Hive – A Petabyte Scale Data Warehouse Using Hadoop

Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Ning Zhang, Suresh Antony, Hao Liu and Raghotham Murthy

Facebook Data Infrastructure Team

Abstract— **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 [1] is a popular open-source map-reduce implementation which is being used in companies like Yahoo, Facebook etc. 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 that are executed using Hadoop. In addition, HiveQL enables users to plug in custom map-reduce scripts 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 - *Metastore* – that contains schemas and statistics, which are useful in data exploration, query optimization and query compilation. In Facebook, the Hive warehouse contains tens of thousands of tables and stores over 700TB of data and is being used extensively for both reporting and ad-hoc analyses by more than 200 users per month.**

I. Introduction

Scalable analysis on large data sets has been core to the functions of a number of teams at Facebook - both engineering and non-engineering. Apart from ad hoc analysis and business intelligence applications used by analysts across the company, a number of Facebook products are also based on analytics. These products range from simple reporting applications like Insights for the Facebook Ad Network, to more advanced kind such as Facebook's Lexicon product [2]. As a result a flexible infrastructure that caters to the needs of these diverse applications and users and that also scales up in a cost effective manner with the ever increasing amounts of data being generated on Facebook, is critical. Hive and Hadoop are the technologies that we have used to address these requirements at Facebook.

The entire data processing infrastructure in Facebook prior to 2008 was built around a data warehouse built using a commercial RDBMS. The data that we were generating was growing very fast - as an example we grew from a 15TB data set in 2007 to a 700TB data set today. The infrastructure at that time was so inadequate that some daily data processing jobs were taking more than a day to process and the situation was just getting worse with every passing day. We had an urgent need for infrastructure that could scale along with our data. As a result we started exploring Hadoop as a technology to address our scaling needs. The fact that Hadoop was already an open source project that was being used at petabyte scale and provided scalability using commodity hardware was a very compelling proposition for us. The same jobs that had taken more than a day to complete could now be completed within a few hours using Hadoop.

However, using Hadoop was not easy for end users, especially for those users who were not familiar with map-reduce. End users had to write map-reduce programs for simple tasks like getting raw counts or averages. Hadoop lacked the expressiveness of popular query languages like SQL and as a result users ended up spending hours (if not days) to write programs for even simple analysis. It was very clear to us that in order to really empower the company to analyze this data more productively, we had to improve the query capabilities of Hadoop. Bringing this data closer to users is what inspired us to build Hive in January 2007. Our vision was to bring the familiar concepts of tables, columns, partitions and a subset of SQL to the unstructured world of Hadoop, while still maintaining the extensibility and flexibility that Hadoop enjoyed. Hive was open sourced in August 2008 and since then has been used and explored by a number of Hadoop users for their data processing needs.

Right from the start, Hive was very popular with all users within Facebook. Today, we regularly run thousands of jobs on the Hadoop/Hive cluster with hundreds of users for a wide variety of applications starting from simple summarization jobs to business intelligence, machine learning applications and to also support Facebook product features.

In the following sections, we provide more details about Hive architecture and capabilities. Section II describes the data model, the type systems and the HiveQL. Section III details how data in Hive tables is stored in the underlying distributed file system – HDFS(Hadoop file system). Section IV describes the system architecture and various components of Hive . In Section V we highlight the usage statistics of Hive at Facebook and provide related work in Section VI. We conclude with future work in Section VII.

II. Data Model, Type System And Query Language

Hive structures data into the well-understood database concepts like tables, columns, rows, and partitions. It supports all the major primitive types – integers, floats, doubles and strings – as well as complex types such as maps, lists and structs. The latter can be nested arbitrarily to construct more complex types. In addition, Hive allows users to extend the

978-1-4244-5446-4/10/$26.00 © 2010 IEEE

system with their own types and functions. The query language is very similar to SQL and therefore can be easily understood by anyone familiar with SQL. There are some nuances in the data model, type system and HiveQL that are different from traditional databases and that have been motivated by the experiences gained at Facebook. We will highlight these and other details in this section.

A. Data Model and Type System

Similar to traditional databases, Hive stores data in tables, where each table consists of a number of rows, and each row consists of a specified number of columns. Each column has an associated type. The type is either a primitive type or a complex type. Currently, the following primitive types are supported:

- Integers – bigint(8 bytes), int(4 bytes), smallint(2 bytes), tinyint(1 byte). All integer types are signed.
- Floating point numbers – float(single precision), double(double precision)
- String

Hive also natively supports the following complex types:

- Associative arrays – map<key-type, value-type>
- Lists – list<element-type>
- Structs – struct<file-name: field-type, ... >

These complex types are templated and can be composed to generate types of arbitrary complexity. For example, list<map<string, struct<p1:int, p2:int>> represents a list of associative arrays that map strings to structs that in turn contain two integer fields named p1 and p2. These can all be put together in a create table statement to create tables with the desired schema. For example, the following statement creates a table t1 with a complex schema.

```
CREATE TABLE t1(st string, fl float, li list<map<string,
struct<p1:int, p2:int>>);
```

Query expressions can access fields within the structs using a '.' operator. Values in the associative arrays and lists can be accessed using '[]' operator. In the previous example, t1.li[0] gives the first element of the list and t1.li[0]['key'] gives the struct associated with 'key' in that associative array. Finally the p2 field of this struct can be accessed by t1.li[0]['key'].p2. With these constructs Hive is able to support structures of arbitrary complexity.

The tables created in the manner describe above are serialized and deserialized using default serializers and deserializers already present in Hive. However, there are instances where the data for a table is prepared by some other programs or may even be legacy data. Hive provides the flexibility to incorporate that data into a table without having to transform the data, which can save substantial amount of time for large data sets. As we will describe in the later sections, this can be achieved by providing a jar that implements the SerDe java interface to Hive. In such situations the type information can also be provided by that jar by providing a corresponding implementation of the ObjectInspector java interface and exposing that implementation through the getObjectInspector method present in the SerDe interface. More details on these interfaces can be found on the Hive wiki [3], but the basic takeaway here is that any arbitrary data format and types encoded therein can be plugged into Hive by providing a jar that contains the implementations for the SerDe and ObjectInspector interfaces. All the native SerDes and complex types supported in Hive are also implementations of these interfaces. As a result once the proper associations have been made between the table and the jar, the query layer treats these on par with the native types and formats. As an example, the following statement adds a jar containing the SerDe and ObjectInspector interfaces to the distributed cache([4]) so that it is available to Hadoop and then proceeds to create the table with the custom serde.

```
add jar /jars/myformat.jar;
CREATE TABLE t2
ROW FORMAT SERDE 'com.myformat.MySerDe';
```

Note that, if possible, the table schema could also be provided by composing the complex and primitive types.

B. Query Language

The Hive query language(HiveQL) comprises of a subset of SQL and some extensions that we have found useful in our environment. Traditional SQL features like from clause sub-queries, various types of joins – inner, left outer, right outer and outer joins, cartesian products, group bys and aggregations, union all, create table as select and many useful functions on primitive and complex types make the language very SQL like. In fact for many of the constructs mentioned before it is exactly like SQL. This enables anyone familiar with SQL to start a hive cli(command line interface) and begin querying the system right away. Useful metadata browsing capabilities like show tables and describe are also present and so are explain plan capabilities to inspect query plans (though the plans look very different from what you would see in a traditional RDBMS). There are some limitations e.g. only equality predicates are supported in a join predicate and the joins have to be specified using the ANSI join syntax such as

```
SELECT t1.a1 as c1, t2.b1 as c2
FROM t1 JOIN t2 ON (t1.a2 = t2.b2);
```

instead of the more traditional

```
SELECT t1.a1 as c1, t2.b1 as c2
FROM t1, t2
WHERE t1.a2 = t2.b2;
```

Another limitation is in how inserts are done. Hive currently does not support inserting into an existing table or data partition and all inserts overwrite the existing data. Accordingly, we make this explicit in our syntax as follows:

```
INSERT OVERWRITE TABLE t1
SELECT * FROM t2;
```

In reality these restrictions have not been a problem. We have rarely seen a case where the query cannot be expressed as an equi-join and since most of the data is loaded into our warehouse daily or hourly, we simply load the data into a new partition of the table for that day or hour. However, we do realize that with more frequent loads the number of partitions can become very large and that may require us to implement INSERT INTO semantics. The lack of INSERT INTO, UPDATE and DELETE in Hive on the other hand do allow us to use very simple mechanisms to deal with reader and writer concurrency without implementing complex locking protocols.

Apart from these restrictions, HiveQL has extensions to support analysis expressed as map-reduce programs by users and in the programming language of their choice. This enables advanced users to express complex logic in terms of map-reduce programs that are plugged into HiveQL queries seamlessly. Some times this may be the only reasonable approach e.g. in the case where there are libraries in python or php or any other language that the user wants to use for data transformation. The canonical word count example on a table of documents can, for example, be expressed using map-reduce in the following manner:

```
FROM (
  MAP doctext USING 'python wc_mapper.py' AS (word, cnt)
  FROM docs
  CLUSTER BY word
) a
REDUCE word, cnt USING 'python wc_reduce.py';
```

As shown in this example the MAP clause indicates how the input columns (doctext in this case) can be transformed using a user program (in this case 'python wc_mapper.py') into output columns (word and cnt). The CLUSTER BY clause in the sub-query specifies the output columns that are hashed on to distributed the data to the reducers and finally the REDUCE clause specifies the user program to invoke (python wc_reduce.py in this case) on the output columns of the sub-query. Sometimes, the distribution criteria between the mappers and the reducers needs to provide data to the reducers such that it is sorted on a set of columns that are different from the ones that are used to do the distribution. An example could be the case where all the actions in a session need to be ordered by time. Hive provides the DISTRIBUTE BY and SORT BY clauses to accomplish this as shown in the following example:

```
FROM (
  FROM session_table
  SELECT sessionid, tstamp, data
  DISTRIBUTE BY sessionid SORT BY tstamp
) a
REDUCE sessionid, tstamp, data USING 'session_reducer.sh';
```

Note, in the example above there is no map clause which indicates that the input columns are not transformed. Similarly, it is possible to have a MAP clause without a REDUCE clause in case the reduce phase does not do any transformation of data. Also in the examples shown above, the FROM clause appears before the SELECT clause which is another deviation from standard SQL syntax. Hive allows users to interchange the order of the FROM and SELECT/MAP/REDUCE clauses within a given sub-query. This becomes particularly useful and intuitive when dealing with multi inserts. HiveQL supports inserting different transformation results into different tables, partitions, hdfs or local directories as part of the same query. This ability helps in reducing the number of scans done on the input data as shown in the following example:

```
FROM t1
  INSERT OVERWRITE TABLE t2
  SELECT t3.c2, count(1)
  FROM t3
  WHERE t3.c1 <= 20
  GROUP BY t3.c2

  INSERT OVERWRITE DIRECTORY '/output_dir'
  SELECT t3.c2, avg(t3.c1)
  FROM t3
  WHERE t3.c1 > 20 AND t3.c1 <= 30
  GROUP BY t3.c2

  INSERT OVERWRITE LOCAL DIRECTORY '/home/dir'
  SELECT t3.c2, sum(t3.c1)
  FROM t3
  WHERE t3.c1 > 30
  GROUP BY t3.c2;
```

In this example different portions of table t1 are aggregated and used to generate a table t2, an hdfs directory(/output_dir) and a local directory(/home/dir on the user's machine).

III. Data Storage, SerDe And File Formats

A. *Data Storage*

While the tables are logical data units in Hive, table metadata associates the data in a table to hdfs directories. The primary data units and their mappings in the hdfs name space are as follows:

- Tables – A table is stored in a directory in hdfs.
- Partitions – A partition of the table is stored in a sub-directory within a table's directory.
- Buckets – A bucket is stored in a file within the partition's or table's directory depending on whether the table is a partitioned table or not.

As an example a table test_table gets mapped to <warehouse_root_directory>/test_table in hdfs. The warehouse_root_directory is specified by the hive.metastore.warehouse.dir configuration parameter in hive-site.xml. By default this parameter's value is set to /user/hive/warehouse.

A table may be partitioned or non-partitioned. A partitioned table can be created by specifying the PARTITIONED BY clause in the CREATE TABLE statement as shown below.

```
CREATE TABLE test_part(c1 string, c2 int)
PARTITIONED BY (ds string, hr int);
```

In the example shown above the table partitions will be stored in /user/hive/warehouse/test_part directory in hdfs. A partition exists for every distinct value of ds and hr specified by the user. Note that the partitioning columns are not part of the table data and the partition column values are encoded in the directory path of that partition (they are also stored in the table metadata). A new partition can be created through an INSERT statement or through an ALTER statement that adds a partition to the table. Both the following statements

```
INSERT OVERWRITE TABLE
  test_part PARTITION(ds='2009-01-01', hr=12)
SELECT * FROM t;
```

```
ALTER TABLE test_part
  ADD PARTITION(ds='2009-02-02', hr=11);
```

add a new partition to the table test_part. The INSERT statement also populates the partition with data from table t, where as the alter table creates an empty partition. Both these statements end up creating the corresponding directories - /user/hive/warehouse/test_part/ds=2009-01-01/hr=12 and /user/hive/warehouse/test_part/ds=2009-02-02/hr=11 – in the table's hdfs directory. This approach does create some complications in case the partition value contains characters such as / or : that are used by hdfs to denote directory structure, but proper escaping of those characters does take care of a producing an hdfs compatible directory name.

The Hive compiler is able to use this information to prune the directories that need to be scanned for data in order to evaluate a query. In case of the test_part table, the query

```
SELECT * FROM test_part WHERE ds='2009-01-01';
```

will only scan all the files within the /user/hive/warehouse/test_part/ds=2009-01-01 directory and the query

```
SELECT * FROM test_part
WHERE ds='2009-02-02' AND hr=11;
```

will only scan all the files within the /user/hive/warehouse/test_part/ds=2009-01-01/hr=12 directory. Pruning the data has a significant impact on the time it takes to process the query. In many respects this partitioning scheme is similar to what has been referred to as list partitioning by many database vendors ([6]), but there are differences in that the values of the partition keys are stored with the metadata instead of the data.

The final storage unit concept that Hive uses is the concept of Buckets. A bucket is a file within the leaf level directory of a table or a partition. At the time the table is created, the user can specify the number of buckets needed and the column on which to bucket the data. In the current implementation this information is used to prune the data in case the user runs the query on a sample of data e.g. a table that is bucketed into 32 buckets can quickly generate a 1/32 sample by choosing to look at the first bucket of data. Similarly, the statement

```
SELECT * FROM t TABLESAMPLE(2 OUT OF 32);
```

would scan the data present in the second bucket. Note that the onus of ensuring that the bucket files are properly created and named are a responsibility of the application and HiveQL DDL statements do not currently try to bucket the data in a way that it becomes compatible to the table properties. Consequently, the bucketing information should be used with caution.

Though the data corresponding to a table always resides in the <warehouse_root_directory>/test_table location in hdfs, Hive also enables users to query data stored in other locations in hdfs. This can be achieved through the EXTERNAL TABLE clause as shown in the following example.

```
CREATE EXTERNAL TABLE test_extern(c1 string, c2 int)
  LOCATION '/user/mytables/mydata';
```

With this statement, the user is able to specify that test_extern is an external table with each row comprising of two columns – c1 and c2. In addition the data files are stored in the location /user/mytables/mydata in hdfs. Note that as no custom SerDe has been defined it is assumed that the data is in Hive's internal format. An external table differs from a normal table in only that a drop table command on an external table only drops the table metadata and does not delete any data. A drop on a normal table on the other hand drops the data associated with the table as well.

B. Serialization/Deserialization (SerDe)

As mentioned previously Hive can take an implementation of the SerDe java interface provided by the user and associate it to a table or partition. As a result custom data formats can easily be interpreted and queried from. The default SerDe implementation in Hive is called the LazySerDe – it deserializes rows into internal objects lazily so that the cost of deserialization of a column is incurred only if the column of the row is needed in some query expression. The LazySerDe assumes that the data is stored in the file such that the rows are delimited by a newline (ascii code 13) and the columns within a row are delimited by ctrl-A (ascii code 1). This SerDe can also be used to read data that uses any other delimiter character between columns.

As an example, the statement

```
CREATE TABLE test_delimited(c1 string, c2 int)
  ROW FORMAT DELIMITED
     FIELDS TERMINATED BY '\002'
     LINES TERMINATED BY '\012';
```

specifies that the data for table test_delimited uses ctrl-B (ascii code 2) as a column delimiter and uses ctrl-L(ascii code 12) as a row delimiter. In addition, delimiters can be specified to delimit the serialized keys and values of maps and different delimiters can also be specified to delimit the various elements of a list (collection). This is illustrated by the following statement.

```
CREATE TABLE test_delimited2(c1 string,
                                                  c2 list<map<string, int>>)
  ROW FORMAT DELIMITED
    FIELDS TERMINATED BY '\002'
    COLLECTION ITEMS TERMINATED BY '\003'
    MAP KEYS TERMINATED BY '\004';
```

Apart from LazySerDe, some other interesting SerDes are present in the hive_contrib.jar that is provided with the distribution. A particularly useful one is RegexSerDe which enables the user to specify a regular expression to parse various columns out from a row. The following statement can be used for example, to interpret apache logs.

```
add jar 'hive_contrib.jar';
CREATE TABLE apachelog(
    host string,
    identity string,
    user string,
    time string,
    request string,
    status string,
    size string,
    referer string,
    agent string)
  ROW FORMAT SERDE
     'org.apache.hadoop.hive.contrib.serde2.RegexSerDe'
  WITH SERDEPROPERTIES(
   'input.regex' = '([^ ]*) ([^ ]*) ([^ ]*) (-|\\[[^\\]]*\\]) ([^
\"]*|\"[^\"]*\") (-|[0-9]*) (-|[0-9]*)(?: ([^ \"]*|\"[^\"]*\") ([^
\"]*|\"[^\"]*\"))?',
  'output.format.string' = '%1$s %2$s %3$s %4$s %5$s %6$s
%7$s %8$s %9$s');
```

The input.regex property is the regular expression applied on each record and the output.format.string indicates how the column fields can be constructed from the group matches in the regular expression. This example also illustrates how arbitrary key value pairs can be passed to a serde using the WITH SERDEPROPERTIES clause, a capability that can be very useful in order to pass arbitrary parameters to a custom SerDe.

C. *File Formats*

Hadoop files can be stored in different formats. A file format in Hadoop specifies how records are stored in a file. Text files for example are stored in the TextInputFormat and binary files can be stored as SequenceFileInputFormat. Users can also implement their own file formats. Hive does not impose an restrictions on the type of file input formats, that the data is stored in. The format can be specified when the table is created. Apart from the two formats mentioned above, Hive also provides an RCFileInputFormat which stores the data in a column oriented manner. Such an organization can give important performance improvements specially for queries that do not access all the columns of the table. Users can add their own file formats and associate them to a table as shown in the following statement.

```
CREATE TABLE dest1(key INT, value STRING)
  STORED AS
     INPUTFORMAT
          'org.apache.hadoop.mapred.SequenceFileInputFormat'
     OUTPUTFORMAT
       'org.apache.hadoop.mapred.SequenceFileOutputFormat'
```

The STORED AS clause specifies the classes to be used to determine the input and output formats of the files in the table's or partition's directory. This can be any class that implements the FileInputFormat and FileOutputFormat java interfaces. The classes can be provded to Hadoop in a jar in ways similar to those shown in the examples on adding custom SerDes.

IV. System Architecture And Components

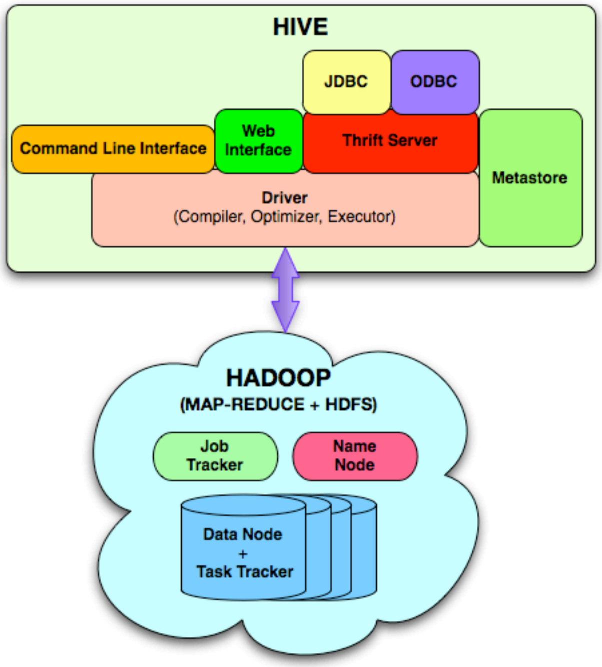

Fig. 1: Hive System Architecture

The following components are the main building blocks in Hive:

- Metastore – The component that stores the system catalog and metadata about tables, columns, partitions etc.
- Driver – The component that manages the lifecycle of a HiveQL statement as it moves through Hive. The driver also maintains a session handle and any session statistics.
- Query Compiler – The component that compiles HiveQL into a directed acyclic graph of map/reduce tasks.
- Execution Engine – The component that executes the tasks produced by the compiler in proper dependency order. The execution engine interacts with the underlying Hadoop instance.
- HiveServer – The component that provides a thrift interface and a JDBC/ODBC server and provides a way of integrating Hive with other applications.
- Clients components like the Command Line Interface (CLI), the web UI and JDBC/ODBC driver.
- Extensibility Interfaces which include the SerDe and ObjectInspector interfaces already described previously as well as the UDF(User Defined Function) and UDAF(User Defined Aggregate Function) interfaces that enable users to define their own custom functions.

A HiveQL statement is submitted via the CLI, the web UI or an external client using the thrift, odbc or jdbc interfaces. The driver first passes the query to the compiler where it goes through the typical parse, type check and semantic analysis phases, using the metadata stored in the Metastore. The compiler generates a logical plan that is then optimized through a simple rule based optimizer. Finally an optimized plan in the form of a DAG of map-reduce tasks and hdfs tasks is generated. The execution engine then executes these tasks in the order of their dependencies, using Hadoop.

In this section we provide more details on the Metastore, the Query Compiler and the Execution Engine.

A. *Metastore*

The Metastore acts as the system catalog for Hive. It stores all the information about the tables, their partitions, the schemas, the columns and their types, the table locations etc. This information can be queried or modified using a thrift ([7]) interface and as a result it can be called from clients in different programming languages. As this information needs to be served fast to the compiler, we have chosen to store this information on a traditional RDBMS. The Metastore thus becomes an application that runs on an RDBMS and uses an open source ORM layer called DataNucleus ([8]), to convert object representations into a relational schema and vice versa. We chose this approach as opposed to storing this information in hdfs as we need the Metastore to be very low latency. The DataNucleus layer allows us to plugin many different RDBMS technologies. In our deployment at Facebook, we use mysql to store this information.

Metastore is very critical for Hive. Without the system catalog it is not possible to impose a structure on hadoop files. As a result it is important that the information stored in the Metastore is backed up regularly. Ideally a replicated server should also be deployed in order to provide the availability that many production environments need. It is also important to ensure that this server is able to scale with the number of queries submitted by the users. Hive addresses that by ensuring that no Metastore calls are made from the mappers or the reducers of a job. Any metadata that is needed by the mapper or the reducer is passed through xml plan files that are generated by the compiler and that contain any information that is needed at the run time.

The ORM logic in the Metastore can be deployed in client libraries such that it runs on the client side and issues direct calls to an RDBMS. This deployment is easy to get started with and ideal if the only clients that interact with Hive are the CLI or the web UI. However, as soon as Hive metadata needs to get manipulated and queried by programs in languages like python, php etc., i.e. by clients not written in Java, a separate Metastore server has to be deployed.

B. *Query Compiler*

The metadata stored in the Metastore is used by the query compiler to generate the execution plan. Similar to compilers in traditional databases, the Hive compiler processes HiveQL statements in the following steps:

- Parse – Hive uses Antlr to generate the abstract syntax tree (AST) for the query.
- Type checking and Semantic Analysis – During this phase, the compiler fetches the information of all the input and output tables from the Metastore and uses that information to build a logical plan. It checks type compatibilities in expressions and flags any compile time semantic errors at this stage. The transformation of an AST to an operator DAG goes through an intermediate representation that is called the query block (QB) tree. The compiler converts nested queries into parent child relationships in a QB tree. At the same time, the QB tree representation also helps in organizing the relevant parts of the AST tree in a form that is more amenable to be transformed into an operator DAG than the vanilla AST.
- Optimization – The optimization logic consists of a chain of transformations such that the operator DAG resulting from one transformation is passed as input to the next transformation. Anyone wishing to change the compiler or wishing to add new optimization logic can easily do that by implementing the transformation as an extension of the Transform interface and adding it to the chain of transformations in the optimizer.

  The transformation logic typically comprises of a walk on the operator DAG such that certain processing actions are taken on the operator DAG when relevant conditions or rules are satisfied. The five primary interfaces that are involved in a transformation are Node, GraphWalker, Dispatcher, Rule and Processor. The nodes in the operator DAG implement the Node interface. This enables the operator DAG to be manipulated using the other interfaces mentioned above. A typical transformation involves walking the DAG and for every Node visited, checking if a Rule is satisfied and then invoking the corresponding Processor for that Rule in case the later is satisfied. The Dispatcher maintains the mappings from Rules to Processors and does the Rule matching. It is passed to the GraphWalker so that the appropriate Processor can be dispatched while a Node is being visited in the walk. The flowchart in Fig. 2 shows how a typical transformation is structured.

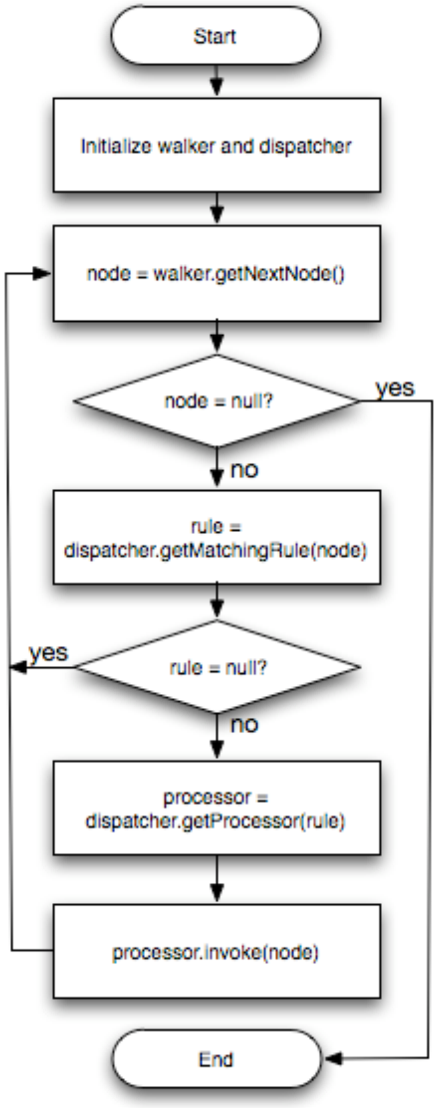
Start
Initialize walker and dispatcher
node = walker.getNextNode()
node = null?
yes
no
rule = dispatcher.getMatchingRule(node)
rule = null?
yes
no
processor = dispatcher.getProcessor(rule)
processor.invoke(node)
End

Fig. 2: Flowchart for typical transformation during optimization

The following transformations are done currently in Hive as part of the optimization stage:

i. Column pruning – This optimization step ensures that only the columns that are needed in the query processing are actually projected out of the row.
ii. Predicate pushdown – Predicates are pushed down to the scan if possible so that rows can be filter early in the processing.
iii. Partition pruning – Predicates on partitioned columns are used to prune out files of partitions that do not satisfy the predicate.
iv. Map side joins – In the cases where some of the tables in a join are very small, the small tables are replicated in all the mappers and joined with other tables. This behavior is triggered by a hint in the query of the form:

```
SELECT /*+ MAPJOIN(t2) */ t1.c1, t2.c1
FROM t1 JOIN t2 ON(t1.c2 = t2.c2);
```

A number of parameters control the amount of memory that is used on the mapper to hold the contents of the replicated table. These are hive.mapjoin.size.key and hive.mapjoin.cache.numrows that control the number of rows of the table that are kept in memory at any time and also provide the system the size of the join key.
v. Join reordering – The larger tables are streamed and not materialized in memory in the reducer while the smaller tables are kept in memory. This ensures that the join operation does not exceed memory limits on the reducer side.

In addition to the MAPJOIN hint, the user can also provide hints or set parameters to do the following:

i. Repartitioning of data to handle skews in GROUP BY processing – Many real world datasets have a power law distribution on columns used in the GROUP BY clause of common queries. In such situations the usual plan of distributing the data on the group by columns and then aggregating in the reducer does not work well as most of the data gets sent to a very few reducers. A better plan in such situations is to use two map/reduce stages to compute the aggregation. In the first stage the data is randomly distributed (or distributed on the DISTINCT column in case of distinct aggregations) to the reducers and the partial aggregations are computed. These partial aggregates are then distributed on the GROUP BY columns to the reducers in the second map/reduce stage. Since the number of the partial aggregation tuples is much smaller than the base data set, this approach typically leads to better performance. In Hive this behavior can be triggered by setting a parameter in the following manner:

```
set hive.groupby.skewindata=true;
SELECT t1.c1, sum(t1.c2)
FROM t1
GROUP BY t1;
```

ii. Hash based partial aggregations in the mappers – Hash based partial aggregations can potentially reduce the data that is sent by the mappers to the reducers. This in turn reduces the amount of time spent in sorting and merging this data. As a result a lot of performance gains can be achieved using this strategy. Hive enables users to control the amount of memory that can be used on the mapper to hold the rows in a hash table for this optimization. The parameter hive.map.aggr.hash.percentmemory specifies the fraction of mapper memory that can be used to hold the hash table, e.g. 0.5 would ensure that as soon as the hash table size exceeds half of the maximum memory for a mapper, the partial aggregates stored therein are sent to the reducers. The parameter hive.map.aggr.hash.min.reduction is also used to control the amount of memory used in the mappers.

- Generation of the physical plan – The logical plan generated at the end of the optimization phase is then split into multiple map/reduce and hdfs tasks. As an example a group by on skewed data can generate two map/reduce tasks followed by a final hdfs task which moves the results to the correct location in hdfs. At the end of this stage the physical plan looks like a DAG of tasks with each task encapsulating a part of the plan.

We show a sample multi-table insert query and its corresponding physical plan after all optimizations below.

```
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
```

This query has a single join followed by two different aggregations. By writing the query as a multi-table-insert, we make sure that the join is performed only once. The plan for the query is shown in Fig 3 below.

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.

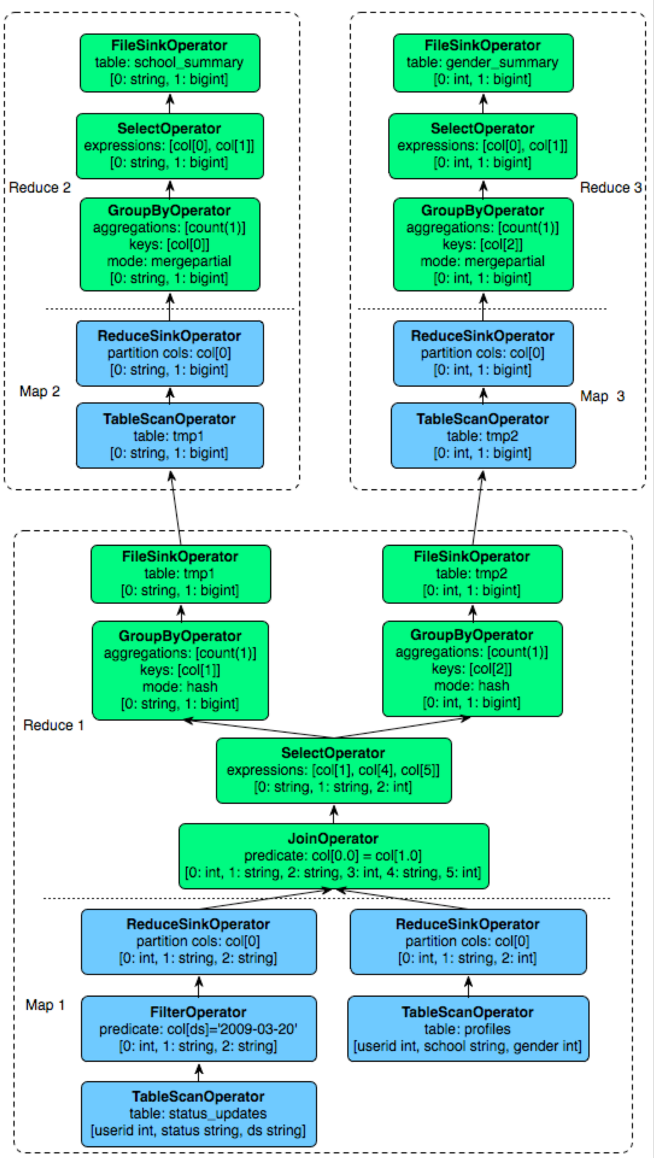

Fig. 3: Query plan for multi-table insert query with 3 map/reduce jobs

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.

C. Execution Engine

Finally the tasks are executed in the order of their dependencies. Each dependent task is only executed if all of its prerequisites have been executed. A map/reduce task first serializes its part of the plan into a plan.xml file. This file is then added to the job cache for the task and instances of ExecMapper and ExecReducers are spawned using Hadoop. Each of these classes deserializes the plan.xml and executes the relevant part of the operator DAG. The final results are stored in a temporary location. At the end of the entire query, the final data is moved to the desired location in case of DMLs. In the case of queries the data is served as such from the temporary location.

V. Hive Usage In Facebook

Hive and Hadoop are used extensively in Facebook for different kinds of data processing. Currently our warehouse has 700TB of data(which comes to 2.1PB of raw space on Hadoop after accounting for the 3 way replication). We add 5TB(15TB after replication) of compressed data daily. Typical compression ratio is 1:7 and sometime more than that. On any particular day more than 7500 jobs are submitted to the cluster and more than 75TB of compressed data is processed every day. With the continuous growth in the Facebook network we see continuous growth in data. At the same time as the company scales, the cluster also has to scale with the growing users.

More than half the workload is on adhoc queries where as the rest is for reporting dashboards. Hive has enabled this kind of workload on the Hadoop cluster in Facebook because of the simplicity with which adhoc analysis can be done. However, sharing the same resources by the adhoc users and reporting users presents significant operational challenges because of the unpredictability of adhoc jobs. Many times these jobs are not properly tuned and therefore consume valuable cluster resources. This can in turn lead to degraded performance of the reporting queries, many of which are time critical. Resource scheduling has been somewhat weak in Hadoop and the only viable solution at present seems to be maintaining separate clusters for adhoc queries and reporting queries.

There is also a wide variety in the Hive jobs that are run daily. They range from simple summarization jobs generating different kinds of rollups and cubes to more advanced machine learning algorithms. The system is used by novice users as well as advanced users with new users being able to use the system immediately or after an hour long beginners training.

A result of heavy usage has also lead to a lot of tables generated in the warehouse and this has in turn tremendously increased the need for data discovery tools, especially for new users. In general the system has enabled us to provide data processing services to engineers and analysts at a fraction of the cost of a more traditional warehousing infrastructure. Added to that the ability of Hadoop to scale to thousands of commodity nodes gives us the confidence that we will be able to scale this infrastructure going forward as well.

VI. Related Work

There has been a lot of recent work on petabyte scale data processing systems, both open-source and commercial. Scope[14] is an SQL-like language on top of Microsoft's proprietary Cosmos map/reduce and distributed file system. Pig[13] allows users to write declarative scripts to process data. Hive is different from these systems since it provides a system catalog that persists metadata about tables within the system. This allows hive to function as a traditional warehouse which can interface with standard reporting tools like MicroStrategy[16]. HadoopDB[15] reuses most of Hive's system, except, it uses traditional database instances in each of the nodes to store data instead of using a distributed file system.

VII. Conclusions And Future Work

Hive is a work in progress. It is an open-source project, and is being actively worked on by Facebook as well as several external contributors.

HiveQL currently accepts only a subset of SQL as valid queries. We are working towards making HiveQL subsume SQL syntax. Hive currently has a naive 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 [9] to measure our progress as well as compare against other systems. In our preliminary experiments, we have been able to improve the performance of Hadoop itself by ~20% compared to [9]. 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 [9]. We have also run the industry standard decision support benchmark – TPC-H [11]. Based on these experiments, we have identified several areas for performance improvement and have begun working on them. More details are available in [10] and [12]. We are enhancing the JDBC and ODBC drivers for Hive for integration with commercial BI tools that 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.

Acknowledgment

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

REFERENCES

[1] Apache Hadoop. Available at http://wiki.apache.org/hadoop.

[2] Facebook Lexicon at http://www.facebook.com/lexicon.

[3] Hive wiki at http://www.apache.org/hadoop/hive.

[4] Hadoop Map-Reduce Tutorial at http://hadoop.apache.org/common/docs/current/mapred_tutorial.html.

[5] Hadoop HDFS User Guide at http://hadoop.apache.org/common/docs/current/hdfs_user_guide.html.

[6] Mysql list partitioning at http://dev.mysql.com/doc/refman/5.1/en/partitioning-list.html.

[7] Apache Thrift. Available at http://incubator.apache.org/thrift.

[8] DataNucleus .Available at http://www.datanucleus.org.

[9] A. Pavlo et. al. A Comparison of Approaches to Large-Scale Data Analysis. In Proc. of ACM SIGMOD, 2009.

[10] Hive Performance Benchmark. Available at http://issues.apache.org/jira/browse/HIVE-396

[11] TPC-H Benchmark. Available at http://www.tpc.org/tpch

[12] Running TPC-H queries on Hive. Available at http://issues.apache.org/jira/browse/HIVE-600

[13] Hadoop Pig. Available at http://hadoop.apache.org/pig

[14] R. Chaiken, et. al. Scope: Easy and Efficient Parallel Processing of Massive Data Sets. In Proc. of VLDB, 2008.

[15] HadoopDB Project. Available at http://db.cs.yale.edu/hadoopdb/hadoopdb.html

[16] MicroStrategy. Available at http://www.microstrategy.com