High performance shared state schedulers

ANTONIOS KOUZOUPISES
High performance shared state schedulers

Antonios Kouzoupis

Master of Science Thesis

Software Engineering of Distributed Systems
School of Information and Communication Technology
KTH Royal Institute of Technology
Stockholm, Sweden

18 October 2016

Examiner: Associate professor Jim Dowling
Supervisor: Professor Seif Haridi

TRITA-ICT-EX-2016:102
Abstract

Large organizations and research institutes store a huge volume of data nowadays. In order to gain any valuable insights distributed processing frameworks over a cluster of computers are needed. Apache Hadoop is the prominent framework for distributed storage and data processing. At SICS Swedish ICT we are building Hops, a new distribution of Apache Hadoop relying on a distributed, highly available MySQL Cluster NDB to improve performance. Hops-YARN is the resource management framework of Hops which introduces distributed resource management, load balancing the tracking of resources in a cluster. In Hops-YARN we make heavy usage of the back-end database storing all the resource manager metadata and incoming RPCs to provide high fault tolerance and very short recovery time.

This project aims in optimizing the mechanisms used for persisting metadata in NDB both in terms of transactional commit time but also in terms of pre-processing them. Under no condition should the in-memory RM state diverge from the state stored in NDB. With these goals in mind several solutions were examined that improved the performance of the system, making Hops-YARN comparable to Apache YARN with the extra benefits of high-fault tolerance and short recovery time. The solutions proposed in this thesis project enhance the pure commit time of a transaction to the MySQL Cluster and the pre-processing and parallelism of our Transaction Manager. The results indicate that the performance of Hops increased dramatically, utilizing more resources on a cluster with thousands of machines. Increasing the cluster utilization by a few percentages can save organizations a big amount of money.
Sammanfattning


Acknowledgements

I would like to thank my examiner Jim Dowling for his invaluable insights and crucial guidelines throughout this project and my supervisor at KTH Seif Haridi. Also, I would like to express my sincere gratitude to my supervisor at SICS Gautier Berthou for his endless support, guidance and patience. He and his expertise was the source of motivation and really helpful regardless the problem.
Contents

1 Introduction ........................................... 1
  1.1 Problem description ................................ 2
  1.2 Problem statement ................................ 3
  1.3 Goals ............................................. 4
  1.4 Reflections of Ethics, Sustainability and Economics .... 5
  1.5 Structure of this thesis ........................... 6

2 Background .......................................... 7
  2.1 Hadoop ............................................. 8
  2.2 HDFS ............................................... 9
    2.2.1 NameNode ....................................... 10
    2.2.2 DataNode ....................................... 11
  2.3 Data computation and Resource management ............ 11
    2.3.1 MapReduce ...................................... 12
    2.3.1.1 MapReduce Fault Tolerance ................. 14
    2.3.1.2 Limitations ................................ 14
    2.3.2 YARN ........................................... 15
    2.3.2.1 ResourceManager .......................... 16
    2.3.2.2 ApplicationMaster ......................... 19
    2.3.2.3 NodeManager ............................... 20
    2.3.2.4 YARN fault tolerance & HA ................ 21
  2.4 MySQL Cluster ..................................... 22
  2.5 Hops-YARN .......................................... 23
    2.5.1 Architecture .................................. 23
    2.5.2 Fault tolerance & HA ......................... 24
    2.5.3 Load balancing ................................ 24
  2.6 Taxonomy of schedulers ................................ 25
    2.6.1 Monolithic ..................................... 26
    2.6.2 Two-level ....................................... 27
    2.6.3 Shared state ................................... 27
3 Methods

4 Implementation

4.1 Transaction state aggregation

4.1.1 Batching system

4.1.2 Aggregation mechanism

4.1.2.1 One TS per commit

4.1.2.2 Multiple TS per commit

4.1.3 Conclusions

4.2 Foreign key constraints

4.2.1 yarn_rmnode

4.2.2 yarn_allocate_response

4.2.3 yarn_ficascheduler_node

4.2.4 yarn_applicationstate

4.2.5 yarn_appschedulinginfo

4.2.6 yarn_schedulerapplication

4.2.7 yarn_appmaster_rpc

4.2.8 Conclusions

4.3 Garbage Collector service

4.3.1 Conclusions

4.4 DTO Caching mechanism

4.4.1 Conclusions

5 Evaluation

5.1 Commit mechanism

5.2 Asynchronous deletion

5.3 DTO caching

5.4 Performance overview

6 Conclusions

6.1 Conclusion

6.1.1 Goals achieved

6.2 Future work

Bibliography
# List of Figures

1.1 Growth of DNA sequencing [5] ......................... 2

2.1 HPC high level architecture ............................ 8
2.2 Hadoop high level architecture ......................... 9
2.3 HDFS architecture [12] ............................... 10
2.4 MapReduce execution overview [11] ..................... 13
2.5 Hadoop 2.0 stack [18] ................................. 15
2.6 YARN architecture overview [19] ....................... 16
2.7 ResourceManager components [19] ...................... 17
2.8 ApplicationMaster interaction [19] ..................... 19
2.9 MySQL Cluster NDB components [30] .................... 22
2.10 Hops-YARN distributed RM architecture ............... 25
2.11 Cluster schedulers architecture [32] ................. 26

4.1 Hops - MySQL Cluster interaction ..................... 32
4.2 Activity diagram of RPC batching system ............... 35
4.3 Example of TS for the queue to be committed .......... 37
4.4 Aggregate commit mechanism example .................. 39
4.5 RPC foreign key constraints micro-benchmark .......... 42
4.6 Hops-YARN database schema ............................ 44
4.7 A node in the cluster from the scheduler’s perspective 45
4.8 Database tables to store incoming RPCs ................. 46
4.9 RPC alternative schema ................................ 51
4.10 Create and commit time for simple DTOs ............... 56
4.11 Hops-YARN DB session pool ........................... 57

5.1 TS commit time with GC service ....................... 63
5.2 Cluster utilization with GC service ..................... 64
5.3 DTO creation and commit with cache enabled .......... 65
5.4 TS commit time with and without cache ................. 66
5.5 Cluster utilization with and without cache ............. 66
5.6 Containers launched with and without cache .......... 67
### List of Figures

5.7  HB processed with and without cache ........................... 68  
5.8  Cluster utilization comparison, Hops-YARN - Apache YARN .... 69  
5.9  Ratio of processed heartbeats ...................................... 70  
5.10 CPU usage for RM (bbc7) and RT (bbc6) .......................... 70  
5.11 Memory usage for RM (bbc7) and RT (bbc6) ....................... 71
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Comparison of commit mechanisms</td>
<td>62</td>
</tr>
<tr>
<td>5.2</td>
<td>Cache mechanism configuration</td>
<td>66</td>
</tr>
</tbody>
</table>
Listings

4.1 ClusterJ annotated interface ........................................ 55
4.2 Caching mechanism configuration file ............................ 59
## List of Acronyms and Abbreviations

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>YARN</td>
<td>Yet Another Resource Negotiator [1]</td>
</tr>
<tr>
<td>HDFS</td>
<td>Hadoop Distributed File System [2]</td>
</tr>
<tr>
<td>RM</td>
<td>ResourceManager</td>
</tr>
<tr>
<td>NM</td>
<td>NodeManager</td>
</tr>
<tr>
<td>AM</td>
<td>ApplicationMaster</td>
</tr>
<tr>
<td>RT</td>
<td>ResourceTracker</td>
</tr>
<tr>
<td>NN</td>
<td>NameNode</td>
</tr>
<tr>
<td>DN</td>
<td>DataNode</td>
</tr>
<tr>
<td>HA</td>
<td>High availability</td>
</tr>
<tr>
<td>NDB</td>
<td>Network Database</td>
</tr>
<tr>
<td>RPC</td>
<td>Remote Procedure Call</td>
</tr>
<tr>
<td>TS</td>
<td>Transaction State</td>
</tr>
<tr>
<td>DTO</td>
<td>Data Transfer Object</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

In the last years, the storage capacity of hard disk drives has increased dramatically while at the same time their price has decreased. This trend of “cheap” storage solutions has led companies and research institutes to store a volume of data that has never been stored before. In 2014 Facebook was processing 600 TB daily [3] while Google in 2008 was processing more than 20 PB of data per day [4].

Another interesting area that is already generating a huge volume of raw data is the DNA sequencing. According to Zachary D. Stephens et al. [5] the Sequence Read Archive maintained by the United States National Institutes of Health National Center for Biotechnology Information already contains more than 3.6 petabytes of raw sequence data for a wide variety of samples including microbial genomes, plant and animal genomes and human genomes. As we can see in Figure 1.1 the need for storage capacity will exceed the order of Petabytes by the year 2025.

It is clear by now that these volumes need a paradigm shift from the traditional way we store and analyze data. It is not possible anymore to store them in a single machine. We need to employ a distributed system to harness the power of those data and extract valuable results within a reasonable time frame. On top of that we have to take under consideration that hardware will fail for various reasons. A minimum requirement would be not to lose our data, so we should keep them in several replicas. Since data are scattered across different computers, processing jobs should also follow a distributed fashion. Going one step further, we would like our analyzing jobs to continue running even though one machine failed. These jobs usually take hours to complete, so re-running them is not the best approach.
CHAPTER 1. INTRODUCTION

1.1 Problem description

As it is already mentioned, datasets now days are in the order of petabytes and exabytes. Distributed file systems like GFS [6], GlusterFS [7], HDFS [2] etc come to extend the traditional file systems located in a single machine. Storing the datasets is one half of the problem though. A cluster consists of several physical machines, sometimes thousands of them, and each machine has numerous CPU and RAM modules and hard disk drives that store the datasets. Users of the cluster issue their jobs with certain CPU and RAM requirements, as well as the files they want to access. The second half of the problem is to manage the available computational resources on that cluster. On a very high level abstraction there is an entity which has knowledge of the available resources and should schedule the jobs accordingly. The view of the cluster from the scheduler perspective is updated frequently with the new cluster utilization.

This project is a work on Hops [8] platform and more specifically on Hops-YARN, which is a modified version of Apache Hadoop YARN [1]. In YARN the entity which is responsible for keeping an updated version of the cluster utilization and scheduling tasks is the Resource Manager (RM). The view of RM regarding the available resources on the cluster is updated frequently (by default 1 second) by a heartbeating mechanism. On each machine of the cluster there is the Node Manager (NM) which periodically sends updates for the machine usage. Users
issue their application requests to the RM which then allocates a container to create the Application Master (AM). The AM service is working independently and is responsible to keep track of the application health and any further resource requests. AM periodically heartbeats the RM (by default 1 second) stating its health, the application progress or any resource increase/decrease.

An aware reader should have already noticed that RM is a crucial part of the Hadoop platform for managing resources. Not only is it vital for the progress of the system but also it can become a bottleneck and a single point of failure. Until recently, Spotify was provisioning a cluster of 1300 Hadoop nodes *. Every single node has to heartbeat the RM every second. On top of that for every single application launched, the AM service should also heartbeat the RM. This produces a considerable amount of load on the RM side which has to handle all those heartbeats and also make scheduling decisions.

In Hops in order to improve performance and High Availability (HA) of the RM we have introduced an in-memory distributed MySQL database which stores all the necessary metadata. One great feature of Hops-YARN is that the ResourceTrackerService (RT) of the RM is distributed into multiple nodes in the cluster. That service is responsible for receiving and handling heartbeats from the NMs. That way each instance handles only a portion of the total NM heartbeats. The updated metadata are then stored into the database and are streamed to the RM to update its view of the cluster. By load balancing the ResourceTrackerService we have increased the performance of the system while decreasing the load of the master RM which can perform the rest of the operations without the load of handling every single heartbeat.

Another equally important feature of Hops-YARN is that RM stores every event received and any scheduling decision into the MySQL cluster. That makes our solution highly available with minimum failover period. When a RM instance fails it re-builds the view of the cluster by reading the latest state from the database. More details on the architecture of both YARN and Hops-YARN will be given in Chapter 2 and 4. All these read/write operations to the database do not come without a cost.

1.2 Problem statement

Even with high throughput, low latency network between the RM/RT and the database, it still takes more time than in-memory operations. Especially in cases where RM operations need more than one round-trip to the database, the difference in performance is noticeable.

A great advantage of using a relational database is the support of foreign key constraints. The information that we store is semantically related. In a SQL schema that is directly translated to foreign key constraints. A trivial example is information regarding the containers running on node and information regarding the node itself. Clearly, we should never run into a situation where we try to persist a container running on a node without having information about the node at all. Moreover, foreign key constraints work the other way around. By removing the information about a node from the database we want the information about running containers on that node to be purged too. Such constraints, although they seem tempting to use, pose a great performance degradation as we will show later in this thesis. Particularly when we aim for millions of operations per second, the use of foreign key constraints should be limited and very well designed.

The transaction state manager of Hops should try to commit various states in parallel as much as possible but on the other hand ensure the order of two colliding states. For example there should never be the case where two states change information about the same node in the cluster and yet be committed in parallel. Similarly, two states that change information in the database about different entities should be committed in parallel.

The distributed MySQL cluster database is a central building block in our architecture. A slow commit and process time will result in less events being streamed and handled by the RM, directly affecting the cluster utilization and the view of the cluster from the RM perspective. A slow commit time will decrease the rate of handled events while increase the rate of queued events that are waiting to be handled. Our goal is to be able to scale up to 10000 NMIs with multiple RTs. With the current mechanism this is not possible due to delays both in the transaction manager mechanism and in the commit time.

1.3 Goals

The primary goals of this project are:

- Increase the cluster utilization of Hops-YARN

- Increase the ratio of heartbeats processed by Hops-YARN scheduler over the total number of heartbeats received

In order to achieve the aforementioned goals, the following sub-goals have been set:

- Profile the execution workflow in order to identify bottlenecks and hotspots

- Minimize the pure transactional commit time to MySQL Cluster NDB
1.4 Reflections of Ethics, Sustainability and Economics

In a wide range of academic areas such as sociology, economics, medical sciences, etc, the ability to effectively process and gain valuable insights from a big amount of data with simpler algorithms, trump other more sophisticated models with less data [9]. Researchers gather a huge volume of data from different sources. Providing tools to process and analyze these data is crucial for understanding the human behavior and validate economic theories or more important theories on medical treatments.

In order to process these data a cluster of computers should be put in place. The goal of this thesis is to increase the number of resources that are allocated at any time in a cluster. This will increase both the number of jobs that researchers can issue to analyze their data and the resource requirements for their processing tasks. Earlier we have given the example of DNA sequencing that generates a lot of data. In cases where medical data are involved, certain procedures should be followed to provide privacy. Although this work does not focus on security, it is of great importance to isolate and fence properly the data stored and filter who can access them. Also, in most of the cases cluster resources are shared among different units in the same organization. Even different competitive organizations might share resources on the cloud. The resource management system should enforce rules on the tenants of the cluster regarding the fair share of the resources and prevent situations where one tenant has allocated resources even though they are not used.

Every big organization nowadays has its own cluster to perform big data analytics. Smaller organizations and companies that cannot afford provision their own cluster turn to cloud service providers. In both cases, computers are operating 24 hours a day, every day consuming a considerable amount of energy. Having a cluster of computers underutilized means that they consume energy while being idle. By increasing the cluster utilization, computers will consume energy for performing more tasks rather than being idle. For the same reason, an organization could profit. In the case where they operate their own cluster, the money paid for energy consumption and hardware provisioning are “invested” for performing tasks and not being idle. In the case where they use a cloud service provider, then

- Parallelize the commit mechanism of Hops-YARN while at the same time guarantee consistency
- Provide asynchronous API for database operations that do not invalidate the consistency model of Hops-YARN
- Evaluate the performance impact of each solution
by utilizing more resources the organizations could lease them (the resources) for less time, thus paying less money.

1.5 Structure of this thesis

The rest of this thesis is organized as follows. Chapter 1 gives a general introduction to the research area and the problem examined in this thesis. Also, it gives an outline of the goals and discusses the ethics and benefits of this project. Chapter 2 gives the reader the necessary background information for the rest of this report to be comprehensible. It goes through fundamental ideas of Hadoop and the technologies that differentiate and realize Hops-YARN. The research methods followed in this thesis are discussed in Chapter 3, while in Chapter 4 there is a detailed technical discussion regarding the solutions examined. The evaluation of the work is done in Chapter 5 where the performance impact of the solutions proposed will be presented. Finally, Chapter 6 makes a conclusion of the work and points directions for future work.
Chapter 2

Background

This chapter will give the reader the necessary background knowledge in order for this work to be understandable. First, it will go through Apache Hadoop, a distributed storage and processing framework. We will give some brief introduction to Hadoop file system (HDFS), then we will dive into the resource manager (YARN) and in what way Hops-YARN extend the Apache YARN project. Later we will introduce a distributed, highly-available, highly-redundant relational database, MySQL Cluster (NDB) and finally we will give some insights on the different types of resource managing systems.

In Figure 2.1 we can see a high level overview of the architecture in HPC. Storage nodes are machines with very high disk capacity and bare minimum processing power. Their main usage is to store data that are going to be processed and analyzed in the future. The second building block of the architecture is the Computing nodes. These machines have no storage capabilities but they are equipped with the state of the art processing units and a lot of RAM. Those modules communicate most probably with a high throughput, low latency network. The most common industry standard for interconnecting nodes in HPC is InfiniBand [10] that can reach 30 Gb/s in each direction and sub-microsecond latency. Users issue their jobs to the Head node which is responsible for transferring the requested datasets from the Storage nodes to the Computing nodes, monitor the tasks and finally return the result to the end user.

In 2003 Google published a paper describing GoogleFS (GFS) [6], a proprietary distributed file system. It was designed to run on large clusters of commodity hardware, that are doomed to fail at some time. That was the main motivation that drove GFS to be fault-tolerant and highly-available. Apache HDFS is the open-source implementation of GFS and it will be analyzed in Section 2.2. In 2004 Google published MapReduce [11], a breakthrough programming model which exploited the locality awareness of GFS and changed the way we process very big datasets. MapReduce was later implemented for Hadoop and paved the way
2.1 Hadoop

Apache Hadoop is an open-source framework for distributed storage of large datasets and processing across clusters of computers. It was created in 2006 after the release of GFS [6] and MapReduce [11] papers from Google. The core part of Hadoop comprises of its distributed file system – HDFS, and the job scheduling, resource management framework – YARN.

The attribute that makes the biggest difference in Hadoop and similar projects is that of data locality awareness. In contrast to the HPC architecture, we do not distinguish anymore between processing and storage nodes. All nodes in a cluster perform both roles. Datasets are split into blocks of data. To provide fault tolerance each block is replicated in several nodes in the cluster. Moreover, there is a central authority which keeps track of the nodes each block is stored.

With that feature in mind, we do not move datasets anymore to the computing nodes but the executable of our job to the nodes where our data reside. When the processing of the individual blocks is done, we gather the result. That is a great paradigm shift from the traditional way of processing big datasets. A high level overview of the Hadoop architecture is depicted in Figure 2.2. Datasets are for YARN, the current resource manager and scheduler which will be analyzed in Section 2.3.2.
split into blocks and are stored in the nodes of cluster, both the original and the replicas. When a user submits a job, the workflow manager will copy the job to the appropriate nodes, which in parallel will execute it. Finally, the workflow manager will gather the individual results, aggregate them and return the final result to the client.

2.2 HDFS

As this project is focused on the resource management framework, a very brief description of the Hadoop Distributed File System will be given. Yet, this section will go through some basic concepts that will make the reader understand better the overall architecture of Hadoop.

HDFS is the distributed file system of Hadoop platform. It is designed with the assumption that hardware failure is the norm and not an exception making it highly fault-tolerant. Also, it is designed to run on commodity, heterogeneous, low-cost hardware making the setup and provisioning of a cluster cheaper than in HPC. HDFS has two main entities, the NameNode (NN) and the DataNode (DN) and its architecture is depicted in Figure 2.3.
2.2.1 NameNode

An HDFS cluster consists of a single active NN that is responsible for the file system metadata, the blocks replication and the health status of the DNs.

HDFS exposes to a user a file system similar to POSIX, in terms that the structure is hierarchical, there are the same file permissions and a subset of POSIX operations. A user through the NN can open, close, read, delete a file as in any file system. The NN uses a transaction log, the EditLog, that persists to the local file system any operation that is done to the HDFS namespace. For example if a file is renamed then a record is added to the EditLog, or if the replication factor for a file is changed.

As HDFS was designed to run on commodity hardware it should be able to handle machine failures. Internally, a file is split in a number of blocks. Typically each block is 128 MB, except from the last one and are stored in the DNs. HDFS replicates the blocks to other machines–DNs according to a configurable replication factor and replication policy. If the DN that holds a specific block has crashed, then that block is read from another replica in another DN. The NN keeps a file in its local file system with the entire namespace and the mapping of blocks to DNs, called FsImage. Upon recovery, the NN reads the FsImage and applies any operation that is logged in the EditLog. That way it can recover from a scheduled maintenance reboot or from a crash.

The NN periodically receives heartbeats from the DNs that have dual purpose. The first one is to maintain a health status for the DNs. If the NN misses a heartbeat from a DN, then it marks that DN as dead. From that point no blocks will be further assigned to that DN and NN will start migrating all the blocks
that reside in that DN to others. The second reason of receiving heartbeats is to maintain an updated view of the \textit{BlockMap}, the mapping of blocks to DNs. When a DN sends a heartbeat to the NN, it piggybacks a list with the blocks that it currently stores. That way, the BlockMap in the NN is kept up-to-date with the blocks that are stored in every DN.

\subsection{DataNode}

The DN is the slave entity in the HDFS master/slave architecture as depicted in Figure 2.3 and the actual storage of blocks. Upon a client request to store a file in HDFS, the NN instructs the client which DNs to contact to store the individual file block. The same procedure is followed when a client requests to read a file. The NN returns a list with DNs that store the blocks forming the whole file. The client then contacts each and every DN, fetching the corresponding blocks.

A DN periodically heartbeats the NN. As it is explained in Section 2.2.1, the heartbeat contains a list of blocks, that the DN stores, so that the NN maintains a map of file blocks and DataNodes. Also, the heartbeat signifies that the DN is alive and can be used for storage and retrieval of blocks. Heartbeats also carry information regarding the status of the DN such as total storage capacity, storage in use etc. These metrics are taken under consideration by the NN when assigning blocks to DN.

The DataNode also perform various operations as instructed by the NN. These instructions are sent to the DN via the heartbeat mechanism as a reply to the heartbeat sent by the DN. Such operations might be creation, deletion or replication of a block. The block replication is done according to the replication strategy. A common strategy for placing replicas with a replication factor of three, is to place the first replica in the same node as the client runs, the second in another node in a different rack (off-rack) and the third one in the same rack as the second but in a different node. Adjusting the replication factor and the replication policy can greatly affect the write and read performance of the HDFS cluster and should be carefully tweaked.

\subsection{Data computation and Resource management}

So far we have discussed how to store datasets in the order of terabytes and even petabytes in a distributed and reliable way. We have gone through the most important ideas that lay behind HDFS, the distributed file system of Hadoop, and how it manages to overcome the fact that machines will fail and avoid transferring of huge files into computation nodes in contrast to the HPC architecture.
That is half the way of extracting valuable results out of big data though. A cluster consists of thousands of physical machines with certain resources in terms of CPUs, amount of RAM, network bandwidth, disk space etc. We need a mechanism to harness all that power while at same time exploiting the locality awareness to minimize data transfer and maximize cluster utilization.

### 2.3.1 MapReduce

After the world wide web explosion at the ending of 1990’s, Google has emerged as one of the most significant web searching companies. The novelty of Google was PageRank [13], an algorithm counting the number of outgoing links of a webpage to determine its importance. In order to apply the PageRank algorithm and form the Google search results, first the webpage has to be scraped and indexed. As of 2004 the raw size of the documents that had been collected was more than 20 terabytes [11]. Although the engineers at Google have distributed and parallelized the algorithm, there were more tasks that other teams have parallelized in a different way making it difficult to maintain such a diverse codebase. That led them in 2004 to publish a paper about MapReduce, a generic framework to write distributed applications that hide all the complexity of fault-tolerance, locality awareness, load balancing etc.

MapReduce programming model borrows two very common functions from functional programming, **Map** and **Reduce**. The **Map** function takes as input key/value pairs and produces as output a set of key/value pairs as well. The **Map** function is written by the user and varies depending on the use case.

The **Reduce** function, takes as input the intermediate key/value pairs produced by **Map** and merge them together producing a smaller set of values. The **Reduce** function and the way it will merge the intermediate pairs is also provided by the user.

A trivial example of the MapReduce programming model is that of counting the occurrences of words in a text. The **Map** function takes as input a list of all the words in the text and emits tuples in the form `(word,1)`, where `word` is every word parsed. The result of the **Map** function is passed to the **Reduce** function which adds the value of the tuples with the same key, in that case is the word. The final result will be a list of tuples with unique keys, where the keys will be all the words parsed from the text and the value would be the occurrences of the word in the text.

Google provided a framework which took advantage of the locality awareness of the already existing GFS and the MapReduce programming paradigm. The execution overview of MapReduce is depicted in Figure 2.4. We can identify two entities in MapReduce architecture, the **Master** and the **Workers**.

The **Master** has the role of the coordinator that pushes the jobs to the worker
2.3. DATA COMPUTATION AND RESOURCE MANAGEMENT

machines. It keeps track of the status of jobs in the workers, informs other workers for the intermediate files produced during the Map phase and pings the workers to verify their liveness.

The Workers reside at the same physical hardware as the GFS nodes to take advantage of the data locality. They are divided into mappers, which execute the Map function and reducers, which perform the reduce phase as instructed. Workers are pre-configured with available map or reduce slots depending on their CPU or RAM.

At the very beginning, a user submits a job to Master. Master forks the submitted job and is responsible to schedule the forks on workers that (a) have available map/reduce slots and (b) have the requested datasets stored locally. Upon the scheduling is done, the Map phase begins in the mappers. They read the datasets from the local hard drive and perform the Map function. Master periodically pings the mappers to get informed about the status of the job and the health of the node itself. When a mapper node completes its task, it writes the intermediate key/value pairs to the local file system and informs the Master node. The Master node in turn, notifies the reducer nodes that an intermediate result is available at a specific node, where the latter reads it (the result) remotely and perform the reduce function. Finally, when all the reducers have completed the Reduce phase, the Master notifies the user program.
2.3.1 MapReduce Fault Tolerance

Primary concern of the engineers was the fact that machines will eventually fail. MapReduce will run on a cluster of thousands of machines so the probability of a failed one would be higher. For that reason they equipped MapReduce with a heartbeating mechanism in order to be able to handle such situations. The Master periodically pings the workers. The workers should respond back within a predefined timeout before they are declared dead. When a node that performs the Map phase is declared dead, the job that was running at that node is set to idle and is rescheduled on another node. Similarly, when a map job has finished, since the intermediate result is written to the local hard drive, the job has to be rescheduled in a different machine. When a reducer node has failed and the job is still in running state, then it is set back in idle state and assigned to another node. In case of a completed Reduce phase, the result is stored in the global file system, GFS in that case. So, even with a failed reducer machine, the result will still be available and the job should not be rescheduled.

While a Worker failure does not greatly affect the MapReduce job, it is not the same case with a Master failure. If a machine that is a Master node fails, then the whole MapReduce job is canceled and the client is informed so that it can retry later on. “However, given that there is only a single master, its failure is unlikely;” [11].

2.3.1.2 Limitations

MapReduce facilitated engineers to “easily” write parallel data processing applications by hiding all the complexity of a distributed system. It provided some sort of fault tolerance and it was generic enough to fit in various domains.

MapReduce and Hadoop over the years has become the industry standard for processing and storing big volumes of data. After some period of heavy usage it became clear that, although the platform itself suited the needs for distributed, reliable storage and cluster management, there were some limitations that had to be addressed. The two key shortcomings were regarding the tight coupling of a programming model with the resource management infrastructure and the centralized handling of jobs [1, 14].

A user who wants to write an application for MapReduce framework, all it has to do is to provide implementation for the two first-order functions Map and Reduce. This static map-reduce pipeline is very limiting though, as every job should have exactly one Map function followed by an optional Reduce function. That workflow is not suitable for large scale computations, such as machine learning programs that require multiple iterations over a dataset. That means multiple individual MapReduce jobs have to be scheduled while the frequent
2.3. Data computation and resource management

write of data in disk or in a distributed file system would impose a considerable latency. A common pattern/misuse [1] was to submit jobs with a map phase only that spawned alternative frameworks or even web servers. The scheduler had no semantics about the job except that they were map jobs with a consequence in the cluster utilization, creating deadlocks and a general instability to the system. The second drawback of MapReduce and Hadoop 1.x was the centralized job handling and monitoring of their flow. The Master or the JobTracker should monitor every single job, receiving liveness heartbeats, resource requests etc. This is a heavy workload for a single machine that drove to major scalability issues.

These two crucial limitations of MapReduce led to a total re-design of Hadoop. Since Hadoop 2.0 there is a resource management module, YARN – Yet Another Resource Negotiator which will be analyzed in section 2.3.2 and MapReduce is just another application running on a cluster of physical machines.

2.3.2 YARN

Considering the limitations outlined in section 2.3.1.2, Vinod Kumar Vavilapalli et al. presented YARN [1] the new resource management layer that was adopted in Hadoop 2.0. The new Hadoop stack now is depicted in Figure 2.5 where YARN is the cluster resource management module and MapReduce is one out of plenty applications running on top of YARN. This architectural transformation paved the way for a wide variety of frameworks like Apache Spark [15], Apache Flink [16], Apache Pig [17], etc to run on the Hadoop platform like any other YARN application.

The new architecture of Hadoop 2.x separates the resource management functions from the programming model. It delegates the intra-application communication and the tracking of the execution flow to per-job components. That unlocks great performance improvements, improves scalability and enables
a wide variety of frameworks to share the cluster resources in a very smooth way.

YARN uses three main components to provide a scalable and fault tolerant resource management platform. The first component is the ResourceManager (RM), a per-cluster daemon that tracks resource usage, node liveness and schedules jobs on the cluster. The second component is a per-node NodeManager (NM) which is responsible for monitoring resource availability on the specific node, reporting faults to RM and managing container life-cycle. Finally, there is the ApplicationMaster (AM) which coordinates the logical plan of a single job, manages the physical resources offered by the RM and tracks the execution of the job. A high level overview of YARN architecture is described in Figure 2.6. RM has a global view of the cluster and provides the scheduling functionality, while the per-job AM manages the dynamic resource requests and the workflow of the tasks. Containers that are allocated by the RM are locally managed by the NM in each node in the cluster.

2.3.2.1 ResourceManager

In YARN the RM acts as the central authority for allocating resources in the cluster. It works closely with the per-node NodeManager getting an updated view of the cluster by the heartbeats received. The RM itself allocates generic resources in the cluster in the form of containers that have specific CPU and RAM requirements, in contrast to MapReduce Map and Reduce slots. Those
resource requests are piggybacked in the heartbeats issued by every AM. As RM is completely unaware about the job execution plan, it is up to the AM to make local optimizations and assign the resources accordingly. RM internally consists of several modules but the three most important are the ApplicationMasterService, the ResourceTrackerService and the Yarn Scheduler as shown in Figure 2.7.

The ApplicationMasterService is responsible for receiving and handling heartbeats from the AMs that are launched in the cluster. Heartbeats are designed to be as compact as possible, still not missing any vital information. For that reason, Google Protocol Buffers [20] are used for every communication among YARN components. Protocol Buffers is a language-neutral, platform-neutral mechanism for efficiently serializing data. The heartbeat mechanism serves both as a scalable way for the RM and AM to communicate, but also for the RM to track the liveness of AMs. ResourceRequests contain information such as the resources per container in terms of virtual cores and memory, the number of containers, locality preferences and priority of requests. The scheduler then tracks, updates and satisfies these requests with available resources in the cluster. The RM builds the view of the cluster with the available resources from the information it receives from the NMs. The scheduler tries to match the locality constraints as much as possible and responds back to AM with the allocated containers along with credentials that grant access to them. The RM also keeps track of the AM health through the heartbeats received. The component that handle the liveness property of every AM is the AMLivenessMonitor. In case of a missed heartbeat, that particular AM is deemed dead and is expired by the RM. All the containers that were allocated for that AM are marked as dead and the RM reschedules the same application (ApplicationMaster) on a new container.
As it was previously mentioned, the RM builds its view of the cluster by the information that NMs send to it. The ResourceTrackerService component is responsible for handling such RPCs and forwarding them to the appropriate modules. Before a new node in the cluster is able to execute YARN jobs, it should first register itself with the RM through the ResourceTrackerService and exchange some security tokens. A heartbeat mechanism is also used in this place to ensure the liveness of NMs and to receive updated information about the available resources in the physical machine. The newly received information about the available resources on that node is forwarded to the YARN scheduler so it can make scheduling decisions. Also, a received heartbeat is forwarded to the NMLivelinessMonitor module which keeps track of the health of NMs. If RM has not received any heartbeat from a NM after a configurable timeout, then it is deemed dead and is expired. All the containers that were currently running on that node are also marked as dead and an event is sent to the scheduling module not to schedule any job on that node. When the node restarts, it registers again with the RM and it makes itself available for scheduling again.

At the core of RM is the YarnScheduler that is responsible of making scheduling decisions based on the available resources on the cluster and the resource requests issued by the AMs. Currently the resource requirements of an AM are limited to the number of virtual cores and to the amount of memory a container should have. YarnScheduler is a pluggable module and at the time of writing there are three different options. The first and original option is the FIFO Scheduler where jobs are served in a simple first-in-first-out order with no sense of priorities. The second scheduling policy is the Capacity Scheduler [21] developed by Yahoo! Capacity Scheduler is primarily built for large clusters with resources that are shared among different units in the same organization. There are different queues serving jobs for various units while guaranteeing some minimum capacity for each queue. Any excess capacity can be temporarily allocated to other queues and a job with high priority will be executed before any other job with lower priority. If a job cannot be scheduled in its respective queue due to lack of resources and that queue is below its fair share, then jobs in other queues can be preempted. Last but not least is the Fair Scheduler [22] developed by Facebook. In Fair Scheduler every application belongs to a queue, by default the “default” queue. The basic idea is that containers are allocated to the application with the fewer resources assigned within the queue, providing a uniform distribution of the available cluster resources in the long run. There can be multiple queues with support for priorities, minimum and maximum shares and FIFO ordering within the queue. Similar to Capacity Scheduler, Fair Scheduler also has support for preemption of containers that are already assigned. Currently the default scheduler for Hadoop is the Capacity Scheduler.
2.3.2.2 ApplicationMaster

Upon a successful submission of an application to RM, the latter creates a special container in a node called ApplicationMaster. AM is a per-application process that coordinates the execution plan of the application, negotiates resources with the RM and monitors the assigned containers. AM periodically heartbeats RM, default value is 10 minutes, to prove it is alive and to dynamically request more resources or release some. When AM is launched, it will compute the necessary requirements and locality preferences, encode them and through the heartbeat mechanism send them to RM. RM depending on the scheduling decisions it has made it might respond back with any empty response or with container leases on different nodes. AM will contact the respective NodeManagers, present them the leases and the NM will create the containers. Afterwards, it (AM) is responsible to monitor the liveness of the containers or implement any recovery functions. An overview of the workflow explained above is illustrated in Figure 2.8.

Since a key requirement for YARN was to decouple the programming model from the resource management, AM is not tightly connected to YARN. Writing an ApplicationMaster process is not an easy task but Hadoop offers some APIs to avoid the complexity of low-level protocols. Users can write their own ApplicationMaster process fitting particular needs of making local optimizations with the containers allocated by the RM, monitoring the containers, define the recovery procedures when a container dies or capture the exit status of a finished task.
2.3.2.3 NodeManager

*NodeManager* is the “worker” daemon that runs on every physical node of a Hadoop cluster. It is responsible for monitoring the health of the physical hardware, authenticating the container leases, preparing, monitoring and tearing down containers and providing a set of services to the application.

When a node joins a cluster, the NM should register with the RM. It will present the total amount of virtual cores and memory available in this machine, some security tokens required to authenticate the container leases, some communication ports etc. From that point NM should periodically heartbeat the RM, default value is 1 second, proving that is still alive and keeping the RM up-to-date with its status. NM regularly runs some scripts to monitor the health of the machine and the status is sent to RM. As a response it might get back a list of container leases or some instructions such as node decommission because of unhealthy node or to kill some containers. In case of a missed heartbeat, the RM declares the NM as dead, excludes that node from its pool of resources and informs all running AMs about the failed NM. AM is responsible to react to such a failure and possibly ask resources from RM to redo the work done in the failed node.

Each container in YARN comes along with a *container launch context (CLC)*. The CLC contains information specific to the application such as environment variables, dependencies stored in HDFS or on-line, security tokens, commands that actually spawn the application etc. Once the NM has validated the container lease with the security tokens provided, it should initialize the container by fetching the requested dependencies, setting the variables and of course run the commands specified by the CLC. At the end of the container life-cycle or if the container dies or if RM instructs NM to kill a container, NM should garbage collect any dependency fetched during initialization and not used any longer by other containers in that node. For the whole duration of a container’s life-cycle, NM monitors its utilization. If a container’s usage exceeds its assigned, then the NM signals the container to be killed so that it does not disrupt the work of other containers sharing the same physical machine.

Finally, NM provides some services to the containers such as log aggregation that will upload anything written to *stdout* or *stderr* to HDFS when the application finishes. NM also provides a set of auxiliary, pluggable services that are used by some applications. For example, in MapReduce the intermediate output of the Map phase should not be garbage collected after the container has finished. The service will flag these data to be persisted even after the container has gracefully exit.
2.3.2.4 YARN fault tolerance & HA

So far we have gone through some key points of the resource management platform of Hadoop. RM is the central authority that makes scheduling decisions and carries all the burden of monitoring both the AMs and the NMs. Although from the beginning Hadoop was designed to run on commodity hardware where machine failures are the norm [24, 25], a ResourceManager failure would drove the whole cluster useless. Moreover, after a RM restart, it had to kill all the containers running on the cluster including the ApplicationMasters and launch new instances of them. Since RM is not responsible for the recovery of the applications AMs had to start over the tasks, unless if they had some recovery policy. Hadoop 2.4 introduced some kind of recovery mechanism that would recover some application metadata and re-submit only the non-finished applications in a way invisible to the user. As of Hadoop 2.6 RM restart has further improved. The rest of this section will briefly explain the recovery and HA mechanism of the RM.

To recover from a failure, RM needs to store some state in a persistent storage. Currently there is support for three alternatives. The first one is to use the Apache ZooKeeper [26], a service that provides highly reliable distributed coordination, naming, storing and group membership services. The second alternative is LevelDB [27], a light-weight key-value store and finally the local file system or HDFS. The default state store is the file system, local or HDFS, although if a requirement of our cluster is also HA, then Apache ZooKeeper is the preferred one. To begin with, RM stores some application metadata to the persistent storage solution. These metadata include the application submission context, the final status of the application, diagnostics of the application and some security related tokens. Moreover, when the RM restarts it will ask from all the NMs in the cluster to re-sync and send back information about all the containers that are currently running. Using that information it can recover the whole state of the scheduler such as resource requests, queues’ usage etc. That way RM does not need to kill and kick-off again all the running applications, just instruct the respective AMs to re-sync with it.

Even with the aforementioned mechanism RM is a single point of failure. In case of an RM crash, the cluster would be essentially useless until the RM restarts and recover. This period, depending on the number of nodes on the cluster, the number of applications running and the policies to detect a dead machine, might take quite a long time. As of Hadoop 2.4 there is a High Availability feature for the RM, using an Active/Standby architecture and a ZooKeeper instance to coordinate the RM nodes. ZooKeeper ensures that at any point of time there is only one Active node that performs all the scheduling decisions and monitoring operations. In case of a crash, a new leader is elected from the Standby pool and
is promoted to Active. The transition from Standby to Active can be done either manually through the administration CLI or automatically where ZooKeeper will detect the crashed node and elect a new leader. The new Active RM will build the current state from the state stored in ZooKeeper. AMs and NMs will keep contacting the crashed, previously Active, RM until they realize it is dead. Then they will contact the next RM in their configuration list until they hit the currently Active node in a round-robin fashion.

### 2.4 MySQL Cluster

This work focuses on optimization techniques for processing and storing metadata in the persistent storage. The solution of our choice is a MySQL Cluster database, therefore this section will briefly explain some basic concepts for the reader.

MySQL Cluster is a standard MySQL server with an in-memory clustered storage engine called NDB [28]. NDB is a real-time, ACID-compliant, relational database storage engine with no single point of failure. NDB was designed for the telecom industry where high throughput and minimum downtime is a necessity. It provides 99.999% uptime with subsecond failure recovery and failover enabled by its shared-nothing architecture. While the performance is similar to other NoSQL solutions, 2.5 Million SQL Ops/Sec for MySQL Cluster 7.4 [29], it does not lack the features of a relational database such as ACID transactions, multiple isolation levels with row-level locking and of course SQL-based declarative query interface.

A typical MySQL Cluster consists of three entities as illustrated in Figure 2.9.
These include:

**Data Nodes** An `ndb` process that stores the actual data – tables or a replica of them. Each Data Node in the cluster should be located in a different physical machine.

**SQL Nodes** A `mysqld` server daemon that is used by applications to query the schema stored in the cluster.

**NDB Management Server** The server which the management client connects to in order to perform maintenance operations.

Depending on the number of Data Nodes and the replication factor, NDB forms *node groups* such that **number of node groups** = **number of Data Nodes / replication factor**. For example, a cluster with 8 data nodes and replication factor of 2, will form 4 node groups. Tables in NDB are partitioned by a user-defined key into these node groups and then replicated in the same group. This architecture results on data always available with at least one data node alive in each group. If all data nodes in any group crash, then the cluster will remain with a corrupted view of the database.

### 2.5 Hops-YARN

Hops-YARN is a drop-in replacement of Apache Hadoop YARN for the Hops platform. From a user’s perspective there is no difference between the two implementations and an Apache YARN application can be scheduled on Hops-YARN without any modification. Although the interface is the same, there are some key characteristics that distinguish the two implementations and can be categorized into architectural, recovery mechanism and load balancing. In the rest of the section the difference will be presented for every category.

#### 2.5.1 Architecture

In Hops we heavily use a MySQL Cluster that was briefly introduced in Section 2.4. We store all kinds of metadata spanning from Hops-YARN to Hops-HDFS, a new distribution of Apache HDFS, and HopsWorks, a web-based UI front-end to Hops. The fact that everything is stored in the database leverages the limited amount of information that can be stored in the JVM heap of a single machine and opens up great opportunities of improvement and experimentation.

Apache Hadoop uses ZooKeeper to detect failures and elect a new leader. Since the MySQL Cluster is already in place storing data, we use a leader election
mechanism proposed by Salman Niazi et al. [31] that uses NewSQL databases in a novel way. The protocol guarantees a single process acting as a leader at any point of time with performance comparable to Apache ZooKeeper. Having the database acting as a persistent storage and as a leader election mechanism, Hops drops ZooKeeper from its stack relieving the operations team from the burden of maintaining one extra service.

2.5.2 Fault tolerance & HA

Section 2.3.2.4 outlined how Apache Hadoop YARN deals with RM failures and provides a highly available solution. In Hops-YARN we follow a different path for storing information for recovery. In YARN, AMs and NMs communicate with the RM through a heartbeating mechanism. These heartbeats carry information such as (de)allocation requests, health status, etc. Since the database allows for millions of operations per second, we store every single RPC that the RM receives and delete them when the request is handled. Moreover, every operation that is done on the scheduler state is reflected on a modification in the database. The main advantage of this approach over the approach followed by Apache YARN is in terms of recovery time. It is much faster to read the complete state of the scheduler from the distributed in-memory database than asking every NM to re-sync and send back a list of all running containers. Particularly when the cluster size grows in the order of thousands of machines. Moreover, in case of a crash in Apache YARN, the RM instructs all the AMs and NMs to re-sync and send again any request that has been sent but not handled. In Hops-YARN, the RM recovers the unhandled RPCs from the database and replays them.

In terms of HA the architecture of Hops-YARN is basically the same with an Active/Standby model for the scheduler, although some improvements have been made for the Standby nodes described in the following section.

2.5.3 Load balancing

Standby is boring! Except for being boring, having a physical machine idle for most of the time is a waste of resources. Although RM is a monolith, its architecture is modular. The components of the RM are illustrated in Figure 2.7. Hops-YARN follows a very original approach of distributing the ResourceTrackerService among the StandBy RM nodes. The ResourceTrackerService is responsible for handling the RPCs from the NMs (see Section 2.3.2.1). Assume a cluster with the moderate size of 5000 nodes and the default value of 1 second for the heartbeat interval. That implies that every second the RM should handle 5000 RPCs just for keeping track the NM status. In Hops-YARN, the StandBy RMs also run the ResourceTrackerService. When NMs register with the
RM they are assigned to the least overloaded ResourceTracker (RT) – StandBy ResourceManager. The information received by each ResourceTracker separately is stored in the database and through the event API of NDB is streamed to the Active RM to update its view of the cluster. In that case, NDB serves as a communication channel between the RT and the RM. With that architecture the load of tracking 5000 nodes is distributed among all the RMs in the cluster. An overview of Hops-YARN distributed ResourceManager is illustrated in Figure 2.10.

2.6 Taxonomy of schedulers

Operating large-scale clusters is expensive both in terms of investment to buy all the necessary hardware equipment but also in terms of human resources that will
maintain them. The variety in the jobs running in a big organization poses a great challenge in the utilization and efficiency of a cluster. There are long-running production jobs that should “never” stop running, short-living memory intensive batch jobs that analyze massive amount of data, testing jobs running with the lowest priority and so forth. At the same time schedulers should be able to scale to tens of thousands of nodes per cluster and be highly available with minimum downtime. In order to tackle these issues there has been a lot of research regarding cluster schedulers or data center operating systems as they are also referred. In this section three different architectures will be presented, based on the taxonomy published in the Omega paper [32]. An overview of these architectures is depicted in Figure 2.11.

2.6.1 Monolithic

The first category of schedulers explored is the monolithic. In this architecture there is a single, centralized entity that makes all the scheduling decisions with no parallelism. A monolithic scheduler still can facilitate different scheduling policies according to the type of the workload by providing multiple code paths. Depending on the type of the job, the execution flow can take different path – policy. Although, it is tempting to support multiple scheduling policies, “it is surprisingly difficult to support a wide range of policies in a sustainable manner using a single-algorithm implementation” [32].

Another drawback of monolithic schedulers is the head-of-line blocking. A small and easy to schedule job might get stack behind a big and demanding job. This will delay the execution of the former, a side effect that is not desirable in the enterprise world. Scalability is another issue that has to be addressed. Since the scheduler runs on a single instance it can be the bottleneck if the cluster size is big enough. On the other hand, a monolithic scheduler has a full view of the cluster and its available resources. For that reason it can make optimal decisions on the
job placement and achieving high utilization (until it becomes the bottleneck).

A slight variation of a monolithic scheduler is the static partitioning of the
cluster. Each partition will run its own monolithic scheduler with a separate policy
according to the jobs type. This approach though leads to fragmentation and sub-
optimal cluster utilization.

A prominent example of a monolithic scheduler is Apache Hadoop YARN
and Hops-YARN. The key distinguishing characteristic of Hops-YARN is that
the state of the scheduler is stored in the MySQL Cluster which opens the way
for various architectural experimentations resembling shared state schedulers (see
Section 2.6.3).

2.6.2 Two-level

The direct shortcoming of the monolithic scheduler is the inability to keep up
with the diverse job types since it can only run one scheduling policy. The two-
level scheduling architecture tries to fix it by dynamically partition the cluster and
adjust the partitions while each partition runs its own scheduling framework. This
approach is explored by systems like Mesos \[33\] and Hadoop-on-Demand \[34\].

The resource allocator dynamically partitions the cluster and offers the
available resources of that particular partition to the scheduling framework running on top of it. That way, at any time a resource is examined for possible
allocation by only one framework. In essence that means that the framework
holds a lock on a partition’s resources for as long as the scheduling decision lasts.
This can be an issue in terms of cluster utilization and scheduling time with long
running tasks or heavy resource requirements. Finally, as the framework does not
have access to the full view of the cluster it cannot preempt containers belonging
to another partitions in favour of high priority, heavy tasks.

2.6.3 Shared state

Last but not least in the taxonomy of schedulers is the shared state scheduling
with systems such as Omega \[32\] and Borg \[35\]. In this category there are
multiple schedulers, each running different scheduling policies, like in two-level
scheduling but the main difference is that every scheduler operates on the full view
of the cluster. Compared to two-level, optimistic concurrency control is used to
deal with state updates increasing the scheduler parallelism. This comes with the
risk of re-scheduling the job in case of a conflicting state between two or more
schedulers.

In Omega they realize it using a persistent storage for the allocations in the
cluster called cell state. Each scheduler has a local, frequently updated copy of
it, on which it operates in order to make its decision. When a scheduler assigns
some resources to a job, it updates the shared state in a transactional way. If during scheduling another scheduler has assigned those resources, then the transaction is rolled-back and the request has to be re-scheduled. On a successful decision, the allocation is persisted in the shared state and other schedulers will update their local copy asynchronously. The commit is atomic and at most one transaction will succeed. To prevent starvation, a scheduler will accept all but the conflicting resources. It will synchronize its local copy with the shared one and in a next round it will try to allocate the rest of the requested resources. That way schedulers work completely independently, maximizing parallelism. Clearly the performance of this type of schedulers is determined by the ratio of ultimately failed allocations over the successful.
Chapter 3

Methods

The aim of this project is to increase the number of events processed by the scheduler and maximize the overall cluster utilization. To achieve this, we have to minimize the time spent for Hops-YARN to commit a transaction in the MySQL Cluster while at the same time parallelize the process. In order to explore the limits of the current system, different optimization techniques were examined and evaluated as presented in Chapters 4 and 5. For that reason a Quantitative Research method was followed. The impact of a new feature in the system was measured in terms of commit time in the database, the percentage of events handled by the ResourceManager or the overall cluster utilization as suited.

The quantitative research method is supported by deductive approach and experimental method. Before starting an iteration for implementing a new feature, a thorough profiling of the workflow of the system was performed, in order to identify the bottlenecks. Afterwards, wherever it was possible, a prototype was created of the new feature and benchmarked to validate any performance gain. The micro-benchmark gave us an incentive on whether it was worth investing on that feature or not. In case of a positive feedback, a concrete version of the feature was implemented and finally the system’s performance was measured. After the completion of one feature, another iteration of the procedure explained above took place to identify more bottlenecks in the system.

Regarding the data collection method, experiments were conducted throughout the project duration and with statistics data analysis calculation the results are presented in Chapter 5. For the purpose of data collection a cluster operated by SICS was used with seven physical machines and two MySQL Clusters with four and two nodes respectively. In order for the results to be reliable each experiment run several times and an average was computed where it was reasonable. Since the experiments took a some time to finish and produced a lot of valuable data, custom code to dump these data to files was employed and process them later. For the validity part of the experiments, a simulator was used that simulated
a configurable number of nodes in a Hadoop cluster and measured in fixed intervals the variables that were interesting for the system performance. After spawning the ResourceManager and the ResourceTracker in multiple machines, it starts simulating the NodeManagers that heartbeat the scheduler. Also, it parses existing trace files and issues a synthetic workload to the scheduler. The simulator used is a modified version of the simulator that ships together with the Hadoop distribution. The difference is that the load of simulating the NodeManagers and sending application launching requests is distributed across multiple machines in the cluster. Since the workload is parsed from trace files, it makes the experiments reproducible by other researchers. A note should be taken here. The performance of the system is affected by the load of the physical machine, the network traffic and the utilization of the database, so a small variation on the results should be anticipated.
Chapter 4

Implementation

This chapter will present the work done throughout this project. Section 4.1 will give a detailed analysis of the new transaction state manager of Hops-YARN and how that boosted our performance. Section 4.2 presents the database schema that we were using and how this evolved to a schema without any foreign key constraints, yet being consistent. Section 4.3 deals with the garbage collector service written for Hops that deletes asynchronously old data from the database. Finally, in Section 4.4 we will explore the shortcomings of the MySQL Cluster connector for Java and how we have managed to overcome them. As it is already mentioned in Chapter 3, each feature completion was followed by a system profiling to guide us to the next bottleneck.

Before diving into the implementation details it is advisable to give a general overview of how Hops and Hops-YARN interact with the NDB cluster in terms of Java packages. The interaction is illustrated in Figure 4.1. Hops distribution is statically linked with the Data Access Layer (DAL) package hops-metadata-dal which provides an API used by both Hops-HDFS and Hops-YARN to interact with the persistent storage. Currently it is implemented a client library for the MySQL Cluster NDB in the package hops-metadata-dal-impl-ndb which also links to the DAL package. Our choice in favour is the NDB cluster but users are free to implement their own client library for any other storage solution as long as the back-end has support for transactions, read/write locks and at least read-committed isolation. Both Hops and the DAL API are released under an Apache 2.0 license and the DAL implementation is licensed under GPLv2.
4.1 Transaction state aggregation

This section will present the new architecture of the Transaction State manager of Hops-YARN. A Transaction State in Hops-YARN is an object that holds all the information that should be persisted in the database back-end in a consistent way. Updates in the RM state are generated either from heartbeats received from NMIs and AMIs or from events that are streamed from the NDB event API (see Section 2.5.3). The distributed ResourceTrackingService of Hops-YARN receive heartbeats from the NodeManagers and persist them in NDB. MySQL Cluster has an event API that can stream events to subscribers. In RM there is a custom C++ library that receives the events emitted from NDB, creates a Java representation of them and put them in FIFO queue. In the background there is a Hops service that polls from the queue and triggers the appropriate events. Every modification in the scheduler state should be reflected with updates in the corresponding tables in NDB. Such modifications include:

- New applications that have been issued to the scheduler through the YARN client. This include name of the application, user of the application, the Application Submission Context, state of the application etc.
- New application attempts including reference to AM, diagnostics, list of nodes that the containers of this application run etc.
- Newly created containers that applications will use
- Containers that have finished their job and should be removed from scheduler's state
- Containers that have updated their state for some reason etc.
- Newly added nodes in the cluster
For the full list of the modifications tracked and persisted in NDB, consult `TransactionStateImpl.java` file.

It is important that the state persisted in the database reflects the state of the ResourceManager in memory. In case of a recovery, the state recovered should be the last known state and be consistent. An example of inconsistency would be a container to be listed as running even though the application that was using it has finished. In order to avoid such situations and achieve the consistency level we desire there is the notion of Transaction State (TS). A TS is committed in an “atomic” fashion. This is facilitated by the transactional nature of MySQL Cluster, so a commit of one TS is one “big” SQL transaction. Using transactions we achieve isolation and atomicity of our Hops TS. With the all-or-nothing architecture of transactions, if the RM crashes in the middle of a commit, then the whole TS will be aborted. Similarly, in case of an error during the commit phase it will roll-back the whole transaction leaving the database in a clean state. More information regarding how a TS is created is presented in Section 4.1.1.

Having our safety property covered, one more reason to use Transaction States is for efficiency. Committing a single Transaction State with a lot of modifications is more efficient than committing small modifications several times.

The Transaction State is implemented generally with concurrent hash maps and accessors in the style of `add*ToAdd` for entries that we want to persist in the database and `add*ToRemove` for entries that we want to delete from the database. Even inside a TS we should be very careful on how we access the fields. A TS holds modifications from several RPC requests that may modify the same element. For example one RPC might create a new container and the next one destroy it. YARN internals are event-based. So there is no guarantee about the order that events will be processed. The second RPC might follow a different code path and finish earlier than the first one. In that case what will be persisted in the database would be the creation of a new container, something completely wrong. That is the reason we hold two separate data structures for the same table, one for insert operations and another for delete operations. First the insert data structure is processed that persists entries in the database and then the delete data structure.

Section 4.1.1 will give some insights on the existing batching system, while in Section 4.1.2 the new queuing mechanism will be presented that improved the commit time.

### 4.1.1 Batching system

So far we have discussed why we need the Transaction State in Hops-YARN and how it is implemented. We still miss the part of how Hops-YARN handles

* https://goo.gl/Ukq4Tp
heartbeats or events from NDB and how the scheduler updates the fields in it. A TS object is managed by the TransactionStateManager a custom service that runs on Hops-YARN. It is responsible for creating new TS, provide the current TS to methods that have requested it and keep track of how many RPCs have requested the TS.

Heartbeats from AMs and NM are perceived as RPCs. Upon an RPC, the method invoked will ask for the TransactionStateManager to provide the current TS. The TS will be piggybacked to every event triggered by RM components and “travel” all the way until that request has been fully handled. Each modification made by the components to the state of the RM will also be added to the insert and delete data structures explained above. TransactionStateManager service provides isolation by batching several RPC requests together, let the requests be handled and then batch the next RPCs in a different TS. The activity diagram of the batching system is depicted in Figure 4.2.

At the beginning the TransactionStateManager service creates a Transaction State object. Heartbeats start coming from the ApplicationMasters and the NodeManagers in the form of RPCs. The methods that are invoked, get the current TS from the TransactionStateManager and piggyback them to the events triggered, while various RM components handle those events in separate threads. The TransactionStateManager service keeps receiving RPCs until a certain threshold of received RPCs – default is 60, or until a timeout has been reached – default is 60 ms. At the point where either of two is satisfied, it blocks responding to further getCurrentTransactionState requests until all the RPCs in the previous batch have been handled properly and put in the queue to be committed in the database back-end. When this is done, it creates a new Transaction State object and unblocks the receiving of new RPCs. At the same time but in a different thread, the previous Transaction State is committed to the database as described in Section 4.1.2.

The Transaction State should also be included in the events triggered by streamed NDB events as the cascading modifications should be persisted in the database. The same Transaction State manager service is used as described in the previous paragraph, so from the manager’s perspective there is no difference between RPC events and NDB events.

### 4.1.2 Aggregation mechanism

Persisting in a non-volatile storage solution is expensive due to exclusive locks, OS buffers, I/O interrupts etc. On top of that, persisting data in a remote database, introduces a few milliseconds of network latency as well. For that reason, when a TS is ready to be committed, it forks a new thread which will handle the actual database operations to persist its state. Having the commit mechanism
4.1. TRANSACTION STATE AGGREGATION

Figure 4.2: Activity diagram of RPC batching system
parallelized, multiple TS can be persisted concurrently. At that point, we have just invalidated our consistency model. Sooner or later we will reach the case where two TS would be committed in the wrong order corrupting the state of the database. The mechanism explained in Section 4.1.2.1 controls the commit phase of each TS guarantying that two conflicting TS will be committed in the correct order. Section 4.1.2.2 will outline the shortcomings of the existing mechanism and describe the new mechanism built that extends the previous.

### 4.1.2.1 One TS per commit

The mechanism that controls the commit phase of TS should allow as much as possible parallelism without violating our constraints. The constraints are:

- If the RPCs that are batched in Transaction State $TS_0$ have been fully handled before the RPCs batched in $TS_1$ and they both modify the same **YARN Application**, then the commit phase of $TS_0$ should have been successfully completed before the commit phase of $TS_1$ begins.

- If the RPCs that are batched in Transaction State $TS_0$ have been fully handled before the RPCs batched in $TS_1$ and they both modify the same **NodeManager**, then the commit phase of $TS_0$ should have been successfully completed before the commit phase of $TS_1$ begins.

- If Transaction States $TS_0$ and $TS_1$ modify different YARN Application *and* different NodeManager, they can be committed in parallel.

Every TS keeps a list with the IDs of the applications it modifies and a list with the IDs of the nodes it modifies. In the commit mechanism, there is a FIFO queue for each and every application ID and node ID. Before committing a TS, the mechanism puts it in the corresponding queues both for the applications and the nodes it modifies. In order for the TS to be committed, it should be in the head of the queues for the corresponding application IDs and node IDs. If it is not in the head of one application or node queue, it means that a previous TS modified the same application or node and should be committed before. The un-committed TS is put in a queue to be examined later.

The following example will make clearer how the commit mechanism works. To make it easier, let us assume that the only lock is on application IDs and we have only three applications running. The FIFO queues for the applications are depicted in Figure 4.3. The initial state of the system is in Figure 4.3a where no TS has been handled yet and the queues are empty. Technically, the queues for the application IDs and node IDs are created lazily when the TS has been fully handled and is ready to be committed. RPCs for $TS_0$ have been handled and is
4.1. TRANSACTION STATE AGGREGATION

Figure 4.3: Example of TS for the queue to be committed

ready to be committed in the database. It modifies entries for applications with ID app_0 and app_2, so it is placed in the respective queues, Figure 4.3b. TS_0 is the head of the queues for the applications it modifies so it can start the commit phase. Since a commit phase may fail and roll-back, it will be removed from the queues when the transaction has been successfully completed.

While TS_0 tries to persist its modifications to the database, two more Transaction States have finished and are put in the queue to be committed, Figure 4.3c. TS_1 modifies applications app_0 and app_1 while TS_2 modifies applications app_1 and app_2. The commit mechanism checks whether TS_1 can be committed. Remember that TS_0 is still committing to the database. The mechanism fetches the applications that TS_1 modifies and checks if it is the head in each queue. For app_1 it is in the head of the queue but not for app_0, TS_0 should finish first. So it will be examined after TS_0 is done. The same applies for TS_2.

TS_0 is now complete and removed from the queues of the applications it modified, Figure 4.3d. TS_1 is now the head for app_0 and app_1 so it starts the commit phase. This is not the case though for TS_2 which still has to wait for TS_1 to finish. TS_1 successfully completes its transaction, removed from the queues and now TS_2 is in the head position, Figure 4.3e so it can start committing to the database.
4.1.2.2  Multiple TS per commit

The existing mechanism described in the section above provided the consistency model we required and parallelism for non-conflicting TSs. The major issue that had to be addressed was that for a conflicting TS (1) it had to wait in the queue for each and every of the previous TSs to finish, in the example above TS_1 and TS_2 and (2) for every transaction committed we paid the network latency penalty and the time to acquire locks in the database etc. In order to solve the first problem we examined different solutions. Initially, a more fine-grained locking system was tried. Instead of locking on application and node IDs, try to lock on containers. Although this solution would increase parallelism, it was very risky that we would end-up with a corrupted state. Next solution was to remove the system with the queues and replace it with exclusive reentrant locks. The expected result was to increase performance but at the end it was the same.

The solution proposed in this section and implemented reduces both the wait time in the queue for a TS to be committed and the RTT for each transaction performed. The mechanism extends the method described in Section 4.1.2.1 by aggregating multiple TSs into a single one while it guarantees consistency. The queue system still gives us the proper order in which the TSs should be persisted. At the beginning a TS is examined to determine if it should be persisted in the database. If it is not possible to be persisted due to conflicting TSs then it is put in the toBeAggregated queue. If it is permitted to commit, then it does so and the commit mechanism constructs an extended TS called AggregatedTransactionState.

The AggregatedTransactionState contains TSs from the toBeAggregated set that are eligible for commit according to the following aggregation rules:

1. A TS was not the head in its respective queues at the time it was examined for commit, but until now the conflicting TS(s) have been committed and removed. So now it is in the head of the queues.

2. A TS is not in the head of the queues, but all the conflicting TSs that should be committed before it, have already been aggregated in the AggregatedTransactionState.

The first rule is trivial and we have examined it in the previous section. For the second rule, the mechanism gets the modified application and node IDs from a Transaction State TS_a. Then it retrieves all the conflicting TSs from the appropriate queues that are blocking TS_a from being committed. If all of the conflicting TSs have already been aggregated – put in the AggregatedTransactionState, then the mechanism aggregates TS_a as well and it proceeds by examining the next TS in the toBeAggregated set. At the end of this process we will end-up with
4.1. TRANSACTION STATE AGGREGATION

Figure 4.4: Aggregate commit mechanism example

a “big” Transaction State, the AggregatedTransactionState that will be committed in the database. The AggregatedTransactionState class actually extends the TransactionState class and for every TS that is aggregated it updates the data structures with the modifications of that TS. Internally, the update process could be either insertion of new values in the data structures or overwriting existing values. The toBeAggregated data structure is a FIFO queue so the TSs that are examined for aggregation are kept in the correct order. At the end of the aggregation process, the data structures of the AggregatedTransactionState hold the correct, most recent, modifications to be persisted.

Consider the state of the queues as in Figure 4.3c. Both TS_1 and TS_2 cannot be committed because they are blocked by TS_0 so they are added to the toBeAggregated queue. At some point TS_0 is committed in the database, removed from the queues as in Figure 4.3d, the AggregatedTransactionState (ATS) is created and the aggregation process is started. TS_1 is the first candidate for aggregation. At that point TS_1 is at the head of its respective queues and following the aggregation rule 1 it is aggregated. All the data structures of TS_1 are copied to the data structures of the ATS. Next in the toBeAggregated set is TS_2. TS_2 is not the head in the queue for application app_1 but the conflicting TS, TS_1, has already been aggregated. So according to aggregation rule 2, TS_2 is also aggregated, probably overwriting the values that TS_1 has put in TSA. The toBeAggregated set is now empty and the “big” AggregatedTransactionState is committed to the database. Compared to the previous commit mechanism, where we have done three commits in the database, we have now reduced it to two.

Another case that demonstrates the improvement we have achieved is described in the following example. Consider the locking queues as depicted in Figure 4.4a for two applications. All the Transaction States are blocked behind TS_0
which is at the commit phase. At that time the `toBeAggregated` set contains `TS_1-TS_5`. `TS_0` finishes its commit phase, removes itself from the locking queues and begins the aggregation phase. First it examines `TS_1`. It is in the head of the queue for application `app_0` and according to aggregation rule 1 it should be aggregated. Next in the `toBeAggregated` set is `TS_2`. The conflicting state is `TS_1`, which is already aggregated, so according to aggregation rule 2 it is aggregated too. `TS_3` is in the head of the queue for `app_1` and all the conflicting TSs for `app_0` have been aggregated so it is also put in the `AggregatedTransactionState`. Similarly, `TS_4` and `TS_5` are also aggregated. Now the `AggregatedTransactionState` contains the modifications of `TS_1`, `TS_2`, `TS_3`, `TS_4`, `TS_5`, Figure 4.4b, and begins the commit phase. With the previous commit mechanism, every TS would have performed the commit phase individually introducing considerable delay due to the RTT to the database, whereas with the new commit mechanism we only require two commits in the database.

Aggregating several TSs into a single one introduced some erroneous behaviour. NDB is very performant with transactions of small size but the aggregation mechanism was overloading the transaction and NDB was throwing errors. In order to mitigate this issue a TCP-like aggregation control was put in place. In the beginning, the mechanism starts to aggregate a small number of TSs. When the commit phase for that `AggregatedTransactionState` is successfully completed, the limit is increased by some delta. It continues to increase until an error message is received from the database. The transaction is rolled-back, the limit for the number of aggregations is set to minimum and the mechanism tries again to aggregate. Users can implement their own policy as it fits their needs without having to change the commit mechanism code.

### 4.1.3 Conclusions

Overall, the new commit mechanism addresses both the waiting time in the queue and the communication latency. TSs do not have to wait in the queue for all the previous conflicting TSs to be committed. Once one conflicting TS is committed then the mechanism aggregates as many as possible TSs. Moreover, since multiple TSs are squashed into a single one, there is only one commit phase reducing the network latency.

### 4.2 Foreign key constraints

In Hops for reasons that have been outlined in previous chapters we persist all the metadata in our persistent storage solution. MySQL Cluster is a relational
4.2. FOREIGN KEY CONSTRAINTS

distributed database, so data are stored into tables with very specific properties. Since version 7.3.1, MySQL Cluster supports foreign key constraints. Foreign keys is a powerful feature of relational databases that guarantee some kind of consistency between two tables. One important aspect of foreign keys is that they map relationships of the “real-world” into relationships in the database.

The database schema of Hops consists of 95 tables, 64 out of them are used by Hops-YARN. The information they store span from incoming RPCs to scheduler state and nodes’ statuses. We make heavy use of foreign key constraints, mainly the ON DELETE referential action, to ensure integrity when a row from a parent table is deleted. The database schema of Hops-YARN is depicted in Figure 4.6. The foreign key relationships are illustrated with a solid line between the parent and child tables. There are two reasons why we actually have relationships between tables. The first one is because there is indeed semantically a relationship between the tables. For example in Figure 4.7, the table `yarn_rmnode` is used by the RM to store information regarding the available nodes in the cluster. Table `yarn_containerstatus` on the same figure stores some status information for the containers that have been launched. Obviously there is a relationship between the containers and the nodes of a Hadoop cluster. A container cannot be launched in a node that has not been registered with the RM yet. Similarly, a container should not exists in the database if the node row has been deleted. The second usage of foreign keys is to group together tables, such as the tables in Figure 4.8. These tables persist the heartbeats sent by the AMs and NM. For a reason that will be explained in Section 4.2.2, the information that a single heartbeat carries had to be partitioned and stored in multiple tables. Again, we do not want orphan entries and we avoid this by the ON DELETE action.

Foreign keys in relational databases is a great feature. They do not come without a cost though. For bulk and frequent operations they kill performance. In Figure 4.5 is presented the result of a micro-benchmark measuring the time needed to perform database operations with and without foreign key constraints. The red line in the diagram is the time needed to remove rows from the tables in Figure 4.8. Then the schema has changed and removed the foreign keys. The time to remove the RPCs from the database without foreign keys is illustrated with the light blue colour. The impact on performance of foreign keys in our schema was huge. The performance for persisting data with (blue line) and without (brown) foreign keys is comparable until 1500 RPCs but then the schema without the foreign keys performed better. After the improvements introduced by the new commit mechanism, the removal of foreign key constraints would improve the performance even more.

Although the main reason we use foreign keys is to get the automatic cascading in the deletion of a row, this has a side effect to insertion operations as well. A row in a child table cannot be inserted in the database before its parent,
otherwise the RDBMS will complain about missing foreign key. To solve it the parent row has to be inserted first, then flush the buffer and finally insert the child row. This imposes an order and cancels out any parallelism.

In our case we aim to scale to 10k nodes in a cluster it means that we have to remove these constraints and replace them with some logic in the program that will cascade the deletion to children tables. It is very important that this logic will consist of primary key operations for two reasons. The first reason, which applies to all relational databases, is that primary keys are indexed, thus avoid making full table scan. The indexes usually are implemented with a B-tree and/or hash table data structure which allows operations in logarithmic or constant time depending on the query. The second reason, which applies specifically to NDB, is that primary keys are also partition keys, if no partition keys are defined explicitly. A partition key defines how data will be distributed across Node Groups in the MySQL Cluster. Executing statements based on the primary key, alleviates the round-trip time between the Data Nodes, since NDB will know exactly in which machine the row you are looking for is stored. In non primary key operations, NDB might have to lookup in all Node Groups to execute the statement. The rest of this section will describe how the foreign keys were replaced with application
4.2. FOREIGN KEY CONSTRAINTS

logic.

4.2.1 yarn_rmnode

NodeManager (NM) is the process that runs on every physical node on a Hadoop cluster. The RM keeps information for every NM represented in memory by the RMNode class. The information this class holds is persisted in the yarn_rmnode table, Figure 4.7. This table is parent table to numerous other tables such as yarn_node which holds information about the name of the node, its location etc. Another table is yarn_latestnodehbresponse which holds the last response sent by the RM to that NM. In total there are 10 tables having foreign key constraints to yarn_rmnode. Ideally we would like to replace them with a logic that performs deletions with primary key operations. The problem in that case is that the children tables are diverse and have multi-column primary keys, so keeping in memory all the primary keys would not be efficient. Especially when the deletion operation of an RMNode would not happen very frequently, only in the case of a dead node. Moreover, in later versions of YARN, RMNodes are never removed from RM’s state, just having their status changed.

With all these in mind we decided to remove the foreign key constraints from the children without doing primary key operations. In the client library, hops-metadata-dal-impl-ndb, application logic was implemented that whenever an RMNode row would be deleted, this deletion would be cascaded to the rest of the tables. Children tables have the rmnodeid either as part of their primary key or indexed, so we avoid a full table scan. Moreover, rmnodeid is the partition key of that table so we avoid unnecessary round trips among the NDB Data Node groups. The implementation was very straightforward, when the remove method in the client library is called for an RMNode, all the rows from the corresponding tables are selected, filtered by the rmnodeid and finally remove them. Considering that this operation would occur rarely and in later versions never, that was a fine solution trading performance with memory usage.

4.2.2 yarn_allocate_response

Next in the list with tables that are referenced with foreign keys is yarn_allocate_response. It holds the response sent by the RM to an allocate request by an AM. This is the case where we had to partition the data persisted. MySQL Cluster has a limit to the size of a single row. That limit is 14000 bytes [36] and in some cases the data that are to be persisted exceeds it. So the allocation response is partitioned to yarn_allocated_containers and yarn_completed_containers_status, both having foreign keys to yarn_allocate_response. Since the deletion
Figure 4.6: Hops-YARN database schema
Figure 4.7: A node in the cluster from the scheduler’s perspective
Figure 4.8: Database tables to store incoming RPCs
operation of an allocate response would happen very often, primary key operation was the only way to achieve high performance. Both tables have a multi-column primary key with the applicationattemptid, the containerid and the responseid. An allocate response is removed from the database when (1) a new response for that application is created, (2) the ApplicationMaster unregisters with the RM. The first case will be examined in Section 4.3. When an application unregisters all the container IDs that the application was using are fetched from the RM in-memory state and the response ID, then the primary keys are constructed and issue primary key deletion operations for those three tables in parallel.

4.2.3 yarn_ficascheduler_node

The scheduler, and specifically Fifo and Capacity, holds information for the nodes in the cluster in its own data structure identified by the class FiCaSchedulerNode. This class is persisted in NDB in the table yarn_ficascheduler_node which is parent for the table yarn_launchedcontainers. The later keeps a map of the container IDs that are launched to a scheduler node. Its primary key consists of the nodeid (FiCaNode) and the containerid. The delete method on yarn_ficascheduler_node Data Access Object is called when a node is deleted from the scheduler’s view. At that point the node representation already holds the IDs of the containers that were running on that node. Similarly, the primary keys of the child table are constructed for every containerid and issue the deletion operation for both yarn_ficascheduler_node row and yarn_launchedcontainers rows.

4.2.4 yarn_applicationstate

Hops-YARN persists in the database back-end the state of every application scheduled. This state includes information such as the name and the user of the application, the Application submission context, diagnostics etc. Since applications and specifically AMs can fail, YARN also tracks the attempts an application made to be scheduled. If the AM fails, then YARN creates a new application attempt for that application and spawns the new AM. The state of the application is stored in yarn_applicationstate table in the database and every attempt for that application in yarn_applicationattemptstate. Application attempt semantically has a relationship with the application and in the relational world is expressed with a foreign key constraint between yarn_applicationstate (parent) and yarn_applicationattemptstate (child). When an application is completed it is removed from the state of the scheduler and all its attempts. Reconstructing the primary key for the attempts


4.2.5 yarn_appschedulinginfo

An application from the scheduler’s (Fifo, Capacity) point of view is represented by the class FiCaSchedulerApp. This class contains data structures that should be persisted such as resource requests, the application attempt, any blacklisted resources etc. All this information is stored in yarn_appschedulinginfo table which is parent to yarn_appschedulinginfo_blacklist containing blacklisted resources for a specific application attempt. In order to efficiently remove blacklisted resources when an yarn_appschedulinginfo row is removed, the primary key of yarn_appschedulinginfo_blacklist should be built, which consists of the application attempt ID and the ID of the blacklisted resource and then execute the delete operations.

4.2.6 yarn_schedulerapplication

SchedulerApplication class is the base class for FiCaSchedulerApp that we have examined previously. This naturally translates into a foreign key constraint between yarn_appschedulinginfo – child, and yarn_schedulerapplication – parent. In this case we have three tables chained together with foreign keys. A deletion in table yarn_schedulerapplication, will trigger a deletion in yarn_appschedulinginfo which in turn trigger a deletion in yarn_appschedulinginfo_blacklist. This greatly deteriorates the performance both for insert and delete operations. The foreign keys dictate a very strict order on the insertion of rows in these tables. If these statements are executed in parallel then there is no guarantee about the order and an error might occur compromising the data. That limitation forces us to excessively flush the buffer of the transaction manager introducing network latency.

4.2.7 yarn_appmaster_rpc

Finally, the last table that other tables had references to, was yarn_appmaster_rpc. The database schema is illustrated in Figure 4.8. These tables are used to persist incoming RPCs so in a crash scenario, the RM would recover and replay them. At the time of replacing the foreign key relationships with primary key operations we have decided to leave this set of tables as it is. Later we saw that this setup was greatly affecting the performance and the mechanism described in Section 4.3 came in place.
4.2.8 Conclusions

Having all, or most of, the foreign key constraints replaced by primary key operations improved performance and made the database schema more flexible. Performance was increased for two reasons. Without any constraint we are able to remove the flush operations from transactions. In some cases we do need to maintain an order in our operations but that is limited. Removing flushes allowed for increased parallelism in database operations while decreased the network latency that we were paying for every flush operation. In most of the cases, building the multi-column primary key was an easy task but it was a great opportunity to get familiar with YARN and NDB and a good warm-up for the changes to come.

4.3 Garbage Collector service

In Section 4.2 I have explained how the foreign key constraints were removed from Hops-YARN. After profiling more the system, we realized that this was not enough. There were still some cases that should be improved. These include the time spent when committing a Transaction State in the database. The actual time required to store some information to the database was too high. Two cases have been distinguished that they were very slow for two different reasons explained below.

The first set of operations that were slow to commit were regarding the RPCs we store for recovery. As it is already mentioned Hops-YARN recovery procedure rebuilds the state of the scheduler from the state persisted in the database. Moreover, every time the ApplicationMasterService or the ResourceTrackerService receive an RPC they persist it in NDB as well. If the RM crashes, it will restart or the standby RM will become active, read the latest RPCs stored in the database and simply replay them. RPCs in the database are stored in the tables that were depicted in Figure 4.8. We can distinguish between two types of RPCs. The first type is the heartbeat that the ApplicationMaster sends to the ApplicationMasterService, in order to report progress of the application, diagnostics and of course allocation requests. The second type is the heartbeats that NodeManagers send with the statuses of the containers, health status of the node etc. Unfortunately, these RPCs can get quite big in size and considering the limitation of a row size in NDB they had to be split in multiple tables. For example, if a NM runs 1000 containers then in the heartbeat it will include the status for all of those. Also, when an AM has just started and makes an allocate request, this request can contain hundreds of requirements. So the RPCs are split and stored in the tables as illustrated in Figure 4.8. The yarn_appmaster_rpc
table holds general information that is common for both RPCs such as the rpcid and the type of the RPC whereas the specific information for allocations or NM status is stored in the other tables.

The life cycle of an RPC is as follows. First a method is invoked by an RPC, it decodes the request, which is encoded into a Protocol Buffers encoding. Then it stores the request in NDB into the respective tables, it gets the current Transaction State where it adds the rpcid. Events are triggered, containing the Transaction State and handled by the RM components. When all the events have been handled the RPCs that were previously stored are deleted from the database. If the RM crashes somewhere in the middle, then AMs and NM do not have to resend the same information, since RM will read the last heartbeats from the database. Although, that seems a very good architecture, persisting and deleting thousands of rows in the database every second is a tough work and a gain of a few milliseconds in commit time improves the overall performance.

If you recall from Section 4.2.7, we have not removed the foreign key constraints from the database schema for these tables. Before removing completely the constraints we have experimented with a tree structure of the foreign keys as in Figure 4.9. In this schema, only the tables yarn_heartbeat_rpc and yarn_allocate_rpc would have foreign keys to yarn_appmaster_rpc. The results of the micro-benchmark in Figure 4.5 has shown that a schema with two foreign key constraints had similar performance with a no foreign key constraints schema. Then the tables specific to AM allocate and NM heartbeat would have foreign keys to yarn_allocate_rpc and yarn_heartbeat_rpc respectively. The problem with this setup was the five child tables of the yarn_allocate_rpc table. Considering the overhead of five foreign keys we have concluded to remove all the constraints from these tables and implement the service described in the rest of this section.

The removal of the previously persisted RPCs is part of the transaction that commits scheduling decisions in the database. If we are slow in this part, this will increment the commit time of the whole Transaction State which would make other Transaction States to be blocked more time in the wait queue having a direct impact on the cluster utilization. Persisting new RPCs is done when an RPC arrives and does not directly affect the Transaction State commit time.

In the new implementation, an extra table in the database has been created, called yarn_rpc_gc which stores the rpcid and the type of an RPC. The type can be either HEARTBEAT or ALLOCATE. When an RPC arrives and invokes a method, this method will get the Transaction State from the Transaction State Manager. We add the rpcid and the type of that RPC in the data structures of the TS and are persisted in the database. When all of the RPCs have been handled properly, at the same transaction, only the rows from the yarn_appmaster_rpc table are removed and not from the others, opposed
Figure 4.9: RPC alternative schema
to the previous solution. The entries at the other tables are taken care by the Garbage Collector service.

**Garbage Collector** (GC) is a Hops-YARN service that purges asynchronously old RPC entries from the database. The main thread of GC reads the rpcids and types from the yarn_rpc_gc table. Then it creates a number of threads that will delete the rows with a specific rpcid from all the other tables. The threads will create queries that will remove the rows where the column rpcid equals the specific rpcid fetched by the yarn_rpc_gc table. The tables affected for the AMs’ allocate RPCs are:

- yarn_allocate_rpc
- yarn_allocate_rpc_ask
- yarn_allocate_rpc_blacklist_add
- yarn_allocate_rpc_blacklist_remove
- yarn_allocate_rpc_release
- yarn_allocate_rpc_resource_increase

The rows for the NMs’ heartbeats are removed from the tables:

- yarn_heartbeat_rpc
- yarn_heartbeat_container_statuses
- yarn_heartbeat_keepalive_app

The only performance drawback that we have with the GC service is that the deletion queries do not operate on primary keys. There is a trade-off here. The current implementation uses only one table for the RPCs to be removed that store the rpcid and the type. For that reason the whole multi-column primary keys cannot be constructed but we persist less data in the Transaction State. Since, the removal of the RPCs is done asynchronously it does not affect the commit time of the TS. Also, the rpcid in the RPC tables is indexed so we do not perform a full table scan and it is also the partition key so we avoid RTT among NDB Node Groups. The other alternative would be to store all the necessary information to build the primary keys for the rows to remove. This would effectively mean that we would need to persist more data during the commit of the TS that might increase the commit time. Also, we should have had more than one table to store them, making the schema more complex.

The asynchronous removal of entries from NDB off-loaded the Transaction State commit phase in that extend that we decided to use it also for another
4.3. Garbage Collector service

case that was taking more time to persist data than that we desired. That is the case of Allocate Response. For every heartbeat the RM receives, it generates an allocate response that is sent back to the AM. This allocate response is persisted in the database through the Transaction State for recovery reasons. In case of an RM crash, it (RM) will recover the last allocate response and send it to the AM. In some cases it can contain a lot of information for example when the AM registers and requests all of its allocation requests. When the scheduling decision is made, the scheduler creates a response with the containers allocated. The tables used in the database to store such allocate response are: yarn_allocate_response, yarn_allocated_containers and yarn_completed_containers_statuses. The main problem in that case is that for every heartbeat received, the allocate response of the previous heartbeat has to be deleted from the database. The delete operation should remove probably hundreds of rows of allocated containers and completed containers’ status. The same method was followed as described above with the RPCs. A new table was put in the schema, yarn_alloc_resp_gc, which is storing the necessary information for the GC to remove previous allocation responses. That way, during the commit phase of the Transaction State we are only adding new responses and not removing the old ones, reducing more the commit time of a TS.

Removing entries from the database in an asynchronous way raises some questions about the consistency model. For example what will be the state recovered by the scheduler if old RPCs or allocate responses have not been deleted yet. Regarding the recovery of RPCs, when the failed RM tries to construct the last state from the database, it fetches all the entries stored in the RPC tables and joins them according to their rpcid. The detail that makes the system work is that it consults the yarn_appmaster_rpc table about the rpcids of the RPCs that had not been handled when the RM crashed. Since the entries for that table are removed synchronously in the commit phase of the Transaction State, we guarantee that this table will not contain any rubbish. So the RPCs that will be joined and re-constructed will be the ones that have not been processed at the time the RM crashed. The only drawback is that it might take more time to read the RPC tables since they might contain more rows than actually needed. The GC runs on the RT so until the new RM makes a transition from standby to active and recover the persisted state, the old RPCs would have been collected. The same philosophy applies for the Allocate Responses. The old allocate response of an application is removed synchronously from table yarn_allocate_response, by the Transaction State. Each response has an incremented ID. At any time, there is only one ID for each application attempt stored in this table. When the scheduler constructs the last allocate response before it crashed, it filters the entries read from yarn_allocated_containers and yarn_completed_containers_
statuses with the valid response ID.

4.3.1 Conclusions

The new transaction state commit mechanism and the removal of foreign key constraints improved the performance of Hops-YARN both in terms of commit time and cluster utilization. Some delete operations that were implicitly dictated by insert operations were replaced by the Garbage Collector service. This reduced the number of SQL queries when committing a Transaction State in the database while guaranteed the required consistency model.

4.4 DTO Caching mechanism

In Hops in order to communicate with the MySQL Cluster NDB we make use of ClusterJ [37], a high level API to perform operations on NDB. In a sense it is similar to other ORM frameworks such as Hibernate [38] and EclipseLink [39] which provide an object-relational mapping but more lightweight and is designed to provide high performance methods for storing and accessing data on a MySQL Cluster from a Java application. Every operation on Hops and Hops-YARN, except for the events received from the NDB Event API, goes through ClusterJ which in turn uses the C++ NDB API. The mapping between a table-oriented view and a Java object is done through specially decorated interfaces. The interface provides signatures for the getters and setters methods.

For example Listing 4.1 shows the interface for accessing database entries for the Garbage Collector service regarding old RPCs. The interface is annotated with the table name and contains signatures for accessing each column of the table. The methods are also decorated with the primary key annotation and the column name. For every table in the database there exists such an interface and all the operations from Hops-YARN are done on the Data Transfer Objects (DTO) defined by the annotated interfaces. DTOs are created from a session object which represent a connection to the MySQL Cluster by calling the newInstance method.

Upon completion of the task discussed in the previous section, we profiled again the commit phase of a Transaction State to discover spots that could be possibly improved. Surprisingly we discovered that we suffered from the overhead of creating DTOs with ClusterJ. To measure the overhead we have created a micro-benchmark that is creating and persisting a number of DTOs. The results are shown in Figure 4.10. The blue line represents the time that ClusterJ needed to create the DTO instances when calling session.newInstance. The yellow line is the time we spent to persist them in NDB, while the red one is the sum of those two. Throughout the benchmark we are spending more time
4.4. DTO CACHING MECHANISM

Listing 4.1: ClusterJ annotated interface

```java
@PersistenceCapable (table = TABLE_NAME)
public interface GarbageCollectorDTO {

    @PrimaryKey
    @Column (name = RPC_ID)
    int getrpcid ();
    void setrpcid (int rpcid );

    @Column (name = TYPE)
    String gettype ();
    void settype (String type );
}
```

creating the object instances than actually persisting them. The time to create the DTOs grows linearly to the number of DTOs and faster than to persist them in the database. Creating more than 6000 DTOs is not an extreme scenario when we aim to scale Hops-YARN over 10000 NodeManagers.

In ClusterJ, DTOs are created from database session object. Creation of a new DTO involves the instantiation of several objects such as the handlers for the type of values the DTO will persist in NDB. Upon creation of all the necessary handlers, it invokes the `session.newInstance` reflective method of Java. Java reflection API is a powerful tool but comes with some pitfalls including performance [40]. Reflection API loads types dynamically therefore JVM optimizations cannot be applied making it a bad candidate for high-performance applications. Changing the implementation of ClusterJ is a very difficult task and was not considered as an option. Moreover, we do not want to maintain one more project.

In Hops, `HopsSessions` are wrapped around ClusterJ sessions. The solution we designed is a DTO cache for the database sessions. A session has its own cache space that is filled up with instances created by the “slow” ClusterJ instantiation process. When we actually need to use a DTO, we fetch it from the cache which is faster since the objects have already been created. When the cache has been used a worker thread fills it up again with new instances. The cache generator service should be used cautiously. We should not max-out CPUs just for creating cached DTOs, so the cache is enabled only for a fraction of sessions and only for heavy-duty DTOs. The reason we decided to have both cache-enabled and cache-disabled sessions, is to avoid marking a cached session as used even though the cache was never used. An overall workflow of Hops-YARN database session
provider is illustrated in Figure 4.11. A transaction requests a cache-disabled session from the cache-disabled session pool (1). If there is a session available – not used by other transactions – then the session provider return a session from the pool, otherwise it creates a new one. When the transaction has been committed to the database, the session is returned back to the pool of cache-disabled sessions (3). The workflow for a cache-enabled session regarding the DB session provider is different. There are two different cache-enabled session pools. The first one, *Preparing pool* contains sessions with their cached used and probably empty. The second pool is the *Ready pool* with sessions whose cache is full and ready for usage. When a transaction requests a cache-enabled session, the session provider first looks on the *Ready pool* (1). If there is an available session it returns it to the requester (2), otherwise it looks into the *Preparing pool*. Finally, if there is no session there either, it creates a new session to NDB. When the transaction has performed its operations, it returns the session to the *Preparing pool*. The cache generator service picks sessions from the *Preparing pool* (1), fills up their cache with the appropriate DTO instances and places them back to the *Ready pool* (2).

The cache itself, `DTOCacheImpl` is implemented as a `ConcurrentHashMap` whose key is the type of DTO cached and its value is a `CacheEntry` object. The `CacheEntry` is supported by an `ArrayBlockingQueue` providing methods for putting and getting cached objects and increasing the cache size. We will
Figure 4.11: Hops-YARN DB session pool
see later how the size of the cache is increased. For every HopsSession that has its cache enabled, there is an instance of DTOCacheImpl which provides methods for (de)registering a DTO type to the cache and methods that delegate putting and getting objects to the appropriate CacheEntry. There are two ways to “instantiate” DTO objects. The first one is the newInstance which makes a call to ClusterJ to create the object. We use this for non-cached DTO types. The other variation is the newCachedInstance which makes a call to the cache instead. In this version the DTO has been instantiated ahead of time and it is fetched from the cache. The semantics of the cache is to return the cached object if it exists in the cache and null if the cache is empty. In case of an empty cache, we fall back to the ClusterJ instantiation method. Every CacheEntry keeps track of how many cache-misses have been occurred. If there are too many, it means that this DTO type is very demanding and the cache size is increased every time the cache-misses exceed a certain threshold. So, at the same cache-enabled session two DTO types might have different cache size depending on their “popularity”.

When a cached-enabled session has been used, it is put back in the Preparing pool. The DTOCacheGenerator is a service that picks sessions from that pool and fills their cache. It removes a number of sessions from the Preparing pool and it spawns threads that populate every CacheEntry that is not full. The instantiation of the cached DTO objects is done with the session.newInstance method of ClusterJ. CacheEntry returns true for every put, until the back-end ArrayBlockingQueue is full when it returns false. When all the CacheEntries of a session are full, it (the session) is placed to the Ready pool and another transaction will use it. One note should be made here. ClusterJ allocates memory for DTO objects out of the heap, with the use of Java direct ByteBuffer. ByteBuffers do not account in Java’s garbage collection mechanism, reducing the GC pause time. Also, they are ideal for heavy duty I/O operations since JVM does not have to copy data from intermediate buffers to native buffers. With the caching mechanism we create more than 6000 DTOs ahead of time per session and the default direct memory size reaches its limit very quickly. In order to avoid related exceptions, the flag -XX:MaxDirectMemorySize should be set to a reasonable value.

All the parameters for the caching mechanism are specified by a configuration file, dto_cache-config.xml in hops-metadata-dal-impl-ndb that is loaded when the service starts. A typical example looks like in Listing 4.2. First, it is specified how many sessions will have their cache enabled, then a hard limit on the number of threads that will be created to populate the sessions’ cache and the number of sessions each worker thread will handle. Then in line 14 is the number of sessions in the Preparing pool required to trigger the cache generator threads. If sessionsInterval > sessionsPerThread * threadLimit, then
Listing 4.2: Caching mechanism configuration file

```xml
<?xml version="1.0" encoding="UTF-8"?>
<dtoCache>

<!-- The number of cache enabled sessions -->
<sessions num="40"/>

<!-- The number of worker threads that will fill up the cache -->
<threadLimit limit="4"/>

<!-- The number of sessions each worker thread will populate -->
<sessionsPerThread sessions="10"/>

<!-- The number of sessions in Preparing poll the workers will wait until they start filling up the cache -->
<sessionsInterval interval="20"/>

<!-- List of cache elements -->
<!-- Class of the DTO interface -->
<dto class="io.hops.metadata.ndb.dalimpl.yarn.PendingEventClusterJ">

<!-- Name of the interface -->
<name>PendingEventDTO</name>

<!-- Initial cache size -->
<initialSize>12000</initialSize>

<!-- Maximum cache size -->
<maxSize>20000</maxSize>

<!-- Increasing step of the cache size after some cache-misses -->
<step>400</step>

</dto>

<dto class="io.hops.metadata.ndb.dalimpl.yarn.NodeHBResponseClusterJ">

<name>NodeHBResponseDTO</name>

<initialSize>2000</initialSize>

<maxSize>7000</maxSize>

<step>200</step>

</dto>

</dtoCache>
```
some sessions will be blocked until a thread has finished its work and scheduled again. Following the mechanism parameters, are the DTO types that should be cached. In this example we want each cache-enabled session to cache two types of DTOs. Line 18 is the class name that encloses the ClusterJ specially decorated interface. Line 21 is the name of the interface and then follows the initial size of the cache, the maximum size and the step which the cache size will be increased after some threshold of cache-misses.

4.4.1 Conclusions

The caching mechanism explained in this section boosted the performance of Hops-YARN as it is presented in Chapter 5. Yet we have not discovered the advantages in full extend since we need to experiment more with different types of DTOs. For sure ClusterJ’s way of creating DTOs is not optimal, imposing a great overhead to our system.
Chapter 5

Evaluation

This chapter will present the results from the evaluation of the project. In each section the impact of this project will be presented in terms of cluster utilization or individual time for a transaction to be committed in the database. The simulations were performed with two different configurations outlined below.

1. Six machines cluster with Intel® Xeon® E5-2630 CPU and 250 GB RAM. The RM and RT were running on two of them and the rest were used by the distributed simulator. Two node MySQL Cluster NDB, with AMD Opteron™ 2435 CPU and 30 GB RAM. Between the simulation cluster and the MySQL Cluster there is a Gigabit connection.

2. Six machines cluster with Intel® Xeon® E5-2630 CPU and 250 GB RAM. The RM and RT were running on two of them and the rest were used by the distributed simulator. Two node MySQL Cluster NDB running on two of the cluster machines with 10GbE network between the simulation cluster and the database.

For the simulations, a distributed simulator was used which parsed workload tracefiles for a various number of NodeManagers and created application requests. To distribute the load, the simulators were running on different machines and reporting back to the master.

5.1 Commit mechanism

The impact of the aggregation mechanism in the Transaction Manager was measured with a synthetic payload that should be persisted in NDB. The total number of transaction commits was measured, the average time required by
each transaction and the total time taken for the payload to be committed. The comparison is between the Simple and Aggregated commit mechanism.

The payload is issuing 100 Applications. Each application would uniformly randomly request or release 1000 containers on 30 nodes chosen also uniformly at random. A new Transaction State object object was created with 60% probability over re-using an existing one. Finally, when all this information was tracked, the Transaction State objects were persisted in the database in a foreign-key free schema. The reason to pick hosts and TS objects randomly is to create conflicts in the commit mechanism of the Transaction Manager that would block TSs. In that way we could measure the impact of the aggregation mechanism.

The results of the evaluation are presented in Table 5.1. It is clear that the aggregation mechanism minimized the commits in the database. In the Simple mechanism each TS is performing a commit phase whereas in the Aggregated mechanism, multiple TS are squashed into a single commit. It is normal that the average time needed to commit a single TS in the database is greater in the new version, since now the TS contains much more information. In some cases the AggregatedTransactionState was aggregating up to 200 Transaction States. Also, some times the commit time per TS was above 150 ms but in general the standard deviation is very low with most of the results near the mean value. Overall, we spend less than the half of the time we spent with the Simple mechanism to persist the whole payload.

### 5.2 Asynchronous deletion

In this section we are going to report the results from simulations for the Garbage Collector service described in Section 4.3. The main problem that led to that solution was the commit time of the Transaction State since it had to perform some extra operations. The variables that were evaluated are the time to commit a Transaction State object and the cluster utilization with and without the Garbage Collector service. The simulations were conducted in the first setup described earlier in this chapter.

In Figure 5.1 is depicted the total time taken in milliseconds to persist the TS for 3000 and 5000 nodes in the cluster. The time to commit affects both
5.2. ASYNCHRONOUS DELETION

Figure 5.1: TS commit time with GC service

the scheduler and the ResourceTracker regarding the updated view of the cluster and this has a direct impact on the cluster utilization. For the simulation with 3000 NodeManagers the average commit time, even though it was already low it dropped to 24 ms from 40 ms. The difference is more visible for the simulation with 5000 nodes in the cluster where the commit time dropped to 60 ms from 150 ms. In general we can observe a tension to half the commit time and this is reasonable for several reasons. First and foremost is because we do not have the foreign key constraints in the tables that store RPCs (Figure 4.8). Also, the removal of old values is done asynchronously so no overhead in the commit phase of the TS. Finally, we have removed some flush operations from the application logic that had to be there before to serialize the queries.

The improvements on the commit time had a direct impact also in the cluster utilization as we can see in Figure 5.2. For a 3000 nodes cluster the utilization remained almost the same but it was already high enough. Nevertheless, we have managed to improve the cluster utilization by a few percentages for a 5000 nodes simulation, reaching 86 %. The number of containers created in a 5000 nodes cluster increased by 10% with the Garbage Collector service while in a 3000 nodes cluster it remained almost the same. Moreover, there is an interesting side effect with decreased commit time. Since, TSs take less time to commit, the blocking TSs that cannot be aggregated also spent less time in the waiting queues of the Transaction Manager. That leads to “smaller” but
Figure 5.2: Cluster utilization with GC service

more AggregatedTransactionStates. For instance, in the simulation with 5000 NodeManagers without the GC service, there were roughly 900 TS commits each containing on average 2000 objects to persist just to update the status of NodeManagers. On the contrary, with the GC service, there were 1690 TS commits with 960 objects to persist for the NodeManagers. This side effect helped vanish the rollback of transactions due to overloading (Section 4.1.2.2), since there were not that many TSs in the queue to aggregate.

5.3 DTO caching

In Section 4.4 the issue of creating Data Transfer Objects with ClusterJ was introduced. In Figure 4.10 is depicted the time needed to create a number of DTOs and persist them in the database. The evaluation of the Caching mechanism was done both with a micro-benchmark but also with simulations measuring the time to commit a Transaction State in the database, the cluster utilization and also the number of heartbeats that the scheduler is processing. The simulations were performed in the second type of setup.

In Figure 5.3 is the result of the same micro-benchmark performed in Section 4.4 but with the Caching mechanism. The blue line is the creation time of DTOs,
Figure 5.3: DTO creation and commit with cache enabled

The yellow line is the actual commit time and the red one is both the creation and the commit time. The cache size of the sessions was 6000 objects. The “creation” time in the second case is almost zero and orders of magnitude less, until the cache limit is reached. Even when the number of DTOs requested are more than the cache size, it is still much faster. At the time the benchmark was done, the cache size of the sessions was fixed to a number. It is expected the performance to be better with the dynamic size of the cache.

The next evaluation parameter is the mean time for a Transaction State to be committed in the database. The parameters of the cache are outlined in Table 5.2. It is not possible to cache all the DTO types mainly for two reasons. First, it would take more time to fill-up the cache with more types which would result in very few ready sessions. So the transactions would use non-cached sessions or even worse fall-back to creating new sessions. The second reason is memory related. Even though DTOs are not allocated on the heap of the JVM and do not account to Java garbage collection, they still consume memory from the machine that could be used for scheduling decisions or handling of heartbeats. In Figure 5.4 is the average commit time of a TransactionState object with cache-enabled and cache-disabled sessions. Even with a cluster size of 2000 nodes there is a small difference in the commit time. As the number of NodeManagers grows, the difference is getting wider reaching almost 20 ms in 10000 nodes cluster.

The improvement in the commit time is reflected in the cluster utilization as
Table 5.2: Cache mechanism configuration

<table>
<thead>
<tr>
<th>Type</th>
<th>Min. size</th>
<th>Max. size</th>
<th>Step</th>
</tr>
</thead>
<tbody>
<tr>
<td>PendingEventDTO</td>
<td>12000</td>
<td>25000</td>
<td>400</td>
</tr>
<tr>
<td>NodeHBResponseDTO</td>
<td>2000</td>
<td>10000</td>
<td>200</td>
</tr>
<tr>
<td>UpdatedContainerInfoDTO</td>
<td>2000</td>
<td>10000</td>
<td>200</td>
</tr>
</tbody>
</table>

Figure 5.4: TS commit time with and without cache

Figure 5.5: Cluster utilization with and without cache
5.4. Performance overview

Until now we have presented the improvements in performance that each step of the implementation has contributed. In order to verify that the goals of this project have been achieved a comparison should be made between the latest version of the Hops-YARN with the version before the thesis. Also, since Hops-YARN is a fork of Apache YARN, the big question is how our implementation performs in comparison with the upstream YARN in terms of cluster utilization. In Figure 5.8 is depicted the cluster utilization for various number of nodes in a cluster. The yellow line is the develop branch of Hops-YARN, the code base before this thesis. The blue line is the merge_tx_no_fk branch which is the final version of this thesis and the red line is Apache Hadoop 2.4.0. There are two observations illustrated in Figure 5.5. Until 3000 nodes there is no difference in the utilization but the more NodeManagers in the simulation, the bigger the difference is. In 10000 nodes simulation the cluster utilization increased from 56% to 77% and the launched containers from 167177 to 247032 (Figure 5.6).

Finally, the number of heartbeats processed by the scheduler node has also been increased but still is quite low affecting the scheduling decisions. In Figure 5.7 is depicted the ratio of heartbeats processed by the RM over the total number of heartbeats. For a 10000 nodes cluster the ratio is still very low but until 5000 NodeManagers is quite acceptable.
CHAPTER 5. EVALUATION

that should be noticed in this figure. The first one is the improvement over the develop branch. Already from 3000 nodes we can notice a small difference. Starting from 3000 nodes, the cluster utilization of the develop branch drops very roughly until 10000 nodes where it is 35%. With the modifications proposed in this thesis the cluster utilization curve is more flat with an acceptable rate for the whole range of simulations.

The second observation is the gap between merge_tx_no_fk and Hadoop 2.4.0. Until 7000 NodeManagers the difference is very small, 88% and 92% respectively. Even with the largest configuration of 10000 nodes the difference is 13% in cluster utilization which is not negligible but considering the lower recovery time of Hops-YARN and the distributed ResourceManagers it might be a valid trade-off.

The next evaluation parameter for the project is the ratio of heartbeats processed by the scheduler over the total number of heartbeats. Hops-YARN has distributed the ResourceTrackerService which receives heartbeats from NodeManagers, persists the information received in NDB and then it is streamed to the master ResourceManager which updates its view of the cluster. In Figure 5.9 is illustrated the heartbeat ratio for the develop and merge_tx_no_fk branch and Hadoop 2.4.0. We can observe a great improvement up until 4000 NodeManagers. Although the difference is small for the rest of the simulations, it is constantly better than in develop branch of Hops-YARN. Apache Hadoop never drops below 94% but it is necessary to mention that Apache YARN is a centralized architecture with no
communication latency between the ResourceTrackerService and the scheduler.

Finally, for the completeness of the evaluation it is worth to mention the CPU utilization and memory consumption (Figure 5.10, 5.11) of the ResourceManager – bbc7 and the ResourceTracker – bbc6. Throughout the simulation with 10000 nodes in a cluster, the average CPU usage of RM never went above 20% while for the RT the maximum average is 45%. Regarding the memory consumption for the same simulation, both RM and RT never used more than 24GB of RAM.
Figure 5.9: Ratio of processed heartbeats

Figure 5.10: CPU usage for RM (bbc7) and RT (bbc6)
Figure 5.11: Memory usage for RM (bbc7) and RT (bbc6)
Chapter 6

Conclusions

This chapter draws the conclusion for this thesis report. In Section 6.1 an overall review of the work is presented and Section 6.1.1 evaluates the goals set in Section 1.3. Finally, Section 6.2 gives some general directions for future work on the project.

6.1 Conclusion

In Hops-YARN, as it is already mentioned, MySQL Cluster NDB is used both as a communication transport and as a persistent storage for recovery. At the beginning of the project a thorough profiling of the execution workflow has been done to identify the bottlenecks of the system. The first step analyzed in Section 4.1 was to improve the commit mechanism of Hops-YARN’s Transaction Manager. The new mechanism squashes several blocked transactions into a “big” one, reducing the number of commits in the back-end database system. Section 4.2 describes the evolution to a database schema with no foreign key constraints and how they were replaced by application logic that performs primary key operations instead. Section 4.3 presents the Garbage Collector service of Hops-YARN that asynchronously removes old values from the database. With that solution the commit time dropped even more improving the overall performance of the system. Finally, Section 4.4 explains how the shortcoming of ClusterJ for creating DTOs was bypassed by having created them ahead of time in a per session cache.

After a detailed explanation of the solutions proposed, in Chapter 5 follows the evaluation. Each solution is evaluated separately by simulating real world traces. In each case key characteristics are examined and how they have been improved. In Section 5.4 there is an overall performance overview in terms of cluster utilization and heartbeats processed by the scheduler. The comparison is made among the version of Hops-YARN before this project, the final version with
all the modification proposed and the upstream Apache YARN. The figures show that there was a clear improvement, in both evaluation parameters, when the two version of Hops-YARN are compared. Finally, as far as the cluster utilization is concerned, the performance is comparable with Apache YARN in clusters with thousands of nodes.

6.1.1 Goals achieved

In Chapter 1 the goals of this project were set. The primary goals was to improve the cluster utilization and the number of heartbeats processed by the scheduler. In order to achieve those goals we have also set some sub-goals. With the solutions proposed in Chapter 4 all the sub-goals were met. In particular, with the removal of the foreign key constraints and the DTO caching mechanism the transactional commit time was decreased dramatically. Some sort of asynchronous API was provided by the garbage collector service. It is provided only for a small subset but still it made big difference to the performance of the system. The new aggregation mechanism of the transaction manager of Hops-YARN helped the blocked transactions to be committed faster which in turn improved the parallelization of the system. Finally, in each step of the implementation an evaluation was done to prove the performance impact and guide us to new bottlenecks.

Since all the sub-goals were met it was expected to achieve the primary goals. As Chapter 5 indicates the two primary goals were also accomplished. Both cluster utilization and the heartbeat ratio was improved.

6.2 Future work

A few things have been left undone due to time limitation and are discussed in this section for future work.

In most of the cases the evaluation has been done using simulations measuring among others the cluster utilization while in others the evaluation has been done using benchmarks. It would be more complete if in all cases the benchmarks were supported by simulation results. That way the performance improvements introduced by each step would be more clear.

Currently we do not have any insight on the content of Transaction State objects, thus they are treated equally. In real world scenarios, during the allocation of resources, the Transaction State would carry more information about allocated containers than when the cluster is full and no further allocations can be made. A fine-grained inspection on the content of a TS might improve performance further more. For example in the commit mechanism, when an Aggregated Transaction
State object is overloading NDB, the transaction will roll-back and the aggregation policy will enforce a lower limit. If we had any information on the content of the TS before hand, this situation could have been avoided.

As it is already mentioned, the Garbage Collector service does not perform primary key operations. It was a design decision that would not complicate the mechanism and burden the commit time by committing more information. It is worth trying to persist all the columns needed for the primary keys to be reconstructed and measure the performance. For sure the deletion time would be lower but it remains to be proven if that will have any impact on the performance of Hops-YARN in general.

Last but not least, the batching system explained in Section 4.1.1 should be improved. Changing the number of RPCs that are batched together did not change the performance. Every RPC arriving in the ResourceManager among others it should get the current Transaction State object from the Transaction State Manager. This operation could take 0.06 ms. If we consider a cluster with 10000 NodeManagers heartbeating every second, it is needed 600 ms just to acquire the object. At the time of writing that was the latest bottleneck encountered and for sure it is worth for future investigation.
Bibliography


[38] Hibernate ORM. Aug. 12, 2016. URL: http://hibernate.org/orm/.
