Eloy Gil Guerrero

Providing key-value data services on HPC infrastructures

Master Thesis

Master in Innovation and Research in Informatics
Computer Networks and Distributed Systems

Supervision

Yolanda Becerra Fontal
Department of Computer Architecture (DAC)

October 2018

Contents

Abstract **iii**

1 Introduction **1**

1.1 Context and Motivation . . . 2

1.2 Scope . . . 3

1.3 Goals of the project . . . 4

2 Background **5**

2.1 Job schedulers . . . 5

2.2 Key-value data stores . . . 6

2.2.1 Cassandra . . . 7

2.2.2 ScyllaDB . . . 10

2.3 Hecuba . . . 10

2.4 Parallel programming models . . . 11

2.4.1 PyCOMPSs . . . 11

2.4.2 MPI . . . 13

3 Requirements **15**

3.1 Database cluster configuration . . . 15

3.2 Application execution . . . 16

3.3 Data persistence management . . . 18

4 State of the Art **21**

5 Design and implementation **23**

5.1 Data persistence solutions . . . 23

5.1.1 Multi-data centre architecture . . . 23

5.1.2 Snapshot storage approach . . . 28

5.2 Automatic cluster allocation and configuration for HPC . . . 32

5.2.1 Job launcher 33
5.2.2 Job Script 36
5.2.3 Run Application 38
5.3 Usage and usability 39

6 Experiments **45**
6.1 Determining scripts overhead 46
6.2 Snapshot performance 47
6.3 COPY vs. Recover snapshot 49

7 Conclusions and Future Work **53**

Bibliography **55**

Abstract

The need of this work arose in the context of a European funded project which one of the goals was to evaluate the performance of neuroscientific applications over different data stores, focusing on the comparison of the traditional approach using HDF5 and the alternative NoSQL-based approach. Compared to relational databases, distributed NoSQL databases provide a better performance, flexibility, and ease in scalability that is critical for data-intensive applications, but they were not intended to be executed on HPC systems and their configuration and persisting data methods need to be manually adapted to this type of environments. The main goal of this project is to automatise these tasks. The experiments need to be executed on a supercomputer based on a queuing system, that makes possible to manage the application and the hardware allocation where it is going to be executed, depending on the selected requirements. The resources are assigned dynamically and the user does not know the allocation that will be used for the job until it is already being executed, and even if this configuration could be done manually, it would be a tedious and complex task that would force the user paying attention to the start of the job to be able to configure the database before its deployment.

HPC installations usually have a network shared file system (such as GPFS) that that can be accessed from any node, before, during and after the execution of a job. However, in order to exploit the potential performance of distributed databases, it is convenient to make use of local disks attached to the computing nodes. Local disks are usually intended to store temporary files because they have limited capacity. For this reason, once a job finishes its execution, it is not guaranteed the durability of its stored data, and if the generated information is not stored properly it will be lost forever. There is a need to ease the way a member of the scientific area can execute applications over this type of storage without needing to be familiar with the system's administration in that low-level, for its advantages, and also a simple and automated procedure to store the results and recover them later, if needed, to check or reuse them in a future execution.

Chapter 1

Introduction

High-Performance Computing (HPC) applications usually rely on traditional file systems, like HDF5, to store their input and output data because traditional relational databases do not fit their requirements. However, this type of file systems exhibit some limitations that complicate the interface with the programmers and that may penalise the final performance of the applications. These limitations could be overcome through the utilization of NoSQL databases.

NoSQL databases are widely adopted in production environments. While relational databases rely on tables, columns, rows, and schemas to organize the stored data, NoSQL databases do not rely on these structures, using more flexible data models. A distributed NoSQL database provides a better performance, flexibility, and ease in scalability that is critical for data-intensive applications.
NoSQL follows a horizontal scale-out methodology that makes simple to add or reduce capacity using commodity hardware and without affecting to the service, in contrast with all the manual tuning needed while scaling relational database management systems. They can be used to ensure a high availability avoiding the complexity that comes with typical RDBMS architectures that rely on different types of nodes and components[1] that are exercised during each interaction. Some NoSQL databases use a masterless architecture that automatically distributes the data among multiple nodes, making the system fault tolerant. The data may be replicated not only among multiple servers and across data centres, minimize latency and ensure a consistent application for any user.

However, NoSQL databases were not intended to be executed on HPC systems and, thus their configuration and persisting data methods need to be manually adapted to this type of environments, which may penalise the productivity of users.

1.1 Context and Motivation

The need for the work done during this Master Thesis arose in the context of a European funded project: the *Human Brain Project, Specific Grant Agreement 1, SubProject 7* or HBP SGA1 SP7. One of the goals of this project[21] was to evaluate the performance of neuroscientific applications using different data stores. In particular, the project focused on the comparison of the traditional approach using HDF5 and the alternative NoSQL approach. The experiments need to be executed on a supercomputer based on a queuing system, that make possible to manage *what* it is going to be executed (an application) and *where* it is going to be executed (which hardware will be used for that allocation), depending on the requirements and needs of this specific *job*. In this type of environments, the option of keeping a database on execution as a global service of the system is not convenient because of several reasons as, for example, in order to avoid interferences with other applications should have allocated permanently in an exclusive way some nodes of the machine.
Thus, the alternative is to deploy the database as a regular job submitted to the queuing system. In this scenario, the resources are assigned dynamically and the user does not know the allocation that will be used for the job until it is already being executed.
Typically, the configuration and deployment of a distributed database require to know in advance the specific nodes and communication ports that can use. For this reason, the user should configure the database once the job allocation is performed. Technically, this configuration could be done manually, but this would be a tedious and complex task that would force the user paying attention to the start of the job to be able to configure the database before its deployment. For this reason, automating this process is a must.

Another consideration to deploy a database in these systems is regarding the storage. HPC installations usually have a network shared file system (such as GPFS) that is physically in a remote location that makes possible to be accessed from any node, before, during and after the execution of a job. However, in order to exploit the potential performance of distributed databases, it is convenient to make use of local

disks attached to the computing nodes: this avoids the potential interferences in accessing the shared file system and simplifies the configuration of the location of the data.
Usually, in an HPC environment, local disks are intended to store temporary files because they have limited capacity. For this reason, once a job finishes its execution, it is not guaranteed the durability of its stored data. If the generated information is not stored properly it will be lost forever. There is a need to ease the way a member of the scientific area can execute applications over this type of storage without needing to be familiar with the system's administration in that low-level, for its advantages, and also a simple and automated procedure to store the results and recover them later, if needed, to check or reuse them in a future execution.

1.2 Scope

This project is focused on solving the needs of a task in the HBP SGA1 SP7. The use case for this task was a workflow with several tasks manipulating brain images. In particular, we focused on the first task of the workflow. The original version was an MPI application. The input data for this task are slices of the human brain, and a segmentation algorithm is going to be applied over them to find individual cells on them. These input images are stored in Hierarchical Data Format files. The proposed alternative was to use Cassandra, a key-value NoSQL database, as storage. In order to adapt the application, the Hecuba set of tools was used in order to facilitate the interface with the database. The utilisation of Cassandra turns the configuration and launch of Cassandra clusters into a really frequent process. During the development of the task, it was explored the use of ScyllaDB as a Cassandra replacement, since they have a common interface. For this reason, some parts of this project were adapted to support both databases even the initial goal was to focus just into Cassandra. Finally, there was the interest in comparing the MPI application with an alternative version using the PyCOMPSs programming model. The experiments were performed on the Marenostrum 3[10] and Marenostrum 4[11] supercomputers, which use the IBM's LSF and SLURM queuing systems respectively. All these requirements bound the scope of this project.

1.3 Goals of the project

The main goal of the project is to provide users with a set of tools to facilitate the utilization of key-value data stores on HPC installations by:

- Automatising the configuration and deployment of a key-value database.
- Configuring and launching an application or a set of applications using the deployed database.
- Automatising the management of the data files of the database to guarantee data persistence while exploiting the potential performance of the local disks.

Chapter 2

Background

During the development of this project, there are many technologies involved that have their own utility inside the proposed system. These tools cover the resource management in a cluster, the distributed database that will be used for storage and the different parallel programming models that must be supported for the application executions.

2.1 Job schedulers

A job scheduler[8] is a computer application meant to control unattended background program execution of jobs. This is commonly called batch scheduling, the execution of non-interactive jobs is often called batch processing, though traditional job and batch are distinguished and contrasted. The data structure of jobs to run is known as the job queue, and they could be executed following a different order depending on the priority given by the number of resources needed or how important they are.

Modern job schedulers, often termed workload automation, typically provide a user interface and a single point of control for definition and monitoring of background executions in a distributed network of computers of any kind. Increasingly, job schedulers are required to orchestrate the integration of real-time business activities with traditional background IT processing across different operating system platforms and business application environments. They typically make possible to choose the number of resources to be used by a specific job, those options include configuring the number of cores and nodes to be used, maximum main memory constraints, grouping similar nodes into labeled queues to make easier to choose which hardware will execute the task, dependencies between jobs or its maximum execution time, among others. In particular, there are two big job schedulers for HPC clusters that are widely used: IBM's LSF and SLURM.

Platform Load Sharing Facility (or simply LSF) is a workload management platform for distributed High-Performance Computing. It can be used to execute batch jobs on networked Unix and Windows systems on many different architectures. LSF was based on the Utopia research project at the University of Toronto. Platform Computing was acquired by IBM in January 2012, and a few months later, in August 2012, an agreement was signed between the Spanish government and IBM in order to update the BSC's supercomputer to what was known as Marenostrum 3[10]. This explains why LSF was the software used for the workload management on its nodes until its next upgrade to Marenostrum 4 during 2017, that, among other technical upgrades[11] it also uses SLURM as the job scheduler.

Slurm (Simple Linux Utility for Resource Management) is an open-source job scheduler that allocates compute resources on clusters for queued researcher defined jobs. Slurm has been deployed at various national and international computing centres, and by approximately 60% of the TOP500 supercomputers in the world. As in the case of LSF, an option for running a job on an HPC cluster is to set up a job script. This script will request cluster resources and list, in sequence, the commands that you want to execute. A job script is a plain text file that can be edited with a UNIX editor such as vi, nano, or emacs. To properly configure a job script, it is needed to know the general script format for this queuing system, the commands to use, how to request the resources required for the job to run, and, in this case, some of the Slurm environment variables.

2.2 Key-value data stores

A key-value database, that is also known as a key-value store[9], is a data storage paradigm considered the most flexible of NoSQL database. They have emerged as an alternative to the limitations of the traditional relational databases, as they are designed to manage associative arrays, that nowadays are better known as dictionaries or hash tables. In a key-value store the values are accessed using a *key* and can store numbers, counters, strings, binaries, among other types, providing a flexible data modeling, high performance, and availability while being massively scalable.

2.2.1 Cassandra

Apache Cassandra[23] is a free and open-source distributed wide column store NoSQL database management system designed to handle large amounts of data across many commodity servers, providing high availability with no single point of failure. Cassandra offers robust support for clusters spanning multiple data centres, with asynchronous masterless replication allowing low latency operations for all clients[27]. Cassandra was originally developed at Facebook, was open sourced in 2008, and became a top-level Apache project in 2010. Cassandra clusters are deployed in production environments in large companies such as Apple or Netflix.
There are many key concepts and ideas[7] to understand how Cassandra works: Among the possible architectures for valid distributed database systems, in the case of Cassandra it follows a shared nothing architecture, meaning that each node is independent of the others and there is no master or distinguished role assigned to any of them. A single logical database is spread across a cluster of nodes, appearing a need to spread the data in a balanced manner among all of them. Each node will be responsible for a portion of the stored data, and the act of distributing it across nodes is known as *data partitioning.*
Also, the partition of data on a shared-nothing system, as each node is in charge of a portion of the total data, the potential failure of one of them results in a single point of failure. The way this is avoided is copying the data to other nodes, known as replicas, so the act of storing copies of data on many nodes is referred to as *data replication*, that is a common practice to ensure fault tolerance and reliability in distributed systems.
While trying to distribute data efficiently, many challenges appear: First, the need of a quick way to determine on which node should reside a specific piece of data, and also the need of minimising the amount of data moved when adding or removing nodes. *Consistent hashing* makes these goals possible, one of these algorithms enables the mapping of Cassandra row keys to physical nodes. The range of values from a consistent hashing algorithm is a fixed circular space that is typically represented as a ring. On average, only k/n keys need to be remapped where k is the number of keys and n is the number of nodes. This is the main difference between consistent hashing and most hashing algorithms, where a change in the number of nodes results in the need to remap a huge amount of keys.
In every distributed system where the data may be modified and checked at some point, there is a need to talk about its *data consistency* since all the information is spread across nodes and it must be ensured that it is synchronized across every

replica. In distributed computing is frequent to find a model known as *eventual consistency*, that theoretically guarantees that if there are no updates all nodes and their replicas will eventually give back the last updated value. A popular example of this kind of consistency is the *Domain Name System* (DNS), that during the time it takes to fully propagate those changes to every replica, the data is not consistent. In Cassandra, the number of replicas in a cluster that must acknowledge an operation before considering it as successful is not only configurable, it is a required parameter in any read and write operation and determines the exact number of nodes that must complete the operation to be considered successful, known as *consistency level*.
Those nodes are physically located somewhere, and this has a real impact in the topology of the cluster. Figure 2.1 represents a single data centre Cassandra cluster. They may be divided between *Data Centres* (DC), that are centralised locations

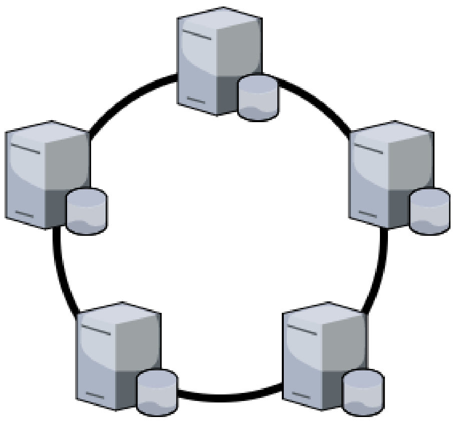
Figure 2.1: 5-node Cassandra cluster

where the servers and networking systems are placed together, a *rack* is a unit that contains multiple stacked servers one on top of another, to conserve floor space inside a data centre. Finally, a *node* is a single server in a rack, and this differentiation is important because Cassandra is typically deployed in data centres where to avoid having a single point of failure it is critical to make an appropriate replication of the data. Replicating data to servers in different racks ensures continued availability in case of a rack failure, and also Cassandra can work in multi-DC environments to support fail-over and a possible disaster recovery that could be performed in a transparent manner for the users.
Cassandra also uses *snitches* and *replication strategies* to determine how data is replicated across nodes, racks and data centres. Those snitches are useful to determine the proximity of nodes within a ring, while replication strategies use that

proximity information to determine the locality of a particular copy.
Moreover, it also uses a *gossip protocol* to discover every node status in a cluster, those nodes discover other nodes while exchanging information about themselves and other nodes that are part of the cluster, facilitating failure detection. Each node does this with a maximum of three other nodes to reduce network load.
There are also many important concepts related to the way data is stored: *Bloom filters* are used to quickly test the existence of a data structure in a set and avoid expensive I/O operation. While in this approach false positives are possible, false negatives are not.
A *Merkle tree* is a hash tree that provides an efficient way to find differences in data blocks. It is a tree in which every leaf node is labeled with the hash of a data block and every non-leaf node is labeled with the cryptographic hash of the labels of its child nodes. Hash trees allow efficient and secure verification of the contents of large data structures. What it is used in this case is called *Sorted String Table* (SSTable), that it is an ordered immutable key-value map and an efficient way of storing large sorted data segments in a file. Each SSTable files of a column family are stored in its respective column family directory. It is possible to backup of all on-disk data files stored in the data directory, creating what is called a *snapshot*.
A *memtable* is a *write-back cache* because the write operation is only directed to the cache and its completion is immediately confirmed, residing in memory which has not been flushed to disk yet. In the RDBMS world, there are *schemas*, while in Cassandra there are *keyspaces* that are seen as containers for all the application data.
When defining a keyspace it must be set a *replication strategy* and a *replication factor* to configure the behaviour of the system when creating replicas. At the same time, a *column family* is analogous to the concept of a table in an RDBMS (in fact, in Cassandra Query Language (CQL) it is referred to as table), but in Cassandra it is a map of sorted maps: A row in the map provides access to a set of columns, which is represented by a sorted map. The *row key* is also known as the *partition key*, it has a number of columns associated with it and it is responsible for determining data distribution across the cluster. Finally, Cassandra will locate any data based on a partition key that is mapped to a *token* value by the partitioner. Tokens are part of a finite *token ring* value range where each part of the ring is owned by a node in the cluster. The node owning the range of a certain token is said to be the primary for that token.
Cassandra also includes a *cqlsh* utility[5], a python-based command line client that

can be used to perform operations executing CQL commands.

2.2.2 ScyllaDB

The typical design of NoSQL data stores consists of a JVM which runs on top of a Linux distribution, utilizes the page cache, and uses complex memory allocation strategies to try to avoid the stop-the-world pauses the JVM garbage collector does, suffering from sudden latency hiccups, expensive locking, and low throughput due to low processor utilization.
Meanwhile, ScyllaDB is one of the newest NoSQL databases which offers extremely high throughput while keeping the latencies under one millisecond and implements almost all of the features of Cassandra but this time built in C++. The Scylla design[20] is based on a modern shared-nothing approach, running multiple engines, one per core, each with its own memory, CPU and multi-queue network interface card. It can reach one million CQL operations on a single commodity server.

2.3 Hecuba

In the 1990s the Internet gained extreme popularity, and relational databases simply could not keep up with the flow of information demanded by users, as well as the larger variety of data types that occurred from this evolution. This led to the development of non-relational databases that can translate strange data quickly and avoid the rigidity of SQL by replacing "organized" storage with more flexibility. Non-relational databases are nowadays a common solution when dealing with huge data sets and massive query workload. These systems have been redesigned from scratch in order to achieve scalability and availability at the cost of providing only a reduced set of low-level functionality, thus forcing the client application to take care of complex logic. As a solution, our research group developed Hecuba[6], a set of tools and interfaces, which aims to facilitate programmers with an efficient and easy interaction with non-relational technologies.
In Hecuba, a developer can interact with objects stored in a Cassandra-like database seamlessly while operating like if they were in memory. In this python example the user is defining a class which inherits from Hecuba's StorageDict:

```
from hecuba.hdict import StorageDict
class Words(StorageDict):
   '''
   @TypeSpec <<int>,str>
   '''
```

This could be placed inside of the python code or an external file (i.e. words.py) and imported from any other script:

```
from words import Words
```

These objects can be instantiated now. After doing so they can be used as regular dictionary objects. In this case, it provides a dictionary that uses integers as index and strings as the values to be stored.

```
ret = Words()
ret[1939] = 'World War II starts'
ret[1969] = 'First man on the Moon'
```

They could be directly created as persistent objects providing a name, thus it will be also stored and changed in the database every time it is modified. If it is not done in the first place it may be persisted later, with a simple instruction used to provide a name for it:

```
ret.make_persistent('historic_events')
```

A name can be used during the instantiation to create directly a persistent object and also later by the same or a different application that is using the same database cluster to access to the values that were already stored there:

```
ret = Words('historic_events')
print ret[1939]
> 'World War II starts'
```

Hecuba enables a simple way to store and retrieve data from a distributed database.

2.4 Parallel programming models

2.4.1 PyCOMPSs

PyCOMPSs is a framework which aims to ease the development and execution of Python parallel applications for distributed infrastructures[28], such as Clusters and

Clouds.
It is the Python binding of COMPSs[3], a programming model and runtime which aims to ease the development of parallel applications for distributed infrastructures, such as Clusters and Clouds. The Programming model offers a sequential interface but at execution time the runtime system is able to exploit the inherent parallelism of applications at a task level. The framework is complemented by a set of tools for facilitating the development, execution monitoring and post-mortem performance analysis.
A PyCOMPSs application is composed of tasks, which are methods annotated with decorators following the PyCOMPSs syntax. At execution time, the runtime builds a task graph that takes into account the data dependencies between tasks, and from this graph schedules and executes the tasks in the distributed infrastructure, taking also care of the required data transfers between nodes.
In this case, the *@task* decorator right before the function declaration is used to both indicate that this function should be executed in parallel and also specify the type of the returned value, that in the example in listing 2.1 a dictionary. PyCOMPSs will identify this function to create the corresponding tasks that will be scheduled to be executed across the worker nodes.

```
@task(returns=dict)
def wordcount(data):
    result = {}
    for entry in data:
        if entry in result:
            result[entry] += 1
        else:
            result[entry] = 1
    return result
```

Listing 2.1: PyCOMPSs parallel python function example

In this case, it will construct a frequency word dictionary from a given list of words. The result will be a dictionary where the key is the word, and its value the number of appearances of each one. It is a very basic example that could be way more difficult to be solved when the data is split into different files, then it should be merged to reduce the multiple partial results into a final one, that could also be performed in parallel. These operations are handled by the use of functions like *compss_wait_on* that are a kind of barrier that guarantees that the execution does not continue until all the related tasks are processed and the workers have returned their data.

2.4.2 MPI

The *Message Passing Interface* Standard (MPI) is a message-passing library[29] standard based on the consensus of the MPI Forum, which has several participating organizations, including vendors, researchers, software library developers, and users. The goal of MPI is to establish a portable, efficient, and flexible standard for message-passing that will be used for writing programs. As such, MPI is the first standardized, vendor-independent, message-passing library.

The advantages of developing software using MPI closely match the design goals of portability, efficiency, and flexibility[12]. MPI has become the *industry standard* for writing message-passing programs on HPC platforms, but it is not an IEEE or ISO standard. MPI is an interface specification for the developers and users of message-passing libraries, the specification of what such a library should be but not a library by itself. In MPI data is moved from the address space of one process to that of another one through cooperative operations on each process.

Simply stated, the goal of the Message Passing Interface is to provide a widely used standard for writing programs while attempting to be practical, portable, efficient, and flexible. Nowadays, MPI runs on virtually any hardware platform, it doesn't matter if they use a distributed, or shared memory or they are a hybrid approach.

Chapter 3

Requirements

This chapter explains the main problems and challenges that lead to this effort to solve them.

3.1 Database cluster configuration

The aim of this section is to describe the minimum number of options that are necessary to configure the deployment of Cassandra in an HPC environment. Notice that in a traditional production environment, if nothing goes wrong, a distributed database is deployed and configured only once. This means that after configuring and testing its performance for some time their configuration remains stable until a change is needed.
If there is no need to add (to scale up horizontally) or replace (to fix the total failure of a node) the cluster may stay as it is for a long period of time. This is convenient for production services that should be highly available while delivering low latencies from reliable data storage. However, in *High-Performance Computing* systems based on queuing systems, the database will be deployed as a regular job and, thus this configuration should be repeated each time the database is executed. For this reason, the automation of the configuration process is still more relevant.

Most of the Cassandra attributes are set in a configuration file. Each Cassandra process can read its own configuration file but the first thing to consider is that all the nodes that are part of the cluster must agree on the settings of some attributes: the *cluster name*, *datacentre*, and also the ports that will be used for different communications (see section 5 for more details).

Other important attributes indicate where to store the data and the auxiliary data. To avoid the interferences these directories should be different for each Cassandra process. As it was mentioned before, to get a higher performance the nodes may use their local storage instead of a shared network file system. As each Cassandra process run on a different computing node it is possible to use the same path for all processes without causing interferences. Some of the paths that need to be specified are the *data* location, the *commit log* file, and the *saved caches* directory.

In the configuration file, there must be some *seeds* that are provided in order to discover the whole members of the cluster through a gossip mechanism. These seeds are physical node interface IPs or hostnames which the DNS must resolve to the IP of a valid network interface to establish the different communications. In a dynamic environment, this step must be done every time a cluster is launched since the allocation may be different in every execution. As a distributed system it may take some time until all the nodes have been discovered and have joined to the cluster, this status must be checked periodically to start an application right after the system is ready.

At this point, it is important to remark that Cassandra expects to have a different IP address for each process in its cluster. In HPC environments the resources reservations are usually made based on the number of CPUs that are needed to process a task, which by default can be assigned inside the same computing node and, thus can share the same IP address. This is something to consider when specifying the job allocation and to force each process in the Cassandra cluster to be in a separate node with different IP addresses.

3.2 Application execution

Regarding the execution of the client application, there are many scenarios that should be supported. The application or applications may be invoked from the same job, sharing the node resources (such as CPU cores and RAM memory). This scenario may be convenient depending on the network topology of the cluster to avoid unnecessary data transfers between the nodes that are storing the information and the ones that will make the computation needed by the application. Moreover, it could be interesting to be aware of the place where each piece of data has been assigned inside the cluster in order to exploit the *spatial locality*, hopefully reducing, even more, the time spent on unnecessary transfers. In this scenario, represented in figure 3.1, the sharing of resources between the application and the database may lead into conflicts that should be taken into account and, thus it is important to make a smart management of the memory in order to avoid crashes that might happen. Another scenario is to use the same job to launch both the database and the applications but on a disjoint set of computing nodes, as seen in figure 3.2.

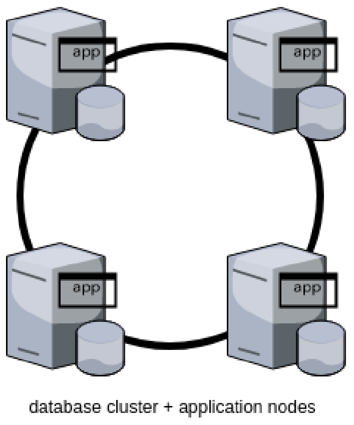

Figure 3.1: Shared nodes scenario

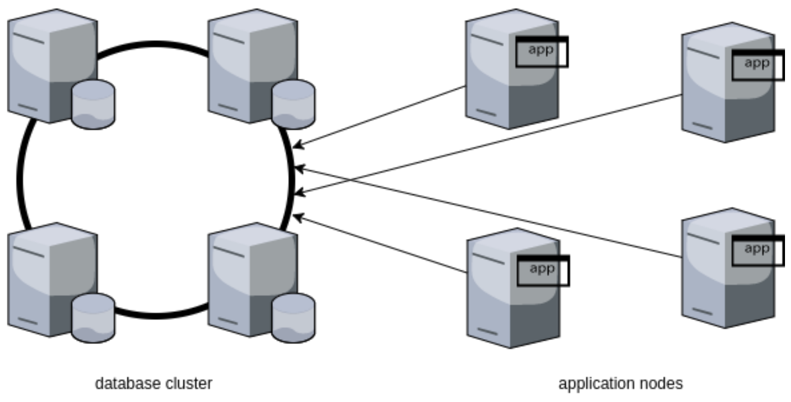

Figure 3.2: Disjoint nodes scenario

It is also interesting to offer the option of launching the application on different jobs with their own assigned resources that use the database system remotely. This allows users to allocate a long-lasting job with the database and different jobs with the applications that need to be executed.

In any case, there is a need to configure the application to find the storage, that might be different depending on the type of the application but surely will be hosted by different resources if the storage cluster is not static.

3.3 Data persistence management

Typically, when an application is being executed, it will allocate memory to store information. Sometimes it will be temporal but also there is data that could be intentionally *persisted* into a file, or in this case, into a distributed database. This could be the case of objects that may be accessed concurrently by different processes, even in different nodes, leaving the consistency tasks to the database system. When the cluster is static, this persistence guarantees that the data is not going to disappear if something goes wrong after this (i.e. an unexpected electricity shortage, job abortion, etc.) but when the database cluster itself is built over a dynamic resource allocation system there is a new layer that leads into new opportunities and considerations to take into account.

While using storage clusters that are alive for a limited amount of time there is an important challenge making accessible the data that is being produced as the result of the execution of the application or set of applications that have been using them during its lifetime for a posterior use of the information. For some tasks it will be fine if the objects and data generated are lost forever right after the execution, for instance, a complex computation that finishes in a single value, or *short* amount of output lines could be printed and this data could be easily obtained from the log files, and the related conclusions made without worrying about the intermediate information, no matter if it was effectively persisted or not. This may be the case for some of the most common use cases, but also there will be cases where the information stored in the database is complex and/or big enough to remain there instead of being printed, and some of the possible needs would be to recover it for a next iteration using existent data or recovering it just to check the results in a different moment or place.

Moreover, as it was mentioned before, to maximise the performance it will be recommended to make use of the local storage in each node. This kind of storage is typically composed by one or several *SSD* devices, *PCIe* ones in some cases, that enables a higher bandwidth, improving the rates for *read* and *write* operations but also creates a new barrier between data and remote accesses. If a job is aborted unexpectedly in the middle of an execution this data will remain stranded in the disk until it is deleted by an automated procedure, if stored in a temporal location, or until the user gets access to the same node to reach it and use it or delete it.

Also, in these systems, the files are not simply plain text files that can be read, or edited, so the *post-mortem* process of re-building the cluster manually would be really hard without a decent amount of work in automatising it. Even if the job finishes without any exception the data will be there, without a simple way to be accessed. Since the physical cluster may be huge, the chances of getting allocated the same resources are low. The user could use some constraints to ask for a specific node there may be other tasks with a higher priority that will be allocated before and accessing this information would be delayed for an unknown amount of time, so a way to solve this problem in order to obtain a higher performance by taking advantage of the local storage is needed.
In this project, it is explored how to automatise this process using two different approaches that are covered during section 5.1: defining multi-datacentre clusters and using a snapshotting mechanism.

Chapter 4

State of the Art

When the first supercomputers appeared they were pure experimental systems that ran their code as a standalone high-performance machine, these were typically single-CPU systems that rapidly evolved into multi-core multi-processor ones. Scaling up by only having faster cores with smaller transistors was not enough anymore, therefore it finally turned into a modern era of supercomputers built in huge data centres to coordinately run tasks across several nodes, that each of them is many orders of magnitude over the first ones.
This change shifted the problem of scaling from the CPU architecture to the networking and shared storage components and the way it is decided which and where the work is going to be executed. When the job is done in a private cluster, this coordination can be performed with the use of queuing systems, while with the increase of public clouds there were many options to coordinate this computational power.
This idea was introduced by R. Buyya et. al[26] when explaining the Nimrod resource management and scheduling system to create a global computational grid which components are spread all around the globe. This had in mind the high cost of clusters of high-end physical resources compared with the inexpensive use of network connections to orchestrate a global grid.
Later C. Bunch and C. Krintz[25] identified an increasing problem that is the lack of automation while using HPC resources, forcing scientists to acquire a knowledge that is part of the technical skills that are more frequent in a system administrator than in these individuals. It is adding a barrier for those scientists who pretend to use HPC resources, while making even more difficult to reproduce the experiments that were documented by other researchers, at least in a trivial manner, to verify or take advantage of their contributions. Their contribution consists in an alternative

system to use one or more public cloud platforms as a back-end for HPC tasks and the creation of public repositories for scientific results, so they also had in mind the need of building hybrid clusters but clearly understanding the need for making easier the use of the HPC resources for the scientists.
In this case, the resources are private, and since the amount of available computational power is big enough[11] even for complex simulations, it will be a matter of focusing in a good management of the local resources in each cluster than adding up that power to reduce the cost of punctual computational needs. Therefore, it makes sense implementing a new integrated solution that makes possible to setup Cassandra-like distributed databases automatically to enable the execution of applications that could make use of this kind of storage.

Chapter 5

Design and implementation

In this chapter, it is described the design and implementation of the tools to provide key-value data services on HPC infrastructures. The chapter is divided into 2 sections: the first one with a discussion about different approaches to manage the data persistence and a section describing the tools to automatise the configuration and deployment of the cluster.

5.1 Data persistence solutions

Ensuring the data persistence in HPC environments without harming the performance of the application is a challenge with many approaches that need to be explored. There are two main possible alternatives taken into account during this project, that have been implemented, tested, and analysed: The multi-data centre Cassandra architecture and the snapshots approach. The first one could be more suitable for a non-HPC environment, while the latter is widely explained during this document since is the most direct approach and also has a better trade-off between the requirements and advantages obtained from their use.

5.1.1 Multi-data centre architecture

Cassandra-like databases enable many possibilities in terms of the distribution and replication of the data among the nodes that are part of the cluster. One of the interesting features is the possibility of splitting the cluster between two data centres. These two groups of nodes might be physically located in a different region or not, but in case of having the second one in a remote location the system automatically gains a live backup mechanism for free, which will have a similar performance than having a single data centre since the data is asynchronously propagated to the second data centre.

Moreover, there will be no latency penalisation[14] if it is ensured the usage of *LOCAL_ QUORUM* instead of *EACH_ QUORUM* since reaching a consensus between data centres would clearly impact in the user's experience. Users must be connected to one or the other data centre, in production environments, it is a common practice to identify the location of the user to route the traffic to the closest replica of the data. Every client that is connected to the Internet is taking advantage of this idea, that in web services consists of transparently getting the, for example, static content of the websites visited from a place that is, if not geographically closer than other copies of this information, for sure closer in terms of number of hops, to guarantee the fastest possible delivery of the requested content, and maintaining some policies to propagate the updates across all the remote locations, that at the end are just a copy of the original one. This service is typically provided by the called Content Delivery Networks (also known for its acronym CDN) that are companies that distribute media content across the globe, they need a big underlying infrastructure to be present in different places, so the key to their business is to make the content of their clients available everywhere and reached as fast as possible.

Nowadays the biggest part of web traffic is served through CDNs, including traffic from major sites like Facebook, Google, Netflix, and Amazon.[22] With Cassandra it is possible to create a similar architecture, that will be divided in two data centres instead of *N*, for the problem that is expected to solve there is no need to worry about a huge increase of latencies because the second logical data centre will be part of the same physical one, but the transparent backup feature enabled by this usage is what makes interesting this option.

There are many approaches to implement this scenario, the first one could be having a 24/7 data centre that is running out of the queuing system that will be eventually synchronised with another data centre that will be using the local storage of each node. This is quickly discarded since the resources in High-Performance Computing environments are scarce and really expensive, so the premise of building a setup that could be created, saved, restored, and destroyed, remains as the first target.

The final implementation of this approach[2] has a job that spawns two clusters, the first (and main, where the application will be executed) is running over the local SSD devices, it may be potentially bigger than the second logical data centre since it should process the petitions on a live execution while the second is passive and simply receives the changes in an asynchronous manner. The size of the clusters, particularly the second one, might be affected by the replication used since there will be at least a number of nodes equal to the number of copies of the data that is needed in the main data centre. While the main data centre uses the local, and faster, storage, the second is using shared storage for both data and configuration files, meaning that even if the job finishes without advice, it could be recovered in a reasonable amount of time, since it does not depend on the resources that were allocated by the queuing system. Figure 5.1 represents a Cassandra cluster with multiple data centres that both data centres have 4 nodes.

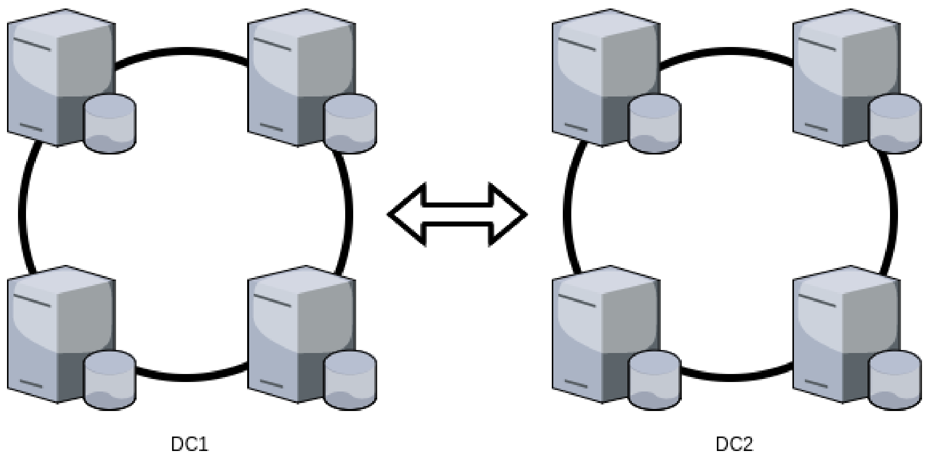

Figure 5.1: Multi data centre Cassandra cluster

Distributed databases were meant to be used as part of large production topologies, that might change over the time but stay available. The unexpected ephemeral nature of the data centres explained during this section creates a new problem to solve, that is a way to detect the synchronisation status between data centres. Cassandra already provides the ability to propagate the changes from the main data centre to the other asynchronously. The critical point where it is needed some intervention is when the main data centre nodes are ready to be switched off. Since the changes, that can be insertions or updates, are compressed by default, they are quickly transferred to be consistent, but it exists a delay between when the last change is made in the main data centre and when it is finally applied in the second data centre.

The most straightforward solution for this problem would be to wait a large amount of time to ensure that the changes have been propagated, but it has the inconvenient of the need of, somehow, finding out which amount of time is large enough. Also, the main drawback is that all the extra time spent in idle without a real need of keeping the clusters and the job alive is completely wasted, and in small executions, the overhead given by this behaviour may be completely unacceptable.

Since it looks like there is no built-in function to fully check this synchronisation status, the way this is checked was a combination of both, checking and waiting, until the status looks stable. The *nodetool status*[19] command provides a full view of the cluster status, it has been used before during the implementation to check if the nodes are up already while launching a cluster and looked to be a useful tool, in this case, listing 5.1 shows an output example for this command.

```
Data center: DC1
===============
Status=Up/Down
|/ State=Normal/Leaving/Joining/Moving
--  Address     Load      Tokens     Owns (eff.)            Host ID                               Rack
UN  10.0.1.1    7.34 GiB  256        100%                   ce5fd40f-ec73-4982-bca0-021e165bf773  Rack1
UN  10.0.1.2    6.53 GiB  256        100%                   b847e998-6b73-2523-62b7-bf8cfdf36b77  Rack1
UN  10.0.1.3    6.88 GiB  256        100%                   584263a4-c0d0-11e8-a7dc-acde4805c080  Rack1


Data center: DC2
===============
Status=Up/Down
|/ State=Normal/Leaving/Joining/Moving
--  Address     Load      Tokens     Owns (eff.)            Host ID                               Rack
UN  10.0.4.1    7.41 GiB  256        100%                   5b24fc47-e3ce-425d-a541-5e0ecef4f041  Rack1
UN  10.0.4.2    7.03 GiB  256        100%                   041fe563-ace4-4a76-bc89-592d6924fe13  Rack1
UN  10.0.4.3    6.66 GiB  256        100%                   a32f94b8-9367-4c69-8a9d-6688ac88e5bc  Rack1
```

Listing 5.1: nodetool status command output example for multi-data centre cluster

The *load* column indicates the amount of data that is stored in each node at the moment of the execution. These values are updated while the data is modified, created, compressed, replicated, rebalanced and deleted, but it has some lag between each update that makes it a worse option than expected. Moreover, the load values are rounded and the units used are those that fit with a value between 1 and 1000, including the first but not the later, so the rounding problem gets even worse when the volume of data increases. So there is a possibility that might be small changes that can be still on their way to be processed but are not taken into account using this view since two consecutive outputs will look exactly the same for different reasons.

Even if the values are exact and correct there could be an extreme case where two opposite operations (i.e. an insert and a delete) take place at the same time for different data that have a similar size, leading into a fake stable status, so only two comparisons won't be enough, or another strategy should be explored. At this point, the use of *nodetool netstats*[16] looks promising, since it prints network information about the host, such as the active, pending, and completed number of commands and responses, that tend to freeze once the application has finished in the main data centre. Listing 5.2 shows an output example for it.

```
Picked up JAVA_TOOL_OPTIONS: —Xss1280k
Mode: NORMAL
Not sending any streams.
Read Repair Statistics:
Attempted: 0
Mismatch (Blocking): 0
Mismatch (Background): 0
Pool Name                    Active   Pending      Completed   Dropped
Large messages                  n/a         0              0         0
Small messages                  n/a         0            578         0
Gossip messages                 n/a         0             72         0
```

Listing 5.2: nodetool netstats command output example

The final implementation of this synchronisation check consists on, once the application has completely finished its execution, getting the values of the *Small messages* row in every node, and comparing them with the previous ones for a given number of times, it is assumed that the status will stay like that in this node, and writing this confirmation into a file that will be what the main job is waiting for.

The command is executed every two seconds and the number of times it must be the same is five, and in any of the tested cases once those values stayed still for ten seconds changed later. Doing this check it in several steps makes possible to identify the stable status a bit faster than if it is done, for instance, only twice but separated by ten seconds. This action is performed in every node, and the active check in the main job will continue once all the files are present, meaning that the status in the second data centre is stable, and it is safe to shut down the cluster.

Looking for the advantages of this system, it makes possible to have a live backup of the data that is being generated during an execution and only needs some time to guarantee that all the data is already in the second data centre, that may be particularly attractive compared to other alternatives when handling huge amounts of information, where other alternatives would need to copy all the files at the end and needing more time to complete this task. Meanwhile, the hardware requirements of this setup are, in the worst case, the double of the needed with a single data centre cluster. It may be worth when the main data centre is big enough that a small part of the allocation is used for the second data centre but when the number of nodes is close to the 50% it will be a waste of resources that could be used for computing purposes while they are being used just as dummy storage infrastructure.

Since it only exists one copy of the data it should be combined with other techniques to allow batch executions, that are useful to make experiments while getting performance results and using the average instead of the value obtained in single executions of the application over the generated data. Moreover, the cluster that is running using as data directory the shared file system is unique, so a single instance of this data centre should be launched at the same time, so it is limited by design and many concurrent executions are not allowed. Even if many applications are launched from different jobs this will be an obvious performance bottleneck.

Since one of the expected usages of this system will be obtaining the best possible performance using the smallest amount of resources, it looks like exploring other options is still needed.

5.1.2 Snapshot storage approach

The process of creating a snapshot in a Cassandra-like database goes through different steps. First, a snapshot must be requested, for instance in Cassandra this can be performed by executing the *nodetool snapshot* command[18] over a host that is part of the cluster. Listing 5.3 shows many of the options it has.

```
nodetool —h <host> —p <JMX port> snapshot <keyspace> —t <name>
```

Listing 5.3: nodetool snapshot command example

The default behaviour makes that if any keyspace is specified the snapshot will be done for every keyspace present in the database. If this is executed from a different node the host IP address or DNS name must be provided, in a similar way if the cluster is not using the default JMX port it must be part of the command. Many other optional parameters such as the tag, that is used to give a name to this snapshot or the username and password are also included.

The *data_ dir/keyspace_ name/table_ name-UUID/snapshots/snap_ name* directory stores the snapshot when this command is successfully executed. Inside of each snapshot directory, there will be many *.db* files that contain the data that was in the cluster at the time it was taken. The snapshots are part of the functions that are already supported, they are scheduled in the launcher with the option *-s* or *–snapshot* and this is written into a file that will be read by the job during the execution. Once the application execution, if any, is finished, it checks for this file and depending on what it was chosen during the job launch it will be accordingly performed or skipped.
Then, depending on the queuing system that can be LSF or SLURM it is used a *blaunch* or a *srun* respectively to execute an auxiliary script in every node that is part of the reservation. This script contains the minimum set of operations needed to safely store the content of the data that is present in the cluster. Since it may happen that the nodes that are part of the database cluster are a subset of the total allocation, the first step is to check if the node is part of the list of nodes that are running the database, skipping the process if it is not the case. By default, a *nodetool snapshot* command flushes the table information that is still in the memtable, this is done to ensure that all the latest changes are part of the snapshot. This action can be also performed in an explicit manner using the *nodetool flush*[15] command at any time, or simply avoiding the use of the *–skip-flush* option of the *nodetool snapshot* command. It is given a *tag* to identify this snapshot that in this implementation is basically the starting time of the job in a year, month, day, hour, minutes format and the job identifier to make it easier to know exactly which execution produced this snapshot and when it was taken while preserving its uniqueness.

As commented before, each keyspace has a directory and inside of each of them there will be a snapshots directory which will have another directory named by the provided tag which will include all the data at the point the snapshot was taken. Since the data directory is typically located in the local SSD device of each node there is a remaining step that consists in moving all this information into a shared location to be recovered when it is needed. As seen during the 5.5 section, the first launch creates a default configuration file that may be modified. One of the variables is the *SNAP_PATH* that will be the place where the snapshots are stored and where the *recover* action will look for which ones that are available to be restored. The user that is executing must have write permissions in that path in order to store data successfully, and after all the information is copied into this remote directory the process will be finished.

Alongside with the copy of the data that was present at the snapshot moment, it is also stored the token rings of each node. This is performed using the *nodetool ring* command[17], which prints a list of big integers that should be properly inserted into the new configuration file of each node that is part of the new cluster that pretends to recover the status of the original one. Figure 5.2 illustrates this process.

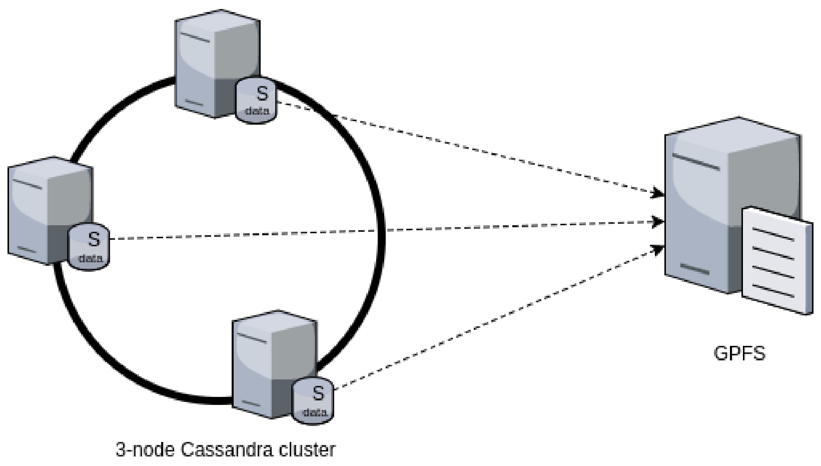

Figure 5.2: Cassandra snapshot process using GPFS

The end of the snapshot process in a node is notified by writing into a file named *snap-status-$CLUSTERNAME-$JOBID-$HOSTNAME-file.txt*, while the main job will be waiting and periodically checking until all the host files are present to continue, and finish, freeing the allocated resources.

Creating a snapshot enables persistent storage even in HPC environments. If the process and the corresponding recover is automated it makes possible to save the results of executions and reuse them once or as many times as wanted, and even more, create new snapshots using a previous one as the starting point, so batch executions are easy to do and rollbacks are just trivial. In any case, if we create a snapshot, even if the amount of data is huge there is no need to have the double of available space in the local storage since the snapshots are mostly hard links to the actual files, while there will be needed to have the whole size of the data as available space in the shared location to complete every copy successfully.

The total time it takes is written in the job output, to analyse how it behaves with different cluster sizes. The main constraint here is the network capacity since the data will be transferred from each node to a remote location and, if the size is big enough, it will reach a point where the network speed is the bottleneck, while for small clusters this process is nearly instantaneous. Sharing snapshots between users is as easy as using a shared snapshots path, or a combination of using other user's path and a correct permissions setting, or simply moving or copying in any way the directory is possible and won't have any problem since this kind of usage is expected and supported.

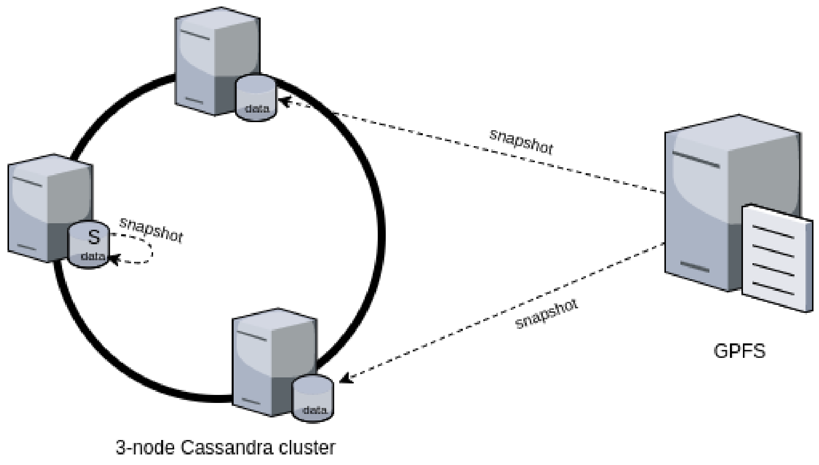

Figure 5.3: Cassandra snapshot recovery process

Figure 5.3 represents the process followed when in a 3-node Cassandra cluster only one node still had the snapshot information when it is tried to be restored. The two missing snapshots are copied from GPFS before the cluster is started. The cost and performance of this process is analysed in detail during section 6.3.

5.2 Automatic cluster allocation and configuration for HPC

This task has been divided into two main blocks:

- Management of the job and interaction with the user (*Job Launcher*): this is a script that acts as the entry point. It receives all the parameters from the users and performs all the tasks regarding the job management: resources allocation, checking of the status, stopping the cluster... And launches the task for configuring and deploying the software.
- Configuration and deployment of the software (*Job Script*): this task is performed by a set of scripts that take the information about the allocation and the user parameters and configures and deploys the database and, optionally, executes a client application. In addition, these scripts manage the data persistence.

Additionally, it is provided other script that allows the users to launch separate jobs with client applications using an already deployed database cluster (*Run Application*).
The communication between the Job Launcher and the Job Script is implemented using temporary files where the Job Launcher stores all the information that the Job Script will need.

In order to facilitate the portability between job schedulers, all the functions that interact with the job scheduler are encapsulated. There are provided implementations for both LSF and SLURM but it would be easy to adapt the scripts to interact with other job schedulers.

The initial step of the Job Script is to adapt the database configuration to the allocation of the job. In the case of Cassandra-like databases it is used a YAML[24] file (which syntax looks like the example in listing 5.4), the file by default provides an example that could run locally in that machine, with many mandatory parameters already set and many others that are optional that may remain commented to, perhaps, be used in the future. This way it will be easier to create some scripts that simplify the process of configuring each node since the common part will be already done and the one that depends on the resources allocated by the queuing system will be set for each node in their own configuration file. Since these services will be running in private networks, sometimes without direct Internet access, there is no need to be worried about security issues. Nevertheless, access control policies may be enforced to avoid unexpected accesses to the data, allowing only to the authorized users to perform operations. Also, there is no need to use other than the default ports for every service, but in any case, it will be set a unique cluster name for each of them to completely avoid the possibility that one node joins to a wrong cluster.

```yaml
# Cluster name is used to prevent nodes in a logical cluster from joining another.
cluster_name: 'Cassandra4HPC-20180822-202338'

# This defines the number of tokens randomly assigned to this node on the ring.
num_tokens: 256

# Authentication backend, used to identify users
authenticator: AllowAllAuthenticator

# Authorization backend, used to limit access/provide permissions.
authorizer: AllowAllAuthorizer

# Part of the A&A backend, used to maintain grants and memberships between roles.
role_manager: CassandraRoleManager

# The partitioner is responsible for distributing row groups across nodes
partitioner: org.apache.cassandra.dht.Murmur3Partitioner

# Directories where Cassandra should store data on disk
data_file_directories:
    - /tmp/cassandra-tmp/data

# Commit log. If not set, the default directory is $CASSANDRA_HOME/data/commitlog.
commitlog_directory: /tmp/cassandra-tmp/commitlog

seed_provider:
    # Addresses of hosts that are deemed contact points.
    - class_name: org.apache.cassandra.locator.SimpleSeedProvider
      parameters:
         # seeds is actually a comma-delimited list of node addresses.
         - seeds: "s04r1b16-ib0,s04r1b17-ib0,s04r1b18-ib0,s04r1b19-ib0"
```

Listing 5.4: YAML configuration file syntax example

5.2.1 Job launcher

The main problem addressed during the solution implementation part of this master thesis is the lack of usability in a complex storage system like the explained before. A user needs to easily deploy clusters and perform operations over it in order to use it daily to develop, test, and run production applications. It is not only important to give a nice performance, but also provide a usable tool that will be crucial if other people, that may be familiarised with these technologies or may be not, should work using it for very different tasks.

To ease all the configuration process it is programmed a *launcher* script that is used to interact with the system. This is a *bash* script that enables different options for each execution, customising the experience depending on the needs of the application and the user. Many of these parameters are directly passed to the queuing system while others are specific and explained in detail in the help section, that is automatically displayed when requested by the *help* option or after executing an incomplete or wrong command. The chosen programming language is *bash* but it could be *python* or any other scripting language. In any case, bash can be found in any machine by default and there is no need to worry about dependencies. The same script is compatible with both *SLURM* and *LSF*, as those were the job scheduler deployed in the two systems used to develop this project (Marenostrum 3 and Marenostrum 4). Typically, the Environment Modules package [13] is used in HPC environments to allow the user to dynamically configure the execution environment. Each user can define their own private modules but, usually, the system offer some public modules with tools and library that can be useful for all users. It has been developed a module with all these tools to make it reachable by any user in the system. However, the utilization of these scripts is very simple as they only need to have the *bin* directory included into the *PATH* environment variable or simply use the full path to the launcher file.

The module is designed to support different Cassandra-like databases, like *ScyllaDB*, or even own modified versions of this database, but in any case, a vanilla Cassandra 3.10 is provided by default.

In order to facilitate the utilization of Hecuba (one of the requirements of the general project), this script sets the *CONTACT_NAMES* variable to identify the Cassandra cluster that should be used. A side effect is that the user can launch additional Hecuba applications from the same terminal without setting the variable if the application script is launched with a leading dot. Notice that the setting of the *CONTACT_NAMES* variable can be overwritten by the user if using a different cluster is needed.

The Job Launcher can be used to configure and deploy a Cassandra cluster and, additionally to run a client application for this cluster. The only mandatory parameter for this option is the number of nodes to allocate. In order to support the different scenarios described in section 3 it is allowed to specify the number of nodes to use for Cassandra, the number of nodes for the application and if the user needs a sharing environment or a disjoint environment. Optionally, the user can manage the data persistence of the data of the cluster. The script has two different options related to this management: restoring the data from a former cluster execution (*recover*) and storing the data from this execution before the cluster is stopped.

When the *recover* action is used it will use the snapshots path defined in the configuration file to look for available snapshots to restore. A list of the ones that are there will be displayed and it will wait until an input is received. Then, it is checked that the given snapshot is part of the ones that are available to be recovered and then a new job will be launched. The main difference with a regular RUN job is that before starting the database cluster it will look for the data in the nodes, if the data is still available in the local storage since the original execution it will move it to the new data directory, otherwise it will copy it from the snapshots path and when it is done it will start the database. By default an allocation does not guarantee that the nodes involved in the new job will be the ones that were part of the job that created the snapshot, even if it is technically possible to ask to the queuing system to only allocate the job in those nodes it may take a long amount of time to fulfil this requirement, so that is why these scripts do a best-effort avoiding to copy unnecessary data through a shared network. In small clusters (in terms of both number of nodes and amount of data stored in the snapshot) the copy process is quick and lasts for only a couple of seconds and if a single node has to copy the data the amount of time needed is similar to when all the nodes need to do so. When the amount of data is higher or the number of nodes is big since bigger movements will be closer or even reach the point when it saturates the network, avoiding unnecessary data transfers looked like an obvious optimisation. In any case, if a node has the information available uses what is in the local storage and otherwise each one of the nodes that do not have it locally gets the snapshot of a node that is not part of the new allocation that was stored in the shared directory, places it in the new data path and appends the token ring information into its configuration file. After this is done the steps that are part of the RUN process take place right after it and the node will be ready to start the database.

This script can also be used to perform several management operations on the job as checking the status of the job or killing the job. Two different options to kill the job are provided: one to indicate that the job should end right after the application launched finishes and the other one to kill the job immediately without storing the data.

5.2.2 Job Script

There are many scripts related to this part, the main one is called *job.sh* and contains all the flow that will be followed once it is sent to the queuing system, allocated, and executed. The launcher prepares the configuration that this job will have, the parameters are given to the queuing system that will be the one that, after assigning a priority and assigning resources, will make possible the execution in a given number of nodes. One of them will follow the *job.sh* script, line by line, until it is aborted by the user, reaches the time limit or the end of the code. This module uses many auxiliary files to store important values that are accessed by different nodes during the execution. By default, during the first execution, it is created a base directory ($HOME/.c4s) that will contain all these files and also every configuration file needed. In each execution, old files that are not needed anymore are removed, if they exist. Also, during the first run, a default configuration file is written in $HOME/.c4s/conf/cassandra4hpc.cfg with different paths that could be edited. Listing 5.5 shows an example of a valid configuration file for Marenostrum 4.

```
LOG_PATH="/home/bsc31/bsc31906/.c4s/logs"
DATA_PATH="/scratch/tmp"
CASS_HOME="/home/bsc31/bsc31906/cassandra"
SNAP_PATH="/gpfs/projects/bsc31/bsc31906/snapshots"
```

Listing 5.5: Configuration file example

The value of LOG_PATH is a default value, each execution may have a different log path that can be set with a parameter. The DATA_PATH shows where the Cassandra data will be stored during the execution. CASS_HOME points to the Cassandra application root, while SNAP_PATH tells the module where to move the data of each node after a successful cluster snapshot.

In the base directory, there are located a bunch of files that makes possible different tasks, all of them created automatically by the scripts executed in this module. The names of all these files have the infix "*" where it represents the job name related to this cluster.

- Files named *hostlist-*.txt* contains the host names, one per line, of the nodes that are part of the allocated job.
- Files named *casslist-*.txt* contains the host names, one per line, of those nodes that are part of the Cassandra cluster, it may have the same content as the previous one or a subset depending on the characteristics given in the launcher options (i.e. if the cluster and the application are sharing nodes it will have the same content)
- Files named *applist-*.txt* contain the host names, one per line, that are going to execute the application as indicated to the launcher.
- Files named *stop.*.txt* contain a zero or one. The default behaviour is zero, that means that the job (and the cluster) will keep alive after the execution of the application, if the flag -f or --finish is used, or during the execution the command STOP is used over this cluster, this will turn to one, finishing the job after the execution ends.
- Files named *app-*.txt* include the application that will be executed, with arguments. A valid content for this file would be: *run.py brain.tif*, where *run.py* is a python executable and *brain.tif* a parameter. This could also be an application that uses an MPI implementation (such as OpenMPI or Intel MPI) to perform a parallel execution across the cores available in each node.
- Also, when PyCOMPSs is used for the application execution, some special files are also created:
  - Files named *pycompss-*.sh* are scripts that load the COMPSs module if it is needed, set the environment variables used by Hecuba and give to PyCOMPSs launcher the configuration indicated, such as the master and workers distribution.
  - Files named *pycompss-flags-*.txt* contain the flags indicated to the launcher for the PyCOMPSs execution, that will be used in the script commented before.

– Files named *pycompss-storage-*.txt* contain the storage nodes in a way that PyCOMPSs can understand, an important difference with previous node files is that here it must include the interface if the default is not where the cluster is bound.

Notice that all these files are deleted periodically to avoid the storage of unnecessary data.
Another auxiliary (and temporal) files are created and deleted right after their use, files named snap-status-$CLUSTERNAME-$JOBID-$HOSTNAME-file.txt indicate that the host $HOSTNAME that is part of the $CLUSTERNAME that is being executed in the $JOBID allocation, already finished performing a snapshot and it is waiting until all the hosts have finished, to proceed with the execution. These scripts check until all of them are present, then they are deleted and the execution continues.

In the conf sub-folder ($HOME/.c4s/conf) there will be a YAML[24] configuration file for each node. This is the result of replacing values in a template with the details of the cluster that is being launched. These names will be *cassandra-*.yaml* for Cassandra clusters, being *** the host name of that node. Moreover, in Scylla clusters, a Scylla template is used and the configuration file is *scylla-*.yaml* instead.

5.2.3 Run Application

This script performs the allocation for a new job that will execute a client application. This script accepts the options to configure the execution of the application, whose are a subset of the Job Launcher script. If there is only a running database cluster, it is assumed that the application will use it. If there are more than one, then the user is prompted to choose which one should be used. Notice that this feature can be useful if the user needs to test the execution of several simultaneous clients. However, this approach has an important drawback: the user has the responsibility of guaranteeing that the database cluster allocation is up during all the execution of the application job. This means that both jobs should be running simultaneously (which is decided by the job scheduler) and the database should be ready before the application starts. For this reason, in order to facilitate the user workflow, using this script is not the recommended working methodology but it may be useful while testing new applications.

5.3 Usage and usability

In this section, it is described in detail the parameters of the Job Launcher that can be used to configure the deployment of the software. It also contains some usability examples.

The Job launcher has four different *actions*: These are RUN, RECOVER, STATUS, STOP and KILL. They are typically used in capital letters but both all upper-case and all lower-case are valid in the current implementation. There are many *options* to be explained for each action, some of them are directly related to the queuing systems and some of them are new and make possible to put together database and application parameters.

The **RUN** action starts a new database cluster and perhaps executes an application, depending on many optional parameters:

- The *nT* option is the total number of nodes to be reserved. If it is not specified it will be set to the DEFAULT_NUM_NODES value, that can be configured but by default, it is four, so not using any number of nodes option is equivalent to execute with *-nT=4*.
- The *nC* option is the number of Cassandra nodes, a subset of nT. If it is not specified a Cassandra will be launched in each of nT nodes.
- The *nA* option is the number of application nodes, a subset of nT. If --appl is not used it will be ignored. If it is not specified the application will run in each of nT nodes.
- To execute an application after the Cassandra cluster is up it is used -a or --appl to set the path to the executable and its arguments, if any.
- If the -d or --disjoint flag is used, the application nodes will be different than the database ones. (In this situation nT>=nC+nA) In case that the number of total nodes is *strictly bigger than* the sum of the number of database and application nodes, nT nodes will be reserved but a warning will be shown, as some resources will be initially unused.
- If the application must be executed using PyCOMPSs the variable --pycompss should be used and contain its PyCOMPSs flags, if any.

- Using -s or --snapshot it will save a snapshot after the execution.

- Using -t or --time it will set the maximum time of the job to this value, with HH:MM:ss format for SLURM and HH:MM for LSF, by default it will be 4 hours.

- Using -f or --finish it will run the database cluster, the application (if set), take the snapshot (if set) and finish the execution automatically. The default behaviour is to keep it alive until it is stopped, killed or it reaches the time limit.

- Using -j or --jobname=<JOBNAME> that will be the job name sent to the queue, this way the job is easily identifiable and performing actions like STOP or KILL is easier than using directly the job identifier provided by the queuing system.

- Using -l or --logs=<directory path> that will be the destination of the log files, a default location is configurable and set during the first execution.

- Using -q or --qos=<queue name> it will run in the specified queue. This can be used to execute in a debug queue that might have some restrictions, like running a single job and a lower maximum time, so if the requirements are impossible under those constraints it will be rejected directly by the queuing system. Also, there are clusters where specific users have access to special queues to use different hardware or send longer jobs, this option will be useful in any of these cases.

The **RECOVER** action restores a saved database cluster and perhaps executes an application, depending on many optional parameters, that are a subset of those commented before for the RUN option:

- Using -s or --snapshot it will save a new snapshot after the execution, it will be like a new *version* of the restored cluster and it will be useful if there are changes in the recovered data that have been recently changed during the last execution and they should be saved as well.

- All the other options but those related to the number of nodes (-nT, -nC and -nA) are supported in a similar way they are for the RUN action.

The **KILL** action is used to abort the execution as soon as possible. By default, a job initializes a database cluster and when it is ready it executes an application, if given. This action simply finishes the job at the point it is, without shutting down the database cluster politely nor taking snapshots, even if they were planned. This KILL action is included to make easier the task of aborting an specific job among many others, which can has an intuitive job name while doing this directly using the job identifier might be more risky to do than let a script handle that translation, while in practice it is completely equivalent to a regular job cancellation.

The **STOP** action is used to make a natural finish of the execution. By default, a job will be running until it is stopped, killed or the time limit is reached. If there is only a cluster running, and the STOP action is used, it will be set to finish right after the application, if any, finishes. If there are many clusters running at the same time it will show them with a cluster identifier that will be needed for the *-c* option order to stop the correct one. i.e. In case that the c4s14189 cluster should be stopped, *STOP* and *-c=c4s14189* are the action and option needed to proceed. The main difference between STOP and KILL is that if the *-s* option was used, the snapshot of the cluster will be performed and stored before shutting down the database cluster and completely finishing the job.

The **STATUS** action, without any option, if there is a single running cluster job, it shows its status. Otherwise, it shows a list of the clusters that are currently running, if any, and providing its *cluster id* using the *-c* option it will show the status of that cluster. i.e. with STATUS -c=c4s14189 the information related to the status of the nodes in the cluster c4s14189 will be displayed, showing each individual node.

Examples of usage

In the Human Brain Project task that was previously mentioned the main target was to compare the performance of an application using a distributed database like Cassandra instead of the file-based approach that was already implemented and also using MPI or PyCOMPSs to make parallel calculations.
The allocations to test the different configurations should be done manually, understanding the options for each queuing system. The user should decide the number of nodes to reserve, and depending on the distribution of the workload, that could be a shared or disjoint scenario, which specific node use to each task. In the case of

the distributed database, every step should be done for each execution, configuring each node before launching the Cassandra process until the cluster is ready and only then, executing the application. With a common interface to choose how the job should be executed and an automated configuration system, all this process is effectively reduced to a single command.
Following there are some examples of usage.

```
c4s RUN --appl="cellseg_mpi.sh brain.tif"
```

The most common execution of the cell segmentation MPI application could be done running this command. In this case, *cellseg_ mpi.sh* is a script that executes an *mpirun* command to execute the python script that uses regular files as storage. If any allocation size or time limit related option is provided, by default it will request for a 4-node job, for a maximum time of 4 hours.

```
c4s RUN --appl="cellseg_pycompss.py brain.tif" --pycompss="-d"
```

This command executes the PyCOMPSs version of the cell segmentation application, distributing the workload across the nodes and Cassandra as storage. This will ask to the queuing system to reserve 4 nodes, that is the default, and when the allocation is done it will configure a 4-node Cassandra cluster and once it is ready it will use the application provided to PyCOMPSs to execute, with the debug option enabled, in all the four allocated nodes. This is a shared execution that will stay alive when the execution ends as it is the default behaviour, and that is also why only four nodes are requested, as the application and the database will be using the same hardware resources, but it can be easily changed to isolate storage and calculation.

```
c4s RUN --snapshot -nC=2 -nA=2 --disjoint --finish
--appl="cellseg_pycompss.py brain.tif" --pycompss="-d"
```

This execution is using the *-s* option that will take a snapshot of the status of the database right after finishing the application execution, *-nC=2* and *-nA=2* defines that 2 nodes will be used for Cassandra and 2 to run the application. At this point, the command would reserve a 2-node job for a shared execution, but setting the *–disjoint* option it makes that the total nodes reserved will be four as the application will be isolated from the database nodes, setting up a 2-node Cassandra cluster and running the application in the remaining two nodes. Moreover, the *–finish* option tells to finish the job as soon as the application has finished and the snapshot is moved to the remote location, instead of keeping alive until the time limit arrives.

```
c4s RUN -nT=4 -t=02:00:00 --qos=debug --appl="matmul_pycompss.py
--use_storage --num_blocks=4 --elems_per_block=128" --pycompss="-d -t"
```

This makes possible to execute any kind of PyCOMPSs application using Cassandra as storage completely forgetting about the configuration of the database. This is the case of a matrix multiplication application that uses Cassandra as storage. It is used a shared 4-node allocation to run both the database and application, it is set a maximum of two hours and the use of the *debug* queue. Many application specific options are provided, such as the number of blocks and elements per block and finally the PyCOMPSs options to be used, that in this case are using the debug mode for more verbose logs, generating a trace of the execution.

The database is empty when it is started, but the alternative is to restore the information from a snapshot. It may make sense to store a snapshot with some information and then execute the same or different applications over it to see how it behaves, get performance measurements or simply resume the calculations that were performed before.

```
c4s RECOVER -s --appl="cellseg_pycompss.py" --pycompss="-d"
```

Returning to the cell segmentation use case, it is possible to recover a snapshot that could store a big image, then apply the cell segmentation algorithm and save the result in a new snapshot. This command will ask to choose a snapshot from a list and the job will be submitted to the queuing system, as seen in listing 5.6.

```
The following snapshots are available to be restored:
20180927D1023h24s-2545335
20181005D1629h01s-2586511
20181005D2301h39s-2586498
20181006D1250h40s-2608510
20181006D1342h29s-2608514
20181008D1130h51s-2635768
20181009D1929h46s-2706628
20181009D2108h49s-2706635
20181010D1525h51s-2715776
20181010D1732h09s-2717648
20181012D1404h34s-2730393
20181012D1746h21s-2730800
Introduce a snapshot to restore:
```

```
> 20181005D2301h39s-2586498
Submitted batch job 2736142
Launching 4 nodes to recover snapshot 20181005D2301h39s-2586498
Launch still in progress. You can check it later with:
    c4s STATUS
```

Listing 5.6: RECOVER action example

Chapter 6

Experiments

The nodes used for all the documented experiments are part of the Marenostrum 4[11] supercomputer which has a general-purpose block with a total of 3456 nodes. Each node has two Intel Xeon Platinum chips, each with 24 processors, amounting to a total of 165,888 processors and a main memory of 390 Terabytes. Its peak performance is 11.15 Petaflops. A big part of these nodes, exactly 3240 of them, have 96GB of main memory in DDR4-2667 DIMMS, that are the ones used during the experiments. About the network connections, each is connected through a 100Gb Intel Omni-Path Full-Fat Tree, and also two 10Gigabit Ethernet interfaces, as seen in figure 6.1.

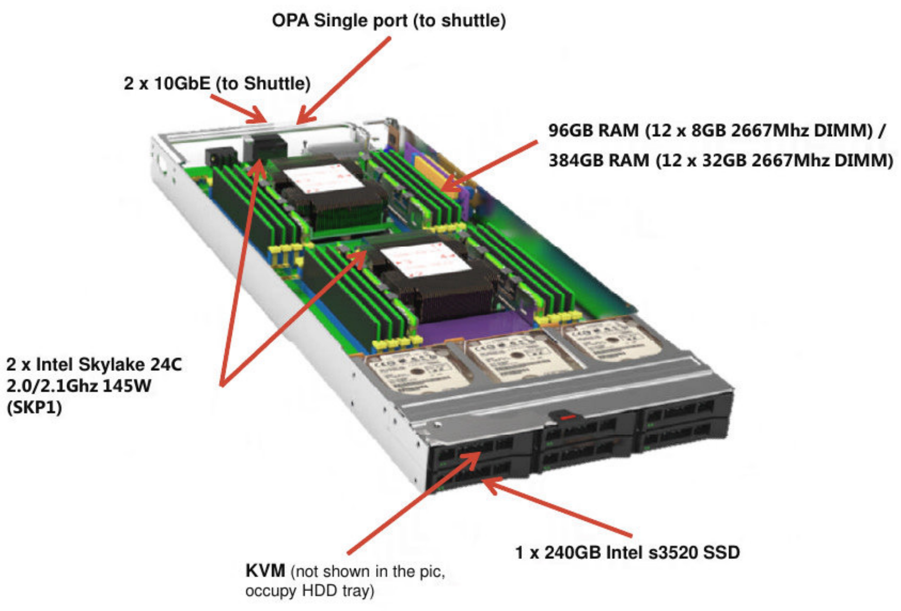
OPA Single port (to shuttle)
2 x 10GbE (to Shuttle)
96GB RAM (12 x 8GB 2667Mhz DIMM) /
384GB RAM (12 x 32GB 2667Mhz DIMM)
2 x Intel Skylake 24C
2.0/2.1Ghz 145W
(SKP1)
1 x 240GB Intel s3520 SSD
KVM (not shown in the pic,
occupy HDD tray)

Figure 6.1: Marenostrum 4 compute node[11]

The methodology followed to extract and analyse the information consists on instrumenting the code to print timestamps at different steps, adding more if needed for a fine-grain check of the actions involved. This way any slowdown that could be easily solved can be detected, fixed and then proceed to make several executions for different dataset sizes, cluster sizes or both of them. The use of additional scripts to extract metrics and make conclusions about them is done to reduce the code changes to timestamp prints to minimize the interferences of what should be a regular execution. Each one is done up to ten times, to detect possible outliers and get representative values.

6.1 Determining scripts overhead

Scripts are frequently used to automatise repetitive processes to avoid human intervention when it is not really needed. In this case, an important part of the time spent in a manual deployment of a cluster and the execution of applications over it has been effectively reduced into seconds of CPU time, but analysing the time spent in the different parts could make possible to find something that can be improved and see if the time spent on these tasks has an actual impact on the overall performance of the system.

It is measured the time spent by the scripts on its configuration and the time the

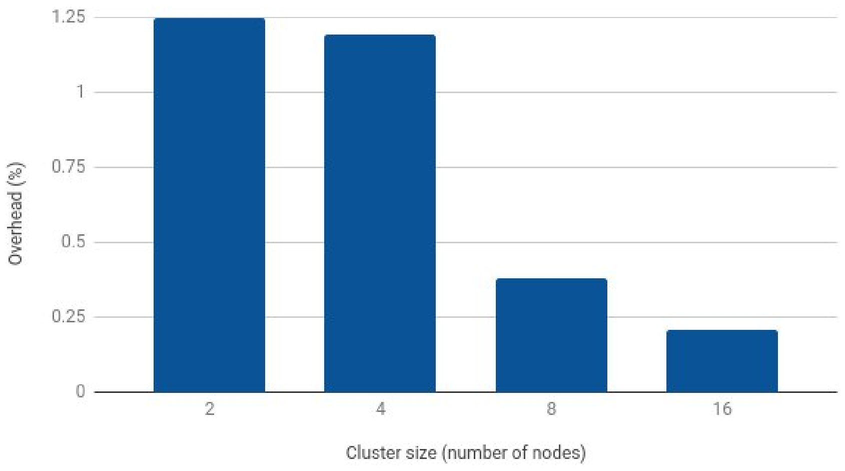

Figure 6.2: Overhead compared with database launching time

database cluster needs to be fully operative. Figure 6.2 shows the percentage the execution of the script represents compared with the time needed for the database

deployment. It is seen that this percentage is meaningless, being always under 1.25%.
It is noticeable that as the number of nodes in the database cluster grows, the scripts overhead diminishes. The reason for this is the following. This implementation determines that a database cluster is fully operative that can proceed with the execution of applications when the first node already recognizes all the nodes that should be part of it. As expected, the time until it is ready grows quickly as the number of members of the cluster increases, since the gossip system needs time to discover and include the new nodes into the cluster. Consequently, the proportion of overhead decreases as the amount of time it takes is between 0.22 seconds, in average for a 2-node cluster and less than 0.35 seconds to set everything that it is needed for a 16-node cluster.
Improving the gossip protocol used by Cassandra is out of the scope of this project but it could help to quickly deploy bigger database clusters.

6.2 Snapshot performance

Once the snapshot approach is chosen to be implemented because it fits the requirements and makes possible a flexible use of the resulting data, enabling the portability between users and clusters, it is important to analyse where the time is spent during this process, once again to see if there is a part that could be improved or detect problems that might be happening.
It is performed the snapshot of the data contained in clusters of different sizes, where the time is printed to be processed afterwards. These *checkpoints* are four and they are written, for each Cassandra node:

- Before executing the *nodetool snapshot* command.
- Right after the previous command finishes, that is before executing the *nodetool ring* command to obtain the *token ring* of this node.
- When the token ring is already written and before the copy of the files to the snapshot location.
- Once the files have been copied.

With these timestamps, it can be found the time spent on the three steps of a snapshot process: the execution of the *nodetool snapshot* and *nodetool ring* commands, and finally the copy of the files. This process is performed in parallel and the timestamps are printed by every node to see if the behaviour of each one inside a cluster is similar among all the tested configurations.

It is observed that the file copy takes around 80% of the time when the number of

Table 6.1: Mean time spent in the snapshot process

Nodes	**Size**	**nodetool snapshot**	**nodetool ring**	**File transfer**	**Time**
2	10GB	11.55%	12.81%	75.64%	10.50s.
4	10GB	23.60%	32.81%	43.59%	7.11s.
8	10GB	22.61%	33.60%	43.79%	6.05s.
16	10GB	26.78%	42.17%	31.04%	5.92s.
2	20GB	6.84%	7.44%	85.72%	18.95s.
4	20GB	11.67%	14.95%	73.38%	11.28s.
8	20GB	17.54%	26.11%	56.36%	7.73s.
16	20GB	23.14%	37.83%	39.02%	6.76s.
2	40GB	4.20%	4.30%	91.50%	31.24s.
4	40GB	7.20%	9.25%	83.55%	17.80s.
8	40GB	12.89%	18.36%	68.75%	10.91s.
16	40GB	17.92%	36.44%	45.64%	8.96s.

nodes is as small as two, and its proportional time usage decreases as more nodes are used for any of the data sizes that are tested. This was expected since the copy is split into more nodes that can start the data transfer in parallel.
Meanwhile, the results obtained surprisingly show that retrieving the token ring for each node using the *nodetool ring* command gets more and more costly while the cluster size increases in the number of nodes.

Since all the nodes are requested to start at the same time, the slowest node finishing the whole process sets the mark for the overall snapshot. Every value for both the average time obtained for each configuration and the mean time spent in each step has a standard deviation under 1, meaning that the expected value is a good representation and it can be confirmed that it is possible to fully backup the status and content of a whole cluster with up to 40GB of unique data in less than a minute. In fact, it is always used replication 2, meaning that the actual data volume in the biggest size cluster is around 80GB. With bigger amounts of data the limitation probably would be given by the data transfer rate that can provide the remote snapshots location, since it looks like a potential bottleneck.

6.3 COPY vs. Recover snapshot

While exploring the different options to initialise a cluster with some data these are reduced into two ways: recovering a snapshot or populating the cluster with rows after initialising a new one.
Recovering a snapshot consists on moving the files stored typically in the GPFS to the local disks located on the nodes. By design, moving files directly to remote nodes will always be faster than repeating the whole process of distributing them across the nodes to finally get there in a similar way.
Using as reference the 40GB dataset (that following a replication two strategy is around 80GB of data) the *recover* process takes in average, as worst case among these experiments, 144.68 seconds in a 2-node cluster. The best case for this amount of data is with a 16-node cluster, that takes only 34.46 seconds to complete the file transfer.
An alternative would be using *COPY*, that is a cqlsh command[4] which makes possible to insert data into a cluster using a file as the source or get information from a cluster and export it into a file. A fair point to use the *COPY* command to insert rows into a new cluster instead of recovering a full cluster state is that storing rows in a, for example, *CSV* file, as it is a plain text file, it can not only be compressed, also it can be modified to attach new rows or delete part of them. On the other hand, this command has its limitations: the ingestion of the data consumes an important amount of time and hardware resources. This command has some options to be explored: by default, it uses an *ingestion rate* of 100,000 rows per second that for this environment sets a conservative rate of row insertions to avoid the congestion of the contact point node and a failure of the whole process.

```
> cqlsh -e "COPY cells.data FROM 'cells_10GB.csv' WITH INGESTRATE=200000;"
Using 16 child processes

Starting copy of cells.data with columns [x, y, val].
Processed: 270000000 rows; Rate:   43264 rows/s; Avg. rate:  198747 rows/s
270000000 rows imported from 1 files in 22 minutes and 38.512 seconds (0 skipped).
```

Listing 6.1: Successful COPY command with custom ingestion rate example

In a cluster where commodity hardware is used it may be too much, but in an HPC environment where the highest class hardware is used it is expected to be lower than what it could be actually supported. The default behaviour is to abort after a maximum number of 5 attempts to import a batch or in case of a server error, if the *INGESTRATE* option is used to specify a higher rate it can be observed that the current rate goes up until reaching it or even going a bit higher, but if some errors appear during these attempts this rate is reduced again or if they keep happening the full ingestion process crashes. It is found that in Marenostrum 4 Cassandra clusters it is possible to insert large sets of rows successfully using *COPY* with an ingestion rate of 200,000 rows per second, as seen in the example provided in Listing 6.1, that is twice the default rate. Since this process is done through a single node this value is independent of the cluster size, and it is also checked using three sets of CSV files with 10GB, 20GB, and 40GB of data each. In this case, these files contain rows representing the location of brain cells, that are basically big integers. In the *keyspace* that stores this data it is used a *replication factor* of two, this means that each data will have a copy in a different node of the cluster, and even with the Cassandra built-in compression enabled it results into having nearly the double of data present in the cluster and spread among its nodes. In Table 6.2 there is a comparison between the time *COPY* needs to ingest different sets of data, with both the default ingestion rate and the maximum rate supported without errors in Marenostrum 4. Comparing the results of ingesting these three sets it is visible that

Table 6.2: COPY time for different ingestion rates

Set size	**Time (default rate)**	**Time (200k row/s)**
10GB	45.0021 minutes	22.3851 minutes
20GB	90.0024 minutes	44.9895 minutes
40GB	180.0042 minutes	89.9837 minutes

the ingesting rate ends as an upper limit for the insertion flow, the needed time does not change a lot from execution to execution, and it is not possible to improve drastically these results without putting in risk the whole process by its congestion. Even splitting a set into pieces to execute many *COPY* commands to different nodes does not reduce the time spent that much to be compared to its main alternative, that is recovering a snapshot. There is a common part that consists on starting each Cassandra node, that takes place in parallel and must be done in both cases, the main difference is that in the case of recovering a snapshot the data must be already present in the node before launching it.

To be completely fair while calculating the speedup of the snapshot system instead of the traditional *COPY* it should be added the time spent creating the snapshot to the time spent being recovered, that represents the worst case scenario when a snapshot is recovered only once. Taking 89.9837 minutes, that is the best mark obtained by $COPY$ for the 40GB dataset as it is shown in table 6.2, and adding up the worst case mark for the snapshot process obtained from table 6.1 that is 31.24 seconds and the average time needed by the *recover* process that in this case is 144.685 seconds. This calculation sets a worst case speedup close to 31.

$$Speedup = \frac{T_{copy}}{T_{snapshot} + T_{recover}} = \frac{89.9837 * 60}{31.24 + 144.685} = 30.69$$

All this difference between the traditional system and what this work proposes has a direct impact on the potential efficiency improvement of users like researchers that can focus on their tasks and avoid unnecessary waiting times.

Chapter 7

Conclusions and Future Work

Following the requirements of a real use case, this work proposes the design and implementation of an integral solution to run applications in High-Performance Computing environments using Cassandra-like distributed databases as storage, effectively reducing the user intervention to providing an application and choosing among a set of cluster options.
The result is a set of scripts that are compatible with at least Slurm and LSF, two of the most important job schedulers, and make possible to deploy database clusters, execute parallel applications that are compatible with the use of Hecuba and PyCOMPSs while providing a data persistence mechanism. By demand, it is possible to share nodes between the database and the applications to take advantage of the data location and also isolate these two components for CPU intensive applications.
The experiments performed are proof that using this set of tools makes easier and faster to deploy applications with a distributed database as storage in an HPC environment. They increase the user's productivity, reducing the time spent on repetitive tasks, providing quick database deployments, and making simple the interaction with different queuing systems.
In fact, due to the goods results obtained and to make easier the integration of this kind of applications, the work done during this project is not only available in a Github repository, it is now part of the Hecuba[6] modules that are published in the public repository of many supercomputers, like Marenostrum 4[11].

The volume of data that was used during the experiments is above what was required for the executions related to the project. To evaluate if it is possible to integrate this kind of database storage for applications that need in order of hundreds of Gigabytes or Terabytes of data it would be interesting to see how is the behaviour of the system

under an even higher data load, in particular when manipulating snapshots for both save and recover clusters.
This work is focused on the use of Cassandra as the database since is part of the original requirements, but Scylla is mentioned in many sections since it is in progress a complete adaptation to this system. Switching between Cassandra and Scylla will be one more option since it already has a subset of the features that are available for Cassandra, such as launching a new cluster and executing applications over it. The current and future work consists on implementing the remaining features to offer this alternative since the potential performance improvements are promising.
Also, at this point, the size of the cluster is chosen when a fresh one is requested, so a snapshot will always be recovered setting up a cluster that has the same number of Cassandra nodes the original had. Scaling up to recover it into a bigger cluster could be done adding new nodes and letting up Cassandra do the rest, re-distributing the load into the empty ones as it usually does while scaling horizontally, but in the case scaling down may be trickier and a deeper evaluation of the possible options to overcome this is needed. Moreover, there are implemented a reduced set of options from the total that are supported by the queuing systems, adding more options gradually is expected in the near future.

Bibliography

[1] Architecture of a Relational Database Management System. http://abiasforaction.net/architecture-of-a-relational-database-management-system/. Accessed: 2018-07-20.

[2] Cassandra with replication data centre over GPFS. https://github.com/eloygil/CassandraMultiDC. Accessed: 2018-07-12.

[3] COMPS Superscalar | BSC-CNS. https://www.bsc.es/research-and-development/software-and-apps/software-list/comp-superscalar. Accessed: 2018-08-29.

[4] COPY | CQL for Cassandra 3.0. https://docs.datastax.com/en/cql/3.3/cql/cql_reference/cqlshCopy.html. Accessed: 2018-10-01.

[5] cqlsh | CQL for Cassandra 3.0. https://docs.datastax.com/en/cql/3.1/cql/cql_reference/cqlsh.html. Accessed: 2018-06-29.

[6] Hecuba - BSC Data Driven group. http://datadriven.bsc.es/hecuba/. Accessed: 2018-07-29.

[7] Introduction to Apache Cassandra's Architecture. https://dzone.com/articles/introduction-apache-cassandras. Accessed: 2018-07-04.

[8] Job scheduler - Semantic Scholar. https://www.semanticscholar.org/topic/Job-scheduler/12096. Accessed: 2018-07-20.

[9] Key-Value Databases Explained | Key-Value Store. http://basho.com/resources/key-value-databases/. Accessed: 2018-07-04.

[10] MareNostrum 3. https://www.bsc.es/marenostrum/marenostrum/mn3. Accessed: 2018-07-20.

[11] MareNostrum 4 - Technical information. `https://www.bsc.es/marenostrum/marenostrum/technical-information`. Accessed: 2018-07-20.

[12] Message Passing Interface (MPI). `https://computing.llnl.gov/tutorials/mpi/`. Accessed: 2018-08-29.

[13] Modules - Software Environment Management. `http://modules.sourceforge.net/`. Accessed: 2018-10-14.

[14] Multi-datacenter Replication in Cassandra | DataStax. `https://www.datastax.com/dev/blog/multi-datacenter-replication`. Accessed: 2018-09-21.

[15] nodetool flush | Apache Cassandra 3.0. `https://docs.datastax.com/en/cassandra/3.0/cassandra/tools/toolsFlush.html`. Accessed: 2018-09-17.

[16] nodetool netstats | Apache Cassandra 3.0. `https://docs.datastax.com/en/cassandra/3.0/cassandra/tools/toolsNetstats.html`. Accessed: 2018-09-21.

[17] nodetool ring | DSE 6.0 Admin Guide. `https://docs.datastax.com/en/dse/6.0/dse-admin/datastax_enterprise/tools/nodetool/toolsRing.html`. Accessed: 2018-08-25.

[18] nodetool snapshot | Apache Cassandra 3.0. `https://docs.datastax.com/en/cassandra/3.0/cassandra/tools/toolsSnapShot.html`. Accessed: 2018-09-17.

[19] nodetool status | Apache Cassandra 3.0. `https://docs.datastax.com/en/cassandra/3.0/cassandra/tools/toolsStatus.html`. Accessed: 2018-09-21.

[20] Scaling Up versus Scaling Out: Mythbusting Database Deployment Options for Big Data. `https://www.scylladb.com/wp-content/uploads/wp-scaling-up-vs-scaling-out.pdf`. Accessed: 2018-08-10.

[21] SGA1 Phase | Human Brain Project. `https://www.humanbrainproject.eu/en/about/governance/deliverables/sga1-phase/`. Accessed: 2018-08-20.

[22] What is a CDN? How does a CDN work? `https://www.cloudflare.com/learning/cdn/what-is-a-cdn/`. Accessed: 2018-09-22.

[23] What is Cassandra? | Apache Cassandra. `http://cassandra.apache.org/`. Accessed: 2018-07-20.

[24] YAML Syntax | Grav Documentation. `https://learn.getgrav.org/advanced/yaml`. Accessed: 2018-09-02.

[25] Chris Bunch and Chandra Krintz. Enabling automated hpc / database deployment via the appscale hybrid cloud platform. In *Proceedings of the First Annual Workshop on High Performance Computing Meets Databases*, HPCDB '11, pages 13–16, New York, NY, USA, 2011. ACM.

[26] Rajkumar Buyya, David Abramson, and Jonathan Giddy. Nimrod/g: An architecture of a resource management and scheduling system in a global computational grid. *CoRR*, cs.DC/0009021, 2000.

[27] Parag Shukla and Kishor Atkotiya. NoSQL Database: Cassandra is a Better Option to Handle Big Data. 5:24–26, 01 2016.

[28] Enric Tejedor, Yolanda Becerra, Guillem Alomar, Anna Queralt, Rosa M Badia, Jordi Torres, Toni Cortes, and Jesús Labarta. Pycompss: Parallel computational workflows in python. *The International Journal of High Performance Computing Applications*, 31(1):66–82, 2017.

[29] David W Walker. The design of a standard message passing interface for distributed memory concurrent computers. *Parallel Computing*, 20(4):657 – 673, 1994. Message Passing Interfaces.