Master In Computational and Mathematical Engineering

Final Master Project (FMP)

A staging area for in-memory computing

Pol Santamaria Mateu

Advisor: Ivan Rodero Castro

15th July 2018

INDEX CARD OF THE FINAL MASTER PROJECT

Title of the FMP:	*A staging area for in-memory computing*
Name of the author:	*Pol Santamaria Mateu*
Name of the TUTOR:	*Ivan Rodero Castro*
Date of delivery (mm/aaaa):	07/2018
Degree:	*Inter-university Master's Degree in Computer Engineering and Mathematics*
Area of the Final Work:	*High Performance Computing*
Language of the work:	*English*
Keywords	*Staging, In-memory, Alluxio, BigData*

Summary of the Work (maximum 250 words): *With the purpose, context of application, methodology, results and conclusions of the work.*

An in-memory staging area provides fast access to different applications. This research is based on evaluating the benefits of a distributed in-memory staging area applied to the field of Big data.

With this purpose, a prototype is designed and proposed to verify the idea. Then, a working version comprised of the in-memory software Alluxio and the processing engine Apache Spark is deployed and evaluated.

In particular, the work demonstrates the increase in performance resulting from updating the data in the in-memory staging instead of allocating space for new objects. The evaluation is conducted by running an analytic with Spark over a continuously changing dataset stored in Alluxio.

The experiments reported a throughput increase of 10x when compared to storing information in a regular parallel filesystem, and an increase of 3x compared to the official deployment methodology. By updating the dataset, the Alluxio in-memory capacity stays constant at a low level compared to current deployments where its capacity decreases linearly, resulting in lower performance.

Contents

List of Figures 4
List of Tables 5

1 Introduction **1**
1.1 Context and motivations 1
1.2 Aims of the Work 2
1.3 Approach and methodology followed 3
1.4 Planning of the Work 4
1.5 Brief summary of products obtained 5
1.6 Thesis organization 6

2 Related work **8**
2.1 Staging area 8
2.2 In-memory computing 9
2.3 Big data architecture and distributed computing 10

3 Related software technologies **12**
3.1 Computation frameworks 13
3.2 In-memory computing 16
3.3 Storage 20
3.4 Auxiliary systems 23

4 Proposed approach: In-memory data staging **28**
4.1 Software stack discussion 28
4.2 Design the architecture 32
4.3 Software components interactions 33

5 Development **38**

5.1 Requirements 38

5.2 Deep code inspection 39

5.3 Deployment plan 42

5.4 Final prototype 43

6 Experimental evaluation **47**

6.1 Methodology 47

6.2 Tests design and implementation 48

6.3 Results 55

6.4 Experimentation summary 66

7 Conclusions **69**

8 Future research **71**

List of Figures

1.1 Gantt chart with scheduled deliveries highlighted in red. 5

3.1 Internal organization of a Kafka topic. 25
3.2 Kafka internals and producer-consumer mechanism. 25

4.1 Spark and Alluxio integration schema. 30
4.2 Software assigned to the proposed functional architecture. 32
4.3 Software and hardware correlation. 33
4.4 Alluxio in-memory blocks organization on a Tier 0 ramdisk. 34

6.1 Mapping services to hardware in worker nodes. 50
6.2 Initial query code. 51
6.3 Fragment of the query used. 52
6.4 Fragment of the Python block updates. 53
6.5 Time needed to update an RDD in GPFS. 56
6.6 Elapsed time since an element is updated until evaluation in GPFS. 56
6.7 Mean time to propagate an update. 57
6.8 Time for an updated RDD to be processed by the query. 58
6.9 Time to write a new RDD into Alluxio, backed by GPFS. 59
6.10 Time to write a new RDD into Alluxio, backed by NVMe storage. . 60
6.11 Time for the query to evaluate an update. 61
6.12 Mean time to propagate an update. 62
6.13 Time between queries. 62
6.14 Time between updates complete. 63
6.15 Node organization used to evaluate the proposal. 64
6.16 Evolution of externally updated data analyzed from Spark. 65

6.17 Time for the prototype to detect and process and update. 66
6.18 Time between query results. 66
6.19 Summary of the experimentation results. 67
6.20 Frequency of the query execution for the different configurations. . . 68

List of Tables

1.1 Concrete dates with scheduled deliveries highlighted in red. 4

3.1 Summary of the analyzed Big data ecosystem projects. 13

4.1 Addressing a file through the Alluxio FileSystem API. 35

6.1 Parallelization of the initial data structure. 51

6.2 Query output example simplified. 52

Chapter 1

Introduction

1.1 Context and motivations

With the information revolution, digital sources increased and the amount of information available growth exponentially. It led to traditional computing approaches being unable to manage or provide valuable insights on large volumes of data. At that point, a new term used to describe these huge quantities of information appeared, the *Big data* era started.

Being able to analyze information is imperative in scientific and business contexts. Gathering and studying data provides insights on inefficiencies or patrons that will support decision-making, discover new opportunities or identify subtle changes.

Since 2012, a wide range of projects tailored to Big data emerged. However, the influence of traditional methodologies such as the presence of a single point of failure or the inefficient use of hardware is still present. The focus of this work lies on the latter, in particular, in the inefficiency of sharing data by storing it on slow and limited bandwidth storage.

Due to the cost and capacity of volatile memory data has been stored in rotational disks (HDD) or solid state drives (SSD) more recently. Although SSDs brought substantial improvements in latency and bandwidth, they are still far from RAM specs. However, with recent advancements in technology where high volumes of RAM are possible and the adoption of the new NVMe technology, delivering near RAM performance at a cheaper cost while being non-volatile, make necessary

a shift on the Big data ecosystem paradigms.

A frequent use case consists of transforming a dataset and applying multiple analytics to extract knowledge. It is commonly done by reading the data from disk, elaborating the data, and storing it again. Others use methods to allow concurrent access, such as databases, or dividing the cluster into compute and storage nodes. All these approaches rely on storing data to disks and performing multiple accesses. This mechanism is time-consuming due to latency and bandwidth while having the same data loaded for different programs reduces the available memory.

The work presented in this document develops the hypothesis that introducing a shared memory region between applications working on the same data will improve their performance. This concept is known as a staging area and defined as intermediate storage where applications interchange data.

In-memory computation can be simplified as doing calculus by only accessing memory and avoiding storage. This concept is tightly tied to this project because it provides unified and efficient data access for applications. Consequently, software that brings in-memory computing to Big data should be used to evaluate the introduction of a staging area. Two relevant open source projects implementing in-memory computation are available, Apache Ignite and Alluxio.

In the context of streaming analytics, latency and processing times are crucial due to the nature of data. Our proposal aims to speed up access to fresh or frequently consumed data. Therefore, it is a promising area for evaluating and testing in-memory computing with a staging area.

In conclusion, this work aims to improve current big data approaches by proposing a solution which considers recent advancements in high-speed memory. In this context, the addition of a staging area to in-memory computation will be investigated and later evaluated in streaming analytics.

1.2 Aims of the Work

The following points summarize the scope of this work:

- Evaluate the introduction of a staging area in the Big data context.
- Design and deploy a big data system to evaluate the proposal through ex-

perimentation.

- Contribute to the actual Big data ecosystem.

1.3 Approach and methodology followed

At least three paths have been devised to demonstrate the benefits of using a portion of the memory as a staging area. The first would be using software already designed for this purpose. Since no solution satisfies our requirements, it becomes necessary to conceive and implement our approach. However, the production of a new tool is time-consuming and out of the scope of a master thesis duration. Besides, there are many tools readily available, stable, supported and adaptable for our purposes. Consequently, modifying an existing and widely used product is the third and best option.

Firstly, a survey of the big data ecosystem needs to be conducted to select the best fitting tools for the realization of an in-memory staging area. Projects that will allow implementing a realistic prototype will be chosen according to the following defined criterion:

- Community support.
- Licensing.
- Open source.
- Innovative.
- Widely used.
- Stable.
- Favors integration.

A proposal should be made consisting of a system able to process amounts of records in the order of thousands, as a standard situation nowadays. Also, the proposed system should be designed to scale horizontally. Since the specifying the correct configuration and deployment of a complete system can be time-consuming,

a smaller prototype providing data caching and computation will be deployed and tested.

Finally, verification of the performance should be carried through experimentation. For this purpose, the Rutgers University offered a cluster to conduct the tests. The first step will consist of evaluating the difference of accessing the data from different supports in terms of performance and usability. The storage systems analyzed will depend on the particularities of the cluster, but will include at least an in-memory system and a typical network filesystem. Secondly, since the available RAM is typically smaller than other storage devices, strategies of memory eviction and reusability will be explored.

To perform a complete evaluation, a short experimentation phase will be carried at the end of the project. The proposed prototype will be deployed and evaluated following synthetic tests which might be based on real use cases. The scope is to get a realistic idea of the potentials of this work by measuring the consumption of resources and confront the results with similar approaches.

1.4 Planning of the Work

The steps illustrated in 1.1 have been planned to achieve the goals with a realistic timeline. Despite 15/07/2018 is the official deadline, we have proposed 31/05/2018 to be the target time with substantial time allocated for unforeseen events that could delay the project until 31/06/2018.

Task	Duration (days)	Date start	Date end
Design masther thesis	23	23/11/2017	16/12/2017
Write memory's introduction	20	17/12/2017	06/01/2018
Research related work	10	07/01/2018	17/01/2018
Evaluate initial prototype	13	18/01/2018	31/01/2018
Elaborate prototype and memory	58	01/02/2018	31/03/2018
Final delivery	60	01/04/2018	31/05/2018

Table 1.1: Concrete dates with scheduled deliveries highlighted in red.

At the same time, a more visual approach to representing the tasks organization and the partial deliveries is shown in 1.1. The initial steps are well defined at this

time and can be split into smaller tasks.

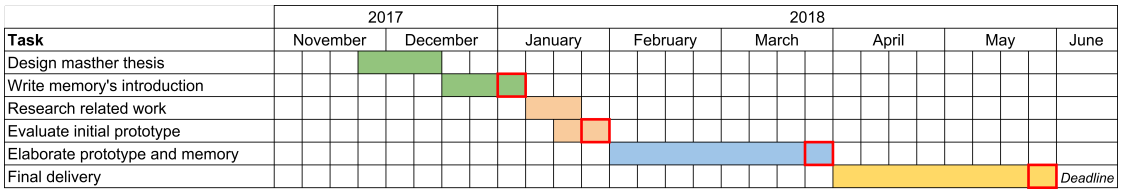

Figure 1.1: Gantt chart with scheduled deliveries highlighted in red.

1.4.1 Planning achieved

The deliverables were performed accordingly to the original plan until March 2018. Then, due external reasons unrelated to the project the advancements were slowed down from that point until the final delivery on 15th of July. The lesson learned with is that unforeseen external factors need to be accounted for when planning a project by allocating an extra period of time between deliverables.

On the other hand, the Big data ecosystem research took longer than expected due to the vast number of projects, their complexity, and coupling. This step, as well as the internal analysis of the selected software, was essential to succeed in proposing and building a realistic prototype.

Finally, evaluation and information extraction from a prototype running in a distributed cluster can be time-consuming. For instance, the first configuration of the prototype did not take advantage of data locality, and thereby tests were running slow until it was fixed.

In overall, the project has suffered a delay of a month an a half due to a variety of reasons. It represents a delay of the 20% with respect to the initial delivery date on 1st of June. It can be considered a typical delay in software development processes such as Agile, particularly when the previous knowledge is reduced.

1.5 Brief summary of products obtained

The work conducted demonstrates the potential benefits of an accessible in-memory staging area. A prototype has been proposed and evaluated featuring Alluxio for an in-memory staging area and tiered storage. The prototype was

responsible to support concurrent running programs continuously modifying the same dataset. This feature is not supported by Alluxio and has been done externally after analyzing the implementation and management of data in both Apache Spark and Alluxio.

Experimentation has been conducted to evaluate a variety of storage configurations for Alluxio and Spark. The results showed a far superior performance obtained with the prototype when compared to actual approaches. In particular, we obtained a constant capacity usage of the in-memory staging area, which is far less than current setups. Typically, datasets stored in Alluxio need to be deleted or evicted after a period of time, since updates are not supported.

On the other hand, the prototype avoided the use of locks, enabling applications to access data without synchronizations. This characteristic resulted in a speedup of 3x in the query execution time in confront of the traditional approach where many reads would have been affected by deletes or writes locking the content for a period of time.

When this approach is compared to accessing data stored in a GPFS, or setting up GPFS as a second tier, a performance speedup of 10x is obtained.

In conclusion, the presented work motivates future research in the field of in-memory data staging, particularly in the field of data streaming and continuous processing.

1.6 Thesis organization

The structure of this thesis is divided into 6 chapters.

The related work is discussed in Chapter 2. Works with the potential to contribute to the knowledge and development of this project are reviewed and discussed.

Then, in Chapter 3 a technological background is performed as a short survey analysis of the Big data ecosystem. This phase enables discussing the selection of the Big data components in Chapter 4 based on technological arguments.

Chapter 5 reviews the research done to propose, configure and deploy successfully a prototype to demonstrate the viability of this project. The taken decisions, the prototype structure, and particular modifications are explained and justified.

A reduced version of the proposed prototype is evaluated on Chapter 6. With this purpose the prototype is deployed in a cluster and evaluated following different metrics, discussing the results obtained. Different configurations are tested to justify the proposed in-memory staging area with enabled updates.

Finally, in Chapter 7 the conclusions extracted during the fulfillment of this project are presented. Results are shortly reviewed and linked with the knowledge gained.

To finish with, ideas on how this work can be continued and improved are presented in Chapter 8. Because this project demonstrates the viability of an idea, different research paths will need to be explored.

Chapter 2

Related work

The scope of this chapter is to review the literature on the relevant techniques applied in this project. Instead, the technological background and related software frameworks will be analyzed in Chapter 3.

Sections 2.1 and 2.2 shortly review the set of techniques to be applied. Then, recent hardware advancements are discussed on section 2.3 to clarify how the mentioned techniques can be applied to obtain software improvements.

2.1 Staging area

In most systems, raw data can come in a variety of formats, is heterogeneous, and may have missing or irrelevant information. It is essential to adapt, transform and clean the data in order to make it available to other processes requests. The place where this process occurs is named staging area, a term introduced in computer science by Business Intelligence specifications.

The scientific community employs this mechanism to improve performance when writing data to storage systems. Firstly, computing applications benefit from executing calculations dedicating less time to write and read data to persistent, low storage. Secondly, the data is gathered and then stored in continuous chunks. Performance for sequential writes is notably higher than random I/O, because of the reduction in seek time and hardware strategies. Moreover, if the data needs to be sent over the network, sending big packets of information is more efficient [15].

The results obtained when introducing an staging area vary according to different factors. For an instance, in some cases Solid State Drives (SSDs) are employed as staging areas of Hard Disk Drives (HDDs) while other set a staging area in the DRAM. In particular, a staging area has proven useful in those cases where data was being re-utilized such as [13].

Finally, raw data needs to be available in a designated location to be accessed on demand. The designated area receives the name of landing zone, which is closely coupled with the staging area. In the landing zone raw data is deposited without pre-processing or post-processing, whereas, in a staging area data can be processed and mutated.

2.2 In-memory computing

In-memory computing is a technique used to speedup computational processes by replacing slow storage accesses for fast DRAM accesses. Its main objective is to reduce the execution time of applications run in datacenters.

In this context, different types of applications with their particular data access patterns are benefited. Data-intensive applications will be rewarded with a reduced latency on data accesses by several orders of magnitude, resulting in faster execution times. Secondly, applications handling large datasets benefit from keeping in memory frequently accessed data, allowing them to read and write huge amounts of data quickly. Lastly, the reduction in latency is crucial in situations where taking quick decisions is needed as in real-time analysis, where also information changes constantly.

A common approach to apply this technique is to keep the data in high speed memory such as DRAM, and then backing up the changes to slow storage. This is done transparently to the user by using parallelization, shared memory technologies and performing operations asynchronously.

In the near future, a closer integration with modern network technologies could extend this approach. New network technologies allow nodes to write remotely into the memory of another node without interrupting the computation on the CPU. This feature enables applications to distribute large volumes of data among nodes without slowing down the execution. The analysis of the hardware particularities

that make in-memory computation feasible are described in section 2.3.

Ensuring that the data stored into volatile memory is not lost and avoiding coherency issues is one of the most challenging parts in in-memory computation. Therefore, it is impractical to develop an in-memory computation software for each use case. Already existing solutions which have demonstrated to have a good performance and behaviour are used instead, reviewed in 3.

On the other hand, including this mechanism in workflows has a significant impact. Processes spend less time accessing and retrieving data, therefore blocked without doing computation, leading to shorter processing times. Straightorward consequences are more users being able to use the system, shorter response times and a reduction of the cost associated of running the cluster.

However, not all types of applications will benefit from this technique. For an instance, simple applications, or computation intensive applications producing few data will incur the penalty of setting up the mechanism and leasing computational resources to the software managing the in-memory mechanism. Therefore, it is clear that this mechanism is particularly beneficial for Big data applications but is not suited for all HPC applications.

2.3 Big data architecture and distributed computing

Current computational frameworks divide the data and computation into logical blocks, and then apply the computation to each distributed piece. Then, results are presented to the user as a single logical entity. A widely adopted implementation of this idea is the MapReduce framework introduced by Google [5]. However, other traditional approaches are based on centralized architectures where some nodes act as masters and the rest as workers []. What they all have in common is that they try to take advantage of data locality by applying the computation where the data resides.

To achieve this distribution of tasks, different computation nodes are used transparently. They are made of processing power with volatile memory attached, interconnected by a network, and sometimes with storage devices attached.

When we analyze storage, two different setups are found. Clusters where each

node has a storage device, called local storage, or clusters with a set of nodes dedicated to storage. In the first example, storage is accessible only during computation, and data needs to be moved at the end of the execution. On the latter, storage is always accessible and physically shared among different users. Hybrid systems are a common setup nowadays.

The second aspect to consider is the network which may have different usages depending on the framework or application. For an instance, some applications send control messages and small pieces of information, while others rely on it to transmit huge amounts of data constantly. Nevertheless, it is important to highlight that new technologies such as IBM's Infiniband and Intel's OPA allow processes to send data from node to node without interrupting the CPU, thus updating other's computer memory remotely with a reduced latency [12].

As an example, commonly used SATA/SAS SSD drives deliver write and read bandwidths in the order of hundreds of MB/s, while NVMe SSD perform around a thousand MB/s in common 4K tests. However, DDR4 at 2.666Mhz delivers bandwidths of 40-60 GB/s in 4 channel mode. A more important factor, latency, is reduced from 40-70 microseconds in SATA/SAS SSDs or a constant 10-20 microseconds obtained with the new NVMe SSD to roughly 14 nanoseconds in DDR4 [4], [6]. As a side note that SSDs performance degrades as the I/O operations queue. However, SATA SSDs usually have a small queue of 32 operations while NVMe SSD may have up to 65.000 queues with a queue depth of 65.000 operations each [14]. As a consequence, it is more difficult to fill up the operations queue of NVMe SSDs and degrading their performance.

In conclusion, huge difference in bandwidth and latency make the in-memory computation to be an interesting field. While at the same time, fast memory and high speed networks provide a good support to include staging areas in current Big data workflows.

Chapter 3

Related software technologies

Since 2003, after the publication of the Google File System [8] paper, different processing and storage solutions have been proposed to achieve horizontal scalability, fault tolerance and availability. Since an increasing number of software solutions exists, their features need to be evaluated to decide how to structure a system. Since big data is about how to organize information for efficient processing, the first characteristic to be reviewed is the data accessing pattern, closely related to the nature of data. As an example, connected data from the internet or logs will be treated differently than scientific data.

As advanced in 2.3, relevant Big data frameworks are inspired by the MapReduce programming model provided by Google and implemented in Hadoop or the resilient distributed datasets of Spark. The following sections will review software available for HPC with the potential to be part of the proposal. Projects discussed in this chapter are summarized in Table 3.1. Analysis will be based focusing on the aspects that will enable developing a proposal that can be verified, and importantly, its impact evaluated.

Category	Software	Subsection
Computation framework	Apache Hadoop MapReduce	3.1.1
	Apache Spark	3.1.2
	Apache Storm	3.1.3
	Apache Flink	3.1.4
In-memory computing	Alluxio	3.2.1
	Apache Ignite	3.2.2
Storage	Apache Cassandra	3.3.1
	Apache Hadoop Distributed FileSystem	3.3.2
	Apache Hbase	3.3.3
	Ceph	3.3.4
Axilliary systems	Apache ZooKeeper	3.4.1
	Apache Kafka	3.4.2
	Apache Beam	3.4.3
	Apache Hive	3.4.4
	Presto	3.4.5

Table 3.1: Summary of the analyzed Big data ecosystem projects.

3.1 Computation frameworks

3.1.1 Apache Hadoop MapReduce

Hadoop was started from the ideas behind the Google File System [8] and the needs to scale data processing to more nodes. The main contribution of Hadoop consisted of the MapReduce programming model that allowed analyzing large datasets in parallel, and efficiently. On the other hand, Hadoop included its filesystem providing support for distributing datasets among nodes. The novel idea of sending the tasks where the data was stored, instead of moving the data, produced a programming model shift.

MapReduce is a strategy to organize the execution of a method over a dataset. The first step consists of executing the user-defined function over the data partitions, spread in a cluster, in parallel. Then, the partial results obtained are unified through the reduce task defined by the user. For instance, to count the number of lines in a file, one would write a map method computing the lines of a chunk of data, and then a reduce method which adds all the partials results.

This model has many advantages respect to previous techniques. First, the

execution takes place where the information is stored, avoiding moving or copying large data sets. Secondly, if a map or reduce task fails in one node, it can be rescheduled in any node with a copy of the data. Fault tolerance is significantly improved through this policy, while performance increases by not restarting the whole execution and minimizing data movements.

Companies and research centers still use Hadoop. However, it makes intensive use of persistent storage, since the results of each MapReduce task are flushed to disk. Therefore, to reuse the data in a another Hadoop job, data must be read again from storage. Secondly, Hadoop is structured around a single master called NameNode and many DataNodes. The architecture produces horizontal scalability issues as the NameNode becomes a bottleneck and availability shortages occur.

3.1.2 Apache Spark

After considering the design issues of Hadoop, presented in the previous section 3.1.1, Apache Spark was proposed by extending the MapReduce framework with substantial changes. The core functionality of Spark resides in the introduction of Resilient Distributed Datasets (RDDs) [16].

RDDs key aspect is immutability, which guarantees that an object won't be modified once written. Through this characteristic, Apache Spark reuses data by keeping it in memory. Also, since no modification can occur, methods applied to data are lazily evaluated. It results in improved performance as operations can be chained and scheduled together. Finally, strategies to maintain coherence become also simpler.

Another feature is the extension of MapReduce by adding more parallel operations working with RDDs [17]. In Spark, there is a difference between actions and transformations. Actions describe procedures applied to RDDs which produce a result presented to the user. On the other hand, RDDs transformations generate new RDDs by taking advantage of lazy computation. Transformations are only computed when actions that require them are applied.

The introduction of Spark represented a step-up in Big data thanks to its flexibility, notable performance and easiness of use. The learning curve allows inexperienced users to interact with Spark easily. Like Hadoop, Spark offers mechanisms for integration and the list of supported software is extensive.

3.1.2.1 Apache Spark Streaming

There are many situations where a constant flow of information needs processing. Typical examples are information provided by sensors or the analysis of system logs. In both cases, the input data needs to be analyzed and probably, stored later.

The most important factor is the time to process information since it is made available. Processing the information as soon as possible is not always possible because many sources of data need to be integrated. For instance, two sensors placed at different places. One may have lower latency and make the data available before, but their information has to be processed together. In this case, computation is delayed until the data from both sources of information, corresponding to the same timestamp, is available. In this context, Apache Spark Streaming seeks to provide support to workflows including streams of data.

Apache Spark Streaming is an API that extends Spark in order to bring processing of continuous flows of data with the Spark core and its RDDs. Until the newest version, the strategy has been to perform microbatching. RDDs are built from the input stream of data and iteratively processed in batches. The user can define a particular window of time of at least half a second. In this way, it is possible to customize the trade-off between performance and data freshness while ensuring that corresponding data from streams are treated together.

With Spark 2.3, the Spark Streaming underlying mechanism has shifted to Structured Streaming through Continuous Processing, tagged as experimental [2]. This change should reduce the processing time from half a second to milliseconds, thus reducing latency and increasing throughput. The idea originated after alternative software solutions started to perform better than Spark Streaming, and the long processing time was the main reason why people were using other solutions. Other advantages of Spark Streaming are the guarantee of not losing or ignoring data, easy integration with other software and the convenience of RDDs, which can be reused from Spark.

3.1.3 Apache Storm

One of the first projects that aimed at processing distributed flows of data. It focuses on unbounded streams of data with processing times in the order of milliseconds. Storm supports both continuous flows of data and microbatching. In fact, Storm is very versatile because allows the user to write programs that will enable any processing pattern.

At-least-once and at-most-once semantics are guaranteed when processing data. However, exactly-once can only be achieved through a different high-level API called Trident [3].

Storm versatility is also one of its drawbacks since users need to implement many functionalities that other systems provide out of the box. On the other hand, users programming for Storm don't need to be highly skilled in Java and can simulate deployments in local mode, which can be paused and debugged.

3.1.4 Apache Flink

In the context of Big data, numerous use cases include continuous flows of data as sources of information. Apache Flink is a project designed to process data streams in a distributed environment, focusing on high performance and accuracy of data.

Flink implements exactly once semantics, which means that each object is evaluated exactly once. Other systems provide weaker constraints of at-least-once or at-most-once where multiple evaluations can occur or none at all respectively.

Occasional delays when analyzing data can occur due to its lightweight fault tolerance. If a Flink node crashes, it is restarted and data is not lost due to distributed checkpoints. Then the execution is resumed, and as a side effect, the analysis of the stream is delayed.

3.2 In-memory computing

After accounting the growth of information to be processed and a change in computing hardware around 2010, novel approaches appeared to speed up data accesses. The hardware elements that have made these solutions possible are a

reduction of the price per gigabyte of volatile DRAM, the introduction of Solid States Drives (SSDs) and the popularization of clusters with high-density memory nodes and local storage.

Different solutions had been implemented to take advantage of those changes to provide memory latency access to distributed datasets that don't fit the available memory. Coherence, fault tolerance, and the interconnection of storage systems are some of the challenging aspects of in-memory computing.

3.2.1 Alluxio

Formerly named Tachyon, started as a project in the UC Berkeleys AMPLab and was open-sourced by the end of 2012. Its community has been growing at a notable speed, mainly thanks to the number of companies and research centers that started to include Alluxio in their workflow with satisfactory results.

Besides achieving data accesses at DRAM speed when possible, Alluxio provides a single layer of storage unification to reduce the efforts of managing tiered, disparate storage and their access from between applications. Because it leverages applications data access to storage systems, it can be said that Alluxio sits in-between computation frameworks and storage systems, reviewed in sections 3.1 and 3.3 respectively.

On the top, an API exposes the distributed in-memory data, streaming sources of data and storage systems as a unified storage. Since the amount of volatile DRAM is typically smaller than the datasets, local caching and eviction strategies are employed to guarantee fast accesses to frequently used data. By letting Alluxio manage the data, different jobs can share results at memory speed and take advantage of data locality.

From a programmers perspective, Alluxio provides an API for memory-mapped I/O and an alternative filesystem interface. The latter allows Hadoop and Spark among others to interact with Alluxio as a storage system. As a non-programmer, datasets, performance metrics and storage usage can be browsed easily through a WebUI.

Recently, with Alluxio v1.6 a new feature named Lineage has been introduced. Upon activation, data is stored in memory and backed up to storage asynchronously. The main benefit is an increased performance at the cost of lower

fault tolerance. Checkpointing allows handling errors and tasks failures. Lineage assumes that applications are deterministic and will restart a task upon a failure.

On the bottom, Alluxio provides an abstraction of storage systems through pluggable backends exposed in a unified API. Consequently, users will interact with an aggregated view of the data from different sources. Due to Alluxio's ability to cooperate with various storage solutions, it can also handle tiered storage transparently.

Lastly, by having Alluxio handle the configuration to access underlying storage systems, the time dedicated to setup storage access for each user is eliminated.

Concerning the architecture, Alluxio follows a master-slaves design. A primary master is in charge of handling the metadata, while a secondary master stores the changes produced in the primary master. Since Alluxio partitions data into blocks which then distributes, the primary master is in charge of keeping track of the blocks.

Alluxio contains a high availability mode that can be activated when ZooKeeper is present. In this scenario, ZooKeeper tracks the master state, and when a failure occurs, elects a new master from a pool of standby masters. During this recovery process, clients will perceive the master node as unavailable and receive timeouts or communication errors.

On the other side, worker nodes are only responsible for managing their blocks. As stated before, workers don't have a concept of the mapping between blocks and files. Consequently, clients need to contact first the primary node to obtain information regarding the workers holding the desired data and the blocks distribution. With that information, clients will contact the workers directly to request storing or reading blocks. Once a block is stored in a worker, it is not moved to a different node for rebalancing purposes, reducing the overall communication.

3.2.2 Apache Ignite

In the field of in-memory computing, a new project starts to gain popularity. Apache Ignite provides in-memory data access for caching but acting like a distributed database, and optionally, persistent storage. The project aims to obtain the benefits of the three mentioned solutions; horizontal scalability, SQL and key-value data access patterns, availability, strong consistency, data locality and fast

data access.

An Ignite cluster is composed of homogeneous nodes which hold data and process tasks. Data is distributed evenly through hashing and rebalanced upon cluster changes. These features allow Ignite to scale horizontally.

Initially, Ignite only provided memory caching and SQL-like data access patterns. It allowed performing SQL operations such as Joins and access data as key-value on the in-memory data. It handled distribution, fault tolerance. On the downside, SQL or scan queries only included results stored in memory, and not in the external database, since Ignite cannot index external data because it would be too computationally expensive.

In recent version, persistent storage has been added. It allows Ignite to perform the same in-memory queries to persistent storage, and be able to provide in-memory data access for datasets which wouldn't fit into the cluster memory. By letting Ignite handle also the persistent storage data can be indexed, which was not supported using 3rd party storage.

Ignite can operate in two modes, transactional and atomic. The first enables ACID properties by grouping multiple operations into a transaction. The latter, performs each atomic operation independently. Since the transactional mode is more restrictive and makes use of locks, it decreases the performance notably. Consequently, it is recommended to use atomic mode whenever ACID properties are not mandatory, for instance, when out of order execution doesn't affect the results.

Lastly, Ignite supports partial operations when running in atomic mode. For example, operations on large datasets can fail but won't rollback the results. Instead, they alert the user of the failure and report which keys weren't modified. With this mechanism, the user can take the opportune course of action and recomputation is not needed.

3.3 Storage

3.3.1 Apache Cassandra

Back in the years when pools of storage were built upon rotational disks, data stores developers and users tried to optimize data access patterns to reduce slow random I/O. Because of the physical layout of data, HDDs deliver fast read and write operations, but mixing random accesses degrades the performance notably.

Apache Cassandra is a NoSQL database designed to reduce random I/O and promote sequential reads, and especially, writes, which are the most expensive operation in terms of latency. Cassandra was open-sourced by Facebook, who got inspired by BigTable and DynamoDB.

A cluster of Cassandra nodes is homogeneous, all nodes have the same role, and data is distributed by hashing which ensures a certain degree of load balance. This mechanism allows clients and nodes to be aware of data locality. When a client performs a petition to a node, it gets propagated to the nodes responsible for the information. Then, the answers are gathered and sent back to the client. Lastly, after adding or removing a node, existing nodes delegate or assume a block of hashes. This architecture ensures horizontal scalability and rebalancing of data.

More internally, Cassandra organizes its data into SSTables of key-value pairs, called commit logs, allowing only append and read operations. In this way, writes are sequential, and only a penalty is incurred when reading is mixed. However, since it maintains order by keys, sequential reads can be easily achieved for most workloads.

On a remove command, a tombstone is written because a deletion would be costly and is unsupported by the commit log. When data is accessed, the SSTable is read backward with the most recent data found earlier, allowing to read consistently.

When the commit log grows as data is added, the time to process a read is also increased. To mitigate this effect, and to save storage space, Cassandra has a mechanism called compaction which runs from time to time. During this step, operations in SSTables are merged updating the most recent value for each key and removing duplicates.

Cassandra can achieve good performance and maintain horizontal scalability.

However, since it runs inside a JVM and makes use of locks for communication other solutions can achieve higher transactions per seconds, particularly with newer storage technologies.

3.3.2 Apache Hadoop Distributed FileSystem

HDFS, as the name indicates, is a FileSystem as opposed to other solutions reviewed on this section such as NoSQL DBMS. HDFS is designed to run in multiple nodes, each owning a portion of the data. Replication and partitioning are used to produce a highly available and fault-tolerant storage.

However, HDFS makes use of a master-slaves architecture, with the master called NameNode. The workers, named DataNodes, hold the information split in blocks. Each file is registered into the NameNode together with the information of the decomposed blocks and their locations.

When a client wants to access the data, it will ask the NameNode who assign a set of blocks. Therefore, when a cluster grows, all clients end up contacting the NameNode and DataNodes sending messages regarding blocks metadata, quickly becoming a bottleneck even if data is kept only in memory.

Also, running applications on top of HDFS should make a balanced use of data since HDFS does not provide rebalancing. However, it includes a utility with this purpose that users can run to rebalance the cluster.

HDFS is a component of the Hadoop framework, which works very well with the MapReduce paradigm thanks to blocks distribution, which allows MapReduce tasks to execute benefiting from data locality.

3.3.3 Apache Hbase

Based on BigTable which is under proprietary license by Google, the main ideas were described in [1] and HBase was built following them with minor modifications. HBase can be described as a NoSQL database storing data as key-value pairs following a columnar model. Additionally, HBase indexes the data which allows for fast look-ups. Therefore, HBase is more prone to obtain a good performance accessing small, random objects, while HDFS was designed for writing or reading sequentially large amount of data. As opposite as many other NoSQL systems,

strong consistency is guaranteed.

From an architectural point of view, HBase is similar to HDFS since it follows a master-slaves design, where the master is responsible for monitoring metadata changes and the RegionServer nodes manage the data. A configuration with multiple master nodes in standby is possible with ZooKeeper, which will be ready to designate a new master upon failure. Since clients communicate directly with the RegionServers, the cluster and clients could withstand a master failure for a short period of time. Lastly, when nodes join or leave a cluster, data is automatically rebalanced

In overall, HBase performs well in use cases involving large amount of data, that can be described following a key-value model with individual accesses.

3.3.4 Ceph

Ceph brings a totally different approach with respect to the storage solutions reviewed in this section. Ceph is a data store designed to be deployed in a small cluster with lots of storage devices. Frequently, computation on storage nodes is forbidden, and users wanting to access the data need to be authenticated. Ceph may have high usages of CPU, making convenient to avoid co-allocating computation and storage.

Designed as a layered storage system, includes a filesystem, a block storage and an object storage. Users wanting to obtain the best performance typically need to interact with the rados block storage, which require a high level of expertise. As many other NoSQL storage systems, Ceph uses hashing to partition, distribute and directly locate data.

To improve performance, Ceph provides a cache tier, recommended to be run with SSDs. This approach allows Ceph to be run with slower technologies and reduce its cost. However, since typically Ceph runs in separate nodes, there is no data locality and the network can become a bottleneck easily, or slow down accesses because of latency.

From an architectural point of view, Ceph is an heterogeneous cluster comprised of Object Storage Daemons (OSD), Metadata Servers (MDS), Monitors and Managers. The most important being the OSD which handles the read and write operations, coherence and distribution. Secondly, the MDS acts as a master

storing metadata information about the objects. A single MDS is active at a time, with other MDS waiting in standby. Lastly, the Monitor holds the cluster information and configuration, while the Manager is only responsible for keeping track of metrics and provide a REST API.

3.4 Auxiliary systems

In this section software with different functionalities are reviewed. The most relevant for this project have been selected. They aim to help running distributed software smoothly, help reducing failures or participate in some form in data governance.

3.4.1 Apache ZooKeeper

Started by the Apache Foundation, Zookeeper has the scope of supporting distributed software by providing coordination mechanisms. It can run in a set of nodes, which will maintain the messages received by the nodes running in the data center. For an instance, a master node can send a heartbeat to ZooKeeper, but upon failure, ZooKeeper will react and contact standby masters to elect a new master node. Typically, ZooKeeper stores node status, configuration of nodes, and cluster layouts.

ZooKeeper runs in multiple nodes, called ensemble, where one is the leader and others the followers who replicate the transactions and serve information to clients directly. This mechanism allows ZooKeeper to be fault tolerant, and perform well under a heavy read load.

The leader is responsible for storing status information in chronological order as forwarded by the followers, who receive the petitions from clients. Therefore, ZooKeeper can serve information from any node but the leader will end up gathering all the updates to be written in serial order, which ends up in great read performance but poor write performance in when writes represent more than a tenth percent of the operations.

Since writes need to be propagated, a node can answer to a client with outdated information, which is why ZooKeeper is eventual consistent.

Changes are applied at the master node, in memory, and then flushed to disk into log files. Nodes keep all data in-memory for fast access, which limits also the amount of information that can be stored in ZooKeeper to the amount of memory available.

3.4.2 Apache Kafka

When internet companies see their user base grow, the number of user interactions increases exponentially. For any large business, analyzing user behavior and reacting in real-time, let it be through recommendations or updating an information panel, is essential.

On the other hand, HPC systems needing to process data streams may encounter inappropriate the streaming processing projects mentioned in section 3.1 due to performance, scalability and fault tolerance properties.

With these ideas, and wanting to unify the flows of data, Linkedin started the Kafka project, later open-sourced. Kafka does not provide elaborated processing capabilities. Instead, organizes events from different sources and makes them available to processing frameworks or storage systems to be consumed. Consequently, Kafka does not aim to replace data streaming processing engines but to complement them.

The smallest unit of data is the record, which follows a key-value format. They are assigned to a topic, which in turn is split into partitions. Thanks to the division of topics into partitions, it is possible to distribute a topic among nodes to ensure scalability and availability. On the other hand, partitions behave like commit log only supporting appends and reads of records. This organization makes possible maintaining the order of the data received, guarantee consistency and perform efficient reads and seeks. Since storing the data infinitely would saturate the persistent storage, data can be marked to expire after a period. Figure 3.1 and 3.2 illustrate how Topics and Partitions are organized internally. is time and can be split into smaller tasks.

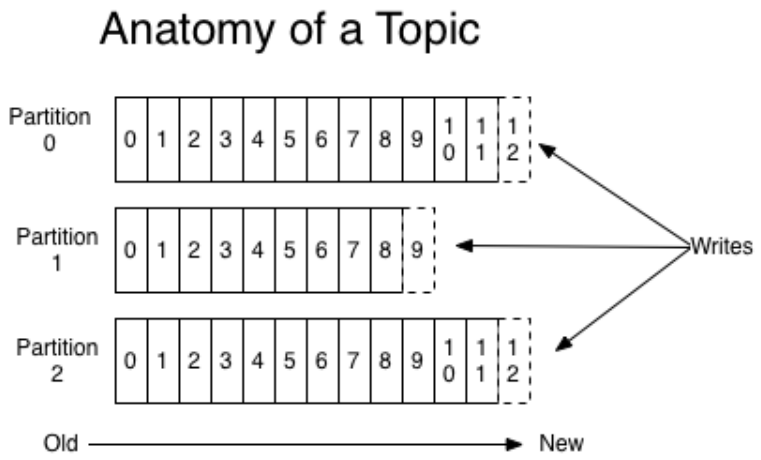

Figure 3.1: Internal organization of a Kafka topic.

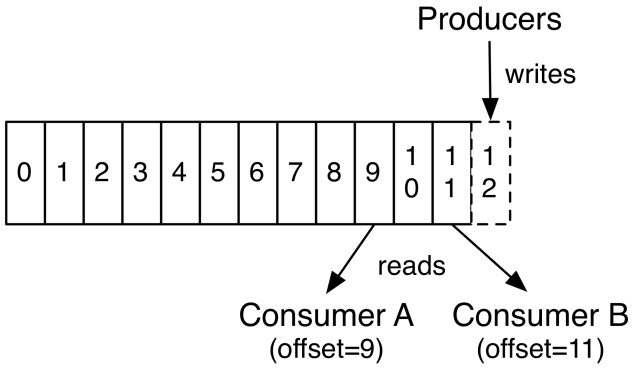

Figure 3.2: Kafka internals and producer-consumer mechanism.

To access the data, Kafka follows a producer-consumer by subscribing to a partition of a topic. When a consumer subscribes, data streams layout is retrieved, and the subscription is performed directly to the node. If a node fails or the layout changes, the connection is dropped and consumers fetch the updated layout. Then, since the consumer keeps track of the event offset, it can jump to the desired position.

Lastly, because Kafka was designed to interact with other systems, it needs to be plugged with connectors. They are responsible for transmitting the data to Kafka following a particular format, which is the same used internally in persistent

storage or to interact with producers and consumers. This aspect permits Kafka to perform zero copies and help improve performance.

To sum up, Kafka delivers a distributed, fault-tolerant and efficient event manager which integrates nicely with other technologies.

3.4.3 Apache Beam

Typically, each data streaming processing software come with its own API. Apache Beam unifies APIs into an abstract API and decouples them from the execution runtime. The same code can be executed with Apache Flink, Spark or some others .

Batch and streaming are the same API. Key concept is portability, easy to migrate and to test on different backends/engines.

3.4.4 Apache Hive

Relational Database Management Systems founded the grounds of data warehouses. They use SQL-like queries, which are intuitive and describe human reasoning. However, unlike RDMBS, big data projects don't guarantee ACID properties nor a strong consistency, or even indexation, making it difficult to support SQL-like queries. However, data scientists need to access the data in a transparent and organized manner describing what they want to analyze in simple statements.

Hive tries to fill this gap by providing a data warehouse built on top of Hadoop MapReduce and its runtime. It can be deployed alongside many big data software to provide near SQL capabilities to data scientist. By expressing queries in HiveQL, which is similar to SQL, data scientist can analyze the data without knowledge of the underlying system. These queries are transformed to Hadoop MapReduce jobs and then executed by Hive.

At its turn, the Hive Metastore component stores information about confgured backends, their layouts and how to contact them. It can be used without Hadoop to provide a service describing the storage systems and their access.

The main flaw is the large amount of time it takes to complete queries, and that simple user queries can be expanded into very complex data accesses consuming lots of resources. It is a consequence of the trade-off taken in order to expose

heterogeneous storage systems as a data lake.

3.4.5 Presto

Because of the need to run light queries over large datasets, from the perspective of a data scientist, Facebook developed Presto and released it under Apache 2.0 Licensing. It is designed to run analytical queries on HDFS without transforming the data and taking addvantatge of its own runtime, instead of Hadoop jobs.

With the popularization of Presto among big data companies, many connectors have been developed. Nowadays, Presto supports a wide variety of data sources. It is designed to run interactive lightweight queries, as opposite of Hive, which is design for complex processing. Thanks to the quick processing of queries, Presto is gaining popularity. It is currently used by Shazam, Facebook and many others while its adoption in HPC is limited to few use cases.

Chapter 4

Proposed approach: In-memory data staging

Reviewing big data software solutions in the previous chapter 3 was the first step towards designing an in-memory big data solution. The second step consists of formulating a working architecture where both individual and collective aspects are considered.

The following sections aim to highlight how individual characteristics will contribute to the projects, as well as, describe their interactions. Through this process, software components will be selected to propose an architecture to fulfill our requirements.

4.1 Software stack discussion

4.1.1 Computation frameworks

Two different approaches are considered in data processing frameworks, batch and stream processing. Apache Spark is the most promising tool in the field of batch processing among the discussed frameworks in section 3.1.

Its efficiency in batch processing is tightly related to smart data-aware scheduling policies. In particular, implements mechanisms to restart and reschedule failed tasks without restarting the complete job while taking in consideration data locality to speed up the execution. Also, its stability and the ability to integrate

with the big data ecosystem help Apache Spark to be some steps ahead of Apache Hadoop MapReduce regarding efficiency and flexibility.

The field of stream processing is evolving fast with many new players joining the big data ecosystem. In the context of stream processing, both Apache Flink and Apache Storm deliver the highest performance in terms of records processed. However, Apache Storm requires a higher degree of expertise and doesn't guarantee the exactly-once semantics, while Apache Flink does. Also, Storm is more robust and tolerant to failures than Flink's fault tolerance, based on checkpoints and restarting. Recently, also Apache Spark has joined the stream processing ecosystem with its new runtime called Continuous Processing. It offers unbounded processing at the same time as batch processing.

Due to the simplicity of being able to test both stream and batch processing, the published performance, and the simplicity to deploy and integration, Apache Spark is preferred for batch processing over the alternatives. Otherwise, for stream processing, combining a setup with Apache Spark and Apache Flink will interesting.

4.1.1.1 In-memory computing

Secondly, the in-memory projects presented in 3.2 are confronted in this section. Both Alluxio and Apache Ignite deliver in-memory speed data access and integrate with data stores. The former has been around for many years, implements the Hadoop FileSystem API and can be deployed as part of Spark replacing its executor and cache. Furthermore, its community has been growing at an incredible rate at the same as its adopters. On the other hand, Ignite is at an early stage and features are still rolling out, with support for persistent storage recently added. Ignite provides nice features for end users, such as SQL like language to explore the data and machine learning capabilities.

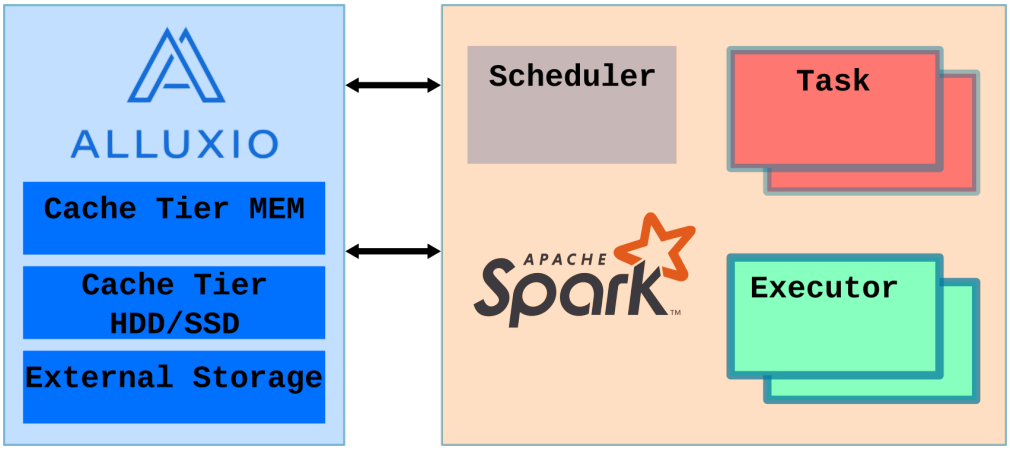

Figure 4.1: Spark and Alluxio integration schema.

Apache Ignite features a homogenous architecture which should scale horizontally, while Alluxio features a single master - multiple slaves architecture. None of the adopters have reported scalability issues with Alluxio while most of the reports focus on the benefits obtained when performing data processing at memory speed.

Regarding the consistency level, both projects provide strong consistency. Alluxio implements a lock at the data block level during writes and moves. On the other hand, Ignite offers tunable consistency at two levels: ACID properties or eventual consistency.

One relevant feature of Alluxio is the tiered storage, that can be defined at different levels and small granularity if needed. Ignite lacks this degree of configuration which is more general.

Lastly, because Alluxio can be easily integrated with Spark, and offers support for Apache Flink and Apache Kafka, it will be used in this work.

4.1.2 Storage

The frameworks selected for computing and memory management constrain the storage solutions. A distributed storage enabling Apache Spark and Alluxio to be aware of data locality is essential. Since Alluxio handles data as storage blocks, we might be more interested in sequential access performance rather than random I/O, which will be intercepted by Alluxio in subsequent data accesses.

Scalability and performance concerns discard Ceph for this proposal. Secondly,

Apache Cassandra can't be easily integrated with Alluxio, whereas HBase and HDFS can. Finally, Hbase is focused on random I/O and analytics on continuous changing data while HDFS provides fast sequential access on both reads and writes. Since the amount of random I/O reaching the underlying storage should be minimal thanks to Alluxio, the best fitting storage to start with is HDFS.

There are situations, such as testing, where the data doesn't have to be persisted after a set of computations. In this case, since the cluster should be always available and failure tolerant, local storage devices such as SSDs and NVMe can be used as the underlying storage.

4.1.3 Auxiliary systems

Nowadays, a big data infrastructure is composed of a set of services running distributed in a cluster. Since managing the individual configurations and node failures is necessary for any deployment, different projects have emerged to fill this gap. In this area, Apache ZooKeeper is the most prominent, which offers support for discovering heterogeneous services. In addition, it can handle services configuration and redeployment upon failures, as well as support election protocols for master nodes. The mentioned benefits compensate for the small amount of resources required for this extra player.

With regards to message brokering, Apache Kafka can play an important role when used with stream processing software such as Apache Flink and Apache Storm. Because of its efficiency and highly scalable properties, Apache Kafka should be deployed in most setups involving large-scale stream processing.

In its turn, Apache Hive and Presto can provide interesting features to an already running cluster for data analysis. They offer data insights with ease and abstract the final user from the underlying storage infrastructure. Consequently, the adoption of these solutions is subjective to the type of data analysis to be conducted, based on user expertise and data characteristics.

Finally, Apache Beam has the potential to speed up data processing workflows. However, since it is not tightly related with this project and will not participate in demonstrating the benefits of an in-memory staging system, it will not be included as part of the proposal.

4.2 Design the architecture

Compute nodes are typically divided into memory, storage and computation units. Due to the use of tiered storage in this project, it is necessary to propose a logical division between storage tiers. Furthermore, as mentioned in the previous section 4.1, our proposal will also provide external services to the running applications.

The *data discovery* group will enable data to be queried directly by the users, responsibility delegated to Apache Zeppelin. In the second layer, Apache Spark and Apache Kafka will perform the *data delivery* process from external sources. Lastly, Alluxio will be in charge of the *in-memory* layer while HDFS will be responsible for the *persistence* of data. Figure 4.2 illustrates the proposed schema along with the proposed software stack.

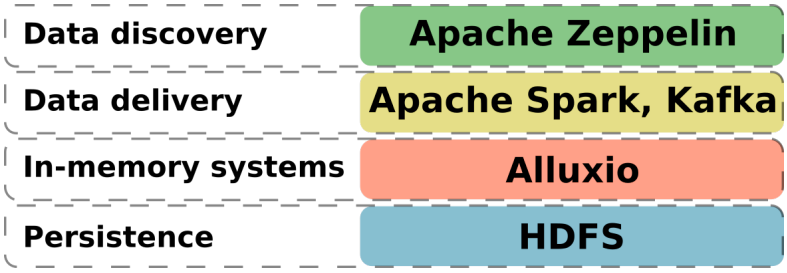

Figure 4.2: Software assigned to the proposed functional architecture.

The presented classification divides and groups the software according to the data management responsibilities. However, a concrete design mapping the software stack and their data movements to hardware is given by Figure 4.3. It is clearly described how the computation, data delivery and data discovery frameworks will access the data stored in the system.

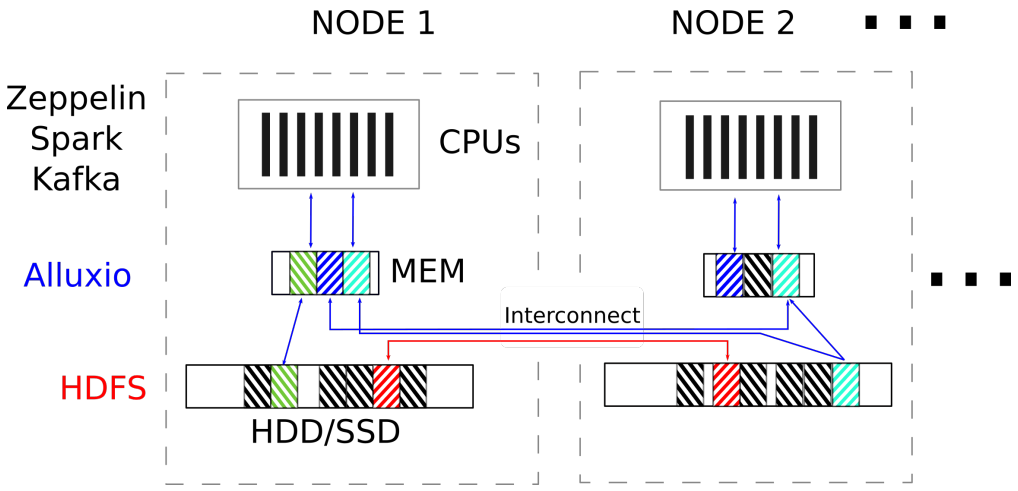

Figure 4.3: Software and hardware correlation.

It can be appreciated the data partitioning schema proposed in this work. Alluxio will be in charge of managing the data at different levels. First, it will provide computation frameworks the ability to access data at memory speed when possible or to load data from an another Alluxio node or HDFS transparently.

Secondly, it will be in charge of forwarding the data to the persistence layer. In its turn, HDFS will be responsible for data replication to avoid fault tolerance and improve the chance to exploit data locality in various nodes.

4.3 Software components interactions

4.3.1 Memory management by Alluxio

Alluxio handles data as files stored in different storage tiers, according to the particular configuration. The primary purpose of Alluxio consists of taking advantage of high-speed volatile memory to save objects and allow external applications to share the data.

The memory management is performed by storing the objects as regular files in a temporal filesystem (*tmpfs*) or a ramdisk with a ram filesystem (*ramfs*), both providing POSIX compliance. The first method will automatically send the data stored from the memory to the swap space when is becoming full. An undesired behavior in this situation because Alluxio manages different tiers of storage with

eviction strategies better than the tmpfs. Also, it would come with a noticeable penalty, since the swap space might be allocated in a slow hard drive. These reasons make the ramdisk the preferred mechanism for Alluxio in-memory storage, which should deliver better performance and stability.

Alluxio applies data placement in blocks, where each block is stored as a file in the storage tier designated. As a consequence, in-memory data can be accessed easily from a terminal by navigating to the Tier 0 path configured in Alluxio. Then, internal data can be accessed and visualized as shown in 4.4, where each block is mapped to a file that can be opened and visualized. File names are given by the block identifier, designated by the master node. And because no encoding takes place, contents can be easily inspected with external tools.

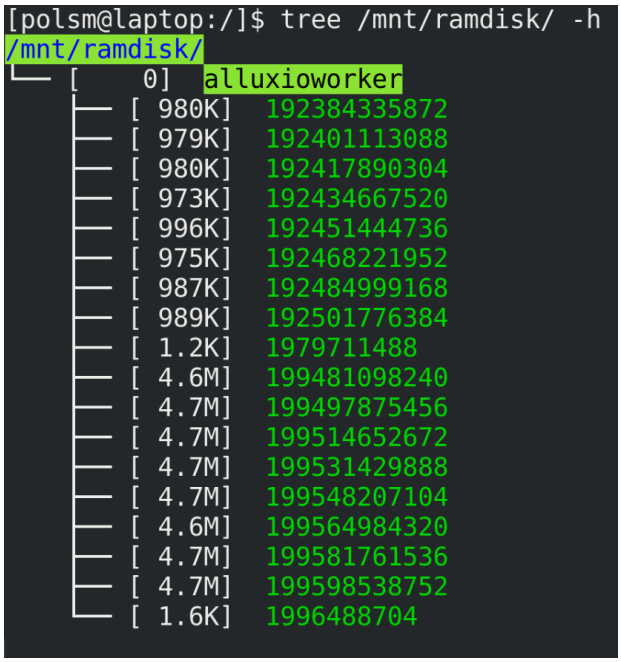

Figure 4.4: Alluxio in-memory blocks organization on a Tier 0 ramdisk.

Objects managed by Alluxio are immutable, and so are the blocks where they are placed. Because management of tiers is done transparently, Alluxio provides a simple FileSystem API to create files identified by a URI as shown in Table 4.1. With this mechanism, it is able to map files to block ids and their locations.

```
AlluxioURI path = new AlluxioURI("/myFile");
```

Table 4.1: Addressing a file through the Alluxio FileSystem API.

4.3.2 Apache Spark Objects

As explained previously in section 3.1.2, the core objects of Spark are the RDDs. Nevertheless, DataFrame and DataSets objects have been introduced recently. A dataset groups different RDDs, while a DataFrame groups a set of DataSets holding rows of RDDs in a columnar manner, with a name given for each column.

Following the Spark methodology, DataFrames and DataSets are lazy evaluated, which means they are computed only when their data is needed. Meanwhile, they are stored in a working plan of how to obtain the data. Since they are composed of RDDs, they will also be immutable. Because an RDD is typically spread among different nodes, immutability guarantees the reproduction of failed tasks, simplifies strategies for allocating computing resources, distributing the datasets and implement failure tolerant techniques.

However, immutability comes with a cost when information is updated continuously. Real-time analyses or simulations are two representative examples. In these situations, recent data is more relevant and has higher probabilities of being accessed. As a result, RDDs need to be created and distributed continuously which is a costly operation.

RDDs are written in the Scala language and can be stored in FileSystems, Data Stores and many others. Different subclasses exist to provide particular operations related to the type of data they represent. For instance, the PairRDDFunctions class provides operations to be applied to keys such as foldByKey or groupByKey. Finally, different mechanism allows persisting RDDs to storage, with the most used being *saveAsObjectFile* and *saveAsTextFile*.

The *saveAsObjectFile* method groups small-sized objects, serializes them, and writes a large SequenceFile. The format of a SequenceFile was developed by the Hadoop community to reduce the number of small objects and files to be tracked, improve I/O and reduce memory consumption by the NameNode.

On the opposite, the *saveAsTextFile* method serializes objects using their String representation. Later, they are stored as Hadoop files with its data optionally compressed.

4.3.3 Alluxio integration with Spark Objects

The recommended procedure to store and retrieve Spark objects from Alluxio is to save objects as text files and then reconstruct them. This mechanism permits accessing objects individually, whereas storing the objects with the java serializer through the method *saveAsObjectFile*, which groups small objects, would have an overhead. The default serializer is provided by Java, with the Kyro serializer being a popular second choice. Custom serializers can also be implemented.

On the other hand, the design of the *saveAsObjectFile* was made thinking on the premises that reading and storing blocks of data performs better than individual accesses. However, the increased throughput that random access memory provides removes the I/O bottleneck. As an example, writing 512 bytes blocks of data outperform 4k writes in any SATA drive. This difference is less significant when looking at random access memory instead of hard disks or solid state drives. Also, it moves the bottleneck from the storage to the processor. Therefore, since also serializing and deserializing puts pressure on the processor, RDDs should be stored into Alluxio through the *saveAsTextFile* method. In this way, RDDs will be used as if they were a regular file.

Because an RDD is spread throughout a set of nodes, having a caching layer in the same node will guarantee that further accesses to the RDD are performed at memory speed. Besides, thanks to the tiered storage management, when the memory starts to fill up, stale data can be sent to lower storage tiers to leave space for new data. Occasionally, Spark stores partial computational results temporally, until they are garbage collected. In concrete, shuffle results are kept for the same object.

Lastly, the newer DataFrame API in Apache Spark includes new support for different storage backends. A frequently used method to access and retrieve DataFrames from Spark is employing parquet files. Through its API, designed to work with HDFS blocks, data is reorganized to obtain increased performance following a columnar model. By reading a metadata file, data chunks are located to allow sequential read of blocks. Since the underlying mechanism is based on files, they can also be stored into Alluxio as parquet files or as text files with *df.write.parquet(alluxioFile)* or *df.write.text(alluxioFile)* respectively. Besides plain text files, other output formats such as JSON and CSV are supported by the API

which will be successfully stored in Alluxio.

4.3.4 ZooKeeper applied to this work

To allow service discovery and organization, ZooKeeper provides a namespace following a tree architecture and the same syntax as a filesystem. Therefore, nodes can register by creating a node in the hierarchy with associated data. Secondly, clients can place a watch on a node to receive a notification if a change occurs.

These mechanisms allow standby masters to monitor the active master and detect failures. Since Spark, Alluxio and HDFS rely on standby masters, ZooKeeper can support their recovery processes when a master experiences a failure.

Standby masters have an open connection with a ZooKeeper maintained with heartbeats. When an active master fails, and a secondary master needs a promotion, an election takes place. By using the namespace mechanism and timestamps, an agreement can is established on which node will be elected master if a determinate node fails.

4.3.5 Alluxio and HDFS data models

HDFS stores data partitioned in blocks of up to 128MB in the current HDFS 3.0.3. In its turn, Alluxio follows a data model based on blocks of 512MB by default. To enhance data locality and optimize space, the best configuration will be to configure the block size to the same value.

However, in situations where data locality is not common, setting bigger blocks in Alluxio than HDFS should deliver better results. In this way, when a piece of information should be requested by the client, Alluxio will receive blocks from different nodes simultaneously and keep them in memory for later access.

In the scenario where the HDFS blocks are bigger than Alluxio's blocks, it would be possible that data not requested and which will never be used will be placed into memory, reducing the amount of memory for other useful objects and triggering evictions.

As in many other cases, the theoretical numbers may be subject to hardware and workload characteristics. Therefore, configuration depends on the use case and experimentation should be conducted to define reasonable parameters.

Chapter 5

Development

Due to time constraints and the inherent complexity of deploying a complex software stack in an HPC cluster, a subset of the proposed system will be implemented as a prototype.

The selected software stack must be able to accomplish the project objectives. With this purpose, a more in-depth analysis of the key aspects will be conducted in section 5.2 to individualize the restrictions and required modifications. The analysis focuses on the features needed to demonstrate the efficacy of staging data in-memory. This chapter aims to link implementation characteristics essential to the prototype and to expose the limitations to be solved to complete this project.

5.1 Requirements

1. **Perform updates on RDDs from Spark.** Two forms of updates are desirable, complete and partial. Being able to update a fraction of a distributed dataset would have a huge impact on performance, and in particular, on continuous processing analytics. Besides, being able to replace RDDs would reduce considerably the amount of memory used and reduce programs complexity, since the most recent data will be present under the same location and id.

2. **Evaluate Spark awareness on modifying external data.** Spark caches intermediate results to speed up executions. Understanding the operations

and situations where data is cached is essential to evaluate modified RDDs. Otherwise, computation will be unaffected by changes on externally stored RDDs or files.

3. **Support operations at block level in Alluxio.** Because datasets loaded into Alluxio or RDDs stored through Spark are divided into blocks, updating their content is essential for this project.

4. **Understand Alluxios operations on metadata.** Characteristics of files such as the number of blocks, their location or size are stored in the master node. Since updating data stored inside the blocks can potentially change these characteristics, understanding the consequences, and possibilities of updating their metadata is relevant to the project to enable shrinking or extending already existing datasets.

5.2 Deep code inspection

5.2.1 Spark RDDs updates and changes detection

Spark, by default, does not provide the option to overwrite objects using the same namespace. To achieve this behaviour, the option

```
spark.hadoop.validateOutputSpecs=false
```

needs to be added in $SPARK_HOME/conf/park-defaults.conf. With this change, data can be replaced with an updated version or an entirely different dataset.

Nevertheless, with this mechanism enabled, the entire RDD will be replaced. As a consequence, no minor updates on distributed blocks is possible with the current Spark implementation.

The internals of Spark are complex, and partially updating an RDD would have consequences on many components such as the scheduler or failure tolerance mechanism. Modifying Spark's internal code represents a complex challenge, which is unnecessary to demonstrate the potential of this project.

With regards to the data access, Spark caches shuffle results, its most expensive data transformation. However, by accessing an external storage, a new RDD is created, avoiding reuse of cached data of former RDDs.

For the scope of this project, operations performing shuffles will be avoided. Thereof, calculations will access the in-memory storage, enabling correct performance measurements. Secondly, the mechanism provided by Spark to update data by replacement enables the evaluation of different methods for in-memory data staging.

5.2.2 Granular operations in Alluxio

The provided operations handle entire blocks of data. The most relevant methods are implemented in the worker process. In particular, the data store (*org.alluxio.worker.block.TieredBlockStore*) is responsible to provide the following functionalities to the worker:

- **Create block:** Given an id and the required bytes, allocates a block and writes its contents.
  Method: *ShortCircuitBlockWriteHandler.handleBlockCreateRequest*();

  Proceeds as follows:

  1. Lock the worker metadata structure.
     Method: *TieredBlockStore.createBlock*();
  2. Allocates space in the worker registry and registers the new block metadata
     Method: *TieredBlockStore.createBlockMetaInternal*();
  3. Creates the path, the file and sets up the permissions for the new block.
     Method: *TieredBlockStore.createBlockFile*();
  4. Write the data through the ProtoBuffer library.
  5. Confirm the write with the Netty library
     Method: *Channel.writeAndFlush*();

- **Lock and unlock block:** Sets a lock on the block's metadata, denying write, delete or move operations. This method enables higher-level operations on blocks.
  Methods: *TieredBlockStore.*[*lockBlock*|*unlockBlock*]();

- **Commit or abort a block:** Confirms a block in order for the caching strategy to take the necessary actions to guarantee space in the storage tier. Abort in its turn frees the space, deletes the contents and cancels temporal metadata.
  Methods: $TieredBlockStore.[commitBlock|abortBlock]()$;

- **Move block:** Changes the path to a block with *Fileutils* inside the same node.
  Methods: $TieredBlockStore.moveBlock()$;

From the list, we can conclude that no method allows a partial or complete update on a block. The explanation can be found in the design decision to force immutability on data. This decision is common among distributed storage systems to avoid coherence, corruption and reduce synchronizations.

Besides, analyzing the methods used by Spark to store data in Alluxio we find the operations provided have the same characteristics. The *org.alluxio.hadoop .AbstractFileSystem* class is responsible for interfacing Alluxio with software following the FileSystem API such as Spark and Hadoop. This class delegates operations to other parts of the client application, which will contact Alluxio nodes to perform the requests. Different methods are provided to create, delete, rename or get information but no method allows altering already existing data:

- **Create a file:** create();
- **Delete a file:** delete();
- **Retrieve hosts providing a data block:** getFileBlockLocations();
- **Make a directory:** mkdir();
- **Rename a file:** rename();
- **Open a file for reading:** open();
- **Confirm a file exists:** ensureExists();
- **Obtain the block that constitute a file:** getFileBlocks();

On the other hand, the Alluxio master node only writes and removes metadata, as requested by the workers. Meaning that no alteration of the metadata is expected. Therefore, shrinking or extending blocks is not contemplated in the original design.

5.3 Deployment plan

To be able to update stored blocks in Alluxio, completely or partially, different approaches have been defined. Their particularities, benefits, and drawbacks are discussed in this section.

1. **Modify the blocks externally, without Alluxio's knowledge.**
   The fastest approach not needing to implement new mechanisms and strategies in Alluxio nor Spark. However, metadata will be kept in the memory of the Alluxio process as seen in the subsection 5.2.2. Consequently, blocks can't be created externally or resized without corrupting Alluxio's memory. To avoid coherence issues in tiered storage environments, an approach where updates are applied in all tiers should work. This strategy has the potential to demonstrate how computations can benefit from this project in terms of data freshness, execution time and memory footprint.

2. **Implement partial or complete updates of blocks inside Alluxio.**
   The required functionality for Alluxio to allow partial or complete updates on blocks should be added. This approach consists of taking actions to avoid major coherence issues between storage tiers, workers and the master metadata. The coherency level could be relaxed for this project, since having eventual consistency could be enough. First, functionality to modify blocks metadata inside the master should be added. Secondly, mechanisms to propagate the data changes in tiered storage should be implemented. And lastly, an API to perform partial updates should be defined.

From the two possible approaches, the first option has been selected. Different reasons have driven this decision, including but not only, the uncertainty behind the complexity of modifying a distributed software without previous knowledge of its internals and the duration of the project.

The selected path allows studying the performance impact of sharing memory regions between applications in the big data environment without the need to develop a production-ready software. Therefore, if the results are promising future work could be done to extend Alluxio with this idea. Also, the work carried out analyzing Apache Spark and Alluxio enables continuity of this project.

5.4 Final prototype

5.4.1 Deployment

The Rutgers Discovery Informatics Institute has offered the Caliburn cluster [10] to deploy, test, and evaluate the prototype. Because configuration in a cluster is more challenging, a set of scripts to support the deployment have been developed. They are in charge of extracting environment information from the nodes and customize the software stack according to the available hardware. The main characteristics of the cluster are the use of NVMe and dense memory technologies, Omnipath internode communication and Quickpath for intranode communication between processors. In its turn, every node consists of 2 processors featuring 18 cores each for a total of 36 cores. The cluster span to 560 nodes and a total of 20.160 cores.

NVMe disks are partitioned to distribute the storage among the network file system and the swap space. In its turn, a mounting point featuring a tmpfs running on top of the RAM space plus the swap space is provided under /dev/shm. Consequently, files stored in this partition will be placed inside the memory RAM as cached data until there is a need for freeing space. At that point, the system will offload RAM data to the NVMe disk.

Regarding user storage, two main systems are provided. The Network FileSystem with low latency and high throughput, and a GPFS partition to store large datasets with a higher capacity and reduced performance compared to the NFS.

5.4.2 Deployed prototype

The deployed prototype consists of the Apache Spark project relying on Alluxio for data storage. Spark has been setup in standalone mode, meaning that Spark's

scheduler is used instead of Apache Mesos or YARN, which are also frequently used. In this mode, applications are submitted and run in the master node, allocating the required executors. In cluster mode, the program called driver would be sent to a node to be executed.

In order to achieve data locality between Alluxio and Spark, it is mandatory to declare the $\$SPARK\_LOCAL\_HOSTNAME$ environment variable, preferably in the configuration file $\$SPARKconfspark - env.sh$. Otherwise, Alluxio will represent hosts by their interface name while Spark will use IP addresses. Then, it is impossible for the scheduler to take advantage of data locality because the naming representation differs as observed in [11] and [7].

The following snippets summarizes the Apache Spark configuration:

```
@File: spark/conf/spark-defaults.conf

spark.driver.extraClassPath {user_home}/alluxio-1.7.0-client.jar
spark.executor.extraClassPath {user_home}/alluxio-1.7.0-client.jar
spark.hadoop.validateOutputSpecs false
spark.executor.cores 1
spark.executor.memory 4g
spark.driver.cores 4
spark.history.fs.logDirector {user_home}/sparkHistory
spark.eventLog.enabled true
spark.executor.heartbeatInterval 30s
spark.network.timeout 300s
spark.driver.maxResultSize 2g


@File: spark/conf/spark-env.sh

SPARK_LOCAL_HOSTNAME=$(hostname)
SPARK_WORKER_MEMORY=64g
SPARK_WORKER_INSTANCES=2
SPARK_WORKER_CORES=16
```

The rationale behind the Spark configuration is tightly related to the cluster characteristics. Since a total of 256GB of RAM are present in each node, half of it has been dedicated to the Apache Spark workers. Having Spark workers with high memory capacity has a negative impact on the Java Virtual Machine (JVM). The garbage collector performance is directly related to the amount of RAM and objects in memory. Therefore, having workers with big amounts of memory is unadvised and for this project, two workers per node have been setup.

As a remarking point, Spark allocates a fraction of the memory more than requested for the JVM. Then, a portion of the available memory will be used by executors to perform computation, or optionally, to cache RDDs which is not applicable to this work. Even if these settings can be customized, they have been left to the defaults values. The document provided by [9] is a good start to understand the effects of modifying the memory fractions.

Regarding the cores dedicated to each worker, we need to leave some cores available for Alluxio and the Operating System. Since Spark is CPU intensive, as opposite to Alluxio, 32 cores have been dedicated to Spark to leave 4 cores for the rest of services.

Finally, depending on the application and the hardware, a different number of executors is recommended. In this case, since we want to analyze parallel applications with almost no data sharing, 1 executor per core has been assigned. This decision leaves each node with 2 Spark workers with 64GB of available RAM each, where each worker has 16 executors available with 4GB of RAM each.

On the other hand, configuring Alluxio turned out to be very smooth. Only a single file has been modified and the only requirement has been to enable ssh to localhost on the laptop during the development since in the Caliburn cluster it was already enabled.

`@File: alluxio/conf/alluxio-site.properties.template.`

```
# Common properties
alluxio.master.hostname=PLACEHOLDER_MASTERHOSTNAME
alluxio.underfs.address=PLACEHOLDER_UNDERFS
```

```
# Worker properties
alluxio.worker.memory.size=128GB
alluxio.worker.tieredstore.levels=1
alluxio.worker.tieredstore.level0.alias=MEM
alluxio.worker.tieredstore.level0.dirs.path=/dev/shm/alluxioworker
alluxio.worker.tieredstore.level0.dirs.quota=128GB

# Options to enable NVMe as a second tier
#alluxio.worker.tieredstore.level1.alias=SSD
#alluxio.worker.tieredstore.level1.dirs.path=/tmp/alluxioworker
#alluxio.worker.tieredstore.level1.dirs.quota=64GB
#alluxio.worker.tieredstore.level1.watermark.high.ratio=0.9
#alluxio.worker.tieredstore.level1.watermark.low.ratio=0.7
```

Finally, in order to manage the deployed services and the running applications a bash script has been created. Because the Caliburn cluster uses Slurm, the first action the script takes is to setup the job to span a determinate number of nodes with 36 tasks on each.

Following next, one node is elected master from the reserved nodes. It will be used as a master node for both Alluxio and Spark, and will run the submitted applications.

Then, storage characteristics are decided. As the UnderFS of Alluxio the path $gpfsgpfsscratch/\{username\}/alluxio$ has been used while the memory tier has been setup in $devshm/\{username\}$. Thanks to this setup we achieve a large storage pool with low latency access for frequently accessed data.

Because the configuration changes from execution to execution, configuration files are adapted with unix commands to the current job setup. For instance, the master node is defined in the Spark and Alluxio configuration files.

Then, Alluxio formats the memory tier, starts its processes and gives way to Spark to start its own processes. Once all systems are up and running, the desired application is submitted to Spark to be run in the distributed cluster.

Upon completion, the script stops all the started services and finished, freeing the resources to be scheduled for another job.

Chapter 6

Experimental evaluation

To demonstrate the viability and potential of this project, a set of tests will be conducted. The deployed infrastructure will be tested with synthetic benchmarks to evaluate the resources used and to assess the performance of different storage layers. In order to study the proposed in-memory staging prototype, different strategies to update and access flows of data will be applied to offer a rich comparison.

6.1 Methodology

Because multiple factors are relevant, logging for post analysis plays a key role. For this purpose, Spark will be configured to log the application summary to be analyzed with the Spark History Server. This service allows launching a SparkUI to analyze applications independently of their status. The history logs can be synchronized to a local computer and opened with a web browser without having a Spark cluster running.

Secondly, coded tests need to print logging information time-stamped. This information gives insights on the time needed to propagate data changes on the storage layer or the frequency of the updated and other relevant aspects. This information will be analyzed in section 6.3 to extract knowledge through different metrics.

Because the main interest is to evaluate the performance of in-memory staging in a distributed architecture, tests will apply different data access strategies based

on storage layers to observe the effects on the executing programs. The scalability factor is excluded from this evaluation since this topic has been already explored and discussed by the community. For this reason, all the experiments will be conducted on the same physical resources.

The strategy chosen to assess the performance is based on a continuous processing of a distributed dataset. On one hand, an application will be responsible for updating the information at the highest rate possible. On the other hand, a second application will continuously analyze the data and extract some characteristics.

To assess the correctness of the query results, both programs should log information to enable post-verification. In concrete, it is important to corroborate that the data changes are being detected and extract the time needed to propagate the updates.

6.2 Tests design and implementation

The approach chosen to evaluate the prototype is by comparing different configurations with metrics. Firstly, a set of tests will be designed to evaluate how can updates be handled when they are performed at different storage layers.

To implement a continuous dataset, a base RDD made of Long elements will be used. New datasets will be derived from this object, according to the particularities of the test. This decision arises from the fact that is easier to verify the correctness of results and detect incoherence with numerical data. Secondly, numerical data is common in the scientific community and represents a realistic workload.

Then, another program will be responsible for extracting statistics from the dataset and log them to a file. This query needs to be I/O intensive to highlight the side effects of the storage setup.

These test will be a proof of concept to demonstrate the feasibility and the power of the idea behind this work.

6.2.1 Test setups

A set of tests has been defined which run in the proposed prototype. They are designed to demonstrate the great impact on the data analysis when using

different data storage systems. Four tests based on the same use case have been setup. Each of them features minor changes on the prototype configuration, where the case number 3 features the proposed architecture and methodology.

Case 0: GPFS based storage. In this setup, Apache Spark will store the RDD to the Network FileSystem provided by the Caliburn cluster for large datasets. The aim of this experiment is to obtain a baseline of the approach which uses a slow storage system, compared to the proposed in-memory proposal.

In this case, each update on the RDD will result in a new Spark RDD being generated and stored under a different namespace. The query will be responsible to wait until the most recent RDD is available to analyze and extract the features.

Case 1: Alluxio in-memory storage with tiered storage. In this setup, new RDDs will be forwarded to Alluxio which will be responsible to manage its memory space and evict data to the GPFS when necessary. For this purpose, the Least Recent Used algorithm (LRU) will be used.

As in Case 0, each update will generate a new RDD which will be forwarded to Alluxio under a different namespace. This approach is a realistic use case since high-density memory nodes are just emerging. As a consequence, Alluxio tends to run out of space and needs to move data to lower storage tiers.

Then, the query implemented for Spark will retrieve the data from the in-memory tier and accesses will be at memory speed. However, it is expected that the update task will be slowed down once evictions start.

Case 2: Alluxio in-memory storage. Data will be stored in-memory inside Spark and keeping the memory usage constant. This will be achieved by configuring Spark to allow overwriting data in the configuration files.

Thanks to this mechanism, it won't be necessary to evict in-memory data stored in Alluxio to other storage tiers. To ensure that the performance consists of only accessing the memory level tier, other storage tiers have been disabled.

Case 3: Alluxio in-memory staging with in-memory blocks being updated.

This scenario implements the proposed prototype with the data being continuously updated inside the Alluxio memory tier by a third program. It is expected that this setup highlights the potential of this approach and arises the required work to deploy this prototype in a production environment.

Apache Spark will be continuously querying Alluxio for the same namespace to obtain fresh data. Then the same query will be performed to extract statistical properties from the dataset.

On the other hand, another program will be running in each node to perform the updates in the staging area. These updates will be done transparently to both Alluxio and Spark, and will use at most the same physical resources as the tasks responsible to derive the RDD in the other test setups reported in this work.

To deploy and evaluate each configuration, 32 nodes will be used. The first one will be designated the master node, where the Spark Master, the Alluxio Master and the Spark driver program will run. The other 31 nodes, accounting for a total of 7.75TB of RAM space and 1.116 cores, will be used to deploy the in-memory staging area and the Spark executors that will access and update the data. Figure 6.1 illustrates the division between services configured in each node. Because of the proposed division, RDDs will be split into 16 slices for each worker node to achieve enough granularity to map each executor with a single slice.

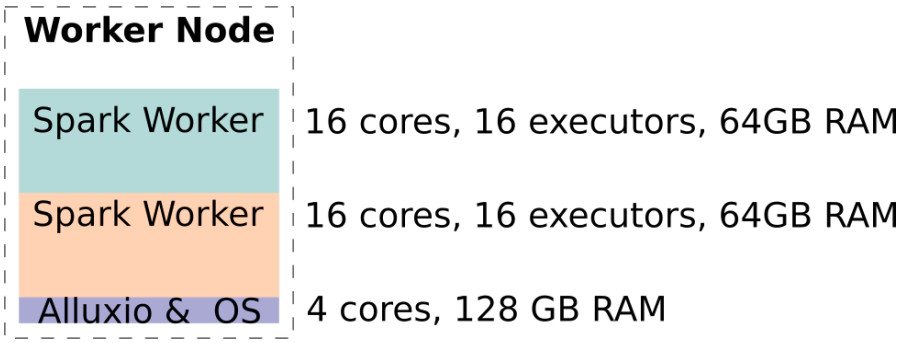

Figure 6.1: Mapping services to hardware in worker nodes.

6.2.2 Codes

To implement the query, a simplistic program in Scala has been written to query the data under a particular namespace, let it be Alluxio cached files or regular files on the GPFS.

First, an initial version 6.2 was written to select a portion of the dataset and extract characteristics. It was done by reordering the distributed dataset and analyzing the top 1% higher elements. This approach resulted in a computational expensive query due to the complete shuffling of the dataset.

```
def storeMean(ss: SparkSession, filepath: String): Unit = {
  val sc = ss.sparkContext
  // Read input dataset
  var largeArray = sc.textFile(filepath).map((a) => a.toInt)
  // Get the percentile
  var sortedArray = largeArray.sortBy((x) => x, ascending = false)
  val size = sortedArray.count()
  var tenth_percent = sortedArray.take((size * 0.1).toInt)
  // Compute the average
  var avg = tenth_percent.sum / tenth_percent.length
  // Create the context with a 1 second batch size
  println("Average:", avg)
}
```

Figure 6.2: Initial query code.

Also, the initial version was based on generating a List and parallelizing the object to obtain an RDD. Nevertheless, this mechanism did not scale in the cluster, since the List is generated in the driver program and then forwarded to the workers, becoming the master node a bottleneck because of the driver program.

val largeArray = sc.parallelize(List.fill(N)(value),slices)

Table 6.1: Parallelization of the initial data structure.

Then, a simplified query applying the method $RDD[Long].stats()$ was implemented and tested successfully, with short execution times bounded by storage accesses. Since the purpose of this tests was to evaluate I/O bound scenarios, the query illustrated in Figure 6.3 query was made final. Nevertheless, a minor modification was applied to adapt the query to test cases not reusing the same

namespace in each iteration. In these situations, the iteration id was added to the end of the namespace both in the query and the program performing the updates.

It is also important to notice that the try-catch enveloping the data access are unnecessary for the test case 3. In this configuration Alluxio always has the data available and updates have no effect on data availability. As a consequence, the query for the Case 3 could be reduced to just accessing the RDD in Alluxio at each iteration and extracting the stats.

```
for (a <- 1 to niter) {
  var done = false
  println("Iteration " + a.toString)
  while (!done) {
    try {
      var largeArray = sc.textFile(filepath).map((v) => v.toLong)
      val stats = largeArray.stats()
      val time = DateTime.now().toString( pattern = "yyyy-MM-dd HH:mm:ss.SSS")
      println(stats.toString() + "," + time)
      done = true
    }
    catch {
      case ioe: IOException => Thread.sleep(300)
    }
  }
}
```

Figure 6.3: Fragment of the query used.

The printed results consist of statistical measurements and a timestamp, written to a file. Figure 6.2 features a simplified version of the captured output.

(count: 899999999, mean: 815189145.93, stdev: 61975544.61,...), 12:48:46.266
(count: 899999999, mean: 877943486.12, stdev: 62113695.39,...), 12:48:48.299
(count: 899999999, mean: 937784556.33, stdev: 57787949.33,...), 12:48:50.489
(count: 899999999, mean: 977382943.85, stdev: 40803947.57,...), 12:48:52.638
(count: 899999999, mean: 992378753.20, stdev: 24666262.19,...), 12:48:54.754
(count: 899999999, mean: 996632734.41, stdev: 13273459.60,...), 12:48:56.861

Table 6.2: Query output example simplified.

Since Alluxio holds the metadata in its process memory, such as the size of each block, the initial RDD and its updated versions need to match the same block size for the Test Case Number 3 to succeed. With this purpose, the initial RDD and its subsequent version have their values in the range [base, base*10-1].

By updating elements according to this schema, the Alluxio in-memory blocks shape is maintained and subsequent reads will succeed. Therefore, all the tests were conducted using an RDD of the same size which is kept during the whole execution.

To avoid the penalty of initializing and finalizing program drivers, both the query and the update programs loop over their operations a user-defined number of times. In this way increases the query and data update frequency.

Finally, to update the data in the Alluxio in-memory staging area, a Bash code has been written. Runs on each node through the queue launching command, in this case, *srun*. Works by iterating over the data blocks, and launching a Python script on each block as sub-processes. Then waits until all the launched processes return to proceed with the next update.

This mechanism allows updating the in-memory data in parallel while doing it step by step. This behavior is important to avoid data races and to be able to confirm the query results for a given timestep.

In its turn, the Python script takes an initial value, the current iteration, the total iterations and the path to the data block. Then updates its contents by computing a Normal distribution where the mean is based on the sinusoidal function as shown in Figure 6.4. This decision was made to obtain a realistic variation of data in a distributed environment analyzing with scientific workloads.

```
distr = np.random.normal(np.sin(2 * np.pi * step), sigma, num_elem)
distr = base + (distr * max_range).astype(int)
distr[distr > max_range] = max_range
distr[distr < base] = base
with open(filepath + '.tmp', 'w') as f:
    f.write('\n'.join(map(str, distr)))

os.rename(filepath + '.tmp', filepath)
now = datetime.datetime.now()
status = (socket.gethostname(), len(distr), s.mean(), s.std(), s.max(), s.min(), str(now))
print("{}:(count: {}, mean: {}, stdev: {}, max: {}, min: {}),{}".format(*status))
```

Figure 6.4: Fragment of the Python block updates.

All these scripts and programs are managed by a job script written in Bash for Slurm. The script executes the following steps to deploy and evaluate applications:

1. **Configure nodes.** Picks a master node from the allocated nodes and leave the rest as workers.

2. **Update configuration.** Configuration files in Spark and Alluxio folders are adapted to the test. In concrete, the paths are updated accordingly and information on which nodes will be used to start the services is updated.

3. **Format Alluxio's memory tier.** To avoid coherence issues with old data, all nodes format the memory tier before starting up.

4. **Launch services.** The following methods are used to deploy Alluxio and Spark inside the job script:

```
$ALLUXIO_HOME/bin/alluxio-start.sh all NoMount &


$SPARK_HOME/sbin/start-master.sh
sleep 15
$SPARK_HOME/sbin/start-slaves.sh spark://$MASTER_NODE:7077
```

5. **Configure and submit applications:** For instance:

```
$SPARK_HOME/bin/spark-submit
--total-executor-cores $((16*$NWORKERS))
--class ArrayGeneration
--master spark://$MASTER_NODE_IP:7077
--driver-memory 4G $HOME/usecase.jar
"alluxio://$MASTER_NODE_IP:19998/myArray" $VAL0 $SLICES



srun --ntasks-per-node=1 --cpus-per-task=4
-n $NWORKERS --nodelist=‘echo ${WORKER_NODES// /,}‘
sh $HOME/NodeTest.sh $TMPFS/alluxioworker/ $NITER $VAL0
>> $GPFS/python_updates &
```

6. **Stop all services and updates** Alluxio and Spark services are stopped through their scripts. If the Python update application is running, it is stop with a simple kill command run in each node.

6.3 Results

In this section, the evaluation results are reported and discussed. For each configuration in subsection 6.2.1 a set of metrics will be gathered and analyzed.

The most important metrics are related to the time needed for an update or query to complete, the frequency in which they succeed, and most importantly, the time passed until an element is analyzed. The latter can be seen as the time an element is present in the system.

Finally, metrics regarding memory and storage will be monitored and discussed thanks to the Spark history server and logging information in Alluxio.

In all configurations, the generated data is split into 16 slices for each worker, to match the number of executors per worker. Also, since on each node both the query and the updating applications will access the same data, this means that the scheduler should be able to divide the executors equally between the two applications, and each application will have the same number of executors as slices.

At its turn, each stage is divided into 496 (16*31) tasks spread to 31 working nodes. The query stages take as input 8.4GB of data without generating an output. Instead, the update application takes 8.4GB of input data and generates 8.4GB.

6.3.1 Case 0: GPFS based storage

As already mentioned, the first setup consists of Apache Spark running distributed and using the available GPFS to read and store data. The program responsible for generating data accepts a value and a namespace. The first step consists in creating an RDD of Longs set to the given value and saved as text to the given namespace. In successive steps, the program iterates and transforms the RDD, which is stored under a different namespace generated from the original.

With this approach, also the program performing the query follows the same pattern to access the next RDD to evaluate. However, since there is a high chance that the query finishes evaluating an RDD before the next one is available, checks that the new RDD exists before accessing it. This situation occurs because reading data from GPFS is faster than generating the distributed the dataset and writing it to the GPFS.

Figure 6.5 reports the period between creations of new RDDs. Only 9 mea-

surements were possible since the quota limit of 100GB was reached. The mean time is high for a creation of a file, but the measurements seem stable with little variance.

count	9
mean	1min 17.95s
std	06.85s
min	1min 07.43s
25%	1min 14.08s
50%	1min 19.38s
75%	1min 21.62s
max	1min 28.40s

Figure 6.5: Time needed to update an RDD in GPFS.

In its turn, the Figure 6.6 reports the time needed for the query to process an update after the data is generated. The variance is small in this situation as observed when creating the dataset.

count	10
mean	29.70s
std	2.65s
min	26.56s
25%	27.45s
50%	29.20s
75%	31.52s
max	34.55s

Figure 6.6: Elapsed time since an element is updated until evaluation in GPFS.

By analyzing the data reported by the Spark history server, the data update job was active for 26 minutes, while the query only 6.5 minutes. Meaning that the query spent roughly 20 minutes waiting for the data to be available in the driver program, while 6.5 minutes were dedicated to data retrieval and process. Therefore, on average, it took 35.5 seconds for the query to execute on iteration.

6.3.2 Case 1: Alluxio In-memory Storage with Tiered Storage

The first experiment to include an in-memory storage layer. Commonly, Alluxio is used in a tiered storage setup to offload the in-memory data when it becomes full. In the following experiments, two setups have been explored, using the GPFS as a secondary tier or the fast NVMe attached to each node.

Because reaching the maximum capacity configured for the Alluxio memory tier will take more than one hour worth of data processing, its capacity has been decreased to 8GB per node. This is a frequent configuration among clusters not provisioned with high-density memory.

Apache Spark has been configured to not overwrite data, which is the default behavior, and new RDDs are stored as new files. This approach was taken to leave the query unaffected by the locking mechanism in Alluxio when blocks are overwritten.

6.3.2.1 GPFS as a secondary tier

In this test, the same GPFS directory used in the Case 0 has been designated as a secondary storage tier. To make sure Alluxio stays inside the quota limit of the GPFS (100GB), it has been configured to allocate a maximum of 64GB of the GPFS.

The time it takes for the query to analyze the updated data is reported in Figure 6.7. The data access and processing time stay within a similar range during the whole execution.

count	35
mean	2.20s
std	0.28s
min	1.71s
25%	2.05s
50%	2.07s
75%	2.22s
max	2.78s

Figure 6.7: Mean time to propagate an update.

However, by observing the time individually we can observe the effects of the eviction. Figure 6.8 illustrates the evolution of the processing time needed after each update. It is clear that evictions have a consequence in performance. However, since the recently updated data stays always in the memory tier, the query is affected by increasing a 34% the time it takes for an update to be detected. However, since data accesses are performed at memory speed, this increase means a delay of 700ms over the 2 seconds it took on average before evictions started.

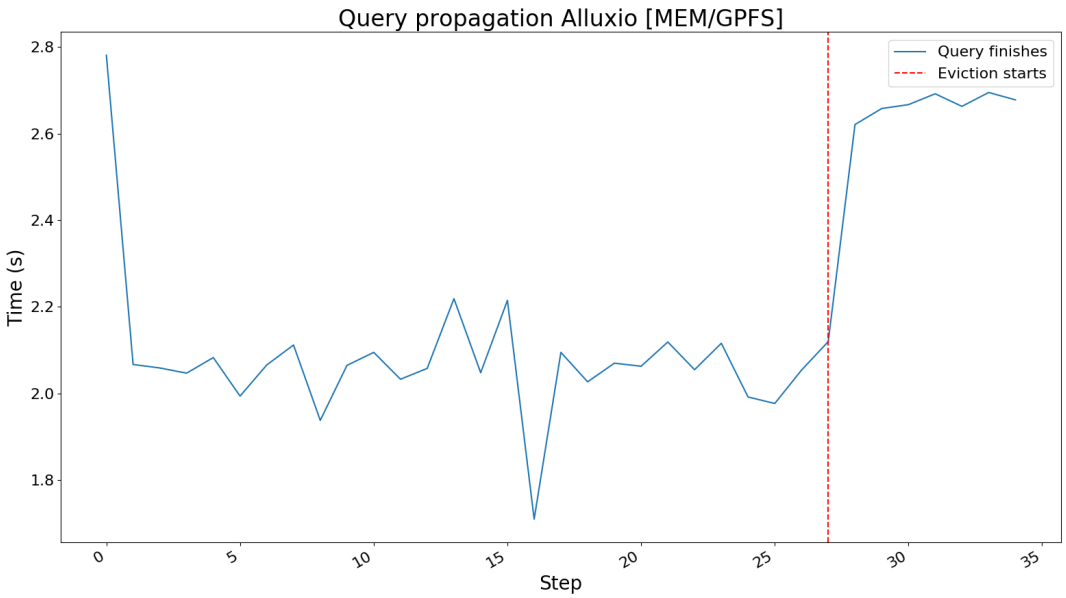

Figure 6.8: Time for an updated RDD to be processed by the query.

On the other hand, the execution time of the updates is greatly affected as shown in 6.9. When evictions start the time to store the new RDD is increased notably as Alluxio moves the least recently accessed data to the GPFS. The increase is smoothed by the fact that the eviction was configured with watermarks in the range of [0.7,0.9]. Meaning that when the capacity of Alluxio is over 70% evictions start until the 90% is reached when Alluxio blocks successive writes.

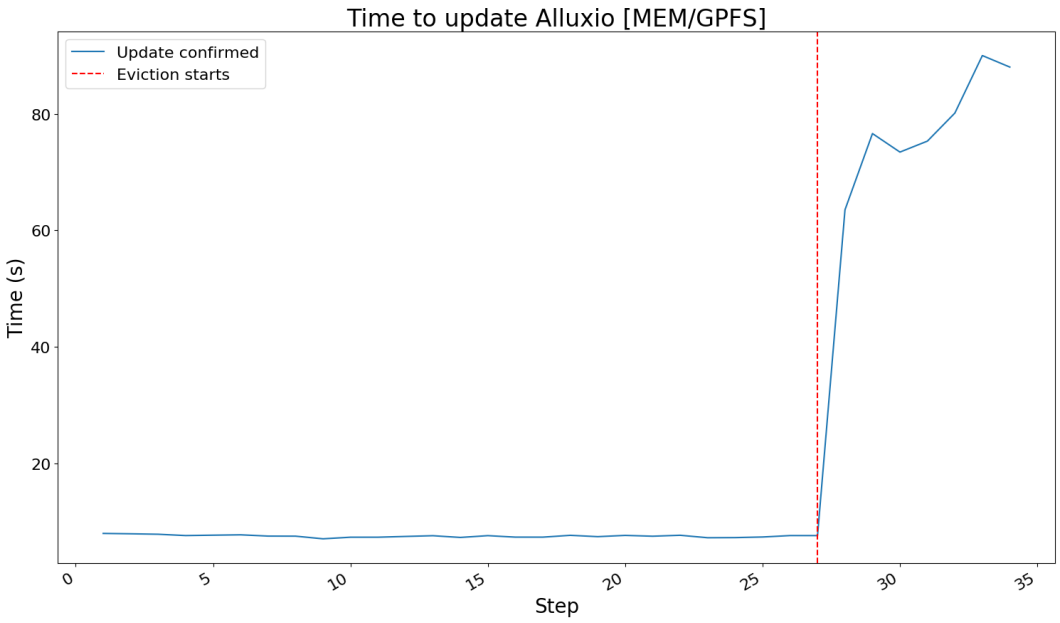

Figure 6.9: Time to write a new RDD into Alluxio, backed by GPFS.

The elapsed time between the first and last query executed is 768 seconds. During this period, a total of 34 queries finished, resulting in a throughput of 0.0455 queries/second.

6.3.3 NVMe as a secondary tier

The situation is completely different when using a secondary tier based on fast storage such as a NVMe. The eviction takes place at the same moment as in the previous experiment but the effect on the updates elapsed time is almost imperceptible as shown in 6.10. The time oscillates slightly during the test but when the eviction takes places the tendency is to stay stable at a similar value.

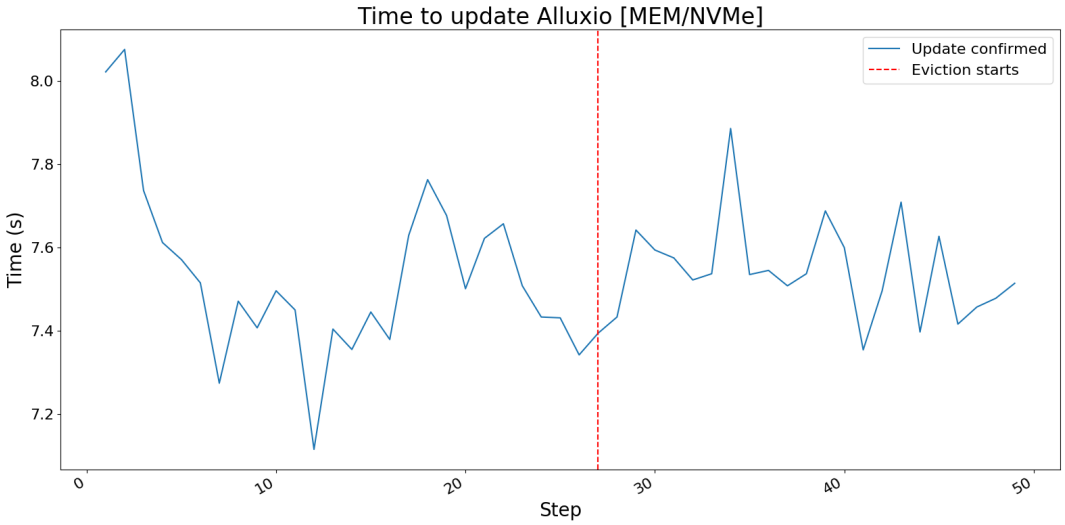

Figure 6.10: Time to write a new RDD into Alluxio, backed by NVMe storage.

The time between the first query result and the last is 368 seconds. In-between 49 queries were performed, resulting in a throughput of 0.1332 queries/second being accomplished. The time dedicated to performing the query is similar to the previous experiment but with smaller variability since Alluxio alteration by evicting blocks to the NVMe disk has been minimal compared to GPFS. Figure 6.11 shows the mean time needed to evaluate an updated RDD. It can be seen that before evictions the time stays in a range. However, after evictions one step took longer than the average due to Alluxio being slowed down by moving data.

When comparing the results with the experiment using the GPFS as secondary storage, there is a noticeable difference but still processing times stay in a low range between 2 and 3 seconds. Compared with the previous experiment where only GPFS was used, these results are comforting if considering that the mean time without Alluxio would be 29.70s compared to 2.6s or 2.1s obtained when using GPFS and NVMe as secondary tiers respectively.

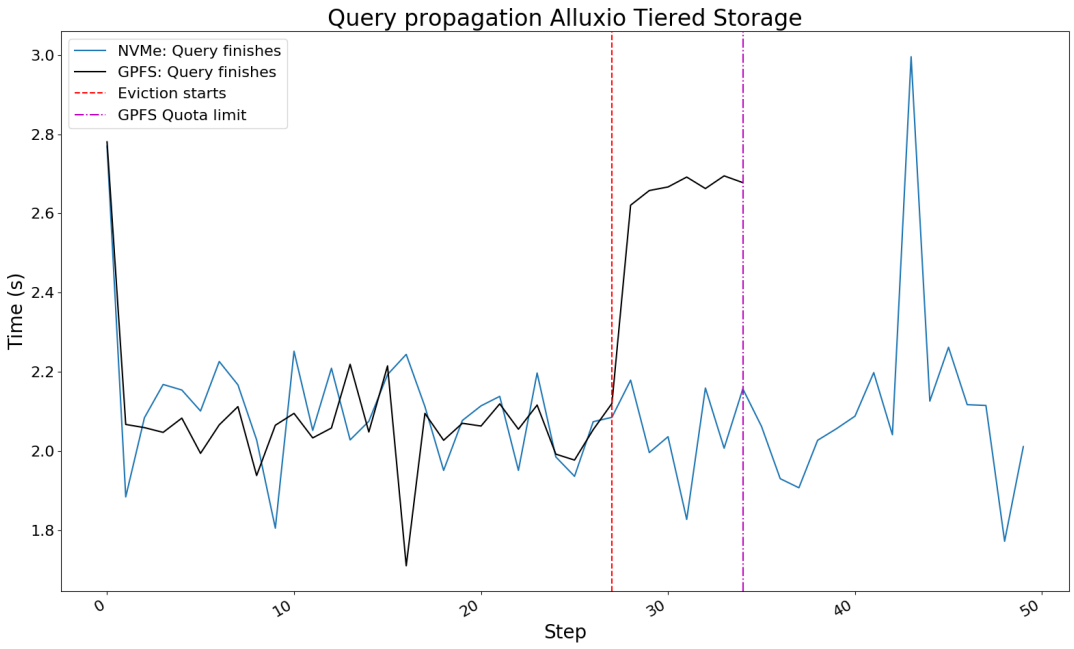

Figure 6.11: Time for the query to evaluate an update.

6.3.4 Case 2: Alluxio in-memory storage

After observing the impact of tiered storage using different technologies, and the slowdown they suppose on I/O intensive workloads, Alluxio has been setup to provide a single layer storage based on RAM memory.

This experiment is designed to obtain a baseline of the performance obtained in the best conditions, where no evictions occur and data is always read and write from and to the memory tier.

For this test, the updated RDD are saved as new files with new namespaces, as in previous experiments. Alluxio capacity has been left intact to 128GB, which provides enough space for the execution to run for nearly one hour.

The average time for an update to be noticed from Spark is reported by Figure 6.12 measured as the time elapsed since the update confirmed the write into the Alluxio staging area until the query logged the updated result.

count	50
mean	2.1s
std	0.25s
min	1.72s
25%	1.98s
50%	2.06s
75%	2.14s
max	2.92s

Figure 6.12: Mean time to propagate an update.

The Alluxio in-memory tier storage fills up during 6 minutes and 6 seconds at a rate of 1.15 GB/s, until reaching 420GB of distributed data. Because the application runs in 32 nodes, one of them being the master, in the end, each one holds an average of 13.54GB.

On the other hand, Spark processes a total of 930.1GB as reported by the History server. In average, each executor takes 1.9GB of input to the process, meaning that the limit of 4GB of RAM per executor defined is adequate.

Because data is stored as new files, the memory usage grows quickly. Alluxio has been setup to manage 128GB of RAM per node, meaning that after an hour of the execution, the Alluxio memory tier will be full and evictions triggered.

However, the query is performed at intervals of more than 7 seconds as shown in Figure 6.13, which is three times the time it takes to execute as seen in Figure 6.12.

count	50
mean	7.50s
std	0.47s
min	6.47s
25%	7.25s
50%	7.50s
75%	7.72s
max	9.42s

Figure 6.13: Time between queries.

And observing the frequency by which the data is successfully updated in Figure 6.14 we can observe a correlation between the two frequencies.

count	49
mean	7.47s
std	0.19s
min	7.21s
25%	7.34s
50%	7.44s
75%	7.58s
max	8.07s

Figure 6.14: Time between updates complete.

6.3.5 Case 3: Alluxio in-memory staging with in-memory blocks being updated

In this experiment, we feature the proposed architecture and evaluate the effects of updating the data in the in-memory staging area, instead of creating new objects.

For this purpose, the data generation program has been replaced by the Python script reported in 6.2.2. The data is being continuously updated in the staging area by this external program while Apache Spark continuously obtains statistics from the data. A schematic organization can be observed in Figure **??**.

This mechanism provides continuous updates with a high frequency and smaller granularity. Updates are performed at the Alluxio block level, corresponding to an RDD slice. With this setup, 34388 updates are performed in a period of time of 109.53s. This leaves an update frequency of 3ms, which compared to the Spark job is increased by a factor of 2345x. However, if we compare the time needed for an RDD to be completely updated, we obtain that every 1.58s in average all the RDD slices are updated, which is slightly shorter than the frequency obtained with Spark in previous experiments.

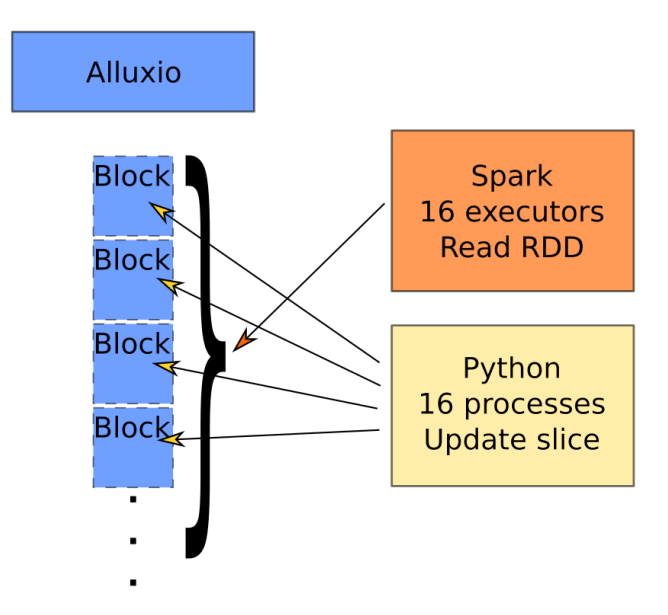

Figure 6.15: Node organization used to evaluate the proposal.

Figure 6.16 reports the mean value of the distributed dataset during the execution. It is confronted with the mean value reported by the Spark query, which graphically shows that the time it takes for the query to compute the mean value of the dataset once it has been updated stays in the range of 2 to 5 seconds. It is also relevant that despite the high update frequency the query is able to follow the data changes and detect inflections, minimums, and maximums.

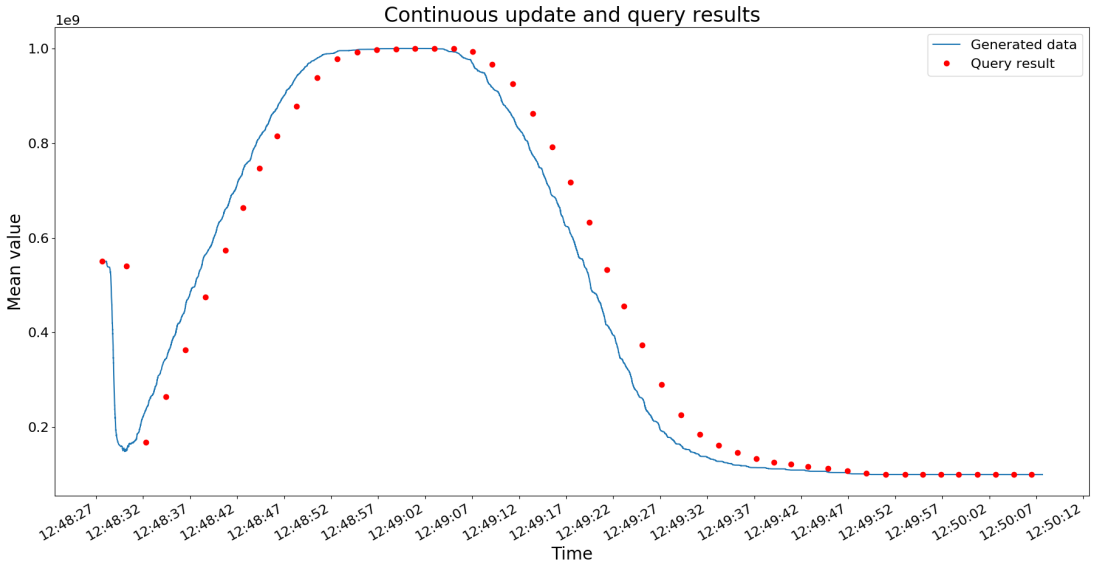

Figure 6.16: Evolution of externally updated data analyzed from Spark.

To compute the time it takes for the prototype to detect an update, the period between [12:48:28,12:49:22] illustrated in the figure 6.16 has been used. Before the timestamp 12:48:28 there were blocks of data who hadn't been updated yet, and thus, the preceding period is considered a warming up phase.

On the other hand, the query results reported after 12:49:22 are influenced by nodes with at least one block of data no longer updated. Consequently, the time after this timestep has also been discarded to compute the elapsed time between an updated until it is processed by the query.

These measurements are reported in 6.17, where the number of queries performed under these circumstances is reduced to 26 from the 50 original measurements. It is clear that the time needed to propagate an update to the query is similar to the previous experiments, where data access were also performed at memory speed.

count	25
mean	2.12s
std	0.37s
min	1.59s
25%	1.81s
50%	2.09s
75%	2.37s
max	2.99s

Figure 6.17: Time for the prototype to detect and process and update.

However, when we analyze the frequency at which queries are executed, shown in Figure 6.18, we find that the time has been notably reduced to 2 seconds compared to other configurations. The explanation resides in that Alluxio doesn't need to allocate space for new blocks and handle the addition of new files. As a consequence, the capacity of Alluxio stays constant at 99.79%, with 8.4GB out of 3.86TB of memory space used. Because in this scenario the query is independent of the updates, all 50 measurements have been used.

count	49
mean	2.02s
std	0.13s
min	1.74s
25%	1.92s
50%	2.03s
75%	2.09s
max	2.57s

Figure 6.18: Time between query results.

6.4 Experimentation summary

In this section results from the different setups are summarized and discussed. Figure 6.19 reports the most relevant aspects of each configuration.

Configuration	Mean time for an update to be evaluated	Alluxio memory usage trend	Throughput (Frequency of the query execution)
GPFS Storage	29.70 ± 2.65s	NA	0.0336 results/s Stable.
Alluxio Tiered Storage **Level 0: MEM** **Level 1: GPFS**	5.39 ± 0.21s	Grows linearly with the size of the input (4GB/s)	Before evictions: Stable at 0.1325 results/s After evictions: Tends to 0.0464 results/s
Alluxio Tiered Storage **Level 0: MEM** **Level 1: NVMe**	2.10 ± 0.20s	Grows linearly with the size of the input (4GB/s)	0.1331 results/s Stable.
Alluxio In-memory Storage	2.11 ± 0.25s	Grows linearly with the size of the input (4GB/s)	0.1334 results/s Stable.
Alluxio In-memory Staging with block updates	2.12 ± 0.37s	Constant with the size of the dataset (8.4 GB)	0.4717 results/s Stable.

Figure 6.19: Summary of the experimentation results.

Firstly, the time elapsed since an RDD is updated until is analyzed by the query is kept constant in situations where the data is read at memory speed, and Alluxio is not being slowed down by a slow secondary tier. This is, using the GPFS in any configuration results in slower data accesses due to the slower throughput or Alluxio being saturated by evicting data to a GPFS secondary tier.

The second relevant characteristic is the memory usage. Only when updates were performed to the Alluxio data blocks the memory usage was kept constant. In all the other configurations growth linearly at 4GB/s, which is unsustainable for long running executions unless a garbage collection mechanism is configured.

Finally, one of the most relevant findings is the number of queries performed successfully. Tests representing current deployments obtained a throughput of 0.1339 queries/second at best. However, the proposed prototype was able to deliver 0.4717 queries/second, representing a speedup of 10x compared to storing data into GPFS or 3x to storing and reading data from the Alluxio memory tier.

The comparison between the different configurations tested during the experimentation is illustrated in Figure 6.20. For each configuration, the time between each query has been computed and displayed. It clearly shows the difference in

performance when using different storage technologies and strategies.

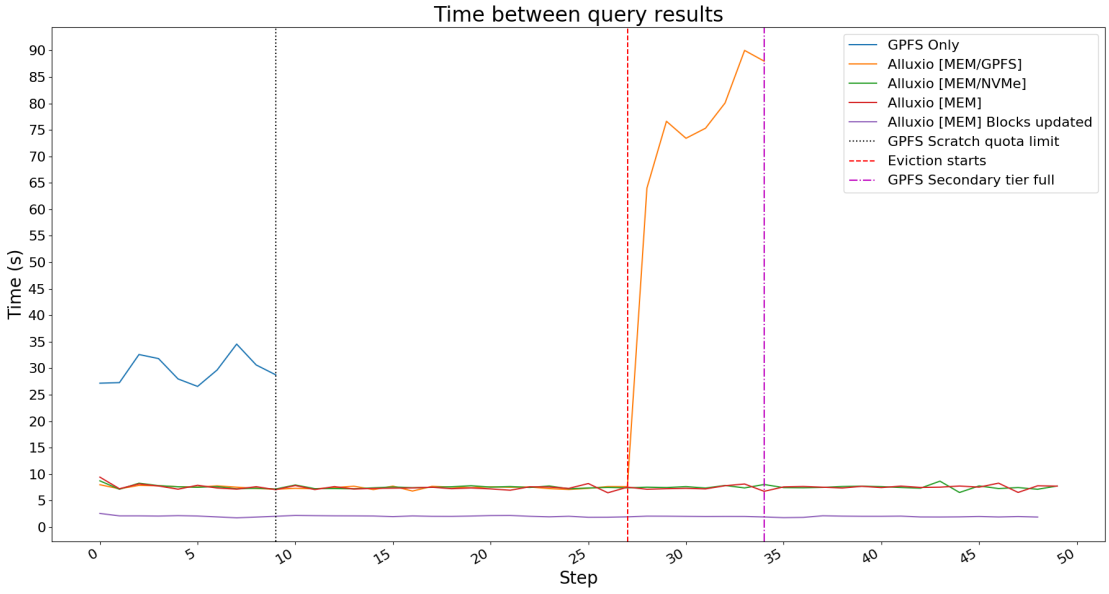

Figure 6.20: Frequency of the query execution for the different configurations.

Chapter 7

Conclusions

The realization of this work has served to evaluate the potential of introducing an in-memory data staging area for Big data. This field is populated with a wide variety of projects, and selecting the best components for a Big data infrastructure is not trivial.

The survey on current Big data projects highlighted a shift to stream computation in the recent years. However, it also revealed the pending developments to integrate stream processing with current in-memory staging frameworks.

To evaluate an in-memory staging area with frequent updates a prototype has been proposed. The objective was to test a prototype that with minor modifications should deliver massive performance to the in-memory staging area provided by Alluxio.

To demonstrate the viability of the proposal a reduced prototype has been tested and confronted to current deployments. It featured Apache Spark as the processing engine and an in-memory staging area provided by Alluxio.

Experimentation showed the prototype obtained a speedup of 10x compared to designating GPFS as the storage layer for Spark. But most importantly, accessing the same object stored in Alluxio, but updated externally provided a 3x speedup compared to reading and writing a new object in each step.

From a numerical perspective, the prototype was able to deliver a throughput of 0.4717 queries/second compared to 0.1339 queries/second with the current deployments of Apache Spark and Alluxio.

In part, the difference in performance is explained by different factors. First,

by updating the data blocks there was no need to allocate new space. Secondly, since the same object was reused, there was no need to register new objects. And finally, when accessing an object that is being written, the reader is blocked until the object is completely written by all workers. However, by updating the blocks individually Alluxio was accessing always the most up-to-date information stored without applying locking mechanisms.

Lastly, by updating the in-memory data instead of creating new objects, the prototype was able to keep the Alluxio capacity stable, while the other deployments failed after an amount of time when the disk or memory space available were exceeded or suffered the penalty of having to evict data to secondary storage layers.

Chapter 8

Future research

This project verified the benefits of implementing an in-memory staging area based on Alluxio. However, to accomplish the objective data updates have been applied by an external program to the in-memory blocks handled by Alluxio.

Therefore, it will be desirable to implement this mechanism as part of Alluxio. The benefits obtained will consist of enabling updates on datasets that alter the block size, a restriction identified in this project, that limits the number of elements and their size during updates.

Secondly, to develop this project Apache Spark RDDs have been used. Future research on supporting Spark datasets and dataframes will contribute on integration Apache Spark Streams with Alluxio efficiently and with a reduced memory usage. With this purpose, implementing a custom connector for Alluxio and Spark that takes advantage of the Alluxio future updates will be necessary.

Finally, this project has been evaluated in a cluster by allocating 32 high-density memory nodes. Spark and Alluxio were responsible for handling a constant flow of 4GB/s wort of data representing nearly 450M records/s. Further testing with workloads that increased the required space and the transfer could reveal relevant performance trends.

Bibliography

[1] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, OSDI '06, pages 15–15, Berkeley, CA, USA, 2006. USENIX Association.

[2] S. Community. Structured streaming programming guide: Continuous processing, 2018.

[3] S. Community. Trident state documentation, 2018.

[4] Crucial. Why cas latency isnt an accurate measure of memory performance. Technical report, Micron Technology, Inc, 2015.

[5] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[6] J. Dean and P. Norvig. Latency numbers every programmer should know, 2012.

[7] A. O. Foundation. Running spark on alluxio - docs, 2018.

[8] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. *SIGOPS Oper. Syst. Rev.*, 37(5):29–43, Oct. 2003.

[9] A. Grishchenko. Spark memory management, 2016.

[10] R. D. I. Institute. Caliburn — rdi2, 2016.

[11] C. Lawrie. Evaluation of the suitability of alluxio for hadoop processing frameworks, 2016.

[12] J. Liu, J. Wu, and D. K. Panda. High performance rdma-based mpi implementation over infiniband. *Int. J. Parallel Program.*, 32(3):167–198, June 2004.

[13] R. Prabhakar, S. S. Vazhkudai, Y. Kim, A. R. Butt, M. Li, and M. Kandemir. Provisioning a multi-tiered data staging area for extreme-scale machines. In *2011 31st International Conference on Distributed Computing Systems*, pages 1–12, June 2011.

[14] Seagate. Nvme - performance for the ssd age. Technical report, Seagate Technology, Cupertino, CA, USA, 2016.

[15] S. Sur, M. J. Koop, L. Chai, and D. K. Panda. Performance analysis and evaluation of mellanox connectx infiniband architecture with multi-core platforms. In *15th Annual IEEE Symposium on High-Performance Interconnects (HOTI 2007)*, pages 125–134, Aug 2007.

[16] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.

[17] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.