Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Automation and Applied Informatics

Analyze and optimize the performance of Hive

Bachelor's Thesis

Author
Barnabas Maidics

Advisor
Peter Vary
Akos Dudas, PhD

November 10, 2018

Contents

HALLGATÓI NYILATKOZAT

Alulírott *Maidics Barnabas*, szigorló hallgató kijelentem, hogy ezt a szakdolgozatot meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy autentikált felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Budapest, 2018. november 10.

Maidics Barnabas
hallgató

1 Introduction

The digital era has led to large amounts of data being amassed by companies every day. Data comes from multiple sources: sensors, sales data, communication systems, logging of system events etc.. According to Forbes [9] 2.5 quintillion bytes of data is created each day. That means 2.5 million Terabytes per day. Bigger corporations can easily create hundreds of Terabytes daily. We need a new solution to process this amount of data. The traditional relational databases (RDBMS) can deal only with Gigabytes. Hadoop provides a software framework to scale up our system for storing, processing and analyzing big data.

In this chapter, I will write about the basics of Hadoop architecture, why Hive was created on top of it and the performance issues it faces.

1.1 Hadoop basics

Apache Hadoop is an open source distributed framework for managing, processing and storing a huge amount of data in clustered systems built from commodity hardware. All modules in Hadoop were designed with an assumption that hardware failures are frequent and should be automatically handled by the framework. One of the most important characteristics of Hadoop is that it partitions the data and computation across many hosts and executes computation in parallel close to the data it uses. [16]

The base of the Hadoop framework contains the following modules:

- HDFS - Hadoop Distributed File System: designed to store large data sets reliably and stream those at high bandwidth to user applications.
- Hadoop MapReduce: an implementation of the MapReduce programming model for large data processing

- YARN - Yet Another Resource Negotiator: a resource management and job scheduling technology
- Hadoop Common: contains libraries and utilities for other Hadoop modules

1.1.1 HDFS - Hadoop Distributed File System

HDFS is the file system of Hadoop. It stores file system metadata and application data separately. The dedicated server that stores metadata is the NameNode. Application data is stored on other servers (DataNodes). These servers are connected and they communicate using TCP-based protocols [13].

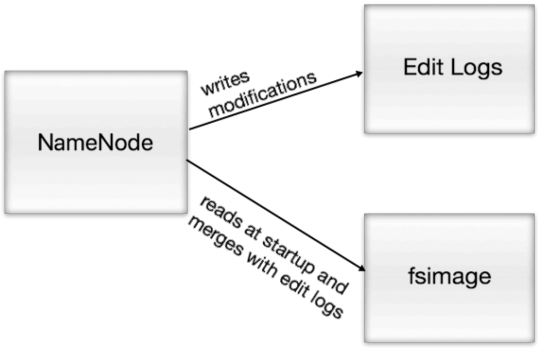
The file system is based on the following goals and principles [1]:

- **Hardware failure**: Hardware failures should be considered as normal, rather than an exception. An HDFS instance consists of hundreds or thousands of components so this means that some of them will always be non-functional. Therefore, fault detection and automatic recovery is a must.
- **Streaming Data Access**: HDFS was designed for batch processing rather than interactive use. Therefore, HDFS users need streaming access to their data. This means that high throughput is more important than low latency.
- **Large Data Sets**: The size of a typical HDFS file is gigabytes to terabytes. Thus, the file system is tuned to support large files.
- **Simple Coherency**: HDFS follows WORM (Write-Once-Read-Many) model. A file, once written should not be changed except for appends and truncates. This assumption simplifies data coherency issues. A MapReduce application fits perfectly for this model.
- **Moving computation**: A computation is much more efficient if it is executed near the data it operates on. It is especially true for big data.
- **Portability**: HDFS was designed to port from one platform to another with ease.

NameNode

NameNode keeps the directory tree of all files in the file system and tracks where data is kept across the cluster, it does not store the files. Clients talk to the NameNode whenever they want to locate a file. The NameNode's response is a list of relevant DataNode servers where the data is available.

As a result of this approach, the NameNode is a Single Point of Failure in the HDFS cluster. Whenever the NameNode goes down, the file system becomes offline.



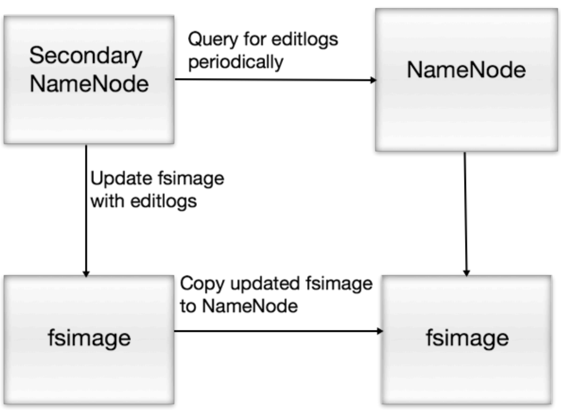
Problem with NameNode

The image shows how NameNode stores information [10]. There are two different files:

- edit logs: the changes made to the file system after the NameNode started
- fsimage: a snapshot of the file system when the NameNode started

In production clusters, the NameNode restarts are very rare. That means edit logs can grow large therefore in case of a crash we will lose a huge amount of metadata since the fsimage is very old.

The Secondary NameNode helps to solve this issue. It is responsible for merging the edit logs with fsimage. It collects edit logs on a regular basis and applies them to the fsimage. NameNode will use this fsimage in case of a crash and it can also be used to reduce the startup time of the NameNode. It is important to remember that the Secondary NameNode is not a real backup NameNode it only merges the edits into the fsimage.



Solution using the Secondary NameNode

DataNodes [13]

On a DataNode, a block is represented by two files in the native file system. The first contains the data itself, the second is the metadata.

On startup, the DataNodes connect to the NameNode and perform a handshake. This will verify the namespace ID and software version of the DataNodes. If one of them does not match with the NameNode's value, the DataNode automatically shuts down. After a successful handshake, the DataNode registers with the NameNode. DataNode will store it's internal identifier. If restart occurs the DataNodes will be recognizable with the ID, even if they get a different IP address or port. After the ID is registered to the NameNode the it will never change.

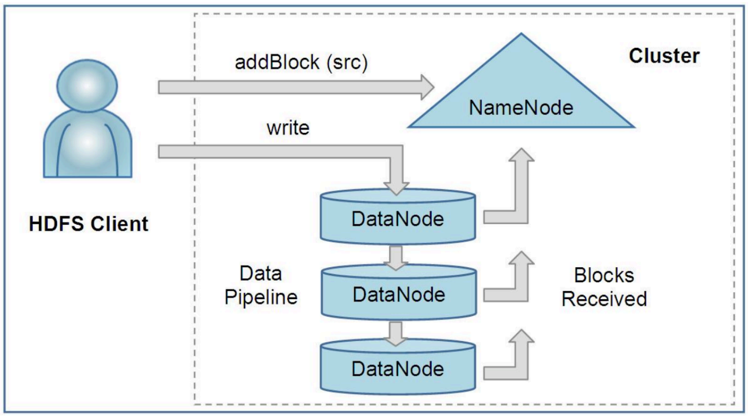
When a DataNode is registered it sends a block report immediately. It contains block id, generation stamp and the length of each block the DataNode hosts. To provide up-to-date information to the NameNode reports are sent every hour.

DataNodes send heartbeats to the NameNode. It ensures the NameNode that the DataNode is operating and block replicas of the server are available. If the NameNode does not receive a heartbeat from a DataNode it will consider the node to be out of service. The default heartbeat interval is three seconds.

HDFS Client [13]

User applications can access the file system using the HDFS client which exports the HDFS file system interface. HDFS supports operations similar to a traditional file system: read, write, create or delete files and create or delete directories. The user can refer to files or directories using paths in the namespace.

When someone reads a file, HDFS Client asks the NameNode for the list of DataNodes that host replicas of the blocks of the file. Then it will directly contact the DataNode and request the desired block.



HDFS file writing

The client creates a new file by giving its path to the NameNode. For each block, the NameNode will return a list of DataNodes to place the replicas. The client pipelines data to the given DataNodes, and they will confirm the creation of the block to the NameNode.

1.1.2 MapReduce

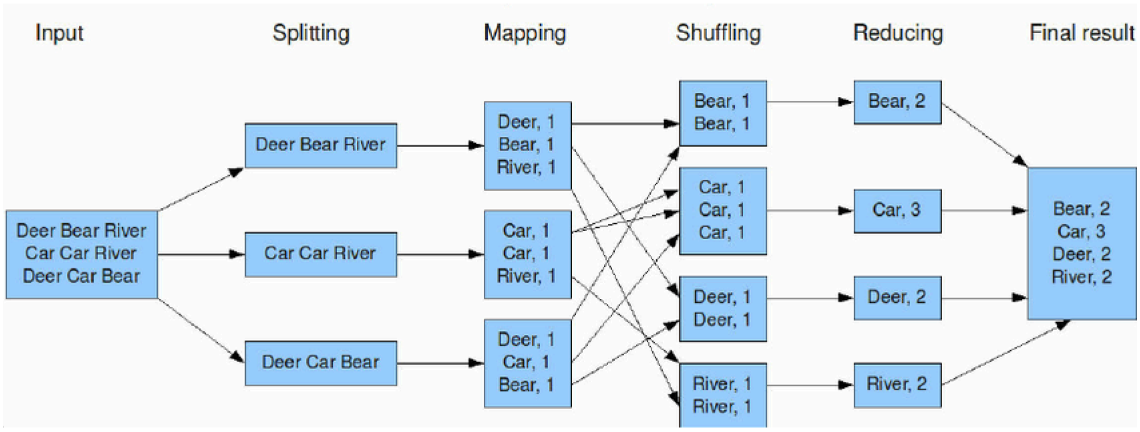
MapReduce is a programming model for processing data sets. Users specify two functions [8]:

- map function: processes a key-value pair to generate a set of key-value pairs
- reduce function: merges the intermediate values associated with the same key

Programs written in MapReduce are automatically executed parallelly on large clusters. Using this, programmers with no experience in parallel programming and distributed systems can utilize the available resources on the cluster.

Example [5] This example shows how MapReduce handles the problem of counting words. We have the following list of words:

Dear, Bear, River, Car, Car, River, Deer, Car, Bear



The MapReduce word count process [11]

- **Splitting**: the first step is dividing the input into splits. This will distribute the work among the Map nodes.
- **Mapping**: tokenize the words in each mapper and giving a value of 1 for each word, since every word in itself will occur once.
- **Shuffling**: partition takes place with shuffling and sorting: this way pairs with the same key will be sent to the same reducer.
- **Reducing**: a reducer gets the list of pairs and counts the number of ones in this list.

Advantages of MapReduce [5]

Parallel processing In MapReduce we divide the job among multiple nodes, so they can work on their part of the data parallelly. This way the data processing is done by multiple machines instead of one, so the time is significantly reduced.

Data locality In Hadoop MapReduce, instead of moving data into the processing unit, we move the processing unit to the data. The traditional approach has its limit when it comes to processing big data. Moving huge data is costly: network issues can occur and the master node (where data is stored) can get overloaded and may fail.

However, the MapReduce approach is very cost efficient, since all the nodes are working simultaneously on their part of the data and there is no chance of a node getting overloaded.

Using Hadoop we just need to provide the map and reduce functions, the rest is done by the framework. The word count example would look like the following in Java:

Map

```
public void map(LongWritable key, Text value, Context context) throws IOException,
   InterruptedException {
  String line = value.toString();
  StringTokenizer tokenizer = new StringTokenizer(line);
  while (tokenizer.hasMoreTokens()) {
    value.set(tokenizer.nextToken());
    context.write(value, new IntWritable(1));
  }
}
```

The input and output of the Mapper is a key/value pair.

Input:

- Key: the offset of each line
- Value: each line

Output:

- Key: the tokenized words
- Value: the hardcoded value 1

Reduce

```
public void reduce(Text key, Iterable<IntWritable> values,Context context) throws IOException,
    InterruptedException {
  int sum=0;
  for(IntWritable x: values) {
    sum+=x.get();
  }
  context.write(key, new IntWritable(sum));
}
```

Both the input and output of the Reducer is a key/value pair.

Input:

- Key: unique words, generated after the sorting and shuffling phase
- Value: a list of integers corresponding to each keys
- e.g. Bear, [1, 1]

Output:

- Key: all the unique words in the input text file
- Value: number of occurrences for each unique word
- e.g. Bear, 2; Car, 3

The traditional way to execute MapReduce operations is that the users specify the Map and Reduce functions in Java. However, this approach has some problems:

- it is not a high-level language for data processing
- data scientists do not understand Java. They came from the world of traditional databases, where SQL is used.
- even a simple problem (like word counting) resulted in hundreds of lines of code.

Although, the Hadoop MapReduce framework is written in Java, with the help of Streaming API we can create Map and Reduce functions in any languages.

MapReduce gives us a solution for many big data problems. However, for some scenarios, MapReduce is not the ideal choice: e.g. real-time analysis. In

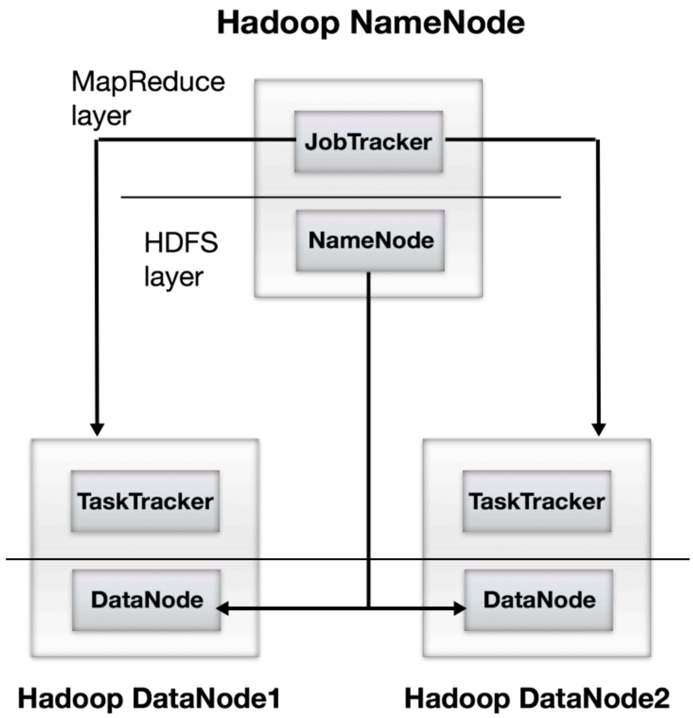
Hadoop 1.0 we could not use components other than MapReduce (for example Apache Storm which is ideal for real-time computation). The desire for utilizing the potential provided by the distributed file system (HDFS) in other solutions has grown. YARN provides a solution to fulfill this claim.

1.1.3 Yarn [4]

Hadoop 1.0 resource management

Previous to Hadoop 2.0, a single JobTracker had the responsibility to monitor the resources and distribute the MapReduce jobs for the DataNodes and monitor these jobs.

In Hadoop 1.0 the MapReduce module was responsible for cluster resource management and data processing as well.



Hadoop 1.0 architecture[12]

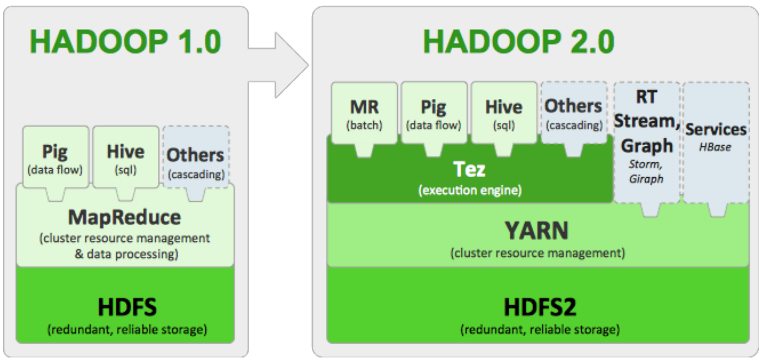
Resource management in Hadoop 1.0 [14]:

Clients submit jobs to the JobTracker which turns to the NameNode. It returns the location of the data. The JobTracker locates TaskTracker nodes with available slots close to the data and sends the job to the chosen TaskTracker nodes. After the job has started the JobTracker monitors the chosen TaskTracker nodes. If they do not send heartbeats frequently, they are deemed to have failed so the task will be scheduled on a different TaskTracker. The JobTracker gets a notification if a task fails. It decides what to do then: it may send the job to another TaskTracker, it can mark the record as something to avoid, or it may even put the TaskTracker to blacklist since it is unreliable. If the JobTracker sees that the task is finished, it will update its status. Clients poll the JobTracker for information.

The architecture of Hadoop 1.0 has many problems [12]:

- It **limits scalability** since the JobTracker runs on a single machine doing multiple tasks it becomes a bottleneck: resource management, job and task scheduling, monitoring are done by the JobTracker.
- JobTracker is a **Single Point of Failure**. If it goes down, all the jobs are halted.
- In Hadoop 1.0 **JobTracker is tightly integrated with the MapReduce** module so only MapReduce applications can run on Hadoop. Although MapReduce is powerful enough to express many data analysis algorithms (mostly batch-driven data analysis), it is not always the optimal paradigm. It is often desirable to run other computation paradigms on Hadoop like real-time analysis and Message-Passing approach, etc.. Since HDFS makes it easy to store large amounts of data it is desirable to utilize this for other big data problems.

Developers recognized that splitting the responsibility to resource management and application monitoring has serious benefits. YARN is a re-architecture of Hadoop that allows multiple applications to run on the same platform. With YARN, applications run "in" Hadoop, instead of "on" Hadoop. This takes Hadoop beyond a batch processing application to a "data operating system" where HDFS is the file system and YARN is the operating system.



From Hadoop 1.0 to Hadoop 2.0

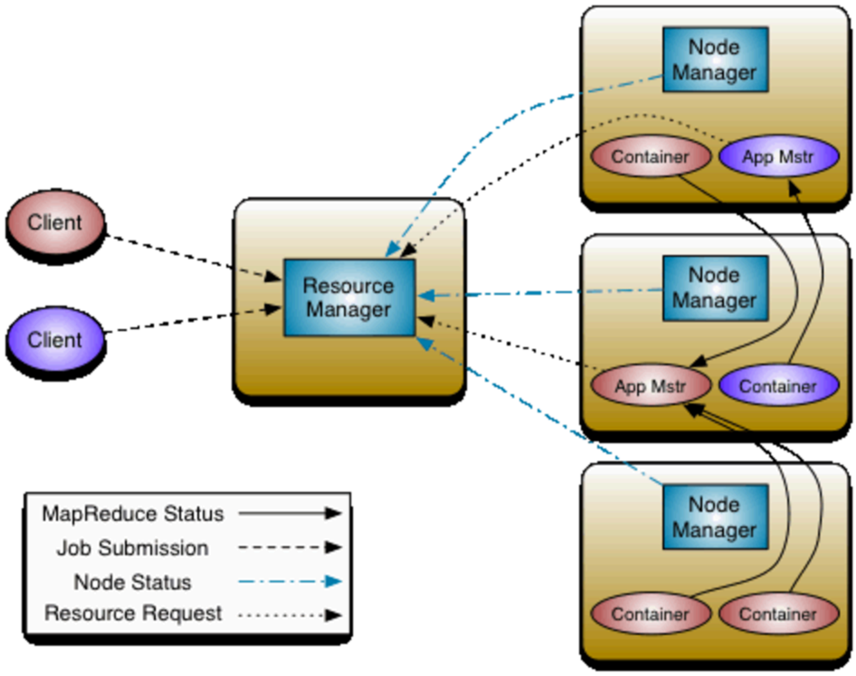
The fundamental idea behind YARN is to split up the functionalities of resource management and job scheduling/monitoring. In YARN we have a global ResourceManager (RM) and ApplicationMaster (AM) for each application.

ResourceManager is responsible for distributing the resources among all the applications in the system. The **NodeManager** is a per-machine agent who monitors the resource usages (CPU, memory, network, disk) of the containers and reports them to the ResourceManager.

The **ApplicationMaster** is framework specific, and its task is to ask the ResourceManager for resources when needed. It is also working with the NodeManager to execute and monitor tasks.

The ResourceManager is divided into two main components: Scheduler and ApplicationsManager.

- The Scheduler is responsible for allocating resources to applications running in the cluster. It schedules based on the resource requirements of each application. The Scheduler does not perform monitoring or status tracking.
- The ApplicationsManager accepts job-submissions. It negotiates the first container for executing the application specific ApplicationMaster. It is also responsible for restarting the ApplicationMaster if it fails.



Yarn architecture

In summary, with YARN Hadoop is able to run applications that do not follow the MapReduce model since it decouples the resource management and scheduling capabilities of MapReduce. With the help of YARN, we can efficiently utilize the resources and can run multiple applications in Hadoop, all sharing a common resources.

1.2 Apache Hive

Hadoop is a popular implementation of the map-reduce model and used widely to process and store extremely large datasets. However, a map-reduce program is very low level and difficult to maintain or reuse. Data scientist come from a world, where SQL is the standard of data processing. Apache Hive gives us a data warehouse solution built on top of Hadoop to write SQL-like queries so we can utilize the advantage of a declarative language. The language similar to SQL is called HiveQL.

1.2.1 Hive vs. RDBMS

This section shows the main differences between Hive and traditional databases (e.g. MySQL, Oracle, MS SQL etc.).

Hive supports SQL interface but it is not a full database. It follows WORM (Write Once Read Many) model while RDBMS is designed for Write and Read many times. Hive uses schema on read and traditional databases offer schema on write. Looking into Hive's approach, data is not validated until it is read. We can define multiple schemas to the same data and it provides a fast initial loading since the operation is just a copy and write. However, the schema on read approach has some drawbacks. Schema check on write ensures that the data is not corrupt and it provides a better query performance: when we read data, schema checking is not needed. Hive is a better choice when the schema is not available at loading time since it can be added later dynamically.

From Hive 0.13, Hive supports transactions [3] and full ACID semantics at row level, but with many limitations. Previous to this, atomicity, consistency and durability were available and only at partition level. With introducing transactions Insert, Update and Delete keywords were added to HiveQL.

The maximum data size allowed in a traditional RDBMS is 10's of Terabytes. However, Hive can easily handle Petabytes.

Conclusion

Hive is a great choice if we want to analyze large, relatively static data sets, fast querying is not necessary and easy, low-cost scalability is required. RDBMS provides fast responses for analyzing data dynamically, but scalability and maximum data size are limited.

1.2.2 Data storage

Hive structure data in the following units [15, 2]:

- **Databases**: namespaces to avoid conflicts of table, partition or bucket names.
- **Tables**: storage unit for data with the same schema. Tables maps to directories in HDFS.
- **Partitions**: Tables can have many partition keys. These will determine how data is stored. In HDFS partitions map to subdirectories in the table's directory. This way we can speed up the analysis. Instead of running the query in the whole table, Hive will only run our query in the relevant partitions (see example below). Partition columns are virtual, which means they are not part of the data itself.
- **Buckets**: Data can be divided into buckets based on the hash value of a column. These are helpful for efficiently sample data. Buckets are stored in files in the table's or partition's directory.

Example

This example shows how Hive data units map to HDFS and and how partitioning tables can speed up queries.

Hive tables map to `<warehouse_root_directory>/table_name` directory. As default, the warehouse root directory is /user/hive/warehouse. This can be changed with the corresponding hive configuration value.

```
CREATE TABLE test_table(c1 string, c2 int)
  PARTITIONED BY (date string, hour int);
```

The above SQL statement will create a table with two columns and two partitions and it will be stored in `/user/hive/warehouse/test_table` directory in HDFS. For every distinct date and hour value, a partititon will exists. Although, the partition columns are not part of the data, they are stored in the table metadata.

New partitions can be added either with the INSERT or the ALTER statement. These commands will create the corresponding HDFS directories:

`/user/hive/warehouse/test_table/date=2018-01-01/hour=12` and `/user/hive/warehouse/test_table/date=2018-01-02/hour=11`.

```
INSERT OVERWRITE TABLE
  test_table PARTITION(date='2018-01-01', hour=12)
```

```
SELECT * FROM t;

ALTER TABLE test_table
  ADD PARTITION(date='2018-01-02', hour=11);
```

Hive can use these information for pruning the directories to be scanned for query execution.

```
SELECT * FROM test_table WHERE date='2018-01-01';
```

In case of this query, Hive will only scan the files in `/user/hive/warehouse/test_table/date=2018-01-01` directory. Partitioning our data has significant impact on the time taken by queries.

Although, data in Hive is always in the corresponding directory (`<warehouse_root_directory>/table_name`), Hive is able to query data stored in other locations in HDFS. In order to do this, we can create EXTERNAL tables as the following statement shows:

```
CREATE EXTERNAL TABLE test_external(c1 string, c2 int)
  LOCATION '/user/example_table/example_data';
```

Hive assumes that the external table in its internal format. The difference between an external and normal (managed by Hive) table is that the drop table command doesn't effect the data itself on an external table. However, on a normal table, it drops the associated data.

1.2.3 Architecture

The main components of Hive are the following [15]:

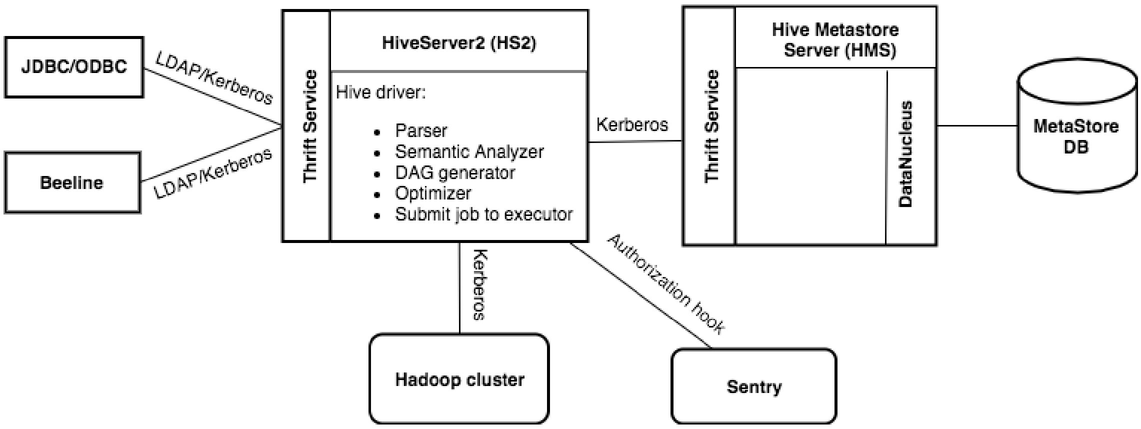
- **Driver**: manages the lifecycle of a HiveQL query by creating a session for it. The driver also collects the result after the execution phase.
- **MetaStore**: stores metadata about tables, columns or partitions. For example, it stores the table schema and location.
- **Compiler**: compiles the HiveQL statement and generates the execution plan using the partition and table metadata obtained from the MetaStore. First, it parses the query and does semantic analysis. Then converts it into AST (Abstract Syntax Tree) and after compatibility checking to a DAG of Map and Reduce tasks (if Hadoop MapReduce is the execution engine).
- **Optimizer**: transformations are done to get an optimized DAG for better performance. It is an evolving component. In the earlier stage, only rule-

based optimization was available which performed column pruning and predicate pushdown. Later, map-side join was introduced and several other join optimization, also cost-based optimization was added.

- **Execution Engine**: executes the plan created by the compiler. The plan is a DAG of stages. The engine manages the dependencies between these stages and executes them in the corresponding component. Hive is compatible with 3 execution engines: MapReduce, Apache TEZ and Spark which can run in Hadoop YARN.

- **HiveServer2**: a service that provides a thrift interface so clients can execute queries against Hive. Thrift is an RPC (Remote Call Procedure) framework for defining services for multiple languages.

- **Clients**: multiple clients are available to interact with Hive. Beeline (CLI), JDBC/ODBC, Python or Ruby client etc..



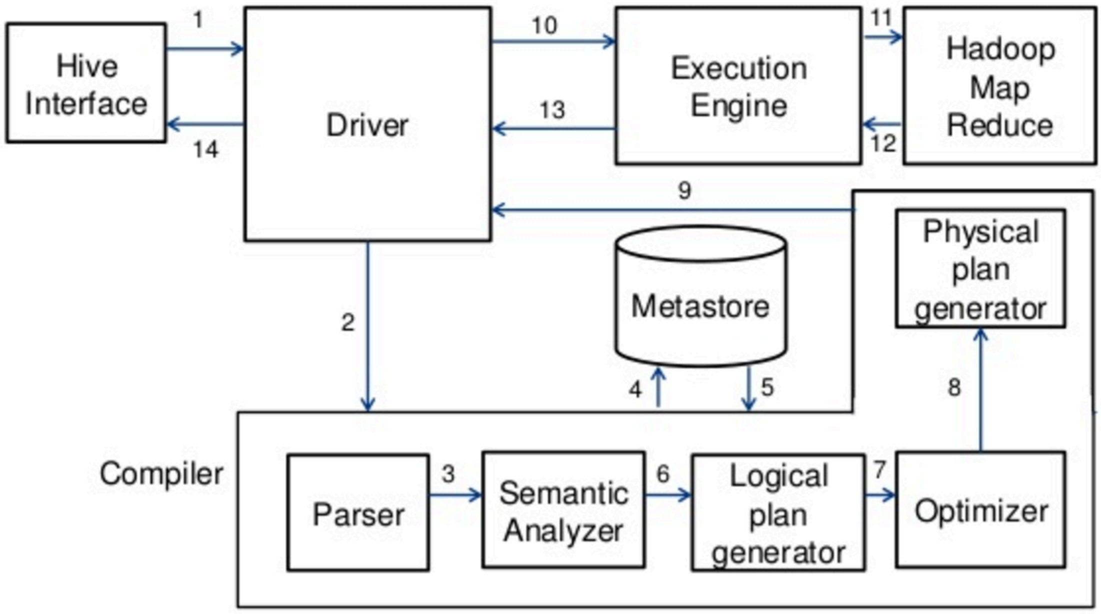
Hive architecture

Clients can connect to HiveServer2 using its Thrift Service. HS2 supports authentication of the clients using Kerberos or LDAP authentication. Hive Metastore server also supports Kerberos authentication for Thrift clients.

Sentry is a role-based authorization module for Hadoop, so we can control the priviliges for authenticated users. Hive can use Sentry over an ”authorization hook”, which Sentry registers to Hive configuration file if secure cluster is enabled.

HMS uses DataNucleus to persist metadata, so any relational database supported by it cal be used: it can be either embedded (e.g. Derby) or remote (e.g. MySQL) Metastore database.

1.2.4 Life of a Query



Life of a query [6]

User submits a query using one of the available Hive clients. HiveServer2 recieves the query through its Thrift interface (1). Compilation takes place:

First, it parses the query (2), so the query string is transformed into a parse tree using the open source Apache ANTLR. The semantic analyzer (3) converts the parse tree into a block-based query representation (Query Block Tree). At this phase connection to Metastore is made (4, 5) to gather information about the tables used by the query, like partition information for later pruning. It checks type compatibility and flag semantic errors.

The logical plan generator (6) transforms the internal representation to an operator tree, which is the logical plan. These operators can either be relational algebra operators (like filter, join) or Hive specific operators (e.g. reduceSink) which are used to convert the query later to a series of MapReduce jobs.

Optimization takes place for better performance (7). Most common optimizations are:

- Column pruning, so only columns needed by the query are kept
- Predicate pushdown: this way the rows can be filtered as soon as possible

- Partition pruning: prunes partitions that do not satisfy the predicate
- Map-side join: if joining two tables and one of them is small, map-side join can be done. This way the small table will be copied to the memory all the Map nodes.
- etc.

At physical plan generation (8) the logical plan is split into a series of Map, Reduce and HDFS tasks (of course if the execution engine is Spark, then to Spark tasks).

The Driver gets back the physical plan (9) and sends (10, 11) it to the corresponding execution engine (which is MapReduce by default). In this case, Hadoop executes the series of Map and Reduce tasks an returns the result to the Driver (12, 13), which sends it to the user (14).

1.2.5 Hive memory limitations

HiveServer2 and Hive Metastore require massive resources, especially memory wise. If we want to run up to 40 concurrent queries the recommended heap size range is 12-16 GB. Once we go above 16 GB it is recommended to split HS2 into multiple instances and load-balance them [7].

Although, with correct configuration and setup we can make our HiveServer2 instance long living, crashes can occur because of Out Of Memory error (OOM) in certain use-cases. Thus, minimizing the memory footprint whenever we can is a must.

The memory reserved by HiveServer2 can grow significantly in these situations [7]:

- When we have many table partitions, HS2 needs to load all the partition metadata from HMS. This can cause a real memory issue, and HiveServer can run out of heap memory and can crash.
- Concurrent connections can be made to HS2, and memory is directly proportional to the number of connections.
- Complex queries that access a large number of partitions from multiple tables can easily crash HS2.

If any of these conditions exists, Hive can slow down, refuse further connections, queries can fail and long query execution can occur. If the Garbage Collector cannot handle the memory reserved, HiveServer2 can go OOM.

2 Memory Analysis

To get a better insight on how Hive uses memory, I needed to build a basic model when and why Hive's reserved memory grows. During my work, I mainly focused on HiveServer2. The first step toward the model building was to get basic knowledge about Hive's code base and the query compilation process. The query life cycle mentioned in the previous chapter helped, but I had to find those steps in the code. If I have these points I am able to start measuring and maybe find some memory wastes.

2.1 Finding the measuring points

Hive has around 2 million lines of code so locating the main steps of the query processing was challenging.

As a starting point, the user (this case me) enters a query in the client. At this point, I run Hive locally, so the simplest way for submitting queries against Hive was using Beeline command line client. HS2 gets the query through its Thrift interface. After this,s the query goes to the *Driver* class, which is the main class for executing queries.

The Driver has two methods that are important for me: run and runInternal. The run method gets called first, which basically just delegates to the runInternal method. These two functions return with a CommandProcessorResponse, once the compilation and execution are done.

The runInternal does the two main steps:

- Driver.compile: gets the command string and parses, analyses the query and generates the execution plan.
- Driver.execute: gets the execution plan and executes it on a specific engine (MapReduce, Spark or TEZ).

2.1.1 Compile

The first step of the compilation is the parsing. It takes a string and returns an Abstract Syntax Tree (AST or the Parse Tree).

```
public int compile(String command, ...) {
...
  ASTNode tree;
  try {
    tree = ParseUtils.parse(command, ctx);
  } catch (ParseException e) {
...
}
```

2.1.1.1 Semantic Analyzis

After the AST is generated, the compile process will continue with the semantic analyzer. The type of the analyzer will depend on the query type we are running. For SELECT and INSERT the SemanticAnalyzer class is used.

```
public int compile(String command, ...) {
...
  BaseSemanticAnalyzer sem = SemanticAnalyzerFactory.get(queryState, tree);
...
  sem.analyze(tree, ctx);
...
}
```

The SemanticAnalyzer class is the main phase during compilation. It checks for semantic errors, fetches metadata from Metastore, generates and optimizes the query plan. The analyze method of the BaseSemanticAnalyzer is called from the Driver's compile method, and it delegates the call to the corresponding Analyzer. From now on, I will write about the SemanticAnalyzer class and its phases (which is executed for SELECTS and INSERTS.

```
void analyzeInternal(ASTNode ast, PlannerContext plannerCtx) {
  //(1)
  if (!genResolvedParseTree(ast, plannerCtx)) {
    return;
  }
  //(2)
  Operator sinkOp = genOPTree(ast, plannerCtx);
  ...
  //(3)
  ...
  resultSchema = convertRowSchemaToViewSchema(...);
  ...
  //(4)
  Optimizer optm = new Optimizer();
  ... = optm.optimize();
  ...
```

```
  //(5)
  TaskCompiler compiler = TaskCompilerFactory.getCompiler(conf, pCtx);
  compiler.compile(pCtx, rootTasks, inputs, outputs);
  ...
}
```

1. Fetches metadata and fill the Parse Tree with it so it becomes a Resolved Parse Tree.

2. The Operation Tree gets created which will contain operators that process data read from the table. This tree is called by the Map and Reduce methods.

3. The semantic analyzer will deduce the schema of the result set from the row schema

4. A logical optimization is done on the Operator Tree. There are two types of optimization: one that transforms the Operator Tree (like removing unnecessary operations) and one that does not (predicate pushdown, vectorization etc.).

5. Physical optimization and translation to the target execution engine take place. The output of this stage is the query plan. It can optimize the tree according to the engine used. For example in MR, common join can be translated to map-side join, if one of the tables is small enough to fit into the memory of the map nodes.

```
public int compile(String command, ...) {
  ...
  sem.validate();
  ...
}
```

After the semantic analysis is completed, the Driver will validate the plan. When it completes, the compilation ends and Hive will continue with the execution.

2.2 Measure the memory of HiveServer2

If I want to measure the memory usage of Hive, I need a something to create heap dumps and generate memory statistics. There are several tools to choose from. In this section, I will present the tools I considered and the method I used to get a better understanding of HS2 memory patterns.

Bibliography

[1] Apache Software Foundation. Apache hadoop hdfs. http://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html, .

[2] Apache Software Foundation. Apache hive data units. https://cwiki.apache.org/confluence/display/Hive/Tutorial, .

[3] Apache Software Foundation. Apache hive transactions. https://cwiki.apache.org/confluence/display/Hive/Hive+Transactions, .

[4] Apache Software Foundation. Apache hadoop yarn. https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html, .

[5] Ashish Bakshi. Mapreduce example. https://www.edureka.co/blog/mapreduce-tutorial/.

[6] Bala Krishna. Apache hive query. https://www.slideshare.net/bkrishnag/hive-30685916.

[7] Cloudera, Inc. Hive memory limitations. https://www.cloudera.com/documentation/enterprise/5-9-x/topics/admin_hive_tuning.html.

[8] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1251254.1251264.

[9] Forbes. How much data do we create every day. https://www.forbes.com/sites/bernardmarr/2018/05/21/how-much-data-do-we-create-every-day-the-mind-blowing-stats-everyone-should-read.

[10] Madhukara Phatak. Secondary namenode. http://blog.madhukaraphatak.com/secondary-namenode---what-it-really-do/.

[11] Óscar Pereira and Micael Capitão. Mediator framework for inserting data into hadoop, 12 2014.

[12] Saurabh. Apache yarn. `http://www.hadoopadmin.co.in/hadoop-administrator/apache-yarn/`.

[13] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, MSST '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society. ISBN 978-1-4244-7152-2. DOI: `10.1109/MSST.2010.5496972`. URL `http://dx.doi.org/10.1109/MSST.2010.5496972`.

[14] Steve Loughran. Hadoop 1.0 resource management. `https://wiki.apache.org/hadoop/JobTracker`.

[15] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Ning Zhang, Suresh Anthony, Hao Liu, and Raghotham Murthy. Hive - a petabyte scale data warehouse using hadoop, 01 2010.

[16] Wikipedia. Apache hadoop. `https://en.wikipedia.org/wiki/Apache_Hadoop`.