



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 23, 2018

Contents

1	Introduction	1
1.1	Hadoop basics	1
1.1.1	HDFS - Hadoop Distributed File System	2
1.1.2	MapReduce	6
1.1.3	Yarn [4]	9
1.2	Apache Hive	13
1.2.1	Hive vs. RDBMS	13
1.2.2	Data storage	14
1.2.3	Architecture	15
1.2.4	Life of a Query	17
1.2.5	Hive memory limitations	18
2	Memory Analysis	20
2.1	Finding the measuring points	20
2.1.1	Compile	21
2.1.1.1	Semantic Analyzis	21
2.2	Measure the memory of HiveServer2	23
2.2.1	Tools for generating heap dumps and statistics	23
2.2.1.1	jmap	23
2.2.1.2	jcmd	23
2.2.2	Tools for analysing heap dumps	24
2.2.2.1	YourKit	24

2.2.2.2	VisualVM	24
2.2.2.3	JXRay	24
2.2.2.4	The selected tool	25
2.2.3	Creating the code to measure memory	26
2.2.3.1	Parsing result	27
2.2.4	How HiveServer2 uses memory	27
2.2.5	Memory consumption of partitions	30
2.2.6	Where does the memory go?	31
2.2.6.1	Memory waste in HDFS Path	32
2.3	Fixing memory waste of URIs - HDFS-13752	33
2.3.1	Solution 1: WeakReference	33
2.3.2	Solution 2: SoftReference	34
2.3.3	Solution 3: On demand URI creating	34
2.3.4	Conclusion	35
2.3.5	A simple performance test for toURI	35
2.3.6	Performance test on a cluster	36
2.3.6.1	Listing files recursively	37
2.3.6.2	Changing the replication factor of a file	37
2.3.6.3	Analyzing the results	37
2.3.7	Effects of the change in Path	38
2.3.7.1	Hive MetaStore OOM	38
2.3.7.2	Hive on Spark memory issue	38
2.3.7.3	”Small files problem” in HiveServer2	39
2.3.7.4	Apache Impala’s solution for the problem	39
2.3.8	Benchmarking on a data center cluster	39

Bibliography	40
---------------------	-----------

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 23.

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 [10] 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. [21]

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 [18].

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

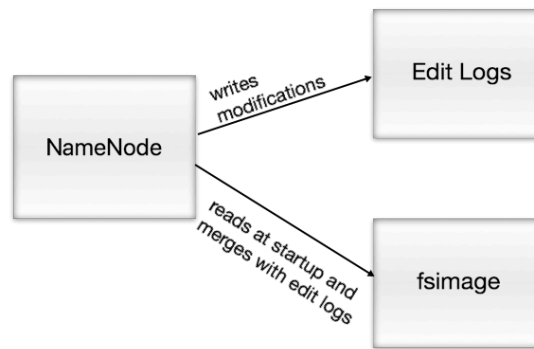
- **Hardware failure:** Hardware failures should be considered as normal, rather than an exception. An HDFS instance consists of hundreds or thousands of components so this means that some of them will always be non-functional. Therefore, fault detection and automatic recovery is a must.
- **Streaming 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.



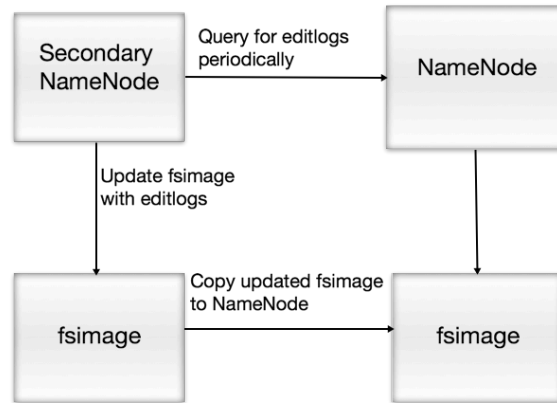
Problem with NameNode

The image shows how NameNode stores information [11]. 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 [18]

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 its 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.

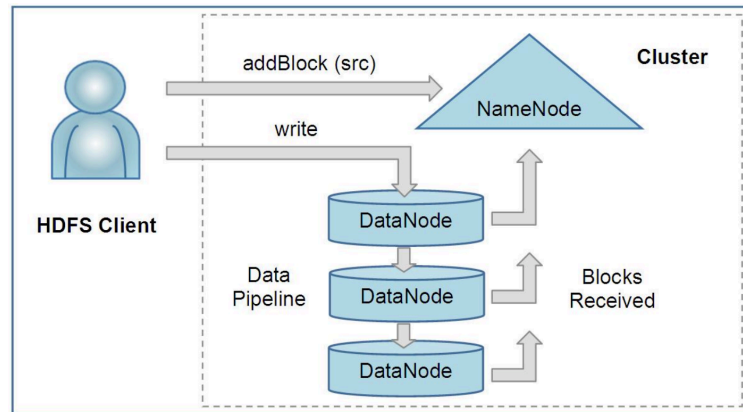
DataNodes send heartbeats to the NameNode. It ensures the NameNode that the DataNode is operating and block replicas of the server are available. If the NameNode does not receive a heartbeat from a DataNode it will consider the node to be out of service. The default heartbeat interval is three seconds.

HDFS Client [18]

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

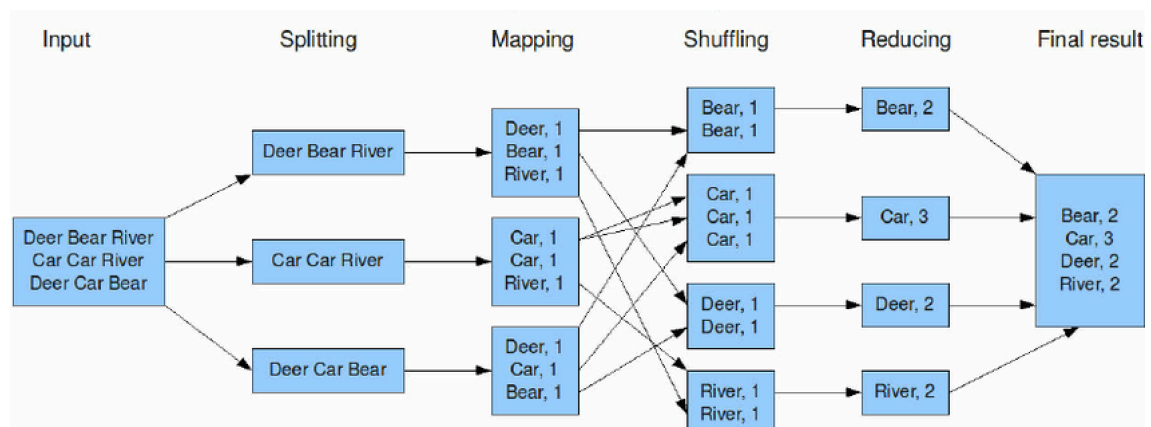
MapReduce is a programming model for processing data sets. Users specify two functions [9]:

- 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 [16]

- **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

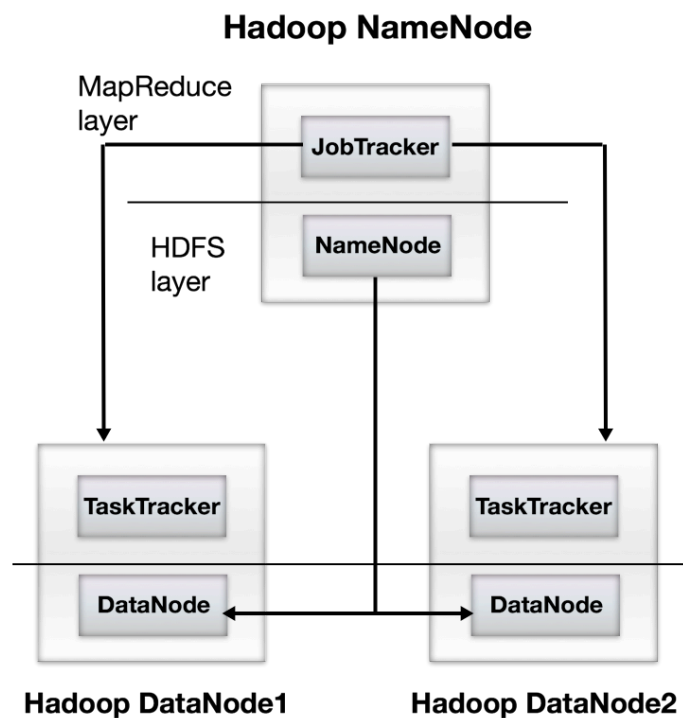
Hadoop 1.0 we could not use components other than MapReduce (for example Apache Storm which is ideal for real-time computation). The desire for utilizing the potential provided by the distributed file system (HDFS) in other solutions has grown. YARN provides a solution to fulfill this claim.

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[17]

Resource management in Hadoop 1.0 [19]:

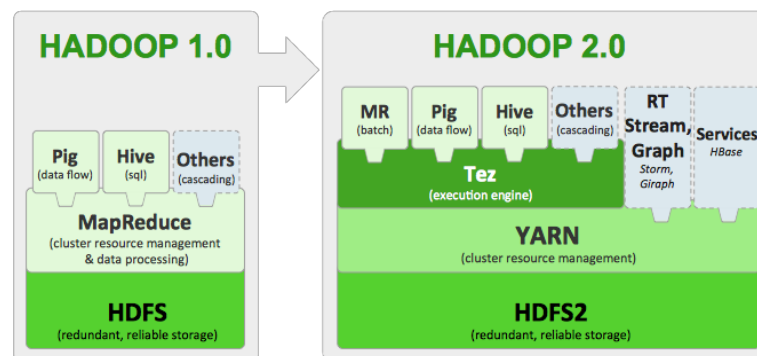
Clients submit jobs to the JobTracker which turns to the NameNode. It returns the location of the data. The JobTracker locates TaskTracker nodes with available slots close to the data and sends the job to the chosen TaskTracker nodes. After the job has started the JobTracker monitors the chosen TaskTracker 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 [17]:

- 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

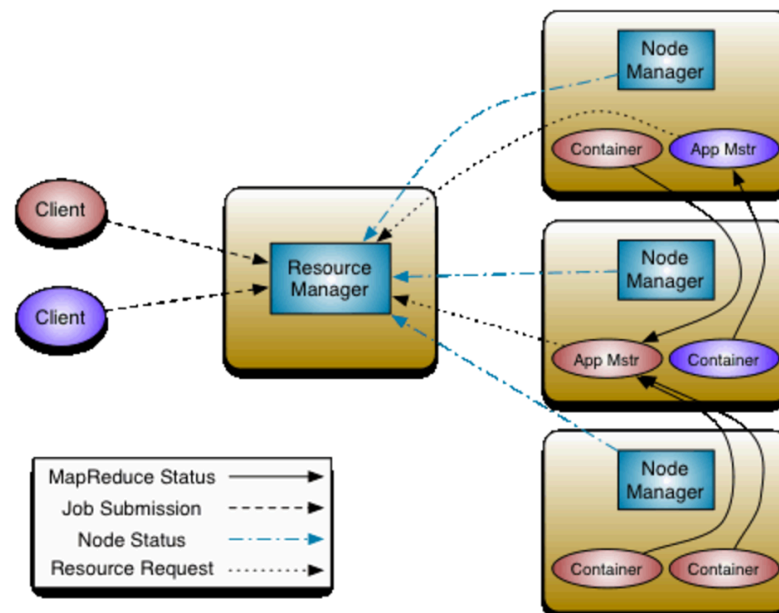
The fundamental idea behind YARN is to split up the functionalities of resource management and job scheduling/monitoring. In YARN we have a global ResourceManager (RM) and ApplicationMaster (AM) for each application.

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 [20, 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 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 partition will exist. 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.

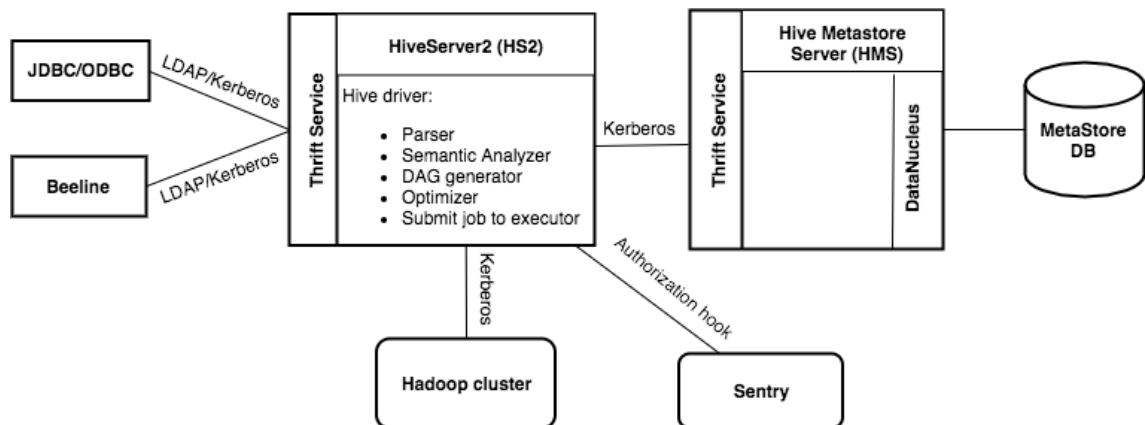
1.2.3 Architecture

The main components of Hive are the following [20]:

- **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..



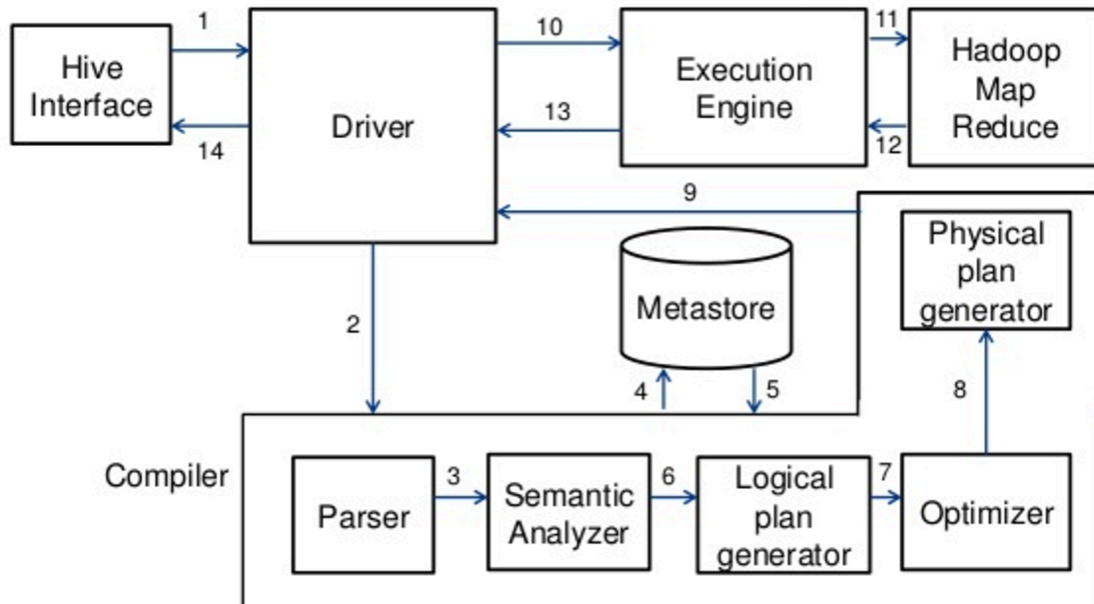
Hive architecture

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 privileges 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 can 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 receives 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 and 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 [8].

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 [8]:

- 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 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, 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 Analysis

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 and analyze 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.

2.2.1 Tools for generating heap dumps and statistics

For creating heap dumps and statistics of the memory, I decided to use a command line tool. In contrast of a UI tool, it has the benefits to generate heap dumps and statistics automatically.

2.2.1.1 jmap

jmap [15] can be used to obtain heap dumps and histograms from a running java process.

To gather information about a running Java program we need its process ID (PID). Using this we can easily create the heap dump with the *-dump* option. If we do not want to analyze large heap dumps, just to get the objects that reserve the most memory, we can use the *-histo* option. It will print a histogram of the heap, containing each Java class, the number of objects and memory sizes per class.

2.2.1.2 jcmd

jmap would have been a perfect choice for measuring memory. However, it is recommended to use the latest utility, which is called *jcmd* [14]. It has enhanced diagnostics capability and its performance overhead is reduced.

The usage of *jcmd* is similar to *jmap*. For creating heap dumps we can use *jcmd <PID> GC.heap_dump*. If we want to obtain class histogram we can do that with the *jcmd <PID> GC.class_histogram* command. With the *jcmd <PID> GC.heap_info* command we can gather basic information of the heap: e.g. young and old memory of the process.

I decided to use *jcmd* in my future work to gather memory information automatically and generate heap dumps at the measuring points identified in the previous section.

2.2.2 Tools for analysing heap dumps

Creating heap dumps with a CLI tool is easier, however analysing those requires a tool with user interface.

2.2.2.1 YourKit

YourKit provides great analysing capabilities for heap dumps. We can use it for profiling local or remote Java applications. It has controllable overhead. Although, it is a great tool for profiling Java applications, it is a commercial software and only has 15 days of free licence.

2.2.2.2 VisualVM

VisualVM provides a great visual interface for profiling Java applications and browsing heap dumps. It offers multiple views for analysing them. We can get the summary of the dump which contains the total bytes, classes and objects.

With the "classes view" we can get information about the reserved memory and number of instances for each class. We can calculate the retained memory, which is the memory that would be freed if the objects would be garbage collected.

2.2.2.3 JXRay

JXRay [13] is a memory analysis tool for Java applications. It creates a HTML report of a given heap dump. It can automatically detect common problems, like data duplication or non-optimal use of data structures.

JXRay can give us a compact report, even if the heap dump analyzed is 10's of GB. It can calculate how much memory taken by a collection internal representation and their workload, since JXRay knows about the internal representation of common java collections (HashMap, ArrayList etc.).

[illegible]

Figure 2.1: Sample JXRay report summary

2.2.2.4 The selected tool

I decided to use VisualVM for visual memory analysis since it has great capabilities for summarizing the most important information about a heap dump and with its help, I am able to walk through the memory tree easily.

During my work, I also used JXRay, since it can automatically detect the most common memory problems and give me an overview of the possible issues a Java application has.

2.2.3 Creating the code to measure memory

With the available tools I am able to generate heap dumps really easily. However, to get a model how Hive uses the memory, I should gather statistics and heap dumps at certain points of the query execution. Clearly, it is not possible by hand. Thus, I needed to create a class in Hive's codebase and integrate the sampling in the identified points found in the previous section.

In order to use `jcmd` from Java, I needed a way to execute a terminal command from code. With the *ProcessBuilder* java class I'm able to create operating system processes with the given attributes. The *Runtime.exec()* method would do the same, but using this command is discouraged.

Now that I can run a command line tool from Java, I will need the process ID of `HiveServer2`. Since Hive uses Java 8, I cannot use the new Process API that Java 9 provides. With the help of the *ManagementFactory* class, we can get the managed bean of the runtime system of the JVM. It will return a class implementing the *RuntimeMXBean* interface. The name of the running JVM contains the ID of the process. With the *RuntimeMXBean.getName()* method, I was able to get the PID of `HiveServer2`. The method will return a string in a format of: *pid@hostname*. Using the split method we can get our application's process ID.

To avoid code duplication I created a method for getting the *ProcessBuilder* which contains the given command and the PID for `HiveServer2` is already set.

```
private ProcessBuilder getProcessBuilder(String subCommand){
    ProcessBuilder builder = new ProcessBuilder();
    //Get own pid
    String pid = ManagementFactory.getRuntimeMXBean().getName().split("@")[0];
    builder.command("sh", "-c", String.format("jcmd %s %s", pid, subCommand));
    return builder;
}
```

Using the *getProcessBuilder* function, it is really simple to run a terminal command. For example, getting information about the state of the heap or creating a heap dump will look like this:

```
...
process = getProcessBuilder("GC.heap_info").start();
...
getProcessBuilder("GC.heap_dump " + path).start();
```

2.2.3.1 Parsing result

I am able to create detailed statistics about the current memory state at certain phases of the query. As a first approach, I decided to just use *GC.heap_info* to see how memory usage looks like at different stages, not caring how much memory is reserved by each class. However, if we look at the result of the *jcmd <PID> GC.heap_info*, it will look quite messy, and hides the most important details, especially if we have 15 result for each query. An example how it looks:

```
21566:
PSYoungGen total 1395200K, used 27419K [0x000000076ab00000, 0x00000007c0000000,
0x00000007c0000000)
eden space 1392640K, 1% used
[0x000000076ab00000,0x000000076c5c6c70,0x00000007bfb00000)
from space 2560K, 0% used [0x00000007bfd80000,0x00000007bfd80000,0x00000007c0000000)
to space 2560K, 0% used [0x00000007bfb00000,0x00000007bfb00000,0x00000007bfd80000)
ParOldGen total 1046528K, used 14704K [0x00000006c0000000, 0x00000006ffe00000,
0x000000076ab00000)
object space 1046528K, 1% used
[0x00000006c0000000,0x00000006c0e5c058,0x00000006ffe00000)
Metaspace used 48314K, capacity 48612K, committed 49280K, reserved 1093632K
class space used 5359K, capacity 5440K, committed 5504K, reserved 1048576K
```

I only need certain values: allocated young and old memory, and a total memory which is the sum of the young and old values. I created a function called *getResult* which will read the result from a given process and return in string format. The returned string will look like above, so parsing is needed if I want to gather the important details.

To do this, I made a function (*printResultToCSV*) which will parse the given string, and print the results to a csv file in a table like format, where the first line is the query, and the columns are the different stages.

select * from people2 limit 10000	Before compile	After parse	Before semantic analyze	After Generating Resolved Parse tree from syntax tree	After Generating OP tree from resolved parse tree	After performing logical optimization	After optimizing physical op tree
Young	10552	8679	10732	13568	13154	25681	23503
Old	12296	14499	14256	14342	51493	85316	85151
Sum	22848	23178	24988	27910	64647	110997	108654

Figure 2.2: Sample output of the parsing

2.2.4 How HiveServer2 uses memory

I have the code to measure the memory of Hive and get a general picture of the usage at certain phases. With this, I am able to get a better understanding when and why memory goes high. I would like to get an answer to these questions: How

does the number of joins increase the memory? Which type of query generates a lot of memory usage: union, group by? If we increase the number of partitions, how it affects the heap size? In this section, these questions will be answered.

The first step is to create a managed table where I will run my queries in the future. Hive faces memory problems when we have a highly partitioned table. I decided that for the first run, 20 000 partitions will be enough.

The second step was to create queries which will be submitted to Hive. The first question was how the number of joins affects memory? To answer it, I have created queries with an increasing number of joins included. I used self-joins and only increased the number to three. The pattern was clear for only four measures: a simple select query without join operator, and queries with one, two and three. From the output of my parsed CSV file, I generated the following chart that shows the memory increase.

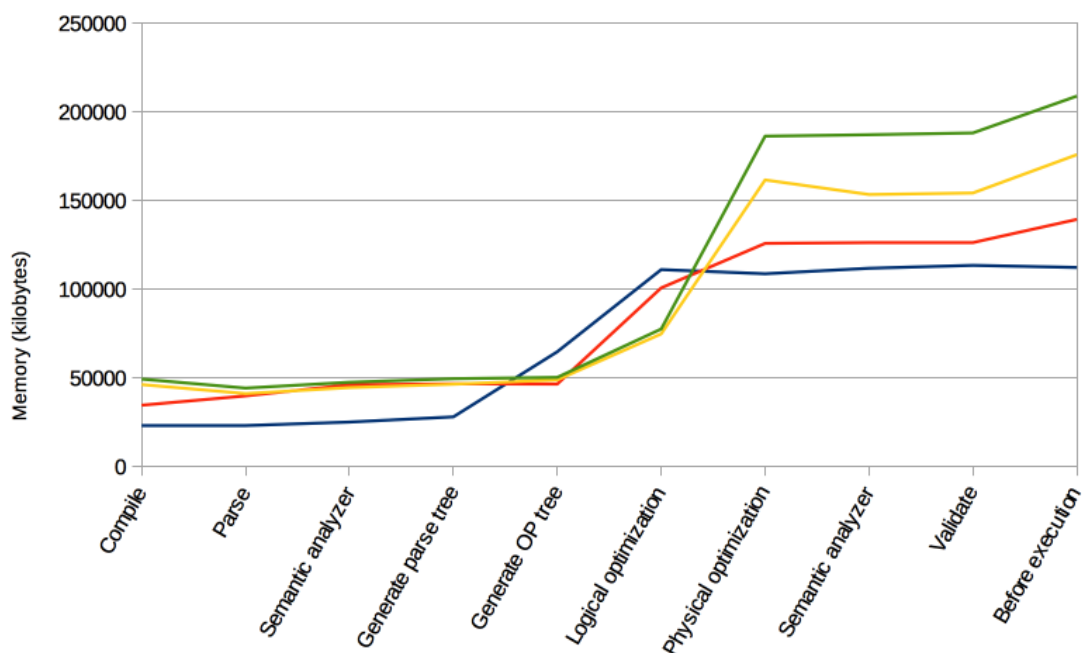


Figure 2.3: Heap size with increasing number of joins

Queries run with their color in the chart:

- select * from tablename (————)
- select * from tablename t1 join tablename t2 on t1.id=t2.id (————)
- select * from tablename t1 join tablename t2 on t1.id=t2.id join tablename t3 on t2.id=t3.id (————)

- `select * from tablename t1 join tablename t2 on t1.id=t2.id join tablename t3 on t2.id=t3.id join tablename t4 on t3.id=t4.id (—————)`

We can see that the memory change caused by the number of join operators is negligible, only around 20 Megabytes per join. The results were as expected: the memory increased significantly when HiveServer2 connected to the MetaStore and asked for metadata. During physical optimization, Hive loaded metadata to memory including partition metadata. In a highly partitioned table, this size is notable. Although real queries submitted to Hive by users are much more complicated and contains multiple tables, the model will be quite similar: heap memory will rise when metadata is loaded.

I also wanted to see the memory effects of *group by* and *union all* operations. I ran 5 more queries including these but did not find anything notable. The pattern remained the same: see the chart below.

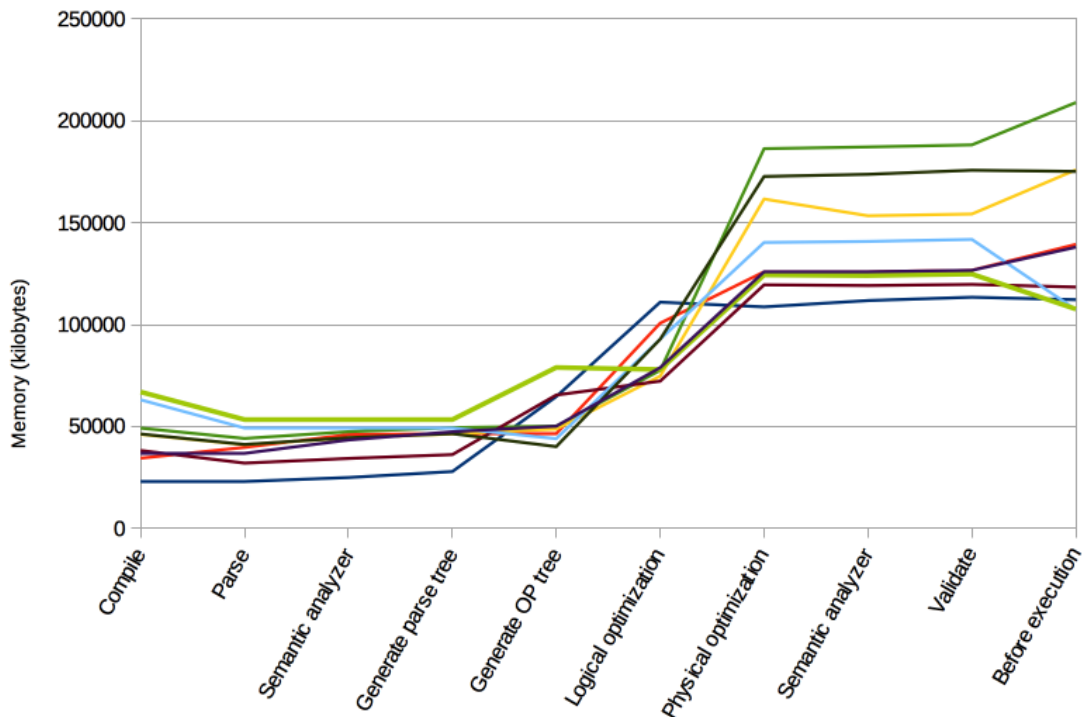


Figure 2.4: Heap size when running various type of queries

I came to the conclusion that for a single connection to HiveServer2, the most important factor that increases heap size is the number of partitions in tables, so I decided to continue my investigation in that area.

2.2.5 Memory consumption of partitions

A well-known issue in Hive is, if we have a highly partitioned table, the memory goes really high in HiveServer2. As a first step, I wanted to recreate the issue. I created tables with an increasing number of partitions: 200, 2000, 5000, 20000 and 100000 partitions. For each table, I executed the same query which contains a simple select with a self-join. During this, I measured the memory and created a heap dump after the semantic analyzer phase of the compilation. At this stage, the partitions are already loaded so I can analyze how much these objects exactly reserve. To get this information, I analyzed the heap dumps with VisualVM and filtered for the Partition objects. The pattern we can observe was as expected, but I the size of the heap reserved by these objects were smaller than I previously anticipated.

From the heap dumps I have found that mainly three kind of objects reserve the memory due to partitions, so I will focus on these objects:

1. hive ql.metadata.Partitions
2. hive.metastore.api.Partition
3. hive ql.plan.PartitionDesc

	ql.metadata. Partitions	metastore.api. Partition	ql.plan. PartitionDesc	Sum
200 partitions	300 Kb	288 Kb	108 Kb	696 Kb
2000 partitions	3057 Kb	2946 Kb	1110 Kb	7113 Kb
5000 partitions	7988 Kb	7709 Kb	2793 Kb	18490 Kb
20000 partitions	29999 Kb	28879 Kb	3173 Kb	62051 Kb
100000 partitions	165002 Kb	158841 Kb	60531 Kb	384374 Kb

Table 2.1: Memory of partitions

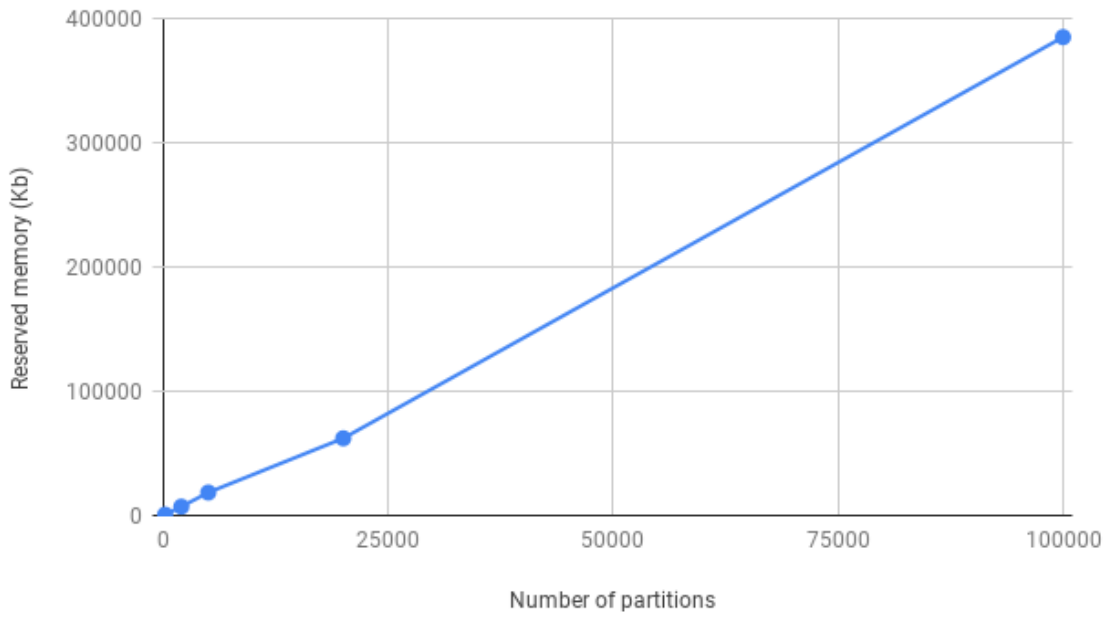


Figure 2.5: Reserved memory by partitions

As we see, for 100.000 partitions the reserved memory is around 385 Megabytes. Before taking the samples, I expected higher memory consumption. However, with a little investigation I found that the memory waste around partitions was already somewhat decreased [12]. `PartitionDesc` objects has a `java.util.Properties` field. These `Properties` fields in the `PartitionDesc` most often are the same. Interning these saves a big amount of memory. In a later section, I will write more about the interning solution because it gave me an idea to fix another memory related issue.

However the memory reserved by partitions is still high enough, so I decided to take a look at the heap dumps and try to find wastes or problems that affects memory.

2.2.6 Where does the memory go?

I analyzed the heap dump generated after the semantic analyzer phase when self-joining a table with 100000 partitions. It showed an interesting fact: 47.1% (853 Megabytes) of retained memory is reserved by Strings. Retained means, that if the Garbage Collector frees them, we win that amount of memory.

Name	Count	Size	Retained
java.util.ArrayList	872,971 (3.2%)	27,935,072 B (1.5%)	1,145,398,414 B (63.3%)
java.lang.Object[]	333,259 (1.2%)	32,852,176 B (1.8%)	1,137,937,642 B (62.9%)
org.apache.hadoop.fs.LocatedFileStatus	812,064 (2.9%)	92,575,296 B (5.1%)	1,068,184,000 B (59%)
java.lang.String	6,007,129 (21.8%)	168,199,612 B (9.3%)	853,253,414 B (47.1%)
org.apache.hadoop.fs.Path	1,022,299 (3.7%)	24,535,176 B (1.4%)	792,212,186 B (43.8%)
java.net.URI	1,022,310 (3.7%)	147,212,640 B (8.1%)	767,682,766 B (42.4%)
char[]	6,007,959 (21.8%)	687,623,070 B (38%)	686,669,538 B (37.9%)
java.util.HashMap\$Node[]	334,856 (1.2%)	34,734,496 B (1.9%)	230,222,408 B (12.7%)
java.util.HashMap	445,434 (1.6%)	28,507,776 B (1.6%)	221,747,828 B (12.2%)
java.util.HashMap\$Node	999,979 (3.6%)	43,999,076 B (2.4%)	216,633,773 B (12%)
java.util.HashSet	1,318 (0%)	31,632 B (0%)	216,352,207 B (12%)
org.apache.hadoop.hive.ql.hooks.ReadEntity	110,003 (0.4%)	11,000,300 B (0.6%)	202,438,419 B (11.2%)
org.apache.hadoop.hive.ql.metadata.Partition	110,001 (0.4%)	6,160,056 B (0.3%)	165,001,500 B (9.1%)
org.apache.hadoop.hive.metastore.api.Partition	110,001 (0.4%)	8,030,073 B (0.4%)	158,841,444 B (8.8%)
java.util.Properties	110,071 (0.4%)	7,925,112 B (0.4%)	151,796,926 B (8.4%)

Figure 2.6: Memory reserved by Strings

We also see in the above image that *hadoop.fs.Path* objects reserve 792 Megabytes. It seems too much considering that we only speak about the location of a file in the filesystem. As a next step, I looked inside the Path objects to see if this is a waste or not.

2.2.6.1 Memory waste in HDFS Path

```

org.apache.hadoop.fs.Path#43
  <fields>
    static <classLoader> = sun.misc.Launcher$AppClassLoader#1
    uri = java.net.URI#53 : file:/Users/barnabasmaidics/data/hive/warehouse/people10/age=1/.hive-staging_hive_2018-07-19_10-07-13_
      string = java.lang.String#57321 : file:/Users/barnabasmaidics/data/hive/warehouse/people10/age=1/.hive-staging_hive_2018-07-19_10-07-13_
      decodedPath = java.lang.String#57319 : /Users/barnabasmaidics/data/hive/warehouse/people10/age=1/.hive-staging_hive_2018-07-19_10-07-13_
      schemeSpecificPart = java.lang.String#57320 : /Users/barnabasmaidics/data/hive/warehouse/people10/age=1/.hive-staging_hive_2018-07-19_10-07-13_
      path = java.lang.String#57319 : /Users/barnabasmaidics/data/hive/warehouse/people10/age=1/.hive-staging_hive_2018-07-19_10-07-13_
      scheme = java.lang.String#57318 : file
      static hexDigits = char[]#14786 : 0123456789ABCDEF
      decodedSchemeSpecificPart = null
      decodedFragment = null
      decodedQuery = null
      decodedAuthority = null
      decodedUserInfo = null
  
```

Figure 2.7: Inside of a Path

In Path objects, we store the location in a *java.net.URI* object. Its internal representation seems really wasteful. As we see, URI stores a path in 3 different Strings. The Strings are almost the same: for example the string field inside the URI contains the scheme, the path field does not, and this is the only difference etc.. We can ask ourselves: is it really necessary?

To determine this we need to inspect the Path.java source code. URIs are always created the same way: we pass four String parameters to the constructor of the URI class: scheme, authority, path, and fragment. In the code below the null

parameter represents the query part of a URI. Passing null means that we do not need that value.

```
newUri = new URI(scheme, authority , path, null, fragment);
```

To summarize, the internal representation of the URIs clearly seems like a waste of memory. So I started to think of an alternative solution to replace these URI objects and save around 66% of memory for each Path.

2.3 Fixing memory waste of URIs - HDFS-13752

The duplication comes from the *java.net.URI* objects so the first thing that came to my mind was to replace this with a more memory efficient implementation in the Path.java class. As a first approach, I created a Jira ticket in the corresponding Apache site to report the issue [7].

The community agreed that the way URIs store these paths is a waste, and could be stored more efficiently. However, many other classes use these URIs. The Path class has a public *toURI* function which returns the URI representation of the path. Obviously, this cannot be removed because others are depending on it. I found three possible solutions to get rid of this memory overhead.

- Keep the URI field with only having a WeakReference to it, and store the 4 parts of the Path in separate fields
- Keep the URI field with only having a SoftReference to it, and store the 4 parts of the Path in separate fields
- Remove the URI field completely and store those parts of the URI that we really use in separate Strings

2.3.1 Solution 1: WeakReference

A WeakReferenced object works in a way, that if the object is only weakly reachable (its only reference is a WeakReference), it will not prevent the object to be garbage collected. We would have a WeakReference to the URI and when the GC runs it can collect the URIs if we do not have a strong reference to it.

When someone wants to get the URI representation of the path and calls the *toURI* function, we need to check if the URI exists through the WeakReference

or not. If it exists we can just get it because we have a WeakReference to it. If it does not, we need to recreate the URI from its parts.

2.3.2 Solution 2: SoftReference

Using SoftReference would be a more conservative solution. If the Garbage Collector finds an object that is only softly reachable, it does not dump the object instantly. An object with a SoftReference will only be collected at memory demand. This way before an Out Of Memory error is thrown, the GC can free all URIs. If the GC collected the URI we would do the same as mentioned before: recreate the URI from the stored parts when needed.

2.3.3 Solution 3: On demand URI creating

A possible solution is to remove the URI completely and only keep 4 parts of the Path which are used: scheme, authority, path and fragment. This way we can immediately win around 66% of memory. The only drawback is that we always have to recreate the URI every time the toURI method is called and this might be expensive CPU-wise.

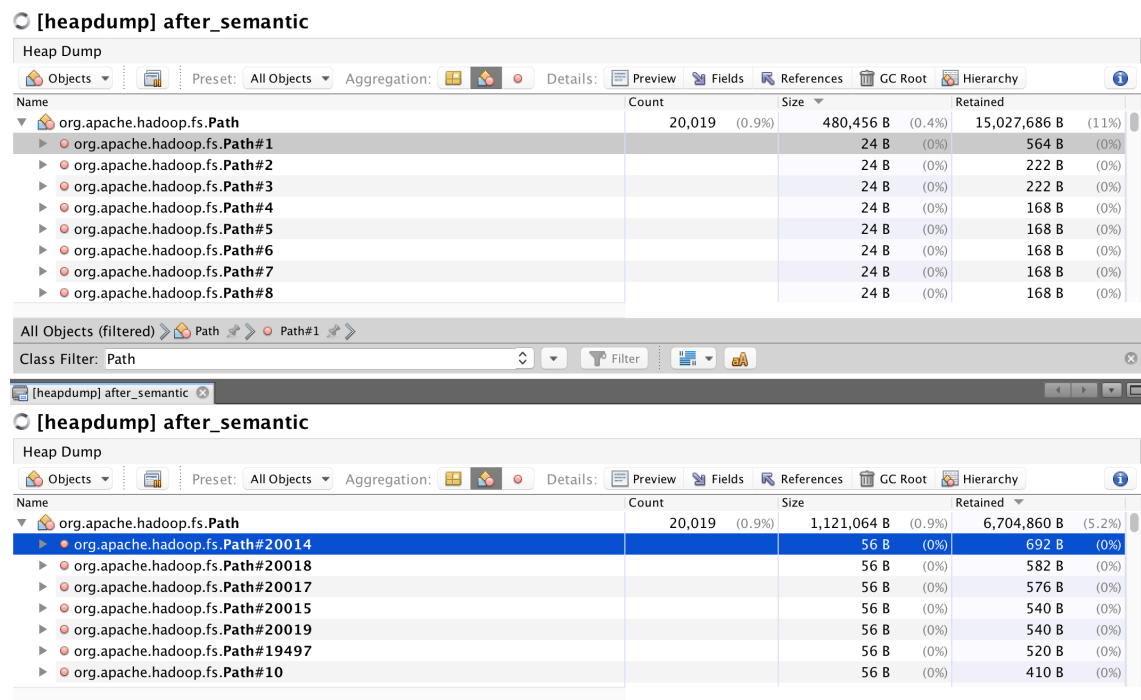


Figure 2.8: Memory save of Paths

2.3.4 Conclusion

For the second thought, I realized that using `WeakReference` would not be the best decision. It would almost be the same as creating the URI on demand, since whenever the GC runs, it will collect the URIs immediately and when `toURI` is called we would have to recreate it anyway. The only way to decide which solution might be the best is to measure the performance of both approach.

I also analyzed the code base of Hadoop to see how `Path.toUri` method is usually used. Most often we just want to ask for certain parts of the Path object and not really using the URI itself. For example: `pathObject.toUri().getPath()` to get the path or `pathObject.toUri().getScheme()` to get the scheme. Considering only this, solution 3 would be the most effective, since we are able to get the path/scheme/authority or fragment directly from the Path object and we do not need to first transform it to an URI and than ask for the value we need.

2.3.5 A simple performance test for toURI

I created a really simple test, to decide which solution would solve the issue without a significant CPU overhead.

I tested the CPU usage of the `toUri()` method, analyzing the different solutions: I created 1.000.000 Path objects (of course this is not a real use case, but it can be a good estimation as the first approach). See the results in the table below (3. row). I've also measured the memory effects of the changes.

The code I used for testing:

```
long startTime = System.nanoTime();
for(int i=0; i<1000000; i++){
    paths[i].toUri().getPath();
}
long estimatedTime = System.nanoTime() - startTime;
```

	Original	SoftReference (solution 2)	New URI (solution 3)
Memory of the Paths	608 MB	664 MB	238 MB
Time of the toURI calls	0.12973 s	0.04142 s	1.29133 s

Table 2.2: Results of the test

The results showed that using `SoftReference` has a bit of memory overhead because apart from the URI itself, we need to store the `SoftReferences` which require heap space as well. Thus, it would only be a save if the heap is almost full, other

than that we it would reserve even more memory. The time of `toURI` calls might be deceptive. Calling *toURI* method only once is slower than with the original solution and we do not use the method as in the example above.

Looking into the results of the third possible solution, the benefits are clear memory-wise. However, the time of the `toURI` calls do not look so promising. The CPU time is almost 10x bigger than with the original approach. But if we take into account other factors, not just the plain CPU time, the overhead might not be that big. If the memory of the paths are reduced, the GC can work much faster, so smaller GC pauses will occur. Also, these Path objects passed through the interface of Hadoop in both direction. For example, NameNode gives the location of files in the format of Paths and Hive passes these objects to Hadoop as well. Considering these, my change would be possibly beneficial for transferring data through the network: Passing objects which are smaller is obviously faster.

An additional factor that should be considered is that if we remove the *toUri* call whenever we want to get the values of path, scheme, authority or fragment, we can improve the performance of the original solution, since we can get those directly. Instead of *Path.toUri().getPath()* we can just say *Path.getPath()*.

To test this theory, I created a simple test. The times of 1.000.000 `toUri().getPath()` calls:

- Original: 0.143235527 s
- Solution 3 if *Path.toUri().getPath()* staying: 1.316882214 s
- Solution 3 with only *Path.getPath()*: 0.004373138 s

With the results of my test, I decided to choose solution 3, which may be the simplest fix: remove the URI field completely and if someone needs it, create on demand from the path, scheme, authority and fragment parts.

2.3.6 Performance test on a cluster

As a next step, I configured a cluster to see if my change would affect CPU significantly. The cluster I created had 4 nodes and only HDFS, YARN and HIVE were installed. I generated the below HDFS directory structure to test the performance of my fix. The "perfctest" directory contains 100 folders, each contains 1000 other folders with one txt file. The code I used for testing can be found in the Jira ticket I reported [7] (HDFS-13752.003.patch).

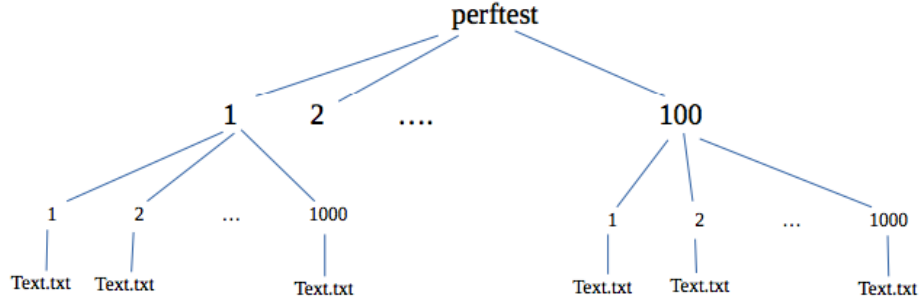


Figure 2.9: Directory structure of the test

For the tests, I used FileSystem shell to interact with the HDFS directly. I choosed one shorter and one longer running operation for measuring the CPU time. The values in the tables are in millisecond.

2.3.6.1 Listing files recursively

The command used: `hdfs dfs -ls -R /user/perftest`

	M1	M2	M3	M4	M5	Avg
Original	42651	38136	40183	41749	36963	39936
New URI	41762	39219	38800	37315	37719	38963

Table 2.3: CPU time of a recursive listing

2.3.6.2 Changing the replication factor of a file

The command used: `hdfs dfs -setrep -w 3 /user/perftest`

	M1	M2	M3	M4	M5	Avg
Original	172240	179050	193963	189105	171446	181161
New URI	185523	169111	171326	171451	170389	173560

Table 2.4: CPU time of a replication factor changing

2.3.6.3 Analyzing the results

Seeing the results I think there is no big performance difference. The new solution is even a little bit faster. A possible explanation for this is that I removed several `toUri()` calls inside the Path class, because we store the Path, Scheme, Authority and Fragment in the Path object. These can also be done outside of the Path class. Another explanation can the network improvement mentioned before or

even the decrease of the GC pauses. Or it may only be just the inaccuracy of the measurement.

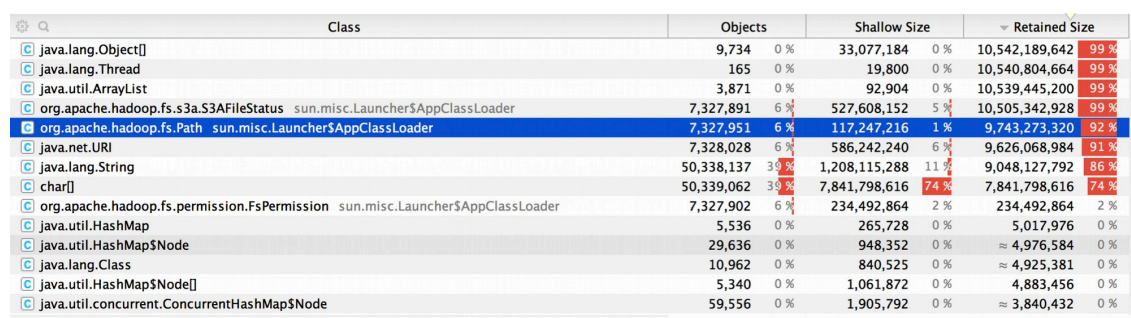
2.3.7 Effects of the change in Path

After publishing the results to the Apache HDFS Jira, I tried to investigate if the change would be really beneficial. These Path objects can be found in every component that uses Hadoop, like Hive, Impala, HBase etc. and used widely everywhere in the code. As a consequence of this, a little CPU overhead can even cause a big performance loss so the patch needs to be tested thoroughly.

If this little change has some possibility to decrease the performance significantly, would it really benefit the components? I did an investigation to see if this issue is real and what would it fixed. Based on the results, my answer is: yes. The following issues was partly or entirely caused by the size of memory these Path objects reserve, so reducing the size with 66% may solve or help these problems.

2.3.7.1 Hive MetaStore OOM

During my investigation I found a HMS memory issue in production use of Hive. The MetaStore server crashed due to an Out Of Memory error. The following heap dump was created before the OOM. Here 9.743 Gigabytes of memory was used by *fs.Path* objects.



Class	Objects	Shallow Size	Retained Size
java.lang.Object[]	9,734	33,077,184	10,542,189,642
java.lang.Thread	165	19,800	10,540,804,664
java.util.ArrayList	3,871	92,904	10,539,445,200
org.apache.hadoop.fs.s3a.S3AFileStatus	7,327,891	527,608,152	10,505,342,928
org.apache.hadoop.fs.Path	7,327,951	117,247,216	9,743,273,320
java.net.URI	7,328,028	586,242,240	9,626,068,984
java.lang.String	50,338,137	1,208,115,288	9,048,127,792
char[]	50,339,062	7,841,798,616	7,841,798,616
org.apache.hadoop.fs.permission.FsPermission	7,327,902	234,492,864	234,492,864
java.util.HashMap	5,536	265,728	5,017,976
java.util.HashMap\$Node	29,636	948,352	4,976,584
java.lang.Class	10,962	840,525	4,925,381
java.util.HashMap\$Node[]	5,340	1,061,872	4,883,456
java.util.concurrent.ConcurrentHashMap\$Node	59,556	1,905,792	3,840,432

Figure 2.10: HMS heap dump before OOM

2.3.7.2 Hive on Spark memory issue

Stress testing of HoS (Hive with Spark execution engine) has revealed that we have memory issues when running with a high value of *spark.executor.cores*. The overhead is mainly introduced by loading multiple copies of HiveConf (Hive configuration

object) and there is a significant overhead when loading lots of Path objects too. In the next chapter I will focus on the multiple HiveConf issue as well.

2.3.7.3 "Small files problem" in HiveServer2

HS2 has a so called "small files problem". Something like *create external table* with lots of small files in the HDFS directory of the table causes HiveServer2 to load the HDFS paths of the files into memory. If we have millions of small files (which is not that rare), these paths can use Gigabytes of heap memory. Hive does a *listFiles* that returns all the Paths, to determine file permissions etc.. So create table hangs and other queries wait a LOT.

2.3.7.4 Apache Impala's solution for the problem

Apache Impala also faced problems caused by the size of memory used by the Paths. Impala is a query engine on top of Hadoop, similar to Hive. They already solved this problem on their side. Impala has a *HdfsPartitionLocationCompressor.java* class which they use to solve exactly this problem. I also thought of the idea to solve the issue in Hive, however for a long term, it may not be the most desirable strategy. Hive would have to always convert the Paths whenever it passes them to Hadoop (Impala does this). I think if the problem can be solved in Hadoop - where it comes from - without a performance overhead, it should be done there. My task is to give the community a proof that CPU overhead will not happen because of the memory fix.

2.3.8 Benchmarking on a data center cluster

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] Barnabas Maidics. Hdfs - 13752: Reduce the memory of path objects. <https://issues.apache.org/jira/browse/HDFS-13752>. [Accessed Nov. 23, 2018].
- [8] Cloudera, Inc. Hive memory limitations. https://www.cloudera.com/documentation/enterprise/5-9-x/topics/admin_hive_tuning.html.
- [9] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on Operating 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>.
- [10] 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>.

- [11] Madhukara Phatak. Secondary namenode. <http://blog.madhukaraphatak.com/secondary-namenode---what-it-really-do/>.
- [12] Mikhail "Misha" Dmitriev. Copyonfirstwriteproperties. <https://issues.apache.org/jira/browse/HIVE-16079>, . [Accessed Nov. 20, 2018].
- [13] Mikhail "Misha" Dmitriev. Jxray. <http://www.jxray.com/>, . [Accessed Nov. 11, 2018].
- [14] Oracle. jcmd - utility. <https://docs.oracle.com/javase/8/docs/technotes/guides/troubleshoot/tooldescr006.html>, . [Accessed Nov. 11, 2018].
- [15] Oracle. jmap - memory map. <https://docs.oracle.com/javase/7/docs/technotes/tools/share/jmap.html>, . [Accessed Nov. 11, 2018].
- [16] Óscar Pereira and Micael Capitão. Mediator framework for inserting data into hadoop, 12 2014.
- [17] Saurabh. Apache yarn. <http://www.hadoopadmin.co.in/hadoop-administrator/apache-yarn/>.
- [18] 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>.
- [19] Steve Loughran. Hadoop 1.0 resource management. <https://wiki.apache.org/hadoop/JobTracker>.
- [20] 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.
- [21] Wikipedia. Apache hadoop. https://en.wikipedia.org/wiki/Apache_Hadoop.