Master's Thesis

HDFS Blocks Placement Strategy

Phokham Nonava

October 2014

Supervisors

Prof. Philippe Cudré-Mauroux
Benoit Perroud, Martin Grund, Djellel Eddine Difallah

eXascale Infolab

VeriSign, Inc.
eXascale Infolab
University of Fribourg

To Sarina Jasmin.

Abstract

We present a blocks placement strategy for the Hadoop Distributed File System (HDFS). HDFS is a core component of Apache Hadoop and provides a virtual file system to client applications. It stores files redundantly across cluster nodes by splitting them into blocks and replicating those according to a replication factor.

Although HDFS distributes blocks reasonably well across cluster nodes, there are cases in which its algorithm creates server hotspots. We show a blocks placement strategy which prevents server hotspots by distributing blocks and their replicas evenly across cluster nodes.

Further, we explore the fact that data centers are not homogeneous anymore. Several hardware generations are now running side by side in server clusters. We show that by introducing a weight factor for hardware generations, we can improve the overall performance of the cluster. This weight factor is used by our blocks placement strategy to place more blocks onto cluster nodes with more processing power.

Finally, we run benchmark applications against the HDFS blocks placement strategy and our blocks placement strategy. We discuss the test cases and show the performance improvement in a small size cluster.

Acknowledgments

During my Master's Thesis, I realized how difficult and time-consuming it is to do research work while having a newborn child at home. There are many people who supported me along this journey.

First and foremost I would like to thank Professor Philippe Cudré-Mauroux for his supervision, encouragement and positive attitude. He was enormously understanding and exceptionally supportive throughout this thesis.

I would also like to thank Benoit Perroud for his extensive support and endless patience. His vast experience about Hadoop was very helpful, and his guidance always pointed me to the right direction.

I thank Martin Grund and Djellel Eddine Difallah for their ideas and discussion. Martin was especially helpful in operating the cluster during the test phase and Djellel gave me useful hints to get started with the project.

I am also deeply thankful for my friends and family. Without their support, this thesis would not exist.

Contents

List of Figures

List of Tables

Listings

1 Introduction

The amount of data created and stored worldwide has exponentially increased over the last decade [ea08]. Today we are surrounded by devices and sensors that record a great deal of information about us and our environment. With the growth of the internet and mobile computing, it is easier than ever to copy and move data from one location to another.

There is a growing demand for new tools from the industry to handle this vast amount of data. Apache Hadoop is one of the upcoming frameworks for storage and large-scale data processing. It consists of two layers at its core. The MapReduce layer provides a programming model for processing large data sets in parallel with a distributed algorithm. The Hadoop Distributed File System (HDFS) layer provides a virtual file system to client applications. In this thesis, we focus on HDFS.

1.1. Motivation

HDFS is a distributed file system which stores files redundantly across cluster nodes for security and availability. To store a file HDFS splits it into blocks and replicates those according to a replication factor. The HDFS default blocks placement policy distributes the blocks reasonably well across the cluster nodes. However, there are cases, in which this algorithm creates server hotspots as shown in chapter 3. Hotspots put unnecessary load onto cluster nodes and lower the overall performance of the cluster. There is already work done to minimize server hotspots [Wan14].

Further by analyzing cluster infrastructures, we see that companies are using different hardware generations in the same cluster. This practice comes from the fact that replacing the whole cluster with a newer hardware generation is costly. So adding a newer hardware generation to an existing cluster is a logical step. However, different hardware generations are not considered by the HDFS default blocks placement policy, and precious computing resources are left unused.

1.2. Goals

In this thesis, we present a blocks placement strategy which addresses both problems. We analyze the HDFS default blocks placement policy and show a blocks placement strategy which prevents server hotspots. The algorithm distributes blocks and their replicas evenly across all cluster nodes.

Additionally we introduce a weight factor for hardware generations. This allows us to describe the processing power of a cluster node by giving it a specific weight factor. Our blocks placement strategy includes this weight factor in its computation and places more blocks on newer hardware generations.

1.3. Outline

This thesis is organized as follows.

Chapter 2 introduces Apache Hadoop in depth. We describe the architecture of Apache Hadoop and see how the two layers interact with each other. Chapter 3 analyzes HDFS and shows how the HDFS default blocks placement policy distributes blocks and their replicas. Chapter 4 presents our blocks placement strategy. It gives a detailed description of our algorithm. Chapter 5 shows some benchmarks. We discuss the results and the impact on performance. Finally, Chapter 6 shows future work and Chapter 7 draws a conclusion. Appendix A describes our balancer tool.

2 Apache Hadoop

The context of this thesis is set in the domain of Big Data. Big Data is a term used to describe data sets that are too large to be processed using traditional data management applications [Wika]. Apache Hadoop was initially created by Doug Cutting in 2005 because he needed a faster data processing framework for the web crawler project called Nutch [Whi09]. Based on the MapReduce paper [DG04] which was published 2004 by Google, Cutting replaced the existing data processing infrastructure with a new implementation and called it Hadoop. At that time Yahoo! decided to invest in the development of Hadoop and agreed with Cutting to make it open source. Today Hadoop is developed under the umbrella of the Apache Software Foundation and is used in many data centers around the globe [Fou].

2.1. Architecture

Apache Hadoop is based on the Google papers [DG04] and [GGL03]. It is "a framework that allows for the distributed processing of large data sets across clusters of computers using simple programming models" [Fou]. Instead of using large scale-up server hardware to process and store data, it uses commodity hardware to scale-out. It is not uncommon for an Hadoop cluster to have hundreds of cluster nodes processing and storing exabytes of data [Hadc].

A large part of Hadoop is written in Java. Critical code is written in C for performance reasons. Hadoop can be run on various Linux distributions and since version 2.2.0 also supports running on Windows.

At its core Hadoop consists of two layers. The computational layer is called MapReduce, and the storage layer is named HDFS. Figure 2.1 shows an overview of the two layers. Both layers are modeled as a master/slave architecture. This greatly simplifies the design, enabling the use of global knowledge for job scheduling and block placement. The disadvantage of this architecture is the master node becoming a single point of failure. In high-availability clusters, an Active/Passive configuration with a hot standby for the master node can be used to minimize downtime.

With an increased number of cluster nodes, there is also an increased chance of a machine failure. Hadoop is explicitly designed with hardware failures in mind. Data stored in the cluster is replicated with a replication factor. This increases both data availability and security. In the case of a node failure, jobs can still access copies of the data on other nodes.

It is important to understand that Hadoop does not provide Online Transaction Processing (although there are components in the Apache Hadoop ecosystem which build on HDFS to provide Online Transaction Processing). By making use of the MapReduce paradigm, Hadoop is best suited for parallelizing jobs using Batch Processing.

2.2. MapReduce

MapReduce is described by Jeffrey Dean and Sanjay Ghemawat as "a programming model and associated implementation for processing and generating large data sets" [DG04]. As the name implies it basically consists of two computational phases.

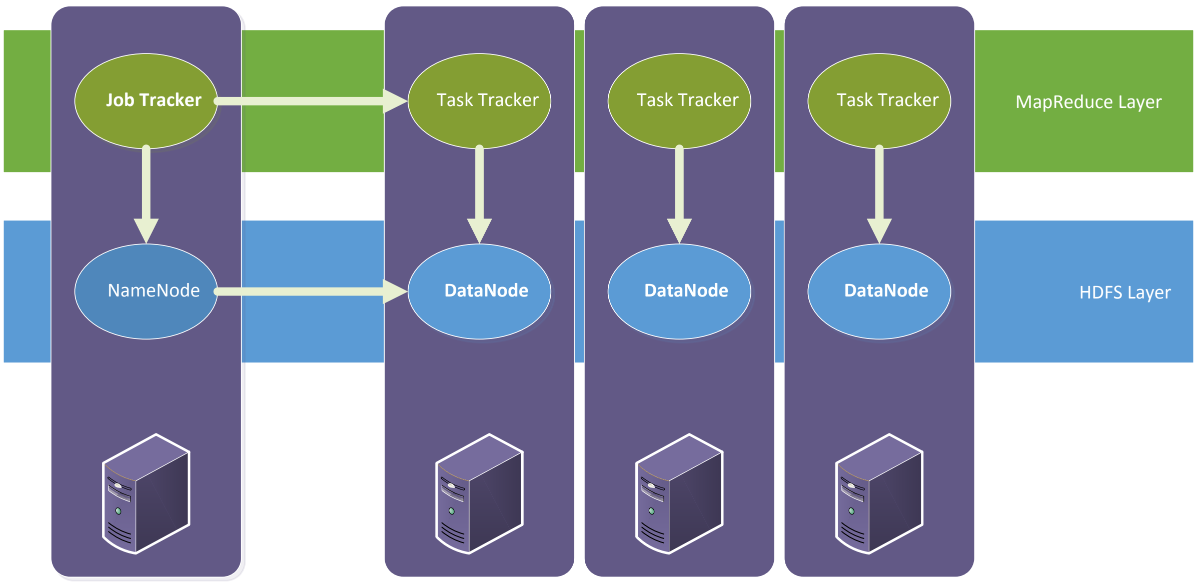
In the map phase, the input data-set is split into chunks and processed in parallel by map tasks. Map tasks are part of a MapReduce job and are being executed on a cluster node. Each map task can output some data-set, which itself forms the input to reduce tasks. Reduce tasks also run on cluster nodes and perform the final computation on the intermediate data-set. Figure 2.2 shows an overview of the MapReduce programming model.

The input and output data-sets of the map and reduce tasks are typically located on HDFS. This allows the MapReduce framework to schedule task operations on cluster nodes where the chunks can be read locally. Instead of moving data to computation as in traditional client/server architectures, moving computation to data results in a very high data throughput. The fact that "computation follows data", plays an important role as we will see in chapter 4.

Since Apache Hadoop 2.2.0, there are two implementations of the MapReduce programming model available [Hada].

Apache Hadoop MapReduce The older Apache Hadoop MapReduce implementation consists of a master component called the JobTracker, and one slave component per cluster node, the TaskTracker (see Figure 2.3).

Figure 2.1.: Hadoop Architecture



The JobTracker is responsible for managing MapReduce jobs. It schedules tasks for execution, monitors their progress and re-executes them in case of a task failure. The TaskTracker simply runs and monitors the tasks as directed by the JobTracker.

Each TaskTracker is constraint by a number of map and reduce task slots. This means that there is a limit on the number of concurrent map and reduce tasks, which can run on a cluster node. If all slots are occupied, no more tasks can be scheduled for execution on this particular cluster node until a task is done, and the slot is available again.

There are utilization issues that can occur if, for example, all map slots are taken and reduce slots are empty. The cluster node would not be running at full performance because the resource limit is calculated with all map and reduce slots occupied.

This issue and the wish to support other data processing models have lead to a new implementation of the MapReduce programming model.

Apache Hadoop Yet-Another-Resource-Negotiator (YARN) To overcome the limitations of the old Apache Hadoop MapReduce implementation, YARN splits the two major responsibilities of the JobTracker into two separate daemons.

The global ResourceManager arbitrates resources among all applications in the system and the per-application ApplicationManager negotiates resources from the ResourceManager. The ApplicationManager works with the per-node NodeManager to execute and monitor the tasks (see Figure 2.4).

Figure 2.2.: MapReduce Overview

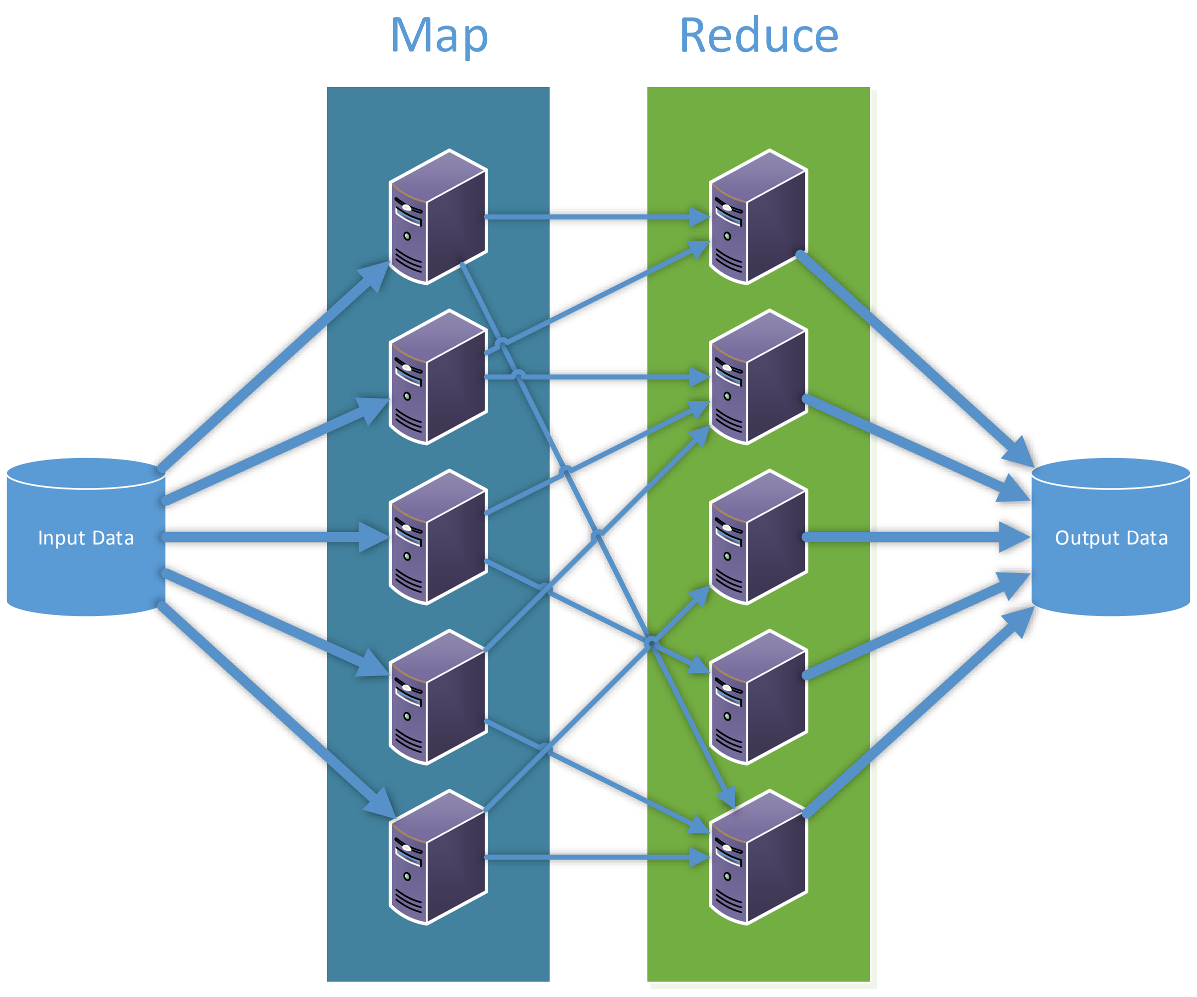


Figure 2.3.: MapReduce1 Architecture (Source [Mur])

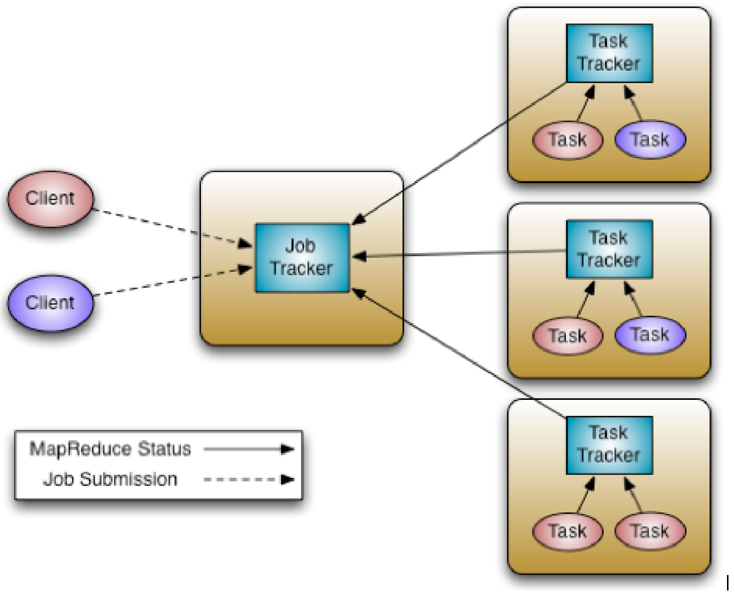
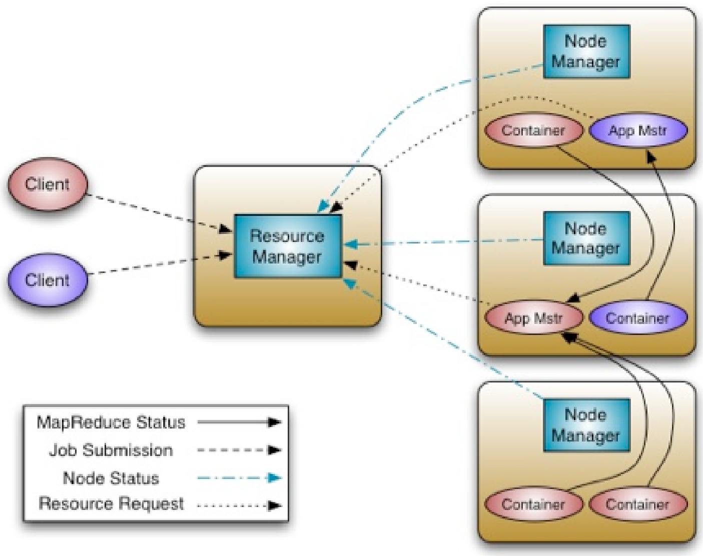


Figure 2.4.: YARN Architecture (Source [Mur])



3 Hadoop Distributed File System

In this chapter, we describe HDFS in depth and explain some of the drawbacks of its blocks placement strategy.

3.1. Concepts and Design

HDFS is a distributed file system designed to store very large files reliably across cluster nodes using commodity hardware. As already mentioned in chapter 2 HDFS uses a master/slave architecture, which greatly simplifies the design. We explain some of the HDFS features to have a better understanding of its inner workings.

3.1.1. Commodity Hardware

Using commodity hardware to scale-out is a cost-effective way to add more computational power and storage space to a cluster. However with increasing number of cluster nodes, the chance of a node failure increases as well. HDFS is explicitly designed with hardware failures in mind. There is no need to bring the cluster down for maintenance. HDFS takes care of data consistency and availability in case a cluster node has to be replaced.

3.1.2. File System

HDFS provides a virtual file system to client applications, with the ability to store very large files. It is not uncommon for an Hadoop cluster to store files with terabytes in size. HDFS is similar to other file systems meaning that it offers the concept of directories and files to structure data storage. However, it has several features that optimize its operation on very large data sets.

Streaming Data Access Although HDFS provides read and write operations on files, it is built around the idea that data is written once and read many times. Once a file is written it cannot be modified. Data can only be appended. This concept comes from the fact that data will, usually, be copied into the Hadoop cluster and analyzed many times. Read throughput plays a bigger role in this scenario. Users, usually, want to analyze the whole data set. Delivering a high throughput is more important than low latency on data access.

Block Size The concept of blocks can be found in many file systems. Usually, file system blocks have a size in kilobytes. This is the smallest unit which can be loaded into memory in one read operation. In HDFS, this concept can also be found. However as we are dealing with very large files, the default block size is 128MB. A block in HDFS is the smallest replication unit. A file is split into blocks during the write operation and distributed across cluster nodes. Also depending on the client application, a block is, usually, the data unit on which an application copy operates on. This fact plays an important role in our blocks placement strategy as we will see in chapter 4. By strategically placing blocks in the Hadoop cluster, we can guide the execution of the MapReduce layer.

Data Integrity Data can get corrupted because of data degradation [Wikb], storage hardware faults, network faults or software bugs. To protect data against these types of error, HDFS saves a checksum for every block stored in the cluster. For every block a client application reads, HDFS compares its checksum against the saved value. If the values differ, HDFS marks the block as invalid and fetches a replica from another cluster node. Eventually, the faulty block gets replaced by a valid replica.

3.1.3. NameNode and DataNodes

In an Hadoop cluster, there are two kinds of nodes. The master node is called NameNode, and the slave nodes are called DataNodes. Both node types are fully interconnected and communicate using TCP. The following paragraphs describe the role of both node types in-depth.

NameNode The NameNode manages the file system namespace and provides an entry point to HDFS for client applications. It provides access control to files and directories and also controls the mapping of blocks to DataNodes. Files and directories are represented by inodes which reccord attributes like permissions, modification and access time.

The inodes and the list of blocks are called the image. The NameNode periodically persists the image to the file system as a checkpoint. Additionally, every change to HDFS is recorded in a write-ahead log called the journal. During startup the NameNode initializes the namespace image from the checkpoint, and then replays changes from the journal.

In an Hadoop cluster, there exists only one active NameNode. To provide fault tolerance in case of a failure, a Secondary NameNode can be deployed on standby. It replicates the image and journal from the master NameNode and merges them as a new checkpoint. If the master NameNode goes offline, the Secondary NameNode can be promoted as the master.

DataNode DataNodes store data blocks of all files. When a DataNode joins a cluster, it records the namespace ID and the assigned storage ID. During startup, a DataNode registers itself with the NameNode using both IDs and then transmits a list of block replicas it owns, as a block report. Further block reports are sent hourly to the NameNode to update the global block list.

To confirm normal operation of a DataNode, it sends a heartbeat every three seconds to the NameNode. If there is no heartbeat within 10 minutes, the NameNode considers the DataNode offline and its blocks as unavailable. To maintain the replication factor, the NameNode eventually will trigger the creation of new replicas on other DataNodes.

3.2. Blocks Placement Strategy

As we already mentioned before, HDFS splits files into blocks before sending them to DataNodes for storage. For security and performance reasons it also replicates a block according to a replication factor. By default, a replication factor of 3 is used. This basically means that the storage requirement for a file is increased by the replication factor. However, in the domain of Big Data, storage space is not an issue and is basically considered as infinite. In the following subsections, we have a closer look at how a block is distributed over nodes in the cluster.

3.2.1. Algorithm

If a client wants to write a file into the cluster for storage, the HDFS layer first looks up whether the client is part of the cluster itself. If it is a node of the cluster, the client places the block locally on itself. Otherwise, it places the block on a random node in the cluster.

For the second block replica, a random node outside the rack of the first replica is chosen. This is done for security reasons. If a rack related to the node of one replica goes offline, HDFS is still able to retrieve another replica from a different rack.

For the third block replica a random node in the same rack as the second replica is chosen. This minimizes inter-rack traffic and is a tradeoff between security and network traffic.

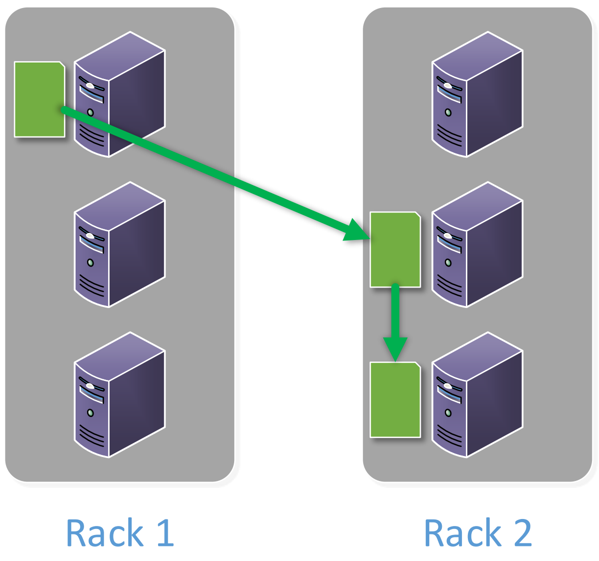
For further block replicas, a random node in the cluster is chosen. Figure 3.1 shows how the block replicas get written.

3.2.2. Pipelining

To minimize inter-node network traffic HDFS uses a technique called pipelining. When a client writes the first block replica to a node, this node is then responsible for writing the second replica to a random off-rack node. Further, this off-rack node is itself responsible for writing the third replica to a random in-rack node.

The result of this pipelining is that client-cluster network traffic is reduced because the client only writes one replica, instead of all three replicas. Additionally, inter-rack traffic is also reduced as the node holding the second replica writes to a random in-rack node and does not cross rack-boundary.

Figure 3.1.: HDFS default blocks placement policy



4 Hadoop Data Placement Strategy

In this chapter, we present our blocks placement strategy and show how it is an improvement to the HDFS blocks placement strategy.

4.1. Goals

Hadoop Data Placement Strategy (Hadaps) has two main goals which are done in the same algorithm. First, it tries to evenly distribute blocks across all cluster nodes. Second, it tries to move more blocks onto hardware generations with more processing power. We discuss these properties in depth in the following subsections.

4.1.1. Even Distribution

As we have seen in chapter 3, HDFS always places the first block replicas onto the writer node if the node is in the cluster. This creates a very unbalanced block placement if the writer node does not leverage MapReduce for writing data. All first block replica (and therefore the whole file) would be placed onto the writer node. The second and third replica end up being distributed randomly across the cluster. This property makes the writer node a hotspot as we see in chapter 5.

We solve this problem by evenly distribute all blocks of a file across DataNodes. This way, we activate all cluster nodes to participate in the computation. This simple idea is the first step of our algorithm.

4.1.2. Hardware Generations

By analyzing the MapReduce layer of Hadoop, we can see that it tries to execute application copies on cluster nodes that have the required data locally available. "Moving Computation is Cheaper than Moving Data" [Hadb] perfectly makes sense in the domain of Big Data, because reading data locally is generally much faster than reading it over the network.

Let's assume that we have a CPU bound application and two hardware generations of which the newer hardware generation has twice as much processing power. We can state that an application copy running on the newer hardware generation will execute twice as fast as an application copy running on the older hardware generation. Or stated differently,

Definition. *An application copy running on a newer hardware generation with twice as much processing power can process twice as much data blocks than an application copy running on an older hardware generation.*

The same definition holds for an I/O bound application. If the disk of a newer hardware generation is twice as fast as the disk of an older hardware generation, an application copy running on the newer hardware generation will process twice as much data blocks as an application copy running on an older hardware generation.

This observation leads us to the following strategy. By placing more data blocks onto newer hardware generation nodes, the performance of an application is expected to increase. Let the weight factor denote the processing power of a hardware generation. Let the quota be the maximum number of blocks of a file for a node. For a specific file with number of blocks n, we can calculate the quota of each node as follows:

$$quota_{node} = \lceil n * \frac{weight\_factor_{node}}{weight\_factor_{total}} \rceil \tag{4.1}$$

Example Lets assume we have a cluster consisting of two hardware generations. The newer hardware generation has a weight factor of 2, and the older hardware generation has a weight factor of 1. There are 6 cluster nodes of which 3 are of the newer hardware generation, and 3 are of the older hardware generation. For a specific file with ten blocks we can calculate the quota of the newer hardware generation:

$$quota_{newer} = \lceil 10 * \frac{2}{1*3+2*3} \rceil = 3 \tag{4.2}$$

And for the older hardware generation we have:

$$quota_{older} = \lceil 10 * \frac{1}{1*3+2*3} \rceil = 2 \tag{4.3}$$

So on the newer hardware generation nodes, a maximum of three blocks can be placed, whereas on the older hardware generation nodes a maximum of two blocks can be stored.

4.2. Algorithm

The algorithm of Hadaps is different from the default HDFS blocks placement strategy because it works on file level instead of the global block list. For a specific file with replication factor r the algorithm works as follows:

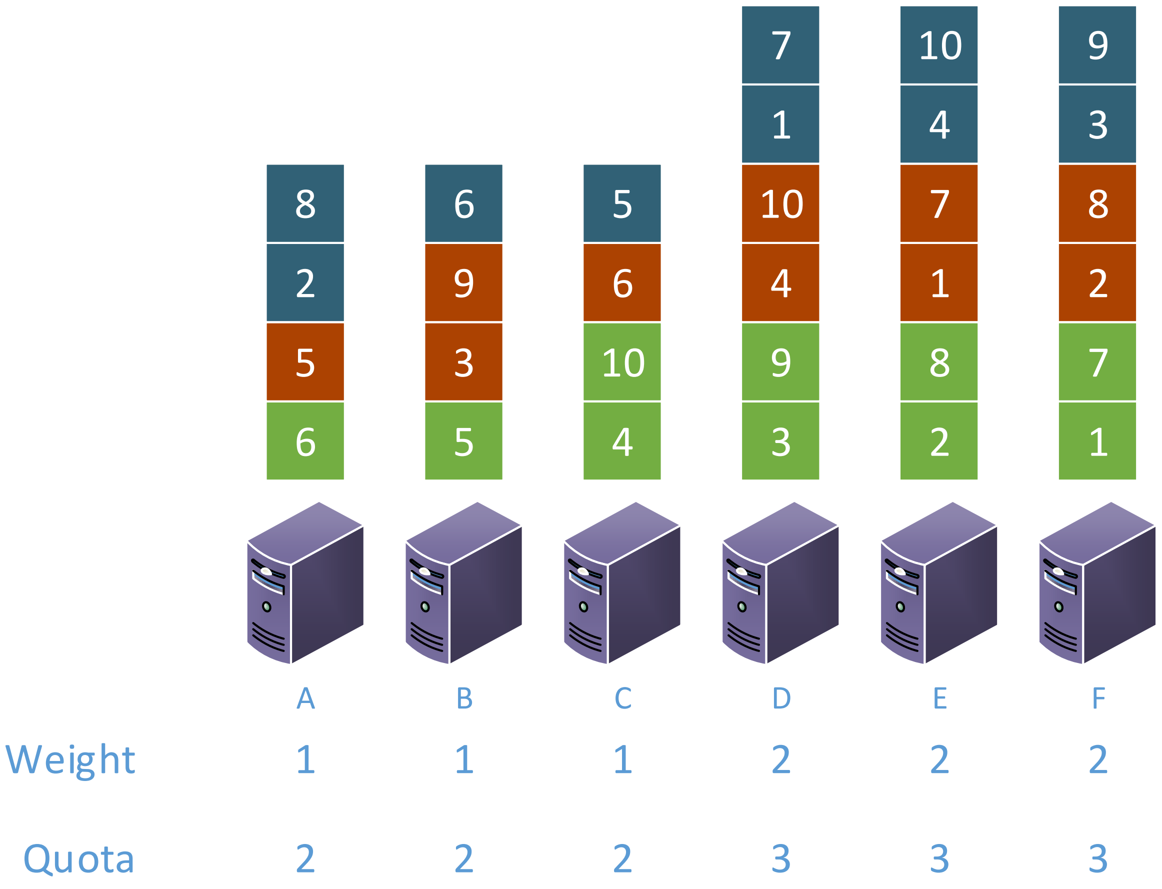
- Enumerate all blocks for file f
- Enumerate all nodes for the cluster
- Calculate the quota for each node based on the weight factor
- Sort nodes by quota (higher quotas come first)
- Sort nodes with equal quota by disk utilization (lower utilization nodes come first)
- For each file replica do the following:
  - Initialize the $block\_list_{queue}$ with all blocks of file f
  - Initialize the $node\_list_{available}$ with all nodes
  - For each block in $block\_list_{queue}$ do the following:
    - Find the first node in $node\_list_{available}$ that does not contain a replica of this block and did not reach its quota.
    - Put the block onto this node and remove it from $block\_list_{queue}$
    - Remove the node found from $node\_list_{available}$
    - If $node\_list_{available}$ is empty, initialize it with all nodes

Example Lets continue our example from the subsection 4.1.2. Figure 4.1 shows the final blocks placement for our cluster.

In the beginning, we have an empty cluster. So our sorted node list is equal to {F,E,D,C,B,A}. Remember, nodes with higher quotas come first. We initialize our block list to {1,2,3,4,5,6,7,8,9,10}. For the first file replica (green) we just evenly distribute the blocks over all nodes.

For the second file replica (red), we initialize our sorted node list to {F,E,D,B,A,C}. Remember, nodes with higher quotas come first ({F,E,D}) and for nodes with equal quota, the algorithm uses lower utilization nodes first ({B,A}). Block 1 cannot be stored on node F because node F already has block 1 from the first replica. So block 1 is put onto node E. For block 2, node F is an available node. Block 3 cannot be put onto the next node in the list which is node D, as it already contains block 3 from the first replica. So block 3 is put onto the next available node in the list which is node B. The rest of the blocks can be placed in the same way.

Figure 4.1.: Hadaps Algorithm



5
Benchmarks

In this chapter, we run two benchmark applications against the HDFS default blocks placement policy, the HDFS standard balancer and our Hadaps balancer.

5.1. Preparation and Setup

To be able to compare the impact of the three block placement algorithms against each other, we use TeraSort, which is included in the Hadoop distribution and HadapsTest, an additional benchmark, we developed for our Hadaps balancer.

The tests have been executed on our cluster located at the University of Fribourg. It consists of 15 hosts, of which 11 are of the older generation and 4 are of the newer generation. The table 5.1 shows the hardware specification of both generations.

We use Ubuntu LTS and Cloudera CDH 5.0 as runtime stack. During our tests only HDFS and YARN have been started as a service. The host diufpc56 was configured as NameNode. To disable the influence of caching we flushed the hard disks and cleared the OS cache before each test iteration using the following command.

```
$ sync
$ sysctl vm.drop_caches=3
```

Additionally, we disable the block cache on DataNodes. Replication factor for all generated files was left at three replicas and for the block size we were using the default value of 128MiB.

Data for the tests have been loaded into HDFS from the node diufpc56. Before loading test data into HDFS, we wanted to have a clean, balanced state of our cluster. So we

Table 5.1.: Cluster Hardware Generations

Component	Old Generation	New Generation
CPU	i7-2600 - 2 core	i7-4770 - 4 core
RAM	32 GB	32 GB
Harddisks	1 x 500 GB	2 x 500 GB
NIC	1 GbE	1 GbE
DataNodes	diufpc301 diufpc302 diufpc303 diufpc310 diufpc311 diufpc312 diufpc313 diufpc314 diufpc315 diufpc316 diufpc317	diufpc54 diufpc55 diufpc56 diufpc57

ran the HDFS standard balancer with a threshold of 10GB.

After this initial balancing we loaded the test data for each test into the cluster. This initial write triggered the block placement using the HDFS default blocks placement policy. To test this policy we ran the two benchmarks and gathered execution time as well as other important metrics.

The second step was to run the HDFS standard balancer to balance the initial block placement. After balancing, we ran the two benchmarks again to see the improvements over the HDFS default blocks placement policy.

The third step was to run our Hadaps balancer with a weight factor of 2 over the test data. Finally, after balancing, we ran the two benchmarks again to see whether there is an improvement over the HDFS default blocks placement policy and the HDFS standard balancer blocks placement strategy. The Figure 5.1 shows the whole test cycle.

After the first test cycle, we noticed that a weight factor of 2 was probably not reflecting the difference of the hardware generations appropriately. We decided to decrease the weight to 1.2 to better match the minor increase of disk performance of the newer hardware generation.

The following sections discuss each test in depth.

5.2. TeraSort

TeraSort is a well-known Hadoop benchmark written by Owen O'Malley. It basically tries to sort 1 TB (or any amount) of data as fast as possible. In 2008, he reported having sorted 1 TB of data in 209 seconds using a cluster of 910 nodes [Owe08]. TeraSort consists of 3 modules, which we will describe in the following paragraphs.

TeraGen generates the official GraySort [Nyb] input data set. It uses MapReduce to write data into the cluster. A data row consists of 100 bytes and is the smallest amount of data to be sorted.

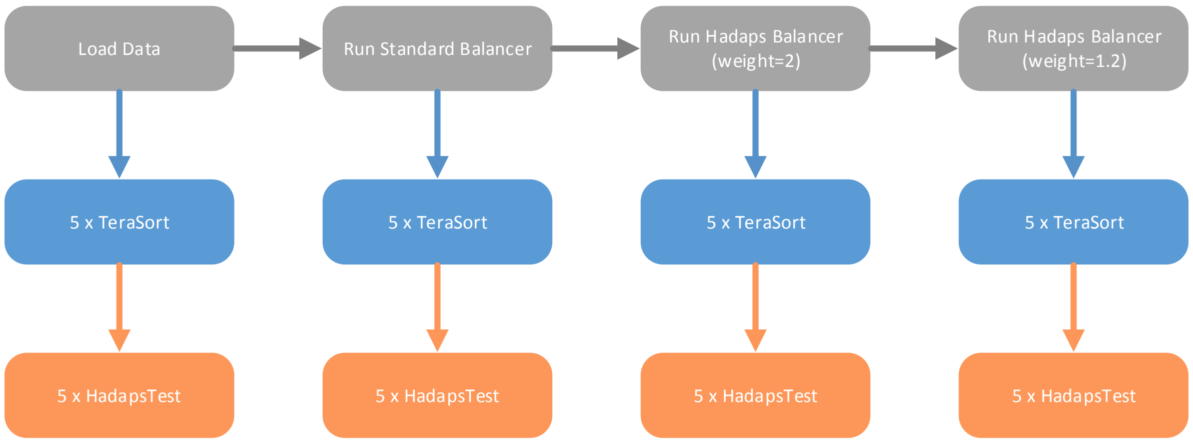
TeraSort itself performs the sort of the generated data set. It also leverages MapReduce to sort the data in parallel.

TeraValidate performs the validation of the sorted data set. It ensures that the output from TeraSort is globally sorted.

5.2.1. Generating data

For generating data, we use the following command.

Figure 5.1.: Test Cycle



```
$ hadoop jar
  /opt/cloudera/parcels/CDH/lib/hadoop-0.20-mapreduce/
  hadoop-examples.jar teragen
  1000000000
  /terasort/terasort-input
```

As we see, we generate a data set of 100GB to be sorted and store it in /terasort/terasort-input.

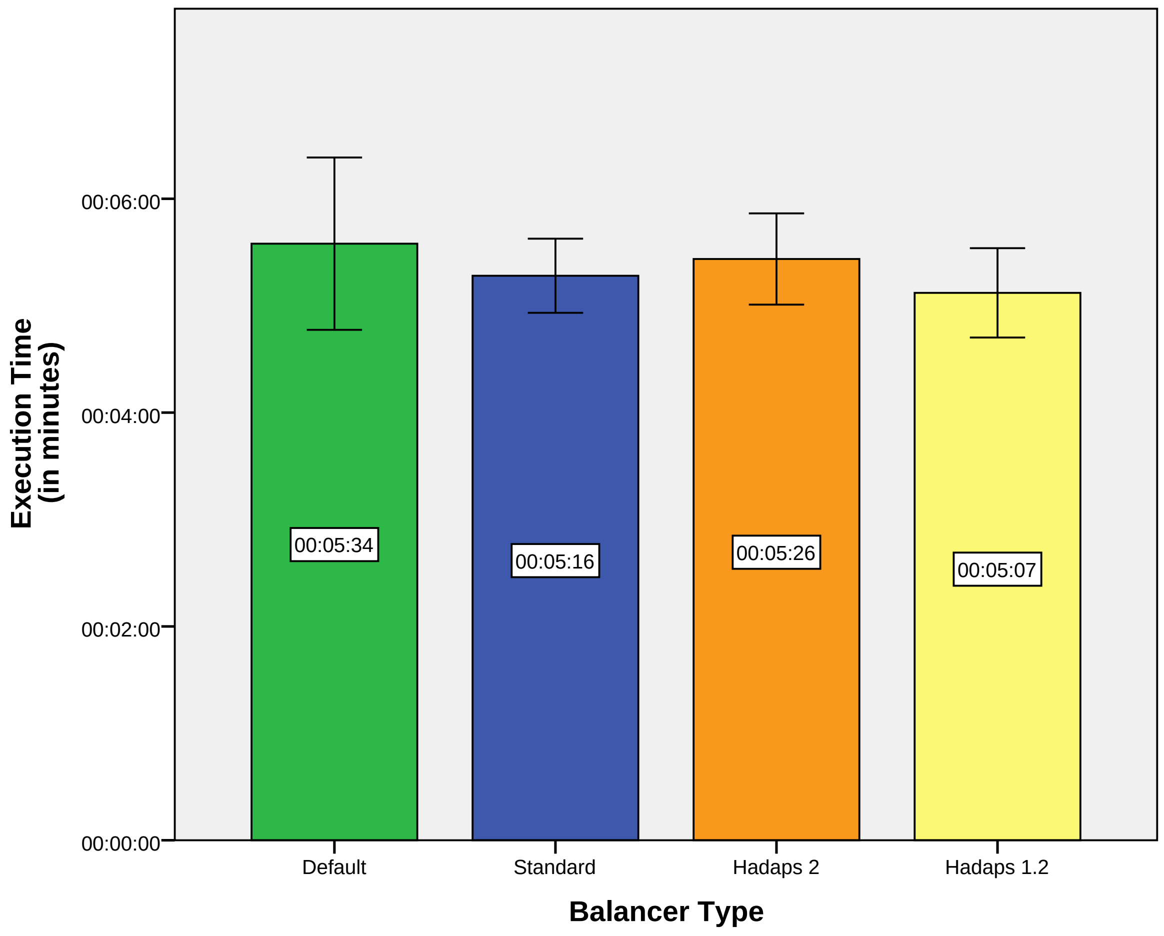
5.2.2. Results

The benchmark was run using the following command.

```
$ hadoop jar
  /opt/cloudera/parcels/CDH/lib/hadoop-0.20-mapreduce/
  hadoop-examples.jar terasort
  /terasort/terasort-input
  /terasort/terasort-output
```

TeraSort takes the input data set from TeraGen and writes the sorted data into /terasort/terasort-output. Figure 5.2 shows the execution time of TeraSort after the initial block placement with the HDFS default blocks placement policy and after rebalancing with the HDFS standard balancer and the Hadaps balancer.

Figure 5.2.: TeraSort Execution Time



As we can see, TeraSort performs equally well after all four block placement policies. To understand the reason behind this, we have to take a closer look at TeraGen.

TeraGen uses MapReduce to write the data set into the cluster. As we described in chapter 3, data is written first onto the local node, then onto a random node in another rack and finally onto a random node in the same remote rack. By using MapReduce, TeraGen produces a well-balanced random distribution of blocks in the cluster already. Rearranging those blocks with the HDFS standard balancer and the Hadaps balancer has no effect on the performance as all nodes are already participating in delivering data to TeraSort.

Additionally, we can observe that the impact of the hardware generation is not as big as we hoped for. TeraSort is an I/O bound application, and therefore disk speed is a key factor. Figure 5.3 and Figure 5.4 show the disk read performance during the execution of TeraSort after rebalancing with our Hadaps balancer for a weight factor of 1.2 and 2 respectively. As we can see, read throughput for both weight factors is approximately the same around 50 MB/s.

These two observations lead us to the conclusion that the random block placement produced by TeraGen is already adequate for high performance and that the difference in disk speed for both hardware generations are not sufficient to make an distinction in performance.

5.3. HadapsTest

The results from the TeraSort benchmark, lead us to develop our own benchmark called HadapsTest to test the effectiveness of our Hadaps balancer. While TeraSort produces test data using the MapReduce framework, HadapsTest writes from a single node. This results in the test data being very unbalanced. We see from the HDFS default blocks placement policy that the first replica of each block will be placed on this writer node. The second and third replica will be uniformly distributed over the remaining DataNodes in the cluster. Thus reading a block will most likely result in a request to the writer node as it is the first one in the node list.

5.3.1. Generating data

For generating data, we use the following command. For a complete usage of HadapsTest, please refer to the subsection A.2.2.

```
$ hadoop jar hadoop-hdfs-hadaps-test-2.3.0-cdh5.0.0.jar
  org.apache.hadoop.hadaps.HadapsTest write
```

Figure 5.3.: Disk Throughput during TeraSort after Hadaps Balancer (weight = 1.2)

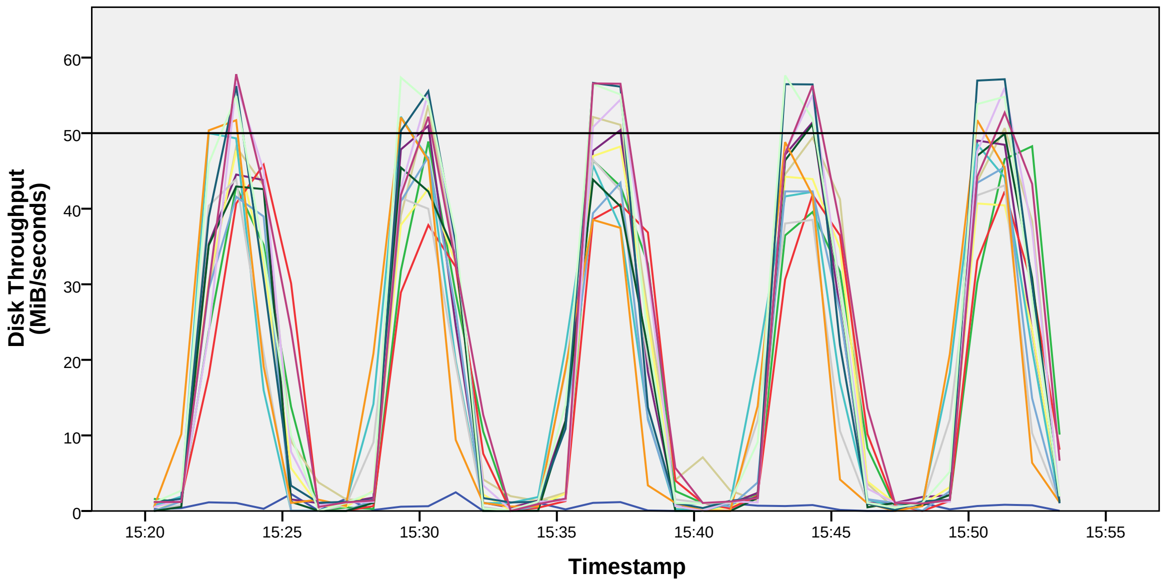
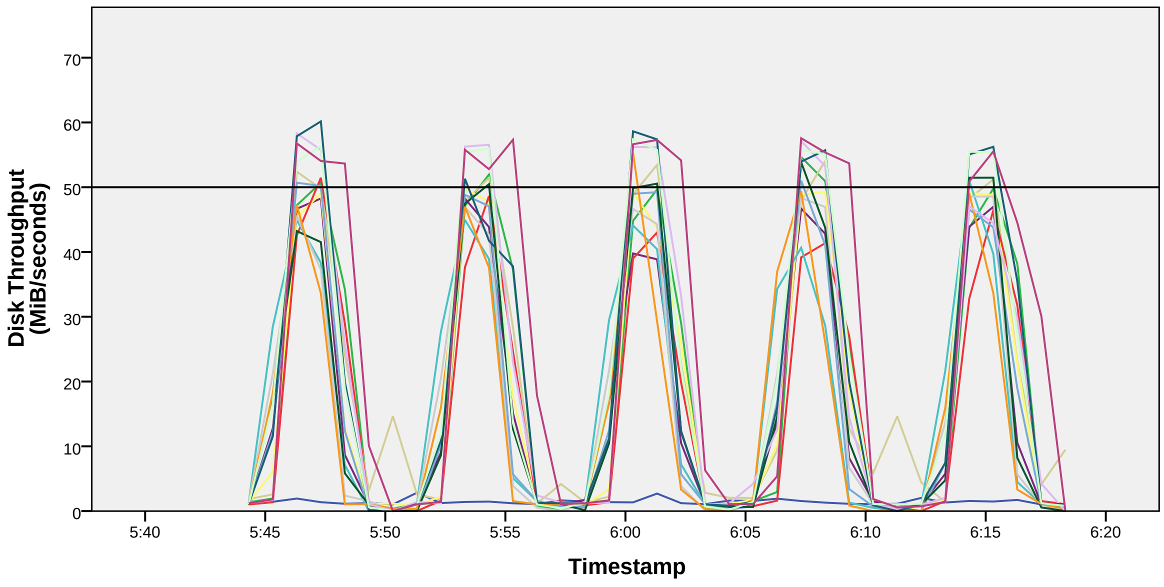


Figure 5.4.: Disk Throughput during TeraSort after Hadaps Balancer (weight = 2)



```
    -out /hadapstest/data
    -count 15
    -size 13000:14000
```

We write 15 files into the directory /hadapstest/data. Each file has a random size between 13GiB and 14GiB.

5.3.2. Results

The benchmark was run using the following command.

```
  $ hadoop jar hadoop-hdfs-hadaps-test-2.3.0-cdh5.0.0.jar
    org.apache.hadoop.hadaps.HadapsTest read
    -in /hadapstest/data
    -out /hadapstest/output
```

Figure 5.5 shows the execution time of the benchmark after the initial block placement with the HDFS default blocks placement policy and after rebalancing with the HDFS standard balancer and the Hadaps balancers.

As we can see rebalancing with the HDFS standard balancer does not decrease the execution time of HadapsTest much. The benchmark finished around the 24 minutes mark for both the HDFS default blocks placement policy and the HDFS standard balancer. However balancing with our Hadaps balancer decreased the execution time of the benchmark to 7:47 minutes for the weight of 2 and to 6:50 for a weight of 1.2. By rearranging the blocks with Hadaps, we could improve the execution time by a factor of 3.

To see the reason of this improvement, we have to look at the network traffic during the benchmark. Figure 5.6 shows the data transmission of all DataNodes during HadapsTest after the blocks have been balanced with the HDFS standard balancer.

We see that the DataNode diufpc56 transmits most of the data to other nodes. The transmission rate is around 125MB/s which means that the outgoing traffic is limited by the network interface bandwidth (1Gbit/s).

To understand why every client requests blocks from the node diufpc56, we have to understand in which order the DataNodes holding the block replica are being returned from the NameNode. When a client requests a list of DataNodes for a block, the NameNode sorts the final list with the method NetworkTopology.pseudoSortByDistance(). The client then uses the first entry in this list to request a block replica and proceeds to

Figure 5.5.: HadapsTest Execution Time

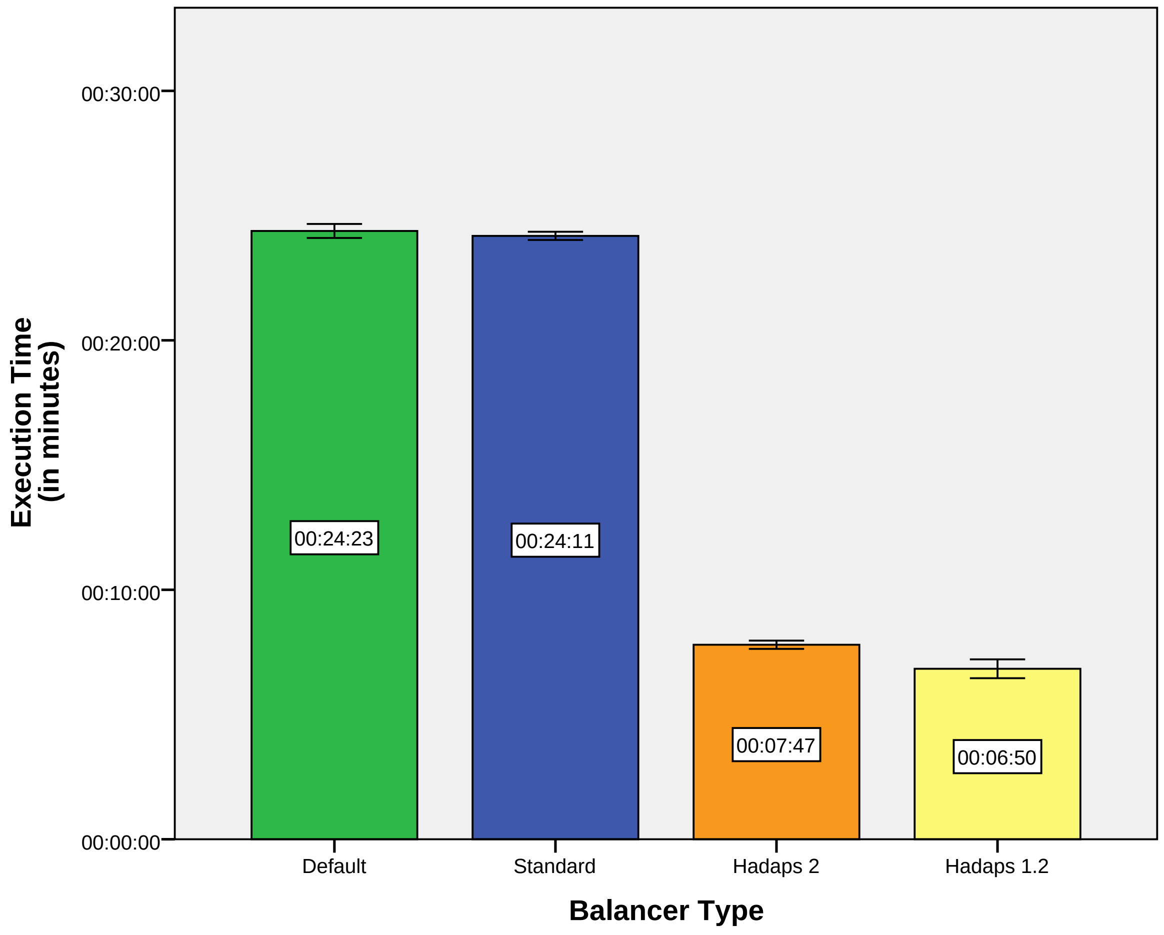
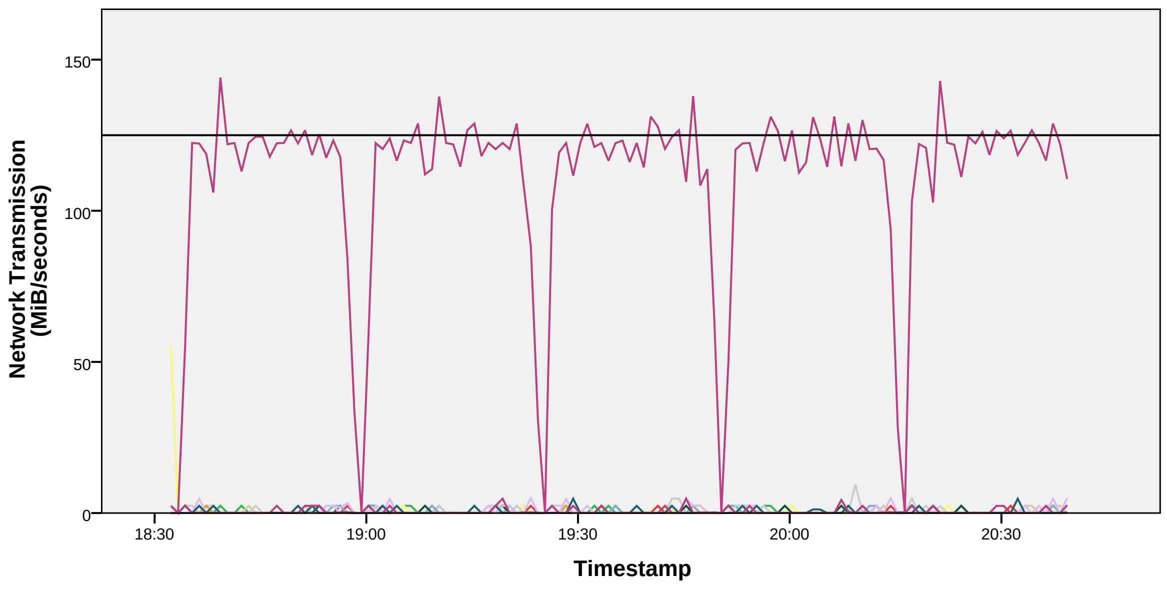


Figure 5.6.: Data Transmission during HadapsTest after Standard Balancer



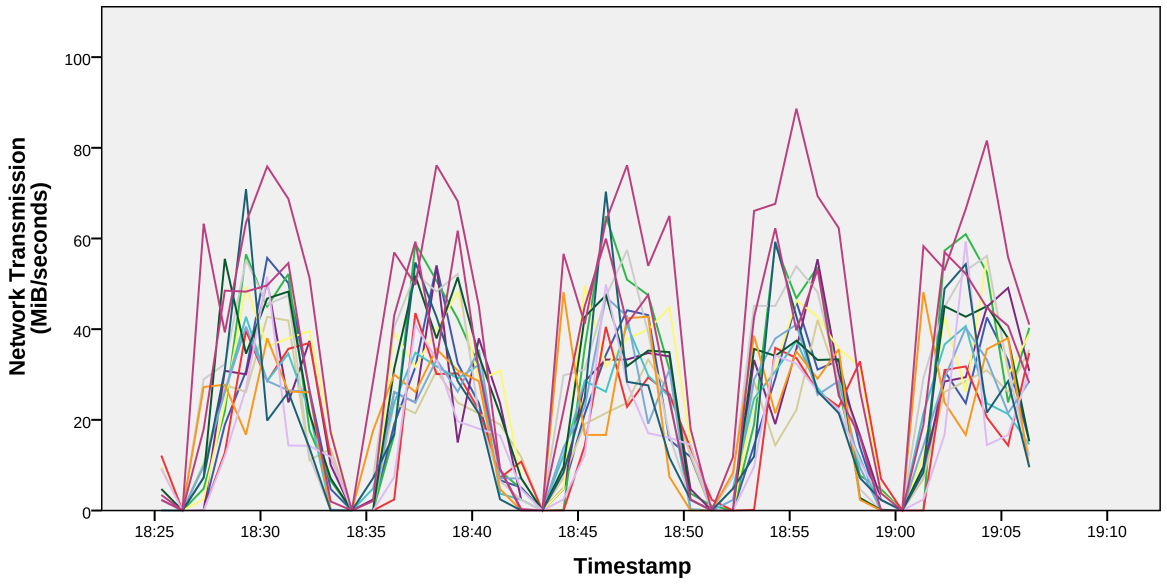
the next entry if it is unable to get the block. NetworkTopology.pseudoSortByDistance() pushes the local DataNode to the front if there is a replica on the same node as the client. Otherwise, it chooses the first entry in the list which is in the same rack. If there is no rack-local DataNode, it chooses a random DataNode from a remote rack.

The important fact to notice is that for rack-local DataNodes, the algorithm always chooses the first matching entry in the list. Because we have only one rack configured in our setup the DataNode diufpc56 is always being pushed to the front of the list. There is a JIRA issue which modifies the method to return a random DataNode from all rack-local nodes holding a block replica [Wan14].

The solution to this situation is either to fix NetworkTopology.pseudoSortByDistance() or to distribute the block replicas over all nodes. The latter is being done by our Hadaps balancer. Figure 5.7 shows how the redistribution activates all DataNodes. All DataNodes are now getting requests for blocks.

Again, the difference of hardware generations has a minor impact on performance. We can see that balancing width a weight factor of 1.2 or 2 does not improve read throughput much. Had we chosen hardware generations with a greater difference in disk speed, we would have observed a bigger difference in execution time.

Figure 5.7.: Data Transmission during HadapsTest after Hadaps Balancer (weight = 1.2)



6 Future Work

In this chapter, we show what improvements could be done in the Hadaps blocks placement strategy as well as in the Hadaps balancer.

Rack Awareness The Hadaps blocks placement strategy is currently not rack-aware. This means that it does not provide the former security level of the HDFS default blocks placement policy. However, extending the algorithm with rack awareness should be straightforward, as we have already several block placement constraints in place. Adding rack awareness as another constraint, modifies the number of optimal cluster nodes for a block. In the case the algorithm has no optimal cluster node, carefully removing one constraint after another should allow it to select a suboptimal cluster node.

Cgroups We could not fully test the behavior of a cluster containing hardware generations with great difference in processing power. The disk speed of our older and newer hardware generation was too similar. However by using cgroups [Men], the processing power of one hardware generation could be limited by a greater factor. Rerunning the benchmarks with this setup should show a bigger performance improvement.

HDFS Encryption The Hadaps balancer tool currently does not support HDFS encryption. However, it could be added with reasonable effort. We would have to keep track of the key changes used for the encryption and use those keys to communicate the block movement to cluster nodes.

BlockPlacementPolicy Framework The Hadaps blocks placement strategy is currently implemented as an external balancer tool. This has the advantage of leaving the hot code of the HDFS default blocks placement policy untouched. We were able to experiment with different algorithms and see their effect on cluster performance.

As the algorithm is getting more and more mature, it would be nice to hook into the BlockPlacementPolicy framework and use the Hadaps blocks placement strategy directly for initial blocks placement. However as we see today, the framework API would need some heavy modifications to support Hadaps.

7 Conclusion

HDFS is an important core component of Apache Hadoop. Not only does it simply store data in a virtual file system, HDFS also greatly affects and guides the MapReduce layer. By carefully placing blocks in the cluster, we can improve the performance of HDFS and thus also the overall performance of Apache Hadoop. This thesis contributes to the research of optimal data placement in large-scale server clusters.

Activating all cluster nodes plays an important role in a distributed computation framework. We show that distributing blocks and their replicas evenly across cluster nodes, improves the read performance of HDFS. The MapReduce layer is able to place a greater number of application copies onto cluster nodes with data locally available.

We also introduce a weight factor for hardware generations. A higher weight factor is assigned to hardware generations with greater processing power. We show that by moving blocks onto cluster nodes with higher weight factor, the overall performance of the cluster is improved.

We have conducted benchmarks on a small server cluster to test our Hadaps blocks placement strategy. The results showed a great improvement in read performance after rebalancing an unbalanced cluster with our Hadaps balancer.

In conclusion, we believe that our Hadaps blocks placement strategy is an improvement over the HDFS default blocks placement policy and that our Hadaps balancer yield a greater performance over the HDFS standard balancer.

A
Hadaps Tools Usage

In this chapter, we show how to build the Hadaps balancer and the benchmark tools and how to execute them from the command line.

A.1. Building

The source code for Hadaps is currently located on a private Git [Git] repository on Bitbucket [Atl]. Hadaps is developed based on the Apache Hadoop 2.3 release branch and has also been ported for the Cloudera CDH 5.0 release branch found on GitHub. There are three branches available in the repository. The master branch is where all development is done. It contains all commits and files which are used during development. This branch is based on Apache Hadoop 2.3. The apache and cloudera feature branches contain only the code to be released squashed into one commit. They are based on Apache Hadoop 2.3 for the apache branch and Cloudera CDH 5.0 for the cloudera branch respectively. We have successfully built the master branch using Windows and Linux. For the apache and cloudera branches, the build procedure has been tested only on Ubuntu LTS.

Getting the source code We assume that you have read access to the repository. Getting the source code is straightforward using the git clone command.

```
$ git clone git@bitbucket.org:fluxroot/hadaps.git
```

Git will download the whole repository containing the full Apache Hadoop 2.3 source code, the full Cloudera CDH 5.0 source code and all Hadaps modifications.

Setup the build environment Hadaps is integrated into the Hadoop build process and has no additional dependencies. You can use the same toolchain as for Hadoop. For a guide on how to setup a build environment for Hadoop, have a look at the file BUILDING.txt at the root of the source code.

Compilation The Hadaps source code adds two additional Maven projects to the tools project. The Hadaps balancer code is located under the `hadoop-hadaps` directory, and the benchmark code can be found under the `hadoop-hadaps-test` directory. Both projects are built as part of the Hadoop build process. To create the distribution archive you can execute the following command.

```
$ mvn clean package -Pdist -DskipTests -Dtar
```

Maven will build the Hadoop distribution archive. If everything has been successfully built, you should see the following output at the end of the compilation.

```
...
[INFO] Reactor Summary:
...
[INFO] Hadoop Data Placement Strategy ................ SUCCESS [1.502s]
[INFO] Hadoop Data Placement Strategy Test ........... SUCCESS [1.576s]
...
[INFO] BUILD SUCCESS
...
```

The distribution archive can be found at hadoop-dist/target/hadoop-⟨version⟩.tar.gz.

A.2. Usage

Hadaps has been fully integrated into the Hadoop distribution. You can install the distribution archive as you normally would. For a guide on how to install and setup Hadoop, please follow the Hadoop documentation.

The Hadaps balancer and benchmark JAR files can be found in the directory `share/hadoop/tools/lib`. You can execute them as you normally would, with the hadoop jar command. The following sections describe the commands in detail.

A.2.1. Hadaps Balancer

The Hadaps balancer needs a configuration file which defines various parameters like generations and weights. You will find an empty XML configuration file called `hadaps.xml` in the `etc/hadoop` configuration directory. The table A.1 lists all available parameters for the configuration file. For a complete example of the configuration file, please have a look at listing A.1.

To run the Hadaps balancer from the command line, you can execute either the JAR file with the hadoop jar command or you can just call the hdfs command. The following command executes the JAR file manually.

```
$ hadoop jar share/hadoop/tools/lib/hadoop-hadaps-<version>.jar
  org.apache.hadoop.hadaps.Hadaps
```

As the Hadaps balancer is integrated into the hdfs command, you can also just execute the following command.

```
$ hdfs hadaps
```

This is the preferred way to run the Hadaps balancer. At startup, the Hadaps balancer parses the configuration file and assigns the given generation weight to the listed nodes. Then it builds a list of all files specified which have to be balanced. The Hadaps balancer works concurrently on three files.

A.2.2. Hadaps Benchmark

The Hadaps Benchmark has two execution modes. The write mode generates data for the benchmark, and the read mode is used to execute the benchmark itself.

Write Mode The write mode writes data from the current host into HDFS without using the MapReduce framework. This will produce intentionally a very unbalanced block placement. It accepts several command line arguments, which are described below.

```
$ hadoop jar share/hadoop/tools/lib/hadoop-hadaps-test-<version>.jar
  org.apache.hadoop.hadaps.HadapsTest write
  [-out <test directory>]
  [-count <number of files>]
```

Listing A.1: hadaps.xml Example Configuration File

```xml
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<configuration>
  <property>
    <name>hadaps.generations</name>
    <value>gen1,gen2</value>
  </property>
  <property>
    <name>hadaps.gen1.hosts</name>
    <value>
      node1.unifr.ch,
      node2.unifr.ch
    </value>
  </property>
  <property>
    <name>hadaps.gen1.weight</name>
    <value>1</value>
  </property>
  <property>
    <name>hadaps.gen2.hosts</name>
    <value>
      node3.unifr.ch,
      node4.unifr.ch
    </value>
  </property>
  <property>
    <name>hadaps.gen2.weight</name>
    <value>1.5</value>
  </property>
  <property>
    <name>hadaps.files</name>
    <value>
      3:/directory1,
      4:/directory2
    </value>
  </property>
</configuration>
```

Table A.1.: hadaps.xml Configuration Options

Name	Value
hadaps.generations	Comma-separated list of generation names
hadaps.⟨generation⟩.hosts	Comma-separated list of host names or IP addresses of target DataNodes
hadaps.⟨generation⟩.weight	Floating point number which defines the weight of the generation. The oldest generation should have a weight of 1.
hadaps.files	Comma-separated list of files or directories in the format ⟨replication factor⟩:⟨path⟩

```
    [-size <minsize in megabytes>:<maxsize in megabytes>]
```

- `-out <test directory>`
  Specifies where the test files will be written.
- `-count <number of files>`
  Specifies the number of test files to be generated.
- `-size <minsize in megabytes>:<maxsize in megabytes>`
  Specifies the size of the test files. To have some variation in size, this argument takes the minimum and maximum file size in megabytes as parameter.

Read Mode The read mode executes the benchmark. It will use the MapReduce framework to read the files. It accepts the following command line arguments.

```
$ hadoop jar share/hadoop/tools/lib/hadoop-hadaps-test-<version>.jar
  org.apache.hadoop.hadaps.HadapsTest read
  [-in <input directory>]
  [-out <output directory>]
  [-iteration <number of iterations>]
  [-csv <filename>]
```

- `-in <input directory>`
  Specifies where the test files are located. This is the output directory from the Write Mode.

- `-out <output directory>`
  Specifies where the result files will be written.

- `-iteration <number of iterations>`
  Specifies the number of iteration, the benchmark will execute.

- `-csv <filename>`
  Specifies the file name of the CSV file, which contains the results.

References

[Atl] Atlassian. Bitbucket. http://bitbucket.org. [Accessed 28-April-2014].

[DG04] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design and Implementation*, OSDI'04. USENIX Association, 2004.

[ea08] Gantz et al. The Diverse and Exploding Digital Universe. http://www.emc.com/collateral/analyst-reports/diverse-exploding-digital-universe.pdf, 2008. [Accessed 28-April-2014].

[Fou] The Apache Software Foundation. Apache Hadoop. http://hadoop.apache.org. [Accessed 28-April-2014].

[GGL03] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, SOSP'03. ACM, 2003.

[Git] Git. Git. http://git-scm.com. [Accessed 28-April-2014].

[Hada] Apache Hadoop. Hadoop Releases. http://hadoop.apache.org/releases.html. [Accessed 28-April-2014].

[Hadb] Apache Hadoop. HDFS Architecture. http://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html. [Accessed 28-April-2014].

[Hadc] Apache Hadoop. PoweredBy. http://wiki.apache.org/hadoop/PoweredBy. [Accessed 28-April-2014].

[Men] Paul Menage. CGROUPS. https://www.kernel.org/doc/Documentation/cgroups/cgroups.txt. [Accessed 28-April-2014].

[Mur] Arun Murthy. Apache Hadoop YARN – Background and an Overview. http://hortonworks.com/blog/apache-hadoop-yarn-background-and-an-overview. [Accessed 28-April-2014].

[Nyb] Chris Nyberg. gensort Data Generator. http://www.ordinal.com/gensort.html. [Accessed 28-April-2014].

[Owe08] Owen O'Malley. TeraByte Sort on Apache Hadoop. http://sortbenchmark.org/YahooHadoop.pdf, 2008. [Accessed 28-April-2014].

[Wan14] Andrew Wang. Better sorting in NetworkTopology#pseudoSortByDistance when no local node is found. https://issues.apache.org/jira/browse/HDFS-6268, 2014. [Accessed 28-April-2014].

[Whi09] Tom White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 3rd edition, 2009.

[Wika] Wikipedia. Big Data. http://en.wikipedia.org/wiki/Big_data. [Accessed 28-April-2014].

[Wikb] Wikipedia. Data degradation. http://en.wikipedia.org/wiki/Data_rot. [Accessed 28-April-2014].

Acronyms

Hadaps Hadoop Data Placement Strategy. vii, 13, 15, 17, 18, 20, 21, 23–25, 27–34

HDFS Hadoop Distributed File System. iii, vii, 1–4, 8–13, 15, 18, 20, 21, 23, 25, 29–31, 34

JAR Java ARchive. 33, 34

TCP Transmission Control Protocol. 9

XML Extensible Markup Language. 34

YARN Yet-Another-Resource-Negotiator. vii, 5, 7

Index