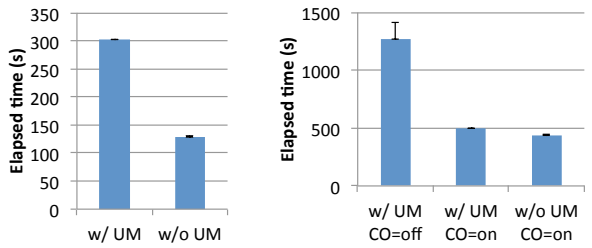
without Snappy compression, datasets of SS-DB and TPC-DS have already had smaller sizes than RCFile with Snappy, which shows the effectiveness of data type specific encoding schemes in ORC File. With Snappy compression, sizes of datasets stored with ORC File were further reduced. For datasets of SS-DB and TPC-DS, this further reduction on sizes is not as significant as that shown in the dataset of TPC-H. It is because every table in TPC-H has a column of `comment`, which contains random strings. The cardinality of such a column is high and the dictionary encoding scheme is not effective. Thus, the size of the dataset of TPC-H can be significantly reduced by using a general-purpose data compression technique, such as Snappy.

Besides sizes of datasets, we also report data loading times at here. Figure 9 shows the times taken to load the plain text datasets to RCFile, RCFile Snappy, ORC File, and ORC File Snappy. For datasets of SS-DB and TPC-DS, the time taken to load a dataset to ORC File was about the same with RCFile. However, the data loading time taken to store the TPC-H dataset in ORC File was around two times as long as that taken to store the dataset in RCFile. We believe that it was caused by those columns with random strings mentioned above. Because the dictionary encoding scheme was not effective for those columns, operations for the dictionary encoding scheme were basically useless work which contributed to the longer time on loading the TPC-H dataset.

To show the data reading efficiency of ORC File, we used three variations of SS-DB query 1. The query template is shown below.

```
SELECT SUM(v1), COUNT(*) FROM cycle
WHERE x BETWEEN 0 and var AND
      y BETWEEN 0 and var;
```

These variations are referred to as query 1.easy, query 1.medium, and query 1.hard. The value of `var` corresponds to these queries are 3750, 7500, and 15000, respectively. With the increase of the difficulty (e.g. from easy to medium), the number of rows satisfying predicates in the query increases, and all rows will satisfy



(a) TPC-DS query 27 (b) TPC-DS query 95

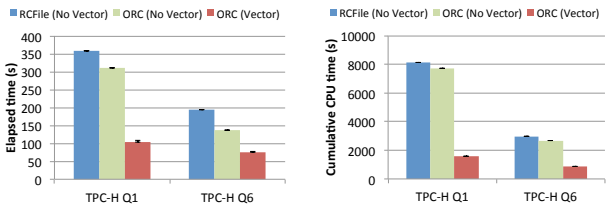
Figure 11: Elapsed times of TPC-DS query 27 and query 95. `w/ UM` means that the plan had all unnecessary map phases. `w/o UM` means that the plan did not have any unnecessary map phase. `CO=off` and `CO=on` means that the Correlation Optimizer was disabled and enabled, respectively.

predicates in query 1.hard. This three queries are suitable for evaluating the data reading efficiency of ORC File and the effectiveness of indexes provided by ORC File. Figure 10 shows elapsed times of these queries and the amounts of data read from HDFS. For ORC File, it can use its indexes to evaluate predicates at the ORC reader level. If there is no row in a stripe satisfying predicates, the ORC reader does not read this stripe. If an index group does not satisfy predicates, ORC File does not read rows in it from HDFS. From Figure 10, we have three observations. First, without using indexes, ORC File's large default stripe size (i.e. 256 MB) can provide better data reading efficiency which is shown by a less query elapsed time and a less amount of data read from HDFS than a smaller default stripe size provided in RCFile (i.e. 4 MB). This observation confirms the study in [28]. Second, with indexes, ORC File is able to significantly further reduce the amount of unnecessary data read from HDFS. For query 1.easy, with the help from indexes, the amount of data read from HDFS was 1.07 GB comparing to 16.91 GB data without using indexes. Third, the overhead of using indexes is low. For query 1.hard, because all rows satisfy predicates, indexes are useless for this query. In this case, with using indexes, ORC File read around 40 MB extra data (storing indexes) and introduced around 2 extra seconds than without using indexes.

7.3 Query Planning

To show the effectiveness of optimizations introduced in Section 5, we show performance of TPC-DS query 27 and query 95. The TPC-DS query 27 first has a five-table star join. Then, the result of this star join is aggregated and sorted. For the star join in this query, it involves a large fact table and four small dimension tables. Without the optimization introduced in Section 5.1, the plan of this query has four Map-only jobs and one MapReduce job. Each Map-only job corresponds to a join between the large table and a small dimension table. The last MapReduce job is used to generate the final result. This plan yields poor query evaluation performance because it has four unnecessary Map phases. After eliminating these unnecessary Map phases, the optimized plan has a single MapReduce job. Those Map Joins are executed in the Map phase. Figure 11(a) shows the performance of these two plans. The speedup of the optimized plan is around 2.34x.

The TPC-DS query 95 is a very complex query. Because the limitation of Hive on supporting sub-queries in the `WHERE` clause, we flatten sub-queries in this query for this experiment. This query can be optimized by eliminating unnecessary Map phases and exploiting correlations. Without these two optimizations, the plan of this query has three Map-only jobs and five MapReduce jobs. By exploiting correlations with the Correlation Optimizer, the plan has three Map-only jobs and two MapReduce jobs. By further eliminating unnecessary Map phases, the optimized plan has only two MapReduce jobs. Figure 11(b) shows the performance of these three plans. With the Correlation Optimizer, we can achieve a speedup of 2.57x. Then, after further eliminating unnecessary Map phases, we can achieve a combined speedup of 2.92x.



(a) Total elapsed times (b) Cumulative CPU times

Figure 12: Elapsed times and cumulative CPU times of TPC-H query 1 and TPC-H query 6. `No Vector` and `Vector` means that the vectorized query execution engine was disabled and enabled, respectively.

7.4 Query Execution

To show the effectiveness of the vectorized query execution engine in Hive, we conducted experiments with TPC-H query 1 and TPC-H query 6. These two queries use the largest table in the TPC-H benchmark, i.e. `lineitem`. The TPC-H query 1 has one predicate and eight aggregations. The TPC-H query 6 has four predicates and one aggregation. Both of these two queries were executed in a single MapReduce job. Figure 12(a) shows the elapsed times of these two queries with and without the vectorized query execution engine. The elapsed times based on RCFile with the original Hive execution engine are also reported at here as references. As we can seen, with the vectorized query execution engine, the elapsed times of these query were significantly reduced. We also report cumulative CPU times in Figure 12(b). For the TPC-H query 1, the cumulative CPU time of the original Hive execution engine is around five times as long as that with the vectorized execution engine. For the TPC-H query 6, the cumulative CPU time of the original Hive execution engine is around three times as long as that with the vectorized execution engine.

8. RELATED WORK

The advancements we introduced in this paper come from a combined result of years' operating experience on Hive, and the foundation of both traditional database research and recent research efforts on Hadoop-based data management systems. In this section, we summarize related work to file formats, query planning, and query execution.

8.1 File Formats

File formats (or storage models in traditional database research) are one of the most important and long-lasting topics. There were several research projects on this topic, e.g. [21] [18] [49]. As shown in [44] and [30], columnar file formats are most suitable for data warehouse workloads. In the ecosystem of Hadoop, there has been several work on file formats. RCFile [27] was the first widely used columnar file format. It has triggered several projects targeting on this topic, including [23] [24] [33] [29] [12].

To introduce columnar file formats into Hive, characteristics of Hive, HDFS and MapReduce need to be considered. First, every HDFS block of every file takes a certain space in the memory of a HDFS master node, which requires that a file format should occupy

as less space as possible in the HDFS master node. Thus, in the design of ORC File, we store all columns of a stripe in a single file instead of storing columns of a stripe to multiple files. This design also significantly simplifies the integration with Hive. Moreover, based on [28], it is not necessary to store a stripe to multiple files when the stripe size is large (e.g. 256 MB default stripe size used in ORC File). Second, the task scheduler in MapReduce tries to co-locate a task with its input data. Thus, ORC File stores metadata and indexes with their corresponding data. Third, because ad hoc queries are an important workload of Hive, the adaptivity of a columnar file format to different query patterns is critical. Thus, in ORC File, columns will not be grouped. Also, a recent study has shown that grouping columns has insignificant performance benefits on I/O when the stripe size is large enough [28].

Another related work is Parquet [13]. Based on the definition of table placement methods provided in [28], ORC File and Parquet share the same basic structure. They both first partition a table to multiple stripes [7] and then store columns in a stripe in a column-by-column fashion. The main difference between them is that they store values of a column in different ways. Parquet stores values of a column in a way based on the nested columnar storage introduced in Dremel [34].

8.2 Query Planning

Planning a query based on its semantics and data properties can be traced back to System R which cooperates ordering information of intermediate results in a query when choosing the join method [40]. There are several projects that aim to infer and exploit intermediate data properties for optimizing the plan of a query. Representatives are [42] on sorting operations; [26] on partitioning operations; [36] on both sorting and grouping operations; and [48] on partitioning, sorting and grouping operations. To increase effective disk bandwidth and reduce unnecessary operations, both data sharing and work sharing are also exploited in [41] [25] [51] [20].

In the ecosystem of Hadoop, there have been several recent research projects exploiting sharing opportunities and eliminating unnecessary data movements, e.g. [31] [37] [47] [32]. The Correlation Optimizer in Hive is a YSmart-based design. YSmart [31] looks at a single query. It exploits shared data scan opportunities and eliminates unnecessary data shuffling operations by merging multiple MapReduce jobs into a single one. The main difference between Correlation Optimizer and other related work (including YSmart) is that Correlation Optimizer is specifically designed for the push-based data processing model used by Hive. Existing work mainly focuses on generating optimized query plans. While, Correlation Optimizer also considers how to execute optimized plans under the push-based model.

8.3 Query Execution

Early work discussed processing a block of rows organized with the N-ary Storage Model in memory [38] and using a special buffer operator to create blocks without modifying regular operators to offer a better instruction cache locality [50]. The vectorized execution design for Hive was motivated by the work on MonetDB/X100 [19]. This work shows the inefficiency of the one-row-at-a-time execution model and shares their experience on building a vectorized query execution system. A recent paper proposed to dynamically generate code to further improve query processing performance [35]. Other recent work argues that for highest query execution performance, vectorization is essential, and it must be judiciously combined with compilation techniques [43].

[7] A stripe in Parquet is called a row group, which is the same term used in RCFile.

In the ecosystem of Hadoop, Apache Drill [14] and Impala [15] also attempt to improve runtime query execution efficiency. Drill uses a vectorized execution model [16] and Impala processes a row batch at a time instead of a single row at a time.

9. CONCLUSION

In this paper, we have presented major advancements in Hive. Specifically, we introduced (1) a highly efficient file format, ORC File; (2) an updated query planner that effectively reduces unnecessary data operations and movements by eliminating unnecessary Map phases and exploiting correlations in a query; and (3) a new vectorized query execution engine that significantly improves the performance of query execution through better utilizing modern CPUs. The performance and resource utilization efficiency of the updated Hive have been demonstrated by our experiments. We have also received positive feedbacks from the Hive-based data processing community. For example, the storage efficiency in Facebook's Hive production system has been significantly improved after adopting ORC File [39].

These major technical advancements come from strong collaborative efforts from both research and development communities. Some academic research projects have directly influenced the new developments of Hive with strong technical and analytical basis. On the other hand, the engineering efforts in systems implementation have addressed several challenges in order to make Hive gain substantial benefits in practice.

Recently, several new features have been added into Hive or under development. Interested readers may refer to [17] for details. From the perspective of functionalities, we have been working on expending Hive's SQL capabilities, such as advanced analytic functions, new data types, extending the sub-queries support, common table expressions, and improved support for join syntax. Hive has introduced limited form of support for transactions. Language level support for transaction is under development and is expected to be released later this year. Also, Hive now supports SQL standard based authorization model. HiveServer2 has been enhanced to support different protocols and Kerberos authentication. From the perspective of performance, we have been integrating Hive with Apache Tez [3] for its support on more general query execution plans and better performance. Except eliminating unnecessary Map phases (it is a MapReduce-specific issue), all of advancements introduced in this paper are still applicable and important to Hive running on Tez. Hive has introduced cost based optimizer. Currently, its used to do join ordering. Work is in progress to use cost based optimizer for wider range of queries.

10. ACKNOWLEDGMENTS

We would like to thank the Hive development community. In recent years, more than 100 developers have made technical efforts to improve Hive on more than 3000 issues. We thank those individuals who have operated, tested, supported and documented Hive in various platforms and applications. We are grateful to the Hive user community who has provided us with numerous valuable feedbacks that drive the development and advancements of Hive. We thank anonymous reviewers for their constructive comments. This work has been partially supported by the National Science Foundation under grants CCF-0913050, OCI-1147522, and CNS-1162165.

11. REFERENCES

[1] https://hadoop.apache.org/.

[2] https://cwiki.apache.org/confluence/display/Hive/Setting+up+HiveServer2.

[3] https://tez.incubator.apache.org/.

[4] https://hbase.apache.org/.

[5] https://svn.apache.org/viewvc/hive/trunk/ql/src/java/org/apache/hadoop/hive/ql/io/orc/WriterImpl.java?view=log.

[6] https://svn.apache.org/viewvc/hive/trunk/ql/src/java/org/apache/hadoop/hive/ql/optimizer/correlation/CorrelationOptimizer.java?view=log.

[7] https://cwiki.apache.org/confluence/display/Hive/Correlation+Optimizer.

[8] https://issues.apache.org/jira/browse/HIVE-4160.

[9] https://issues.apache.org/jira/secure/attachment/12603710/Hive-Vectorized-Query-Execution-Design-rev11.pdf.

[10] http://www.tpc.org/tpch/.

[11] http://www.tpc.org/tpcds/.

[12] http://avro.apache.org/docs/current/trevni/spec.html.

[13] https://github.com/Parquet/parquet-format.

[14] https://incubator.apache.org/drill/.

[15] https://github.com/cloudera/impala.

[16] http://www.slideshare.net/ApacheDrill/oscon-2013-apache-drill-workshop-part-2.

[17] https://hive.apache.org/.

[18] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis. Weaving Relations for Cache Performance. In *VLDB*, 2001.

[19] P. A. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *CIDR*, 2005.

[20] Y. Cao, G. C. Das, C.-Y. Chan, and K.-L. Tan. Optimizing Complex Queries with Multiple Relation Instances. In *SIGMOD*, 2008.

[21] G. P. Copeland and S. N. Khoshafian. A Decomposition Storage Model. In *SIGMOD*, 1985.

[22] P. Cudre-Mauroux, H. Kimura, K.-T. Lim, J. Rogers, S. Madden, M. Stonebraker, S. B. Zdonik, and P. G. Brown. SS-DB: A Standard Science DBMS Benchmark. http://www-conf.slac.stanford.edu/xldb10/docs/ssdb_benchmark.pdf.

[23] A. Floratou, J. M. Patel, E. J. Shekita, and S. Tata. Column-Oriented Storage Techniques for MapReduce. In *VLDB*, 2011.

[24] S. Guo, J. Xiong, W. Wang, and R. Lee. Mastiff: A Mapreduce-based System for Time-Based Big Data Analytics. In *CLUSTER*, 2012.

[25] S. Harizopoulos, V. Shkapenyuk, and A. Ailamaki. QPipe: A Simultaneously Pipelined Relational Query Engine. In *SIGMOD*, 2005.

[26] W. Hasan and R. Motwani. Coloring Away Communication in Parallel Query Optimization. In *VLDB*, 1995.

[27] Y. He, R. Lee, Y. Huai, Z. Shao, N. Jain, X. Zhang, and Z. Xu. RCFile: A Fast and Space-efficient Data Placement Structure in MapReduce-based Warehouse Systems. In *ICDE*, 2011.

[28] Y. Huai, S. Ma, R. Lee, O. O'Malley, and X. Zhang. Understanding Insights into the Basic Structure and Essential Issues of Table Placement Methods in Clusters. In *VLDB*, 2013.

[29] A. Jindal, J.-A. Quiané-Ruiz, and J. Dittrich. Trojan Data Layouts: Right Shoes for a Running Elephant. In *SOCC*, 2011.

[30] A. Lamb, M. Fuller, R. Varadarajan, N. Tran, B. Vandier, L. Doshi, and C. Bear. The Vertica Analytic Database: C-Store 7 Years Later. In *VLDB*, 2012.

[31] R. Lee, T. Luo, Y. Huai, F. Wang, Y. He, and X. Zhang. YSmart: Yet Another SQL-to-MapReduce Translator. In *ICDCS*, 2011.

[32] H. Lim, H. Herodotou, and S. Babu. Stubby: A Transformation-based Optimizer for Mapreduce Workflows. In *VLDB*, 2012.

[33] Y. Lin, D. Agrawal, C. Chen, B. C. Ooi, and S. Wu. Llama: Leveraging Columnar Storage for Scalable Join Processing in the Mapreduce Framework. In *SIGMOD*, 2011.

[34] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: Interactive Analysis of Web-Scale Datasets. In *VLDB*, 2010.

[35] T. Neumann. Efficiently Compiling Efficient Query Plans for Modern Hardware. In *VLDB*, 2011.

[36] T. Neumann and G. Moerkotte. A Combined Framework for Grouping and Order Optimization. In *VLDB*, 2004.

[37] T. Nykiel, M. Potamias, C. Mishra, G. Kollios, and N. Koudas. MRShare: Sharing Across Multiple Queries in Mapreduce. In *VLDB*, 2010.

[38] S. Padmanabhan, T. Malkemus, R. C. Agarwal, and A. Jhingran. Block Oriented Processing of Relational Database Operations in Modern Computer Architectures. In *ICDE*, 2001.

[39] J. Parikh. Data Infrastructure at Web Scale. http://www.vldb.org/2013/video/keynote1.flv.

[40] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access Path Selection in a Relational Database Management System. In *SIGMOD*, 1979.

[41] T. K. Sellis. Multiple-Query Optimization. *ACM Trans. Database Syst.*, 13(1):23–52, Mar. 1988.

[42] D. Simmen, E. Shekita, and T. Malkemus. Fundamental Techniques for Order Optimization. In *SIGMOD*, 1996.

[43] J. Sompolski, M. Zukowski, and P. A. Boncz. Vectorization vs. Compilation in Query Execution. In *DaMoN*, 2011.

[44] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik. C-Store: A Column-oriented DBMS. In *VLDB*, 2005.

[45] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Anthony, H. Liu, and R. Murthy. Hive - A Petabyte Scale Data Warehouse Using Hadoop. In *ICDE*, 2010.

[46] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler. Apache Hadoop YARN: Yet Another Resource Negotiator. In *SoCC*, 2013.

[47] X. Wang, C. Olston, A. D. Sarma, and R. Burns. CoScan: Cooperative Scan Sharing in the Cloud. In *SoCC*, 2011.

[48] J. Zhou, P.-Å. Larson, and R. Chaiken. Incorporating Partitioning and Parallel Plans into the SCOPE Optimizer. In *ICDE*, 2010.

[49] J. Zhou and K. A. Ross. A Multi-resolution Block Storage Model for Database Design. In *IDEAS*, 2003.

[50] J. Zhou and K. A. Ross. Buffering Database Operations for Enhanced Instruction Cache Performance. In *SIGMOD*, 2004.

[51] M. Zukowski, S. Héman, N. Nes, and P. Boncz. Cooperative Scans: Dynamic Bandwidth Sharing in a DBMS. In *VLDB*, 2007.