Hive - A Warehousing Solution Over a Map-Reduce Framework

Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao,
Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff and Raghotham Murthy

Facebook Data Infrastructure Team

1. INTRODUCTION

The size of data sets being collected and analyzed in the industry for business intelligence is growing rapidly, making traditional warehousing solutions prohibitively expensive. *Hadoop* [3] is a popular open-source map-reduce implementation which is being used as an alternative to store and process extremely large data sets on commodity hardware. However, the map-reduce programming model is very low level and requires developers to write custom programs which are hard to maintain and reuse.

In this paper, we present *Hive*, an open-source data warehousing solution built on top of Hadoop. Hive supports queries expressed in a SQL-like declarative language - *HiveQL*, which are compiled into map-reduce jobs executed on Hadoop. In addition, HiveQL supports custom map-reduce scripts to be plugged into queries. The language includes a type system with support for tables containing primitive types, collections like arrays and maps, and nested compositions of the same. The underlying IO libraries can be extended to query data in custom formats. Hive also includes a system catalog, *Hive-Metastore*, containing schemas and statistics, which is useful in data exploration and query optimization. In Facebook, the Hive warehouse contains several thousand tables with over 700 terabytes of data and is being used extensively for both reporting and ad-hoc analyses by more than 100 users.

The rest of the paper is organized as follows. Section 2 describes the Hive data model and the HiveQL language with an example. Section 3 describes the Hive system architecture and an overview of the query life cycle. Section 4 provides a walk-through of the demonstration. We conclude with future work in Section 5.

2. HIVE DATABASE

2.1 Data Model

Data in Hive is organized into:

- Tables - These are analogous to tables in relational databases. Each table has a corresponding HDFS directory. The data in a table is serialized and stored in files within that directory. Users can associate tables with the serialization format of the underlying data. Hive provides builtin serialization formats which exploit compression and lazy de-serialization. Users can also add support for new data formats by defining custom serialize and de-serialize methods (called *SerDe*'s) written in Java. The serialization format of each table is stored in the system catalog and is automatically used by Hive during query compilation and execution. Hive also supports external tables on data stored in HDFS, NFS or local directories.
- Partitions - Each table can have one or more partitions which determine the distribution of data within sub-directories of the table directory. Suppose data for table `T` is in the directory `/wh/T`. If `T` is partitioned on columns `ds` and `ctry`, then data with a particular `ds` value `20090101` and `ctry` value `US`, will be stored in files within the directory `/wh/T/ds=20090101/ctry=US`.
- Buckets - Data in each partition may in turn be divided into *buckets* based on the hash of a column in the table. Each bucket is stored as a file in the partition directory.

Hive supports primitive column types (integers, floating point numbers, generic strings, dates and booleans) and *nestable* collection types — *array* and *map*. Users can also define their own types programmatically.

2.2 Query Language

Hive provides a SQL-like query language called HiveQL which supports select, project, join, aggregate, union all and sub-queries in the from clause. HiveQL supports data definition (DDL) statements to create tables with specific serialization formats, and partitioning and bucketing columns. Users can load data from external sources and insert query results into Hive tables via the **load** and **insert** data manipulation (DML) statements respectively. HiveQL currently does not support updating and deleting rows in existing tables.

HiveQL supports multi-table insert, where users can perform multiple queries on the same input data using a single HiveQL statement. Hive optimizes these queries by sharing the scan of the input data.

HiveQL is also very extensible. It supports user defined column transformation (UDF) and aggregation (UDAF) functions implemented in Java. In addition, users can embed custom map-reduce scripts written in any language using a simple row-based streaming interface, i.e., read rows from

VLDB '09, August 24-28, 2009, Lyon, France

standard input and write out rows to standard output. This flexibility does come at a cost of converting rows from and to strings.

We omit more details due to lack of space. For a complete description of HiveQL see the language manual [5].

2.3 Running Example: StatusMeme

We now present a highly simplified application, StatusMeme, inspired by Facebook Lexicon [6]. When Facebook users update their status, the updates are logged into flat files in an NFS directory `/logs/status_updates` which are rotated every day. We load this data into hive on a daily basis into a table

`status_updates(userid int,status string,ds string)`

using a load statement like below.

```
LOAD DATA LOCAL INPATH '/logs/status_updates'
INTO TABLE status_updates PARTITION (ds='2009-03-20')
```

Each status update record contains the user identifier (`userid`), the actual status string (`status`), and the date (`ds`) when the status update occurred. This table is partitioned on the `ds` column. Detailed user profile information, like the gender of the user and the school the user is attending, is available in the `profiles(userid int,school string,gender int)` table.

We first want to compute daily statistics on the frequency of status updates based on gender and school which the user attends. The following multi-table insert statement generates the daily counts of status updates by school (into `school_summary(school string,cnt int,ds string)`) and gender (into `gender_summary(gender int,cnt int,ds string)`) using a single scan of the join of the `status_updates` and `profiles` tables. Note that the output tables are also partitioned on the `ds` column, and HiveQL allows users to insert query results into a specific partition of the output table.

```
FROM (SELECT a.status, b.school, b.gender
      FROM status_updates a JOIN profiles b
           ON (a.userid = b.userid and
               a.ds='2009-03-20' )
      ) subq1
INSERT OVERWRITE TABLE gender_summary
                       PARTITION(ds='2009-03-20')
SELECT subq1.gender, COUNT(1) GROUP BY subq1.gender
INSERT OVERWRITE TABLE school_summary
                       PARTITION(ds='2009-03-20')
SELECT subq1.school, COUNT(1) GROUP BY subq1.school
```

Next, we want to display the ten most popular memes per school as determined by status updates by users who attend that school. We now show how this computation can be done using HiveQLs map-reduce constructs. We parse the result of the join between `status_updates` and `profiles` tables by plugging in a custom Python mapper script `meme-extractor.py` which uses sophisticated natural language processing techniques to extract memes from status strings. Since Hive does not yet support the `rank` aggregation function the top 10 memes per school can then be computed by a simple custom Python reduce script `top10.py`

```
REDUCE subq2.school, subq2.meme, subq2.cnt
       USING 'top10.py' AS (school,meme,cnt)
FROM (SELECT subq1.school, subq1.meme, COUNT(1) AS cnt
      FROM (MAP b.school, a.status
                USING 'meme-extractor.py' AS (school,meme)
            FROM status_updates a JOIN profiles b
                 ON (a.userid = b.userid)
            ) subq1
      GROUP BY subq1.school, subq1.meme
      DISTRIBUTE BY school, meme
      SORT BY school, meme, cnt desc
) subq2;
```

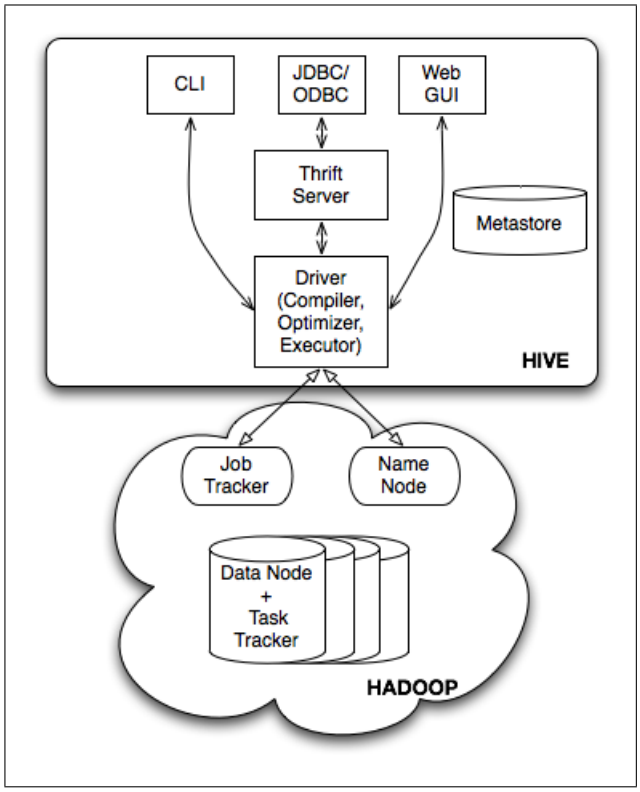

Figure 1: Hive Architecture

3. HIVE ARCHITECTURE

Figure 1 shows the major components of Hive and its interactions with Hadoop. The main components of Hive are:

- **External Interfaces** - Hive provides both user interfaces like command line (CLI) and web UI, and application programming interfaces (API) like JDBC and ODBC.
- The Hive **Thrift Server** exposes a very simple client API to execute HiveQL statements. Thrift [8] is a framework for cross-language services, where a server written in one language (like Java) can also support clients in other languages. The Thrift Hive clients generated in different languages are used to build common drivers like JDBC (java), ODBC (C++), and scripting drivers written in php, perl, python etc.
- The **Metastore** is the system catalog. All other components of Hive interact with the metastore. For more details see Section 3.1.
- The **Driver** manages the life cycle of a HiveQL statement during compilation, optimization and execution. On receiving the HiveQL statement, from the thrift server or other interfaces, it creates a session handle which is later used to keep track of statistics like exe-

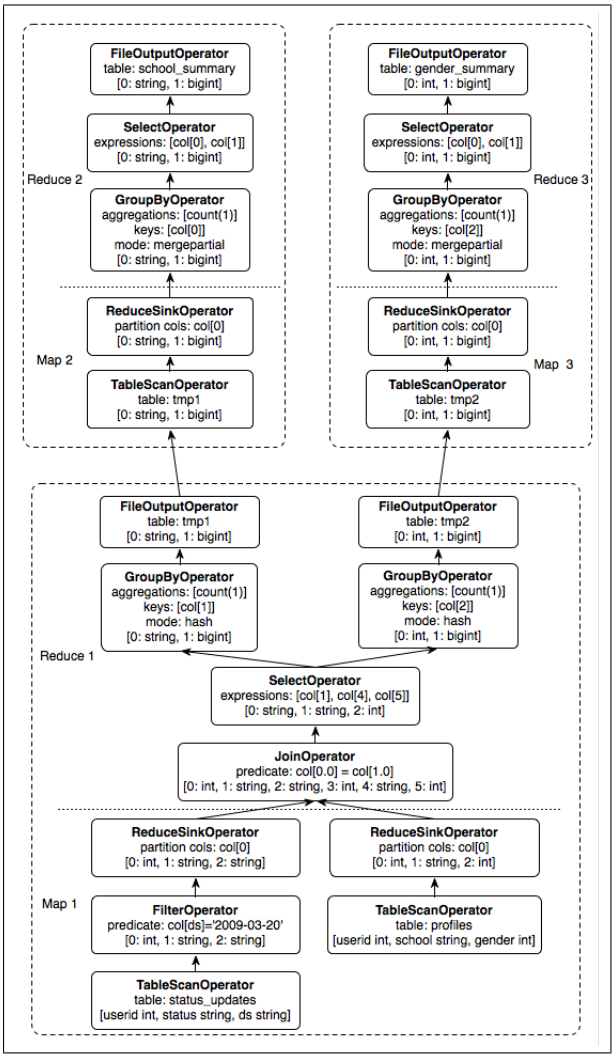

Figure 2: Query plan with 3 map-reduce jobs for multi-table insert query

cution time, number of output rows, etc.

- The **Compiler** is invoked by the driver upon receiving a HiveQL statement. The compiler translates this statement into a plan which consists of a DAG of map-reduce jobs. For more details see Section 3.2
- The driver submits the individual map-reduce jobs from the DAG to the **Execution Engine** in a topological order. Hive currently uses Hadoop as its execution engine.

We next describe the metastore and compiler in detail.

3.1 Metastore

The metastore is the system catalog which contains metadata about the tables stored in Hive. This metadata is specified during table creation and reused every time the table is referenced in HiveQL. The metastore distinguishes Hive as a traditional warehousing solution (ala Oracle or DB2) when compared with similar data processing systems built on top of map-reduce like architectures like Pig [7] and Scope [2].

The metastore contains the following objects:

- Database - is a namespace for tables. The database 'default' is used for tables with no user supplied database name.
- Table - Metadata for table contains list of columns and their types, owner, storage and SerDe information. It can also contain any user supplied key and value data; this facility can be used to store table statistics in the future. Storage information includes location of the table's data in the underlying file system, data formats and bucketing information. SerDe metadata includes the implementation class of serializer and deserializer methods and any supporting information required by that implementation. All this information can be provided during the creation of table.
- Partition - Each partition can have its own columns and SerDe and storage information. This can be used in the future to support schema evolution in a Hive warehouse.

The storage system for the metastore should be optimized for online transactions with random accesses and updates. A file system like HDFS is not suited since it is optimized for sequential scans and not for random access. So, the metastore uses either a traditional relational database (like MySQL, Oracle) or file system (like local, NFS, AFS) and not HDFS. As a result, HiveQL statements which only access metadata objects are executed with very low latency. However, Hive has to explicitly maintain consistency between metadata and data.

3.2 Compiler

The driver invokes the compiler with the HiveQL string which can be one of DDL, DML or query statements. The compiler converts the string to a plan. The plan consists only of metadata operations in case of DDL statements, and HDFS operations in case of `LOAD` statements. For insert statements and queries, the plan consists of a directed-acyclic graph (DAG) of map-reduce jobs.

- The Parser transforms a query string to a parse tree representation.
- The Semantic Analyzer transforms the parse tree to a block-based internal query representation. It retrieves schema information of the input tables from the metastore. Using this information it verifies column names, expands `select *` and does type-checking including addition of implicit type conversions.
- The Logical Plan Generator converts the internal query representation to a logical plan, which consists of a tree of logical operators.
- The Optimizer performs multiple passes over the logical plan and rewrites it in several ways:
  - Combines multiple joins which share the join key into a single multi-way join, and hence a single map-reduce job.
  - Adds repartition operators (also known as ReduceSinkOperator) for join, group-by and custom map-reduce operators. These repartition operators mark the boundary between the map phase and a reduce phase during physical plan generation.
  - Prunes columns early and pushes predicates closer to the table scan operators in order to minimize

the amount of data transfered between operators.

- In case of partitioned tables, prunes partitions that are not needed by the query
- In case of sampling queries, prunes buckets that are not needed

Users can also provide hints to the optimizer to

- add partial aggregation operators to handle large cardinality grouped aggregations
- add repartition operators to handle skew in grouped aggregations
- perform joins in the map phase instead of the reduce phase

- The Physical Plan Generator converts the logical plan into a physical plan, consisting of a DAG of map-reduce jobs. It creates a new map-reduce job for each of the marker operators – repartition and union all – in the logical plan. It then assigns portions of the logical plan enclosed between the markers to mappers and reducers of the map-reduce jobs.

In Figure 2, we illustrate the plan of the multi-table insert query in Section 2.3. The nodes in the plan are physical operators and the edges represent the flow of data between operators. The last line in each node represents the output schema of that operator. For lack of space, we do not describe the parameters specified within each operator node. The plan has three map-reduce jobs. Within the same map-reduce job, the portion of the operator tree below the repartition operator (ReduceSinkOperator) is executed by the mapper and the portion above by the reducer. The repartitioning itself is performed by the execution engine.

Notice that the first map-reduce job writes to two temporary files to HDFS, `tmp1` and `tmp2`, which are consumed by the second and third map-reduce jobs respectively. Thus, the second and third map-reduce jobs wait for the first map-reduce job to finish.

4. DEMONSTRATION DESCRIPTION

The demonstration consists of the following:

- Functionality — We demonstrate HiveQL constructs via the *StatusMeme* application described in Section 2.3. We expand the application to include queries which use more HiveQL constructs and showcase the rule-based optimizer.
- Tuning — We also demonstrate our query plan viewer which shows how HiveQL queries are translated into physical plans of map-reduce jobs. We show how hints can be used to modify the plans generated by the optimizer.
- User Interface — We show our graphical user interface which allows users to explore a Hive database, author HiveQL queries, and monitor query execution.
- Scalability — We illustrate the scalability of the system by increasing the sizes of the input data and the complexity of the queries.

5. FUTURE WORK

Hive is a first step in building an open-source warehouse over a web-scale map-reduce data processing system (Hadoop). The distinct characteristics of the underlying storage and execution engines has forced us to revisit techniques for query processing. We have discovered that we have to either modify or rewrite several query processing algorithms to perform efficiently in our setting.

Hive is an Apache sub-project, with an active user and developer community both within and outside Facebook. The Hive warehouse instance in Facebook contains over 700 terabytes of usable data and supports over 5000 queries on a daily basis. This demonstration show cases the current capabilities of Hive. There are many important avenues of future work:

- HiveQL currently accepts only a subset of SQL as valid queries. We are working towards making HiveQL subsume SQL syntax.
- Hive currently has a naïve rule-based optimizer with a small number of simple rules. We plan to build a cost-based optimizer and adaptive optimization techniques to come up with more efficient plans.
- We are exploring columnar storage and more intelligent data placement to improve scan performance.
- We are running performance benchmarks based on [1] to measure our progress as well as compare against other systems [4]. In our preliminary experiments, we have been able to improve the performance of Hadoop itself by 20% compared to [1]. The improvements involved using faster Hadoop data structures to process the data, for example, using `Text` instead of `String`. The same queries expressed easily in HiveQL had 20% overhead compared to our optimized hadoop implementation, i.e., Hive's performance is on par with the hadoop code from [1]. Based on these experiments, we have identified several areas for performance improvement and have begun working on them. More details are available in [4].
- We are enhancing the JDBC and ODBC drivers for Hive for integration with commercial BI tools which only work with traditional relational warehouses.
- We are exploring methods for multi-query optimization techniques and performing generic n-way joins in a single map-reduce job.

We would like to thank our user and developer community for their contributions, with special thanks to Yuntao Jia, Yongqiang He, Dhruba Borthakur and Jeff Hammerbacher.

6. REFERENCES

[1] A. Pavlo et. al. A Comparison of Approaches to Large-Scale Data Analysis. *Proc. ACM SIGMOD*, 2009.
[2] C.Ronnie et al. SCOPE: Easy and Efficient Parallel Processing of Massive Data Sets. *Proc. VLDB Endow.*, 1(2):1265–1276, 2008.
[3] Apache Hadoop. Available at `http://wiki.apache.org/hadoop`.
[4] Hive Performance Benchmark. Available at `https://issues.apache.org/jira/browse/HIVE-396`.
[5] Hive Language Manual. Available at `http://wiki.apache.org/hadoop/Hive/LanguageManual`.
[6] Facebook Lexicon. Available at `http://www.facebook.com/lexicon`.
[7] Apache Pig. `http://wiki.apache.org/pig`.
[8] Apache Thrift. `http://incubator.apache.org/thrift`.