Doctoral Thesis

A Scalable Data Store and Analytic Platform for Real-Time Monitoring of Data-Intensive Scientific Infrastructure

A thesis submitted to Brunel University London
in accordance with the requirements
for award of the degree of Doctor of Philosophy

By

Uthayanath Suthakar

in

the Department of Electronic and Computer Engineering
College of Engineering, Design and Physical Sciences

November 23, 2017

Declaration of Authorship

I, Uthayanath Suthakar, declare that the work in this dissertation was carried out in accordance with the requirements of the University's Regulations and Code of Practice for Research Degree Programmes and that it has not been submitted for any other academic award. Except where indicated by specific reference in the text, the work is the candidate's own work. Work done in collaboration with, or with the assistance of, others, is indicated as such. Any views expressed in the dissertation are those of the author.

SIGNED: ..... DATE: .....

(Signature of student)

"Progress is made by trial and failure; the failures are generally a hundred times more numerous than the successes; yet they are usually left unchronicled."

Sir William Ramsay, Chemist

Abstract

Monitoring data-intensive scientific infrastructures in real-time such as jobs, data transfers, and hardware failures is vital for efficient operation. Due to the high volume and velocity of events that are produced, traditional methods are no longer optimal. Several techniques, as well as enabling architectures, are available to support the Big Data issue. In this respect, this thesis complements existing survey work by contributing an extensive literature review of both traditional and emerging Big Data architecture.

Scalability, low-latency, fault-tolerance, and intelligence are key challenges of the traditional architecture. However, Big Data technologies and approaches have become increasingly popular for use cases that demand the use of scalable, data intensive processing (parallel), and fault-tolerance (data replication) and support for low-latency computations. In the context of a scalable data store and analytics platform for monitoring data-intensive scientific infrastructure, Lambda Architecture was adapted and evaluated on the Worldwide LHC Computing Grid, which has been proven effective. This is especially true for computationally and data-intensive use cases.

In this thesis, an efficient strategy for the collection and storage of large volumes of data for computation is presented. By moving the transformation logic out from the data pipeline and moving to analytics layers, it simplifies the architecture and overall process. Time utilised is reduced, untampered raw data are kept at storage level for fault-tolerance, and the required transformation can be done when needed.

An optimised Lambda Architecture (OLA), which involved modelling an efficient way of joining batch layer and streaming layer with minimum code duplications in order to support scalability, low-latency, and fault-tolerance is presented. A few models were evaluated; pure streaming layer, pure batch layer and the combination of both batch and streaming layers. Experimental results demonstrate that OLA performed better than the traditional architecture as well the Lambda Architecture. The OLA was also enhanced

by adding an intelligence layer for predicting data access pattern. The intelligence layer actively adapts and updates the model built by the batch layer, which eliminates the re-training time while providing a high level of accuracy using the Deep Learning technique.

The fundamental contribution to knowledge is a scalable, low-latency, fault-tolerant, intelligent, and heterogeneous-based architecture for monitoring a data-intensive scientific infrastructure, that can benefit from Big Data,technologies and approaches.

Acknowledgements

I would like to express my sincere gratitude to *Dr. David Ryan Smith*, who supervised my research and guided me with great dedication throughout my time as a PhD student. His view of research has proved valuable to my research and supporting work. His personality, together with his high standards for research, makes him an ideal advisor. I would like to thank *Prof. Akram Khan* for introducing me to the support for Distributed Computing group, at the CERN IT department, and for providing me support and advice throughout my time as a PhD student.

I would like to express a special thanks to *Dr. Luca Magnoni* for supervising me from CERN and for sharing his extensive knowledge with me while conducting this research. I am thankful to all the members of Support for Distributed Computing group, at the CERN IT department for guiding me and supporting me while I was on site.

I am appreciative to the *Thomas Gerald Gray Trust* for funding and supporting my research.

I would like to thank all my family members for supporting me while conducting this research. Without them, I could not have reached this stage in my life. Finally, I have made a few exceptional friends at Brunel University London, *Nikki Berry*, *Rhys Gardener*, *Hassan Ahmad*, *Asim Jan*, and *Sema Zahid*. I would like to thank them for making my experience enjoyable while conducting this research.

List of Publications

The following papers have been accepted / to be submitted for publication as a direct or indirect result of the research discussed in this thesis.

Suthakar, U., Magnoni, L., Smith, D.R., Khan, A., Andreeva, J., An efficient strategy for the collection and storage of large volumes of data for computation, *Springer: Journal of Big Data*, (2016) [This paper corresponds to Chapter 3].

Magnoni, L., **Suthakar, U.**, Cordeiro, C., Georgiou, M., Andreeva, J., Khan, A., Smith, D.R., Monitoring WLCG with lambda-architecture: a new scalable data store and analytics platform for monitoring at petabyte scale. *Journal of Physics: Conference Series* 664(5), 052023 (2015) [This paper corresponds to Chapter 4].

Suthakar, U., Magnoni, L., Smith, D.R., Khan, A., Optimised lambda-architecture for monitoring WLCG using Spark and Spark Streaming, *IEEE Nuclear Science Symposium*, (2016) [This paper corresponds to Chapter 5].

Suthakar, U., Magnoni, L., Smith, D.R., Khan, A., Optimised Lambda Architecture for monitoring scientific infrastructure, *IEEE Transactions on Parallel and Distributed Systems*, (2017) [This paper corresponds to Chapter 5].

Contents

List of Figures

List of Tables

List of Abbreviations

AAA	Anytime, Anywhere CMS storage systems using XRootD protocol
AI	Artificial Intelligence
CEP	Complex Event Processing
CERN	Conseil Europen pour la Recherche Nuclaire
CPU	Central Processing Unit
CSV	Comma-Separated Value
DAG	Directed Acyclic Graph
DAP	Data Access Pattern
DDM	ATLAS Distributed Data Management
DL	Deep Learning
DT	Decision Tree
DQ2	Distributed Data Management using Don Quijote
EC2	Amazon Elastic Compute Cloud
ED	Experiment Dashboard
EPL	Event Processing Language
ETL	Extraction, Transformation and Loading
FAX	Federated ATLAS storage systems using XRootD protocol
FTS	File Transfer Service
GB	Gigabyte
GBM	Gradient Boosting Model
GC	Garbage Collection
GFS	Google File System
GUI	Graphical User Interface
H2O	Sparkling water Machine Learning library

HDFS	Hadoop Distributed File System
HTTP	Hypertext Transfer Protocol
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
KNN	K-Nearest Neighbors
LA	Lambda Architecture
LHC	Large Hadron Collider
LR	Linear Regression
MB	Megabyte
ML	Machine Learning
MLLIB	Apache Spark Machine Learning library
OLA	Optimised Lambda Architecture
OOM	Out of Memory
PB	Petabyte
PD2P	PanDA Dynamic Data Placement
PL/SQL	Procedural Language/Structured Query Language
POJO	Plain Old Java Object
R^2	R-Squared
RDBMS	Relational Database Management System
RDD	Resilient Distributed Dataset
RDDM	ATLAS Rucio Distributed Data Management
RF	Random Forests
RMSE	Root-Mean-Square Error
UI	User Interface
WDT	WLCG Data acTivities
WLCG	Worldwide LHC Computing Grid
YARN	Yet Another Resource Negotiator

Chapter 1

Introduction

Life-changing scientific discoveries are vital for human existence. Try to imagine life without antibiotics. People would not live nearly as long without them. So, these findings are essential. A few decades ago the speed of scientific discoveries was limited by technologies; an example would be computing power. Fortunately, things have changed in recent years; in particular, recent developments in the field of computing have led to the creation of a platform to support and speed up scientific discoveries. An example of such a scientific infrastructure is located at CERN, which is one of the largest in the world where physicists are trying to understand how the universe evolved. In order to obtain an answer to this question, they are using the Large Hadron Collider (LHC) to accelerate and steer billions of proton-proton collisions. The Worldwide LHC Computing Grid (WLCG) provides the computing resources to store, distribute and analyse the ∼30 Petabytes of data generated annually by the LHC, serving a community of more than 3,000 physicists distributed among 170 computing centres around the world [14]. Every day, thousands of files are transferred, and hundreds of thousands of processing jobs are executed at the different computing sites to perform scientific analysis of LHC data. This is the case for many other scientific infrastructures such as the Earthscope and the Fermilab experiments to name a few. Monitoring such a distributed, geographically sparse and data-intensive infrastructure is a core functionality and one of the greatest challenges to providing a reliable scientific platform.

Scientific infrastructure has to cope with the increases of the volume, velocity and the

variety of the data being monitored; characteristics of what does become called Big Data. Although Big Data has emerged rapidly there is still some uncertainty about the term as is evident from the result of an online survey [1]. Figure 1.1 shows the executives' interpretation of Big Data, and it varies. However, in the context of this thesis the appropriate definition is provided by the TechAmerica foundation, which gives the definition "Big Data is a term that describes large volumes of high velocity, complex and variable data that require advanced techniques and technologies to enable the capture, storage, distribution, management, and analysis of the information" [1].

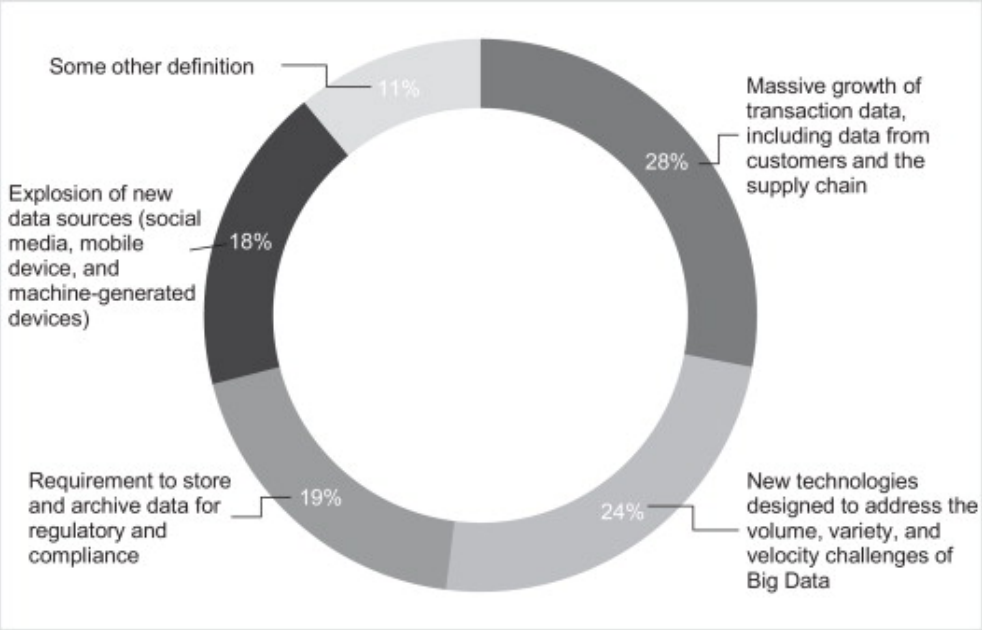


Figure 1.1: Executives' interpretation of Big Data based on an online survey [1].

The traditional architecture where relational database management systems (RDBMS) are used to store, to process and to serve monitoring events has clear limitations and has become inadequate for obtaining timely intelligence; an example can be seen in Figure 1.2 [2]. Scalability, the capacity of the system to accommodate heightened demands in terms of data processing, is hard to achieve with the traditional system as the architecture cannot handle the overwhelming amount of data that need to be processed. Fault-tolerance is the quality of a system to remain in operation accurately with the occurrence of a malfunction of one or more components, and high fault-tolerance is tricky to achieve. A technique,

like sharding, is a method of splitting up a large dataset into numerous, much smaller, datasets that can be dispersed across multiple servers for load balancing. However, this leverages complexity at the application level, which leads to higher maintenance, increased operational costs and increased possibility of human error. Moreover, effective monitoring requires low-latency read access to real-time data which is not possible using traditional architecture.

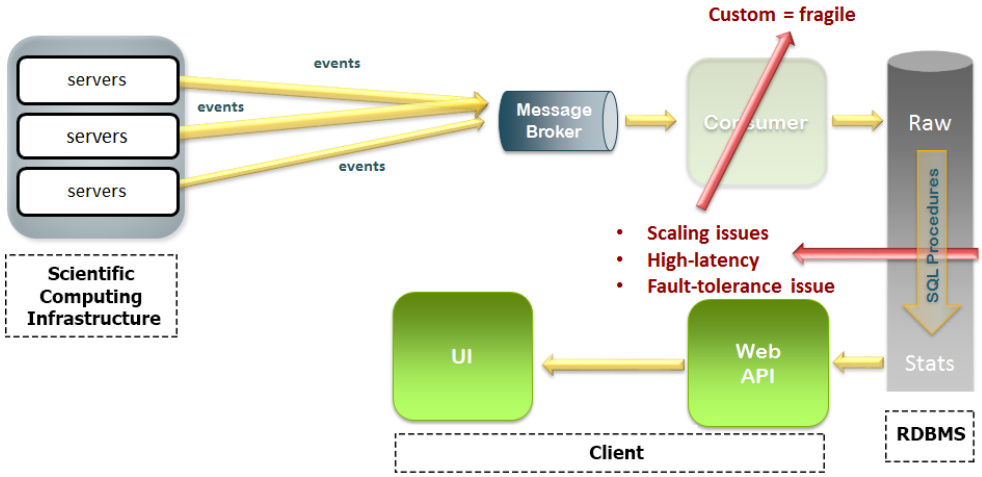


Figure 1.2: An example of a traditional monitoring architecture [2].

In recent years, the challenge of handling big volume (referring to the size of data), velocity (referring to the speed at which data are produced and the rate at which the data should be interpreted) and variety (referring to the structural diversity in the data) has been taken on by many companies, particularly in the Internet domain, leading to a full paradigm shift in data archiving, processing and visualisation. A number of new technologies have emerged, each one targeting specific aspects of large scale distributed data-processing. However, all these technologies, such as a batch computation system (which offers an efficient method of processing large volumes of data where the collection of events is accumulated over a span of time) and unstructured databases (which support text, images, audio, and video) fall short of the structural arrangement required by the traditional database management system. They can handle very large data volumes with little economical cost using commodity hardware (moderately inexpensive, broadly available and compatible with other hardware of its standard) but with serious tradeoffs.

For example, Hadoop is a distributed processing framework that can run large scale batch computations on very large data volume, but with high latency [18]. Another example is non-relational database systems such as Cassandra that can offer scalability but only support a very limited data model with no or relaxed consistency.

1.1 Motivations

The challenges highlighted above form the basis of this research study. The aim was to attempt to tackle a number of critical constraints with the traditional architecture in monitoring scientific infrastructure with regard to developing an efficient, large-scale, real-time and intelligent system using distributed commodity infrastructures. In doing this it was important to prevent and minimise any system failures. The knowledge gained in the development of a monitoring system can be used for automating or improving the infrastructure such as efficient job allocation, resource deployment and so forth.

The main contribution of the presented thesis work was a scalable data store and analytics architecture for real-time monitoring of data-intensive heterogeneous scientific infrastructure, designed to profit from parallel processing on the collaborative infrastructure provided by the WLCG group. Distributed and parallel computing methods are considered to be fundamental characteristics for handling Big Data. Although distributed and parallel programming in dispersed environments is not a trivial task, several frameworks and models are available that make distributed and parallel computing more convenient. The Spark framework is one such Big Data programming framework designed to decrease large-scale computation challenges and facilitate a development of automatic distribution and parallelisation, which are analysed in-depth in Chapter 2. However, the examination of Big Data technology-based approaches in the context of distributed infrastructure is not contemporary. It is a field that one can recognise as being relatively new and is evolving rapidly with regard to capitalising its potential for application in the scientific field. From this standpoint, Big Data technology can be used to deal with both data (e.g. data mining for identifying patterns in unstructured data) and computationally intensive challenges (e.g. processing data produced at high velocity) with monitoring the scientific

infrastructure. The fundamental aims of the thesis work were to explore the required accuracy, efficiency, intelligence, scalability, simplicity and cost effectiveness of the monitoring architecture.

1.2 Methodology

The data (monitoring events in the form of metadata) for the prsented work were collected from three major experiments carried out at CERN, which were ATLAS, CMS and LHCb. This investigation takes the form of case studies of some of the important scenarios such as checking jobs, data management, system failures and so forth for evaluating the proposed monitoring architecture. A limitation of this study was the lack of evaluation of the proposed architecture in a scientific infrastructure other than WLCG due to lack of association. However, it is comforting that WLCG approved evaluation of the proposed architecture on their experiments such as ATLAS and CMS, which are immense and isolated in their own right.

An experimental Hadoop cluster with fifteen nodes was set up to evaluate the proposed work presented in this thesis. The specifications and configurations of the nodes are presented in Chapter 4. Eight nodes were assigned with 32 CPU cores, 64 GB RAM and 500 GB hard disk storage and seven nodes was assigned with 4 CPU cores, 8 GB RAM and 500 GB hard disk storage. The Linux-2.6 operating system was installed on all fifteen nodes. Hadoop-2.6, Flume-1.6 and Spark-1.6 version were configured on all nodes. A custom-made metrics application was developed for collecting the proposed architecture statistics.

To further evaluate the work presented in this study, a virtual cluster was created in the Amazon Elastic Cloud (EC2) using a general purpose instance “m4.2xlarge” that had eight virtual CPUs, 32 GB of memory, and 20 GB of storage per instance. The cluster was configured with four nodes, one name node and three data nodes. The same Hadoop, Flume and Spark versions, and operating system were installed on each instance.

1.3 Major Contributions to Knowledge

The principal contributions to knowledge by the work presented in this thesis can be summarised as follows:

The thesis presents an efficient approach for the collection and storage of high volumes of data for analytics, which is the discovery and interpretation of meaningful patterns in data. The proposed data pipeline is responsible for processing and moving data between different elements at specified intervals where the output of one component is the input of the next one, and is based on custom made daemon using Hadoop framework. The approach was also tried on a custom-Flume implementation. A custom Java daemon was developed to read monitoring events from the message queues and write them into the Hadoop Distributed File System (HDFS) with and without data transformation and serialisation, a process of translating data into binary formats for efficiency. Also, a custom-made Flume was developed to take advantage of the Hadoop appending mechanism to flush data into HDFS. A simple data analytics algorithm was implemented on MapReduce to process large volumes of monitoring events. The analysis was evaluated from the aspect of execution time and scalability in comparison with the traditional method used in the data pipeline. A working prototype based on the custom-Flume pipeline was deployed on the WLCG Hadoop cluster and made available during the LHC second run, in a context that reads events from multiple sources and writes them into the HDFS ready for analytics.

The thesis presents an architecture for monitoring scientific infrastructure using the Lambda approach [19], a Big Data processing architecture. This method has been theoretically and practically tested on a smaller scale. While some studies have been carried out on the Lambda approach, there have been few empirical investigations conducted on a larger scale and also there has been little quantitative analysis conducted. Therefore, in this thesis, the Lambda approach has been adopted, implemented and evaluated on one of the largest scientific infrastructures for monitoring and an in-depth comparison was made with the traditional architecture. The first objective was creation of the batch layer, to store a constantly growing dataset providing the ability to compute arbitrary functions on it. The second objective was creation of the serving layer, to store the batch-processed

views, using indexing techniques to make them efficiently queryable. The third objective was creation of the real-time processing layer able to perform analytics on fresh data with incremental algorithms to compensate for batch processing latency.

The evaluation of the Lambda architecture indicated that it performed well, but the complexity of joining various types of technology together is very time consuming in implementation and maintenance, which requires a dual code base for streaming and batch analysis. The Lambda approach also relies a lot on the client side for joining the statistics produced by both the batch and streaming processes. This thesis presents an optimised Lambda architecture using Apache Spark technology, which involved modelling an efficient way of joining batch computation and real-time computation without the need to add complexity to the User Interface (UI). A few models were explored for WLCG Data acTivities (WDT): pure streaming, pure batch computation and the combination of both batch and streaming. Evaluation of these methods was conducted in a Hadoop cluster and results are presented.

In a scientific experiment, there is usually a lot of data movement; information on these movements can be collected for analysis. Using this information, models can be built to forecast data popularity for efficient data management and placement. This thesis presents how the intelligence can be integrated into the monitoring infrastructure. It presents some insight studies of data access pattern with the ATLAS experiment. This study evaluated the distributed and parallel processing Machine Learning library from the Apache Spark stack for using supervised regression algorithms for forecasting dataset popularity and the number of data replicas required for efficient resource utilisation. Based on this information a decision can be made as to whether enough (do nothing), too few (add) or too many (delete) replicas have been found. It also presents both a batch learning technique, which makes predictions based on historical training data using Spark, and an online learning technique, which dynamically adapts and make predictions as the data are streamed in from Spark Streaming. Also, it presents a complex Deep Learning technique using Sparking Water (H2O) library for the data access patten prediction. Accuracy and performance are the primary properties for effective data popularity prediction for an efficient data management; hence, they were evaluated and presented.

The tasks presented in this thesis have been intensively evaluated on a Hadoop cluster provided by the WLCG. Also, the optimised-Lambda architecture was assessed on the EC2 cloud as well as on the WLCG cluster. Evaluation results are presented and analysed extensively.

1.4 Thesis Organisation

Following this introductory chapter the thesis is organised as follows:

Chapter 2 provides a general background on Big Data architectures for batch and real-time processing, Big Data technologies and machine learning libraries.

Chapter 3 presents the design, implementation and evaluation of an efficient strategy for the collection and storage of large volumes of data for computation. The performance of the strategy is evaluated from the aspects of data ingestion, with data transformation and without data transformation, intermediate data transformation using a MapReduce job and a simple statistical analytic computation using a MapReduce job and a Procedural Language/Structured Query Language (PL/SQL) procedure with and without data transformation.

Chapter 4 presents the design, implementation and evaluation of an architecture using the Lambda approach for monitoring a scientific infrastructure. The architecture models three core layers for supporting real-time monitoring. The architecture was evaluated from the aspects of scalability, data I/O rate, fault tolerance, real-time processing speed, data size/partition, data format and data compression.

Chapter 5 presents the design, implementation and evaluation of an optimised Lambda approach for monitoring the scientific infrastructure that improved upon the performance of analytics and real-time processing using the standard Lambda Architecture.

Chapter 6 evaluates machine learning algorithms for forecasting dataset popularity and the number of data replicas required for efficient resource utilisation. Also, both a batch learning technique, which makes predictions based on historical training data, and an online learning technique, which dynamically adapts and predicts as the data are streamed in, are examined.

Chapter 7 concludes the thesis and discusses some limitations of the research. In addition, suggested future research is pointed out for further improvements and extensions to the thesis work.

Chapter 2

Review of Literature

There have been numerous efforts to review Big Data architectures and with various scopes such as scalability, fault-tolerance and low-latency. In the past few years, there were a number of Big Data technologies emerged for supporting the Big Data architectures. These technologies support the development of distributed analytics platform for solving large-scale, low-latency and intelligent problems. This chapter provides an extensive review of Big Data architectures, Big Data technologies and Machine Learning libraries.

2.1 Architecture

In modern times, there are numerous architectures that are used in monitoring scientific infrastructure systems for storage of data and analysis. These range from the traditional architectures that can only do the basic activities and functionalities in a monitoring infrastructure, to modern approaches that are able to support additional necessities emerging from the monitoring infrastructure. The traditional architectures used relational database systems for the basic purposes of storage, processing and serving the monitoring events in an infrastructure system. On the other hand, the current and more modernised architectural approaches have been able to cover these functionalities and also overcome a number of limitations existing in the use of the traditional architectures [20].

This chapter acknowledge the presence of all the old and the new architectures in the modern times by providing a review of their use.

2.1.1 Review

In modern times, there are numerous architectures that are used in monitoring scientific infrastructures for storage of data, and analysis. Some of the traditional architectures are still in use in many monitoring infrastructures, regardless of the fact that there are numerous new and more effective approaches that have been introduced [21]. However, these traditional approaches, are slowly being replaced with more advanced and modern approaches. This is because while the traditional architectures use relational database systems for the basic purposes of storage, processing and serving the monitoring events in an infrastructure, they result in the creation of a number of limitations for the monitoring infrastructure. For instance, they make the system fail to cope with extension of volume, variety of data, and velocity of the monitoring events and data. This is according to Hellerstein, Stonebraker, and Hamilton [20] who explain that most of the current and more modernised architectural approaches have been able to cover these functionalities and also overcome a number of limitations existing in the use of the traditional architectures.

In agreement with the above assertions, Marik, Schirrmann, Trentesaux and Vrba [22] state that at present, monitoring infrastructure requires large volumes of heterogeneous data to be gathered for analysis. This data normally comes from varied frameworks and services such as data transfers, job monitoring, and site tests and it aims at providing a flexible but rather uniform interface for use by scientists and the monitoring sites. This is according to Martinez, Garcia, and Marin-Lopez [23] who explain that in modern times, the architecture design used has a number of limitations. This architecture has a number of relational database systems that are vital for the processing, storage, and servicing of monitoring data. However, the architectural design can not cope with an increase in volume of the data transfer, nor is it able to foresee the variety of monitoring events that may be required.

Martinez, Garcia, and Marin-Lopez, Andreeva et al. [24] argue that the current mon-

itoring systems are able to deliver solutions that are also reliable for the purposes of supporting the operations and functionalities of large volume datasets. However, in the near future, monitoring systems will be forced to be more robust and able to cope with changes likely to take place in the next generation of architectures. This is because the current traditional architecture designs cannot provide for scalability and they are too costly to maintain especially after implementation of techniques such as sharding (which is the method of splitting up a large dataset into numerous, much smaller, datasets that can be dispersed across multiple servers for load balancing). Further, traditional architecture designs have room for possible human faults and high maintenance costs. Current architecture designs also fail to offer up-to-date monitoring of the infrastructure, therefore, it is critical to have real-time system to access real-time data and have access to low-latency readings. The database procedures built-in within the monitoring system should also be able to impose constraints on the timeliness of the monitoring data, granularity, and the format.

Lambda Architecture

In examining the performance of various traditional architectures as well as some of the new ones introduced in modern times, Hausenblas and Bijnens [3] talk about the Lambda architecture. They observe that the term was coined by Nathan Marz to describe a generic scalable data processing architecture that is also fault-tolerant. They reveal that Nathan Marz introduced this term while working on various distributed data processing systems on social media sites such as Twitter and Backtype.

According to Hausenblas and Bijnens [3] the Lambda architecture's basic objective is to fulfil the requirements for any infrastructure that is fault-tolerant, robust, and prone to human as well as hardware faults. This is because it is able to function in a wide variety of use cases and workloads in situations where it is critical to ensure that the system has low-latency and provide regular updates to users. Therefore, the final system developed using the Lambda architecture is linearly mountable and it scales horizontally.

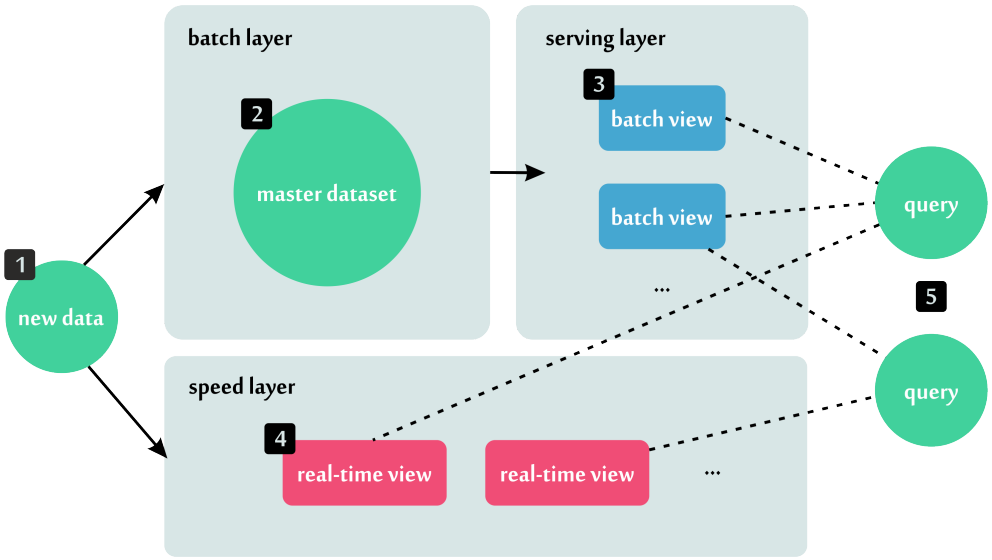


Figure 2.1: The high level perspective of Lambda Architecture (adapted from Hausenblas and Bijnens [3]).

As shown in Figure 2.1, the Lambda architecture has five critical layers, or stages for servicing a system. The first stage involves the entrance of raw data into the system. At this stage, the data is dispatched to two different layers, the speed and the batch layers, where it is processed. In the batch layer, the data is managed within the master data set and pre-computed into batch views. Then, it is forwarded to the serving layer where the batch views are indexed to allow for the data to be queried in low-latency. In the speed layer, only recent data is processed and in this layer, the Lambda architecture is able to compensate for high-latency of updates for the data in the system. Queries entering the system are answered when the results of the batch views in the serving layer and the speed layer are merged [3].

Druid architecture

In examining a real-time analytical data store, Yang, Teshetter, and Leaute [25] discuss the Druid architectural approach. To them, this open source data store is able to support faster processing of aggregated data compared to Lambda architecture and it is also flexible for filtering information as well as offering low-latency data ingestion process. This system is also able to combine a storage layer in column orientation as it is a distributed

and advanced layer for advanced indexing structures that allow exploration of numerous row tables within sub-seconds latency.

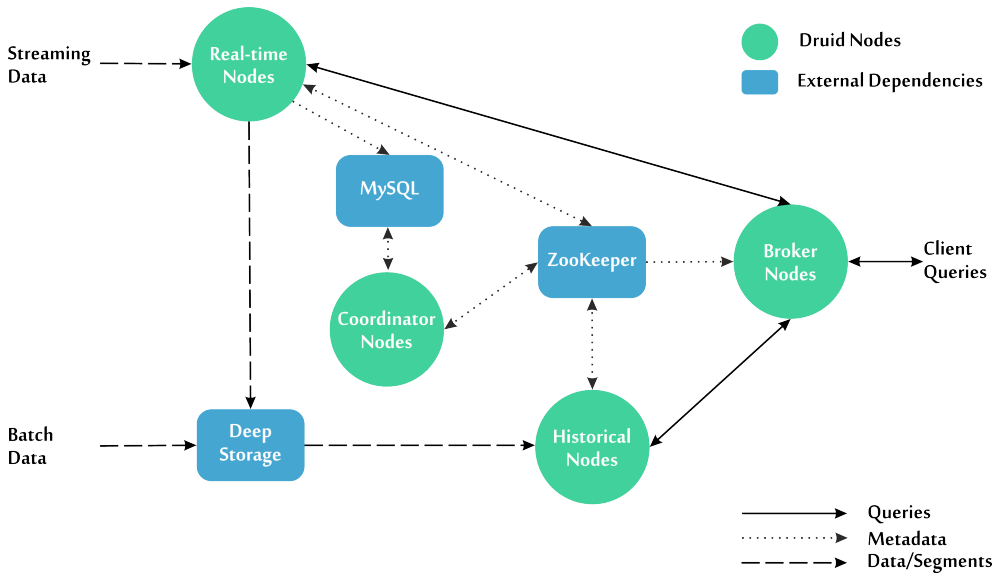


Figure 2.2: The Druid Architecture (adapted from Hausenblas and Bijnens [3]).

In discussing the Druid architecture, Yang, Teshetter, and Leaute [25] argue that this approach is made up of varied nodes that are designed specifically for varied functions. In agreement with the view by Yang, Teshetter, and Leaute, Mukhopadhyay [26] notes that the nodes are crucial for the Druid architecture as it supports the complex nature of the entire system as well as separating the concerns of delivering queries from those of latency issues. In any case, as shown in the Figure 2.2, there is evidently low interaction activity between each node thus lowering the chances of encountering communication failures ensuring high data availability [25].

With regard to the solving of complex data analysis issues, Yang, Teshetter, and Leaute [25] note that the nodes of the Druid approach come together and thus form a complete system that is fully functional. Then, each node is able to provide real-time information by encapsulating the operations during ingestion and querying a stream of events. In order to deliver queries immediately, the nodes also ensure that they index events. This is because events that are indexed together via the nodes of any system or approach are able to quickly respond to queries. For any modern architecture, the aspect of scalability

is very important but very hard to achieve as sharding pushes for complexity within the systems. However, in order for effective monitoring to be achieved in any architecture, there is a need for low-latency (real-time) support.

Kappa architecture

In agreement with the views of Chen, Yang, Teshetter, and Leaute, Forgeat [4] explains that another approach to a real-time system and analytics platform is the Kappa architecture. This approach uses software architecture approach and it avoids the implementation of relational databases. Instead, it has an immutable log for append only. According to a report by Uesugi [27], it is from this log that data is streamed and fed into stores for serving via a computational system. In seconding the assertions by Uesugi, the report [28] reveals that Kappa is a simpler architecture compared to that of Lambda approach. In fact, it is a simpler and easier version of Lambda approach. This is because, excluding the batch processing part of the Lambda architecture, the parts and functionality of the Kappa architecture are very similar to those of the Lambda approach.

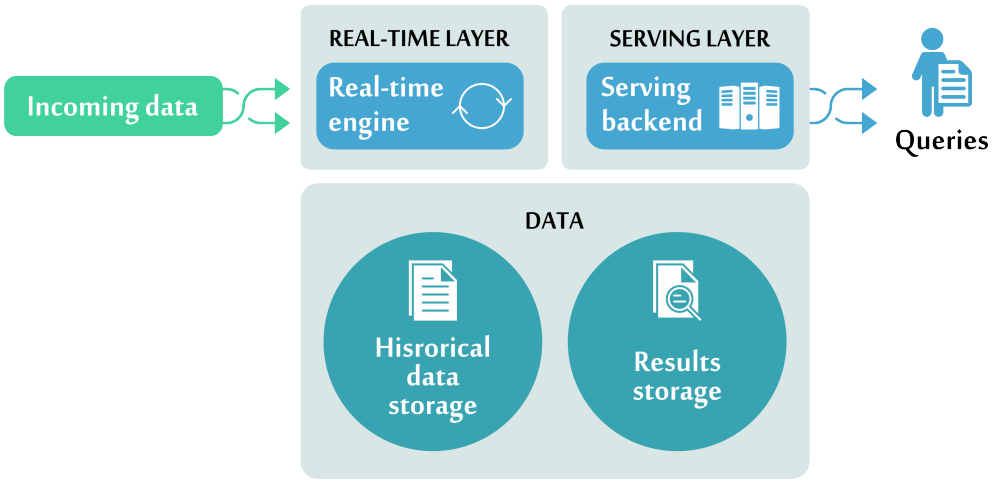


Figure 2.3: The Kappa architecture (adapted from [4]).

In explaining some of the benefits associated with the use of Kappa architecture, the report [28] explains that it was initially invented in order to avoid the issue of maintaining two different codes frameworks for the speed and batch layer respectively, as is the

case with the Lambda architecture. Therefore, this implies that the main idea behind the Kappa architecture is to ensure that real-time data processing systems and the continuous data reprocessing systems are integrated into one effective and efficient system. This is because both parts are very critical to analytics platforms [4].

As shown in Figure 2.3, Kappa architecture only has two layers, one for stream processing and the other one, serving layer. It also has a data section that is able to support other basic functionalities of a real-time system by storage of results and historical data. In this approach, the stream processing jobs are tackled first, then the data reprocessing jobs are carried out when some stream processing jobs need to be modified, altered or reprocessed. Kappa is mainly stream processing, reliable when coupled with tools offering certain guarantees (e.g. Kafka), but Kafka is a temporary buffer, where the retention policy can be hours, days at maximum. Processing an arbitrary set of historical data, a feature which is relevant in the scientific infrastructure to handle recomputation is limited with this architecture.

The serving layer in the Kappa architecture, just like in Lambda architecture, is used for forwarding the queries based on the results of the processing carried out. With regard to its application and use, Hellerstein, Stonebraker, and Hamilton [20] argue that where the algorithms for both real-time data and historical data processing systems are different from each other, the Lambda architecture should be used. According to Ellis [29] who notes that the main benefit associated with Kappa architecture is the fact that it allows developers of real-time systems to operate, test, as well as debug their systems on a single framework for processing.

An exploration of a scientific infrastructure

In exploring how monitoring of infrastructure is carried out in modern times, the monitoring of the Worldwide LHC Computing Grid (WLCG) infrastructure was explored. It is necessary for the heterogeneous data to be gathered and analysed. For instance, this process supports data transfers, testing of sites, and monitoring of processing jobs among

other requirements of the heterogeneous infrastructure. In providing a flexible interface, that is also unique, the monitoring process handles this data regardless of whether it is coming from the service framework or the experiment framework. The current architectural approach implemented in modern times for storing, processing, and serving data uses relational database systems that include Structured Query Language among others. This is for the purposes of serving monitoring data, processing the same data, and storing the data [24].

While this architecture manages to support most of the functionalities where monitoring of the infrastructure is concerned, Andreeva et al. [24] note that this is not adequate. It has limitations in coping with higher Large Hadron Collider luminosity, new data transfer protocols, and cloud computing among other newly introduced techniques of the WLCG events monitoring process. Therefore, Ellis and Jung argue that there is a need for the development of a new scalable data store and analytics platform for use in the current times [23].

Chen [30] agrees with the views discussed above and notes that in a scientific workflow system, it is critical to track the status of workflow in real-time while the user is being notified of any anomalies and failures automatically. In order to add real-time monitoring and troubleshooting capability to the system, an individual could seek to integrate some monitoring infrastructure to their generic system such as the Stampede monitoring infrastructure [31]. This infrastructure is able to address interoperable monitoring jobs by the use of a three layer architecture. It is also useful in describing workflow and job executions as well as delivering tools that are of high performance quality to load workflow logs that conform to the model of the data within the data store system. Stampede is also described as an infrastructure that offers an interface via which the user can query events and extract data from the storage unit [32]. However, this infrastructure uses a traditional relational database for storage.

2.1.2 Summary

Traditional architectural approaches to real-time monitoring of computing infrastructure are slowly being replaced by modern approaches for the storage of data and data analysis. This is because these modern architectures support additional necessities such as low-latency, fault tolerance and scalability emerging from the Big Data unlike the traditional architectures that use relational database systems for the basic purposes of storage, processing and serving the monitoring events in an infrastructure system. In addition, the current approaches also overcome the limitations (e.g. high-latency) existing in the use of the traditional architectures. This section has examined the scalability of data storage and analytics platforms and reviewed the use of the architectures, discussed a scientific experiments and how data as well as the analytics jobs are used in the infrastructures.

2.2 Technology

The aim of this section is to review some of the state-of-the-art technologies that can be used to design a monitoring system for storing and handling Big Data and perform real-time analytics on them.

2.2.1 Batch Process

A batch process is required to store constantly growing Big Data and for historical data analysis that is used to identify patterns such as job failures, popular data, busy sites and so forth. Many individuals consider Hadoop as the de facto framework for analysing Big Data. However, there are many technologies available for application in a distributed system such as the Internet that go beyond MapReduce, which is a programming model for processing big data that was introduced by Google [33]. In this section an attempt will be made to review such technologies.

Apache Hadoop: MapReduce and HDFS

The Hadoop ecosystem has been used for many research and commercial products [34][35]. It has gone through rigorous implementation and testing, which makes it robust. There are many Hadoop ecosystems, but the MapReduce and HDFS are most relevant as they support parallel processing as well as a large data storage with higher data availability. Therefore, they will be analysed in this section. MapReduce is a programming model that was designed to remove the complexity of processing data that are geographically scattered around a distributed infrastructure [33, 36]. It hides the complexity of computing in parallel, load balancing and fault tolerance over a large range of inter-connected machines from the users.

There are two simple parallel methods, map and reduce, which are predefined in the MapReduce programming model and are user-specified methods, so users have control over how the data should be processed [36]. Hadoop was designed to take into account that moving computing to where the data reside is better than vice versa as it will reduce bottlenecks in the network, especially when the data that are being transferred are at the size of terabytes to petabytes [33]. Therefore, map and reduce jobs will be allocated to where the data reside, which will be scheduled by a job task manager as shown in Figure 2.4. The data will be read from a local disk (file system); mapped, with all records being independently processed and key/value pairs assigned; intermediate results stored to a local disk and shuffled (transferred to where the reduce jobs are located); and reduced, so that records with identical keys are processed together and the output is written back to the disk (this output could be an input to another MapReduce job) [33]. Fault tolerance in MapReduce is supported by periodically checking whether the worker nodes are active. Master failures can be protected against by using check-pointing, an approach used to enable applications to recover from failure by loading the last check pointed state at the start-up stage.

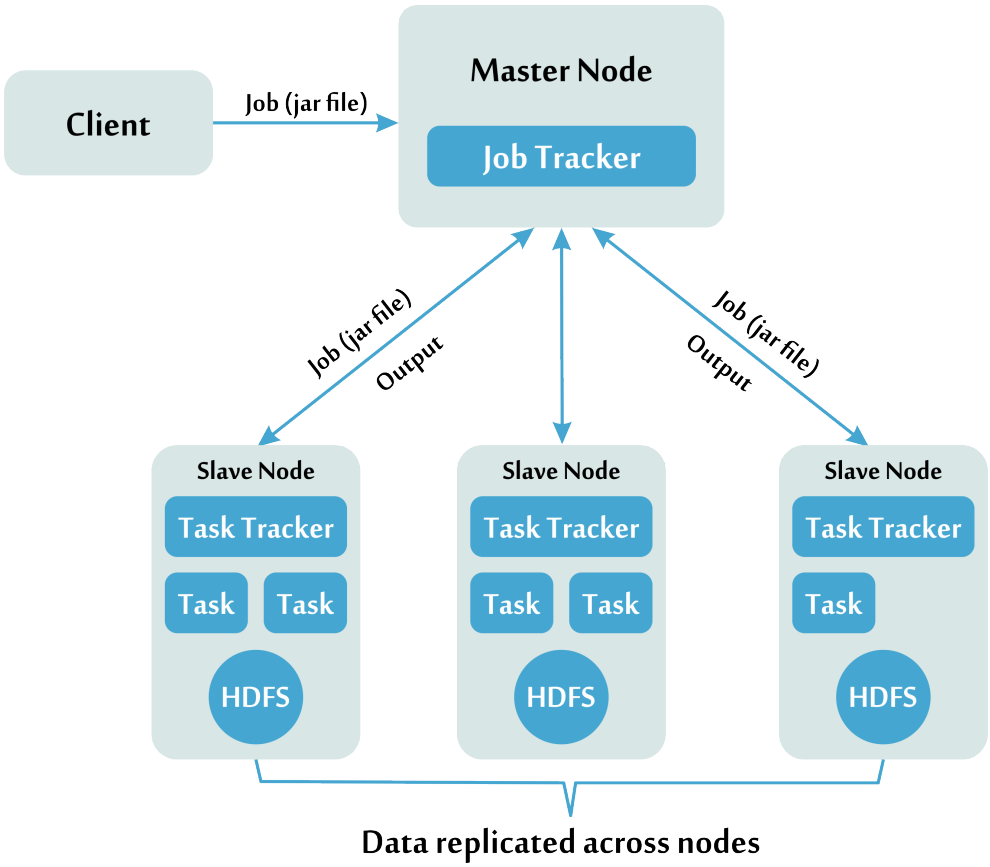


Figure 2.4: Hadoop architecture.

The MapReduce framework is built to on run HDFS and executes I/O operations on it. The HDFS guarantees: scalability on commodity hardware, fault tolerance, high throughput, load balance, data integrity and portability [37]. It employs master-slave architecture, which is prone to single point failure. However, it facilitates failover to the standby server but this is prone to downtime. Data are replicated across disk nodes for load balancing, fault tolerance and high availability [37].

Apache Spark

Apache Spark is an in-memory distributed computing framework [38, 39]. It provides a general programming model supporting iterative classes of algorithms, interactive applications and algorithms containing common operators such as Map, Reduce, Join, Filter, GroupBy, Sort, LeftOuterJoin, RightOuterJoin, Count, Union, Cross and so forth [38].

Spark allows the dataset to be kept in the memory by moulding a new memory abstraction, called Resilient Distributed Datasets (RDDs) [39]. Instead of repeated I/O operations, Spark fetches the dataset once from the file system and store it into a memory and then directly accesses it from the memory thereafter, which improves the performance. By storing intermediate results in the memory, it provides a mechanism for reusing the data to perform other operations such as iterations (e.g. deriving many results from the same data).

Comparison

MapReduce is largely a solution for batch processing [33]. However, MapReduce hardly deals with the instances where the development of functions requires the arbitrary mixture of a set of operations, iterative jobs and multiple inputs. Nevertheless, the above mentioned actions could be achieved by implementing multiple Map and Reduce operations. On the contrary, reloading the same data multiple times from the disk will seriously downgrade the performance. The Spark framework provided a mechanism for overcoming this issue by using inbuilt in-memory processing and extending the MapReduce framework to support many operators such as: Join, Group, Union and Cartesian as shown in Table 2.1.

Table 2.1: Operators comparison.

Spark	**MapReduce**
flatMap	map
map	combine
mapPartition	reduce
union	
rightOuterJoin	
leftOuterJoin	
join	
intersection	
filter	
groupByKey	
sortByKey	
updateByKey	
reduceByKey	
combineByKey	

In brief, MapReduce and Spark have some common features: load balancing, and fault tolerance. Spark provides an in-memory processing mechanism, interactive analysis and additional operations.

2.2.2 Interactive ad-hoc query engine

Batch processing jobs are expected to run for hours, weeks, months or even years. However, this requires the technical skills to implement these jobs. Therefore, a simpler framework is necessary for a non-technical user, hence, an ad-hoc interactive queries engine was explored. A few commercial companies and research community have developed tools to resolve this issue, which has been reviewed in the following section.

Apache Drill

Apache Drill is a distributed execution engine that facilitates interactive, ad-hoc querying of heterogeneous data sources on a large scale, which was inspired by Google's Dremel [40, 5]. Its design goal is to scale to 10,000 servers or more and to process petabytes of data and trillions of records in seconds [5]. As shown in Figure 2.5, Drill's architecture is made up of three major components: query languages, which is responsible for parsing the users query and constructing an execution plan; a low-latency distributed execution engine that provides the scalability and fault tolerance needed to efficiently query petabytes of data; nested data formats, which are responsible for supporting various data formats [5]. The initial goal was to support the column-based format used by Dremel [5]. Finally, scalable data sources are responsible for supporting a variety of data sources. The initial focus is to leverage Hadoop as a data source [5].

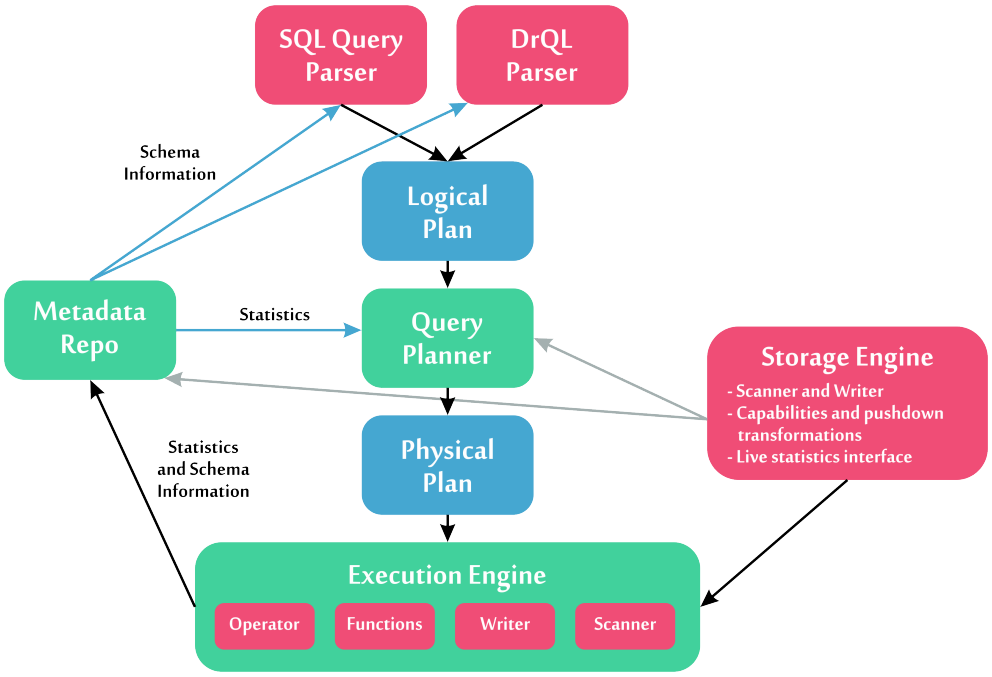


Figure 2.5: Apache Drill architecture [5].

From a distribution perspective, a Drillbit, a specific node instance of Drill, uses local memory and data. Queries can be made from any such instance [5]. The co-ordination, query planning and optimisation, scheduling, and execution are then distributed.

Cloudera Impala

Cloudera Impala is a massively parallel processing (MPP) architecture for performing SQL-like queries on HDFS and HBase (a column-based data store built to run on top HDFS) storage as shown in Figure 2.6, which does not employ the MapReduce model as other alternatives such as Hive (a data warehouse tool for querying and analysing large datasets) [6]. It leverages techniques such as columnar storage for performing really fast scans in the order of seconds of huge amounts of data in memory. All data in HDFS or HBase do not require Extraction, Transformation and Loading (ETL) so can be queried directly without any data movement or predefined schemas using SQL-like commands. Impala inherits inbuilt Hadoop security by integrating with Kerberos (a protocol for validating service requests between hosts) for authentication and role-based authorisation [6].

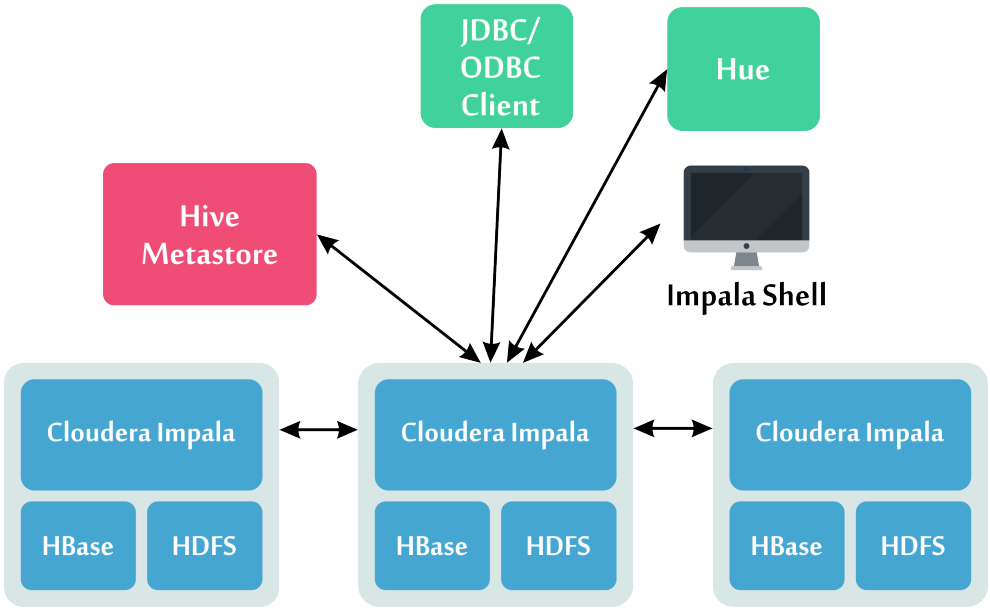


Figure 2.6: Cloudera Impala [6].

Presto

Presto is a distributed low-latency, interactive and SQL-compliant query engine optimised for ad-hoc analysis developed by Facebook [7]. It also supports the majority of ANSI SQL

subgroups, including complex queries, aggregations, joins, and window functions [7]. All processing is carried out in-memory and pipelined across a network between steps, which should reduce the read/write time to disk thus improving performance. The shortcomings of the system are its inability to write output data back to tables as it only supports a read-only mode. In Presto architecture as shown in Figure 2.7, there is a coordinator that receives SQL queries from the client, which it then analyses, parses and then plans the execution [7]. Then the scheduler connects to the execution pipeline and assigns the jobs to worker nodes that reside closer to the data [7]. The client then fetches the results.

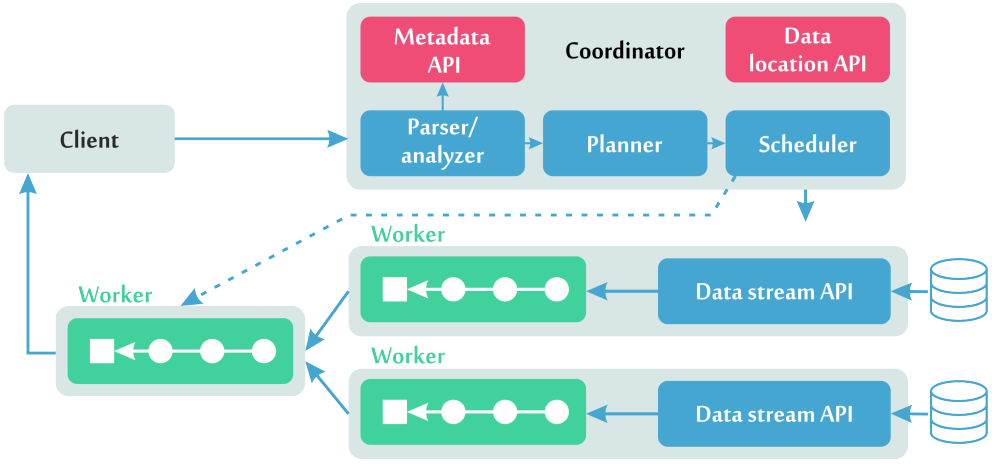


Figure 2.7: Presto architecture [7].

The Presto framework is extendable so that any variety of storage can be plugged in; however, it requires a connector that provides Presto with metadata, information on which nodes hold the data, and a way to actually fetch the data as a stream. These are a few storage plug-ins supported by the Presto: HDFS, Hive, HBase and Scribe [7].

Comparison

Hadoop was not built for interactive ad-hoc querying; it mainly focuses on offline batch processing. This has resulted in a need for new technologies that resolves interactive ad-hoc querying demand. In recent years a few tools have emerged to address this issue, which have been discussed above. In brief, Drill, Impala and Presto were developed to take advantage of in-memory temporary data locality. Drill supports long-running queries

and ad-hoc queries, whereas Impala and Presto do not support long running queries.

No fault-tolerance is implemented in Impala or Presto; when a node fails at the execution time then the queries need to be re-executed. Impala was designed to take advantage of the existing Hive infrastructure, which uses the same metadata. In contrast, Drill and Presto were developed to provide distributed query abilities across various data stores.

2.2.3 Real-Time Processing

Real-time processing is required to perform real-time analytics on fresh data as they are received. This is required to monitor an infrastructure proactively and trigger actions so that the operation will run smoothly.

Apache Storm

Apache Storm is a distributed real-time data system [8]. It is considered as an alternative to high-latency batch processing for processing data in low-latency near real-time. Storm can be embedded within the queuing and database technologies. It facilitates scalability by enabling users to determine how many worker nodes are required to execute a job and the amount of parallelism (number of threads) required on the topology configuration. It also uses an independent Apache technology called ZooKeeper for coordinating the cluster, which also supports a cluster scale [8]. The architecture employs a master-slave model [8]. The master node has a daemon called Nimbus, which is responsible for distributing user applications to worker nodes, allocating jobs to the worker queue and monitoring the status of the worker nodes, which on failure will restart the node or reassign the task to other nodes [8]. The slave nodes have a daemon called supervisor, which is accountable for checking the queue for new jobs [8].

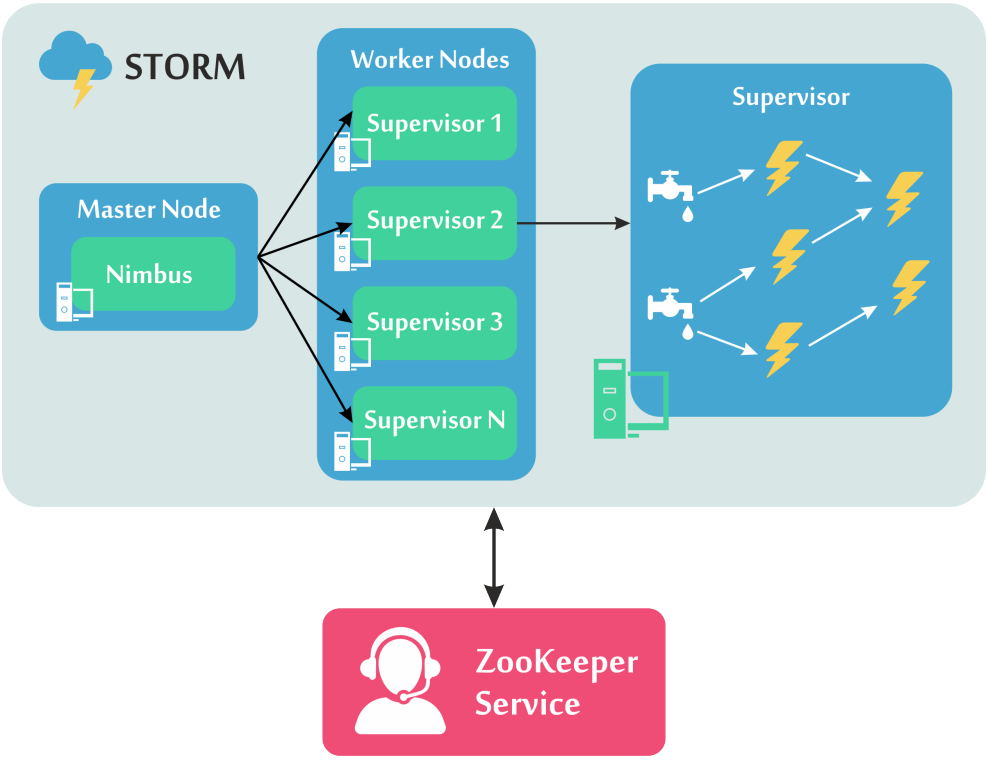


Figure 2.8: Storm topology [8].

Storm uses tuples as its data model, which consist of a list of values. Groups of spouts (a source of streams) and bolts (computational process) are packaged into a topology, which is then deployed into clusters that will run infinitely, until killed manually. As shown in Figure 2.8, the topology will consist of spouts, which are the source of streams; bolts, which consume the stream and process them; and stream grouping, which states how the data should flow [8]. Storm also provides a tool called Distributed Remote Procedure Call (a protocol that a program can utilise to communicate with a program reside in another system on a network), which enables developers to implement complex functions and execute them in Storm utilising parallelism.

Simple Scalable Streaming System (S4)

The Simple Scalable Streaming System (S4) is a distributed general-purpose platform that processes continuous unbounded streams of data [9]. S4 employs the MapReduce and Actor (a mathematical model of concurrent computation) programming models [9]. Therefore,

S4 utilises a concurrent, decentralised and symmetric architecture, with each node sharing the same functionality and responsibility, which is imposed by utilising Apache ZooKeeper in order to coordinate the cluster. There are not any special nodes with special functions. The S4 model facilitates high availability and scalability on commodity hardware, low-latency by utilising local memory, fault-tolerance by check-pointing and summoning the standby server to take over the failed server tasks, and a pluggable framework so that it is more generic and new components can be plugged in [9].

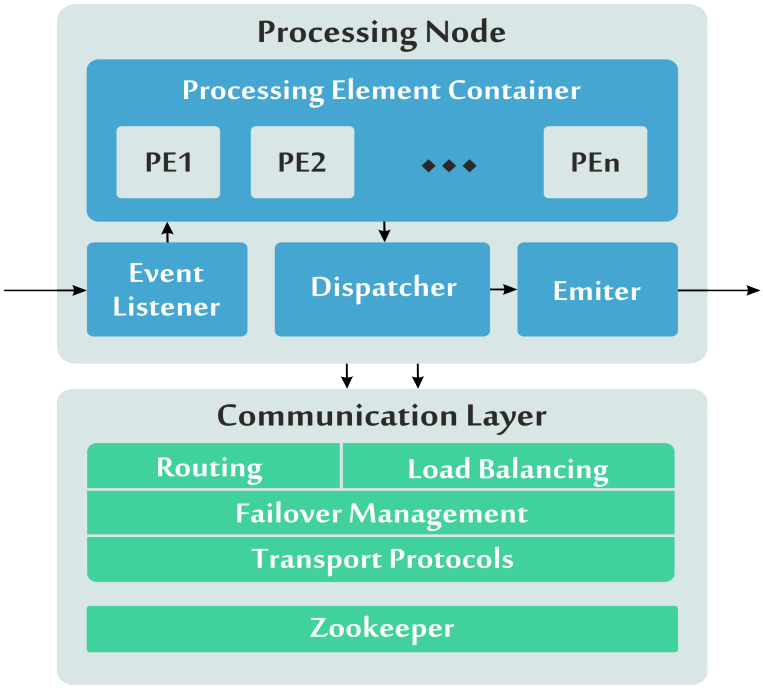


Figure 2.9: S4 architecture [9].

As shown in Figure 2.9 processing nodes are the logical clusters of Processing Elements (PE), an entity that performs computation and transmits messages between PEs by using data events. The processing nodes are responsible for listening to events, executing functions on the incoming events, transmitting events and emitting output events. An event listener in the Processing Node (PN) passes incoming events to the processing element container, which invokes the correct PEs corresponding to a unique key or generates new instances of PEs [9]. An application can be defined in terms of PEs with simple processing logic, and the framework instantiates one PE for each unique key in the stream. The

communication layer provides load balancing, failover management and transport management [9]. There are numerous special PEs that are available for performing tasks such as: Count, Aggregate, Join and so forth [9].

Amazon Kinesis

Amazon Kinesis is a cloud-based service for real-time processing of high-volume stream data [10]. Just as with any cloud service the Kinesis service is based on a metering system, which means you pay for the amount of throughput and Hypertext Transfer Protocol (HTTP) put transactions used [10]. Kinesis is proficient at consuming any amount of data from any number of sources, scaling up and down as needed. The Kinesis client library supervises load balancing, coordination and error handling automatically, so that the developer only needs to focus on processing the data as it becomes available.

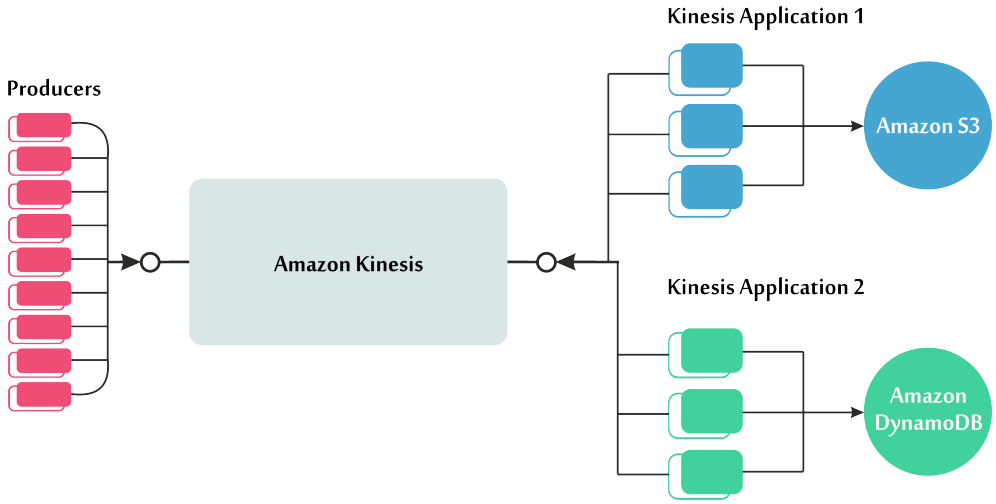


Figure 2.10: Kinesis architecture [10].

As shown in Figure 2.10, Kinesis expects two components, which are the producer and worker [10]. The producer accepts data from a source and converts them into a Kinesis stream, which can now be as large as 1 MB data segments, then transferred into stream using HTTP Put method [10]. The worker then takes the data from the Kinesis stream and processes them. For scalability, the user has to take care of two things; adding or removing shards, depending on the required throughput capacity, and using the Kinesis

client library and deploying the application into an Elastic Compute Cloud (EC2) instance with the auto-scaling group.

Apache Samza

Apache Samza is a distributed stream processing pluggable framework to run continuous computation on infinite streams of data [11]. It is designed to sit on top of the Kafka (a distributed publish-subscribe messaging system) messaging queue for stream processing. It also utilises Apache Yet Another Resource Negotiator (YARN) for resource management and execution, which is responsible for deploying tasks in a distributed cluster, stream processor locality, co-partitioning of streams and providing security [11]. The Samza framework is similar to batch processing as shown in Figure 2.11.

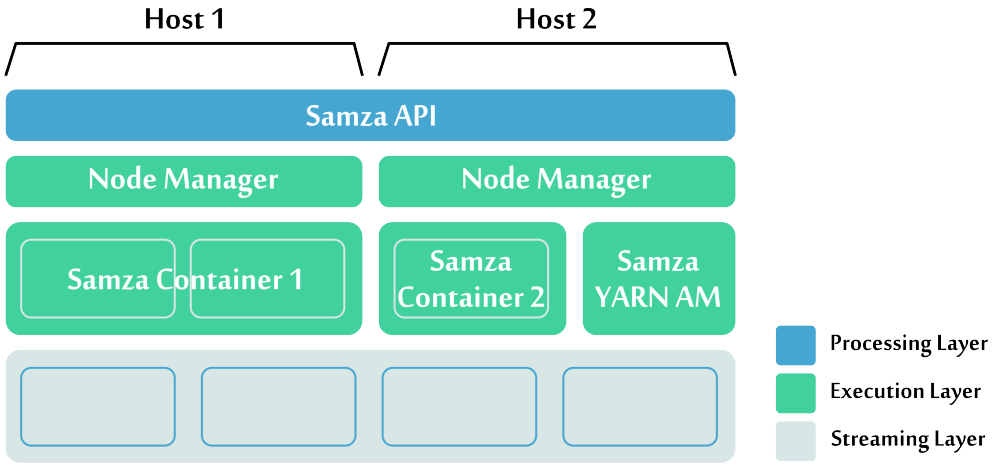


Figure 2.11: Samza architecture [11].

Samza partitions the message, assigns the partition key and sequence ID, and orders them in strict sequence. All messages matching the partition key go in to that particular partition. It also facilitates a replayable mechanism so that a message can be reread when required. The stream processing is done by Samza Job, which performs a logical transformation on a set of input data and emits outputs [11]. Fault tolerances are managed by check-pointing, which enables failure recovery, and state management. This maintains the state of the intermediate data that need to be passed between processes; this is kept in the local disk with each task [11].

Spark Streaming

Spark Streaming is an extension of Spark that supports continuous processing [41, 12]. As shown in Figure 2.12, Spark Streaming is inspired by a batch system, such as dividing processes into sufficient sets so that they can be replayed, assigning failed tasks to other nodes and decreasing batch sizes to tackle low latency [41].

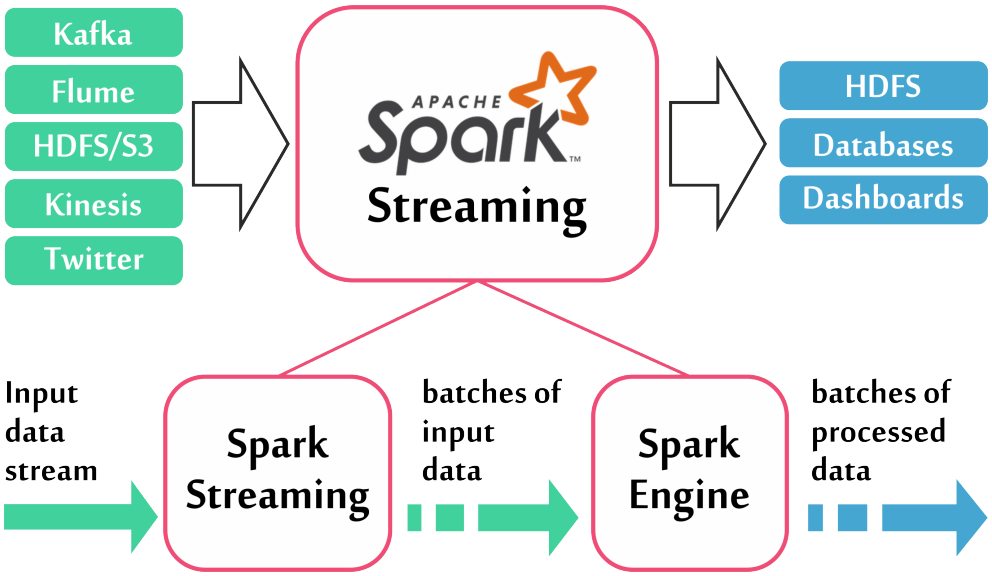


Figure 2.12: Spark Streaming data flow [12].

Spark Streaming provides transformation function, which produce a new Discretized Stream (a continuous sequence of micro RDDs) from one or more parent streams and cumulative computation function, which supports cumulative computations while streaming in new data [41]. Spark Streaming supports all operators that are supported in Spark such as: Map, Reduce, GroupBy, Join and so forth. It also provides a mechanism to aggregate data within a given window of time. It also allows the developer to apply Spark's in-built machine learning algorithms, and graph processing algorithms on data streams[12]. It supports check pointing and fault tolerance, which it inherits from Spark.

Comparison

The existing large-scale MapReduce data-processing platforms are highly optimised for batch processing, which typically operate on static data. Therefore, a paradigm was required to process data in real-time so that business critical decisions can be made on time. This is what motivated the evolution of the stream processing technologies discussed above.

Storm, S4, Samza, Spark Streaming and Amazon Kinesis share the same aim, which is to provide a distributed, scalable and fault-tolerant infrastructure for processing continuous streams of data. Storm, Kineses, Samza and S4 are fundamentally like a pipeline where the source pushes discrete messages, which are then processed a record at a time. On the other hand, Spark Streaming follows a batch processing model where messages are collected and then processed at short-time intervals in a batch manner. However, this is prone to latencies of order seconds compared with former technologies. Nevertheless, Spark does not replicate messages as a mechanism for fault tolerance as with the other systems, which are liable to high disk I/O, network bandwidth usage and overhead associated with the operation itself. Spark utilises in-memory storage abstraction (RDD), which tracks the "lineage" (series of recreation instructions) steps used to build it, so in the event of failures, it can recompute the lost data using the cached steps. Storm does not support managing states, whereas S4, Samza and Kinesis provide tools to manage them locally or remotely. Storm is user-oriented, as it gives full control to the users on how it should be configured so that an external database can be used to store the states; however, this is costly, in terms of performance. Nevertheless, Samza provides a much better mechanism to minimise remote communication by keeping the state located locally with the tasks and only when a state is modified will it invoke a remote method for an update. All the stream platforms discussed above utilise an in-memory mechanism for processing, except for Amazon Kinesis. Although in-memory processing improves low-latencies, this could raise a new issue in terms of flushing out the memory, in particular for S4. A complex application in utilising the S4 framework will generate a large number of unique processing elements as it has been designed to do so, which will occupy a large portion of the memory and could degrade performance. However, there is a mechanism called Time-to-Live that explicitly configures how long the PE should live without any event communication before

the memory is reclaimed, but this will result in loss of the state of the PE. However, there is a method to overcome this issue by applying priority or importance to the PE object, which will be customised by the user.

2.2.4 Summary

This section has given a brief overview of the different technologies that support batch processing, interactive ad-hoc queries and real-time analytics. Most of the technologies were developed by companies concentrating on their specific use cases. For some, performance is important, whereas for others fault tolerance and recovery are important, and only one can be achieved by trading-off the other. So it is not practical to have a perfect technology tailored for one requirement. Therefore, it is important to distinguish and prioritise what is essential for the desired system and what can be compromised.

When there is a separate technologies for each layer such as batch / serving / speed layers, it will become very difficult and complex to maintain an infrastructure. However, the Spark stack support batch processing as well as stream processing. It uses the same processing model and data structures for batch processing and Spark Streaming.

2.3 Machine Learning techniques

One of the problems associated with Big Data is data access and placement. Efficient data placement strategies have a large impact on the computation speed. Data placement is one of the reasons, along with data movement and task assignment for increasing the cost of data centers. This problem gets worse when the data is located in a distributed data center. Therefore, in this section, some of the machine learning libraries and techniques are explored to understand how an adaptive data placement method can be implemented.

2.3.1 Machine Learning libraries and techniques

When Facebook suggests a new page or connects a user to new friends, or Netflix recommends a new movie to its users or Amazon suggests a new product to its customers, all these activities involve Machine Learning (ML). ML is one of the rapidly growing subfields of Artificial Intelligence (AI), with the aim of machines adopting human intelligence. Machine learning is the intersection of computer science and statistics, which is used in numerous real world applications; these include speech recognition, computer vision, bio surveillance, empirical science etc. [42]. Machine learning tasks can be divided into three major categories [43]:

- Supervised learning: Labelled input data is presented to the system with the desired output and the goal is to train the system so that when unlabelled input data are provided, the system should predict the desired output.
- Unsupervised learning: Where the input data are not labelled and the goal is to discover the hidden pattern in the data.
- Reinforcement learning: Reinforcement learning enables a machine to learn from the feedback of a specific environment, for example in game theory or in video games.

Supervised learning tackles classification and regression problems. Classification is the task of identifying a proper class label for new data, based on a training dataset where a number of regression algorithms are proposed for a regression problem. Regression deals with estimating the relationship between variables for feature extraction.

The world is besieged with data. The size of data is increasing constantly. As we can see on the Internet a huge amount of data is produced every day, which the research community has named Big Data. There is always a need to understand that data. It is beyond human capability to analyse Big Data and understand it. Big Data is a collection of huge datasets and requires complex processing, which is not possible on traditional computational systems. According to a study conducted by Dell EMC [44], the growth of digital content doubles every two years. They predict that by 2020 digital data will reach 44 zettabytes or 44 trillion gigabytes in total size. They also recorded that 22% of the total data in 2013 was useful and that less than 5% was analysed, and by 2020, 35% of data will

be useful. With such a large increase in data, the machine learning research community faces the challenge of efficiently using this Big Data. Traditional machine learning toolkits like R or WEKA are not capable of handling Big Data. Hadoop [45], a framework for the Big Data problem, offers distributed storage and provides Big Data processing solutions. Along with data processing there is always a need for machine learning algorithms for implementation in analysis systems for Big Data.

A Hadoop project comprises four modules [18]: HDFS to store data; MapReduce to process the data; YARN, a resource navigator; and Common, a set of common utilities. Most of the machine learning tasks are performed by the processing engine. MapReduce [46] was introduced by Google in 2004 as a programming model for processing a huge dataset. Programs are executed in parallel on a distributed machine and the model is responsible for partitioning input data, scheduling, inter machine communications and machine failure. However, MapReduce does not support machine learning out of the box. Therefore, the open-source community has implemented Map-Reduce for machine learning on a multicore framework. It follows the approach of batch learning, where a set of training data is read off HDFS and a list of key-values pairs is written to the disk. MapReduce has lost its importance in the machine learning community as it is very slow in terms of processing time, requiring a lot of computational resources and network bandwidth, although it is much better than a standalone approach. Another problem associated with MapReduce is its fault tolerance mechanism, where intermediate data is written to disk and replicated in case of a machine failure. Data replication and disk I/O consume 90% of the total running time on MapReduce [46].

To overcome the above mentioned problems the University of California, Berkeley, developed a new model, named Spark [38], which is based on MapReduce but eliminates the above problems. To resolve the problem of replication Spark provides a new storage primitive, RDD [47], which lets the user store data in-memory [48]. RDD is read-only shared memory [49], designed for iterative algorithms (machine learning and graph applications) and interactive data mining tools. Spark performs better than MapReduce and it can also support real-time data processing [50].

Storm [8] is another Apache open-source distributed real-time computation system. Storm received attention in the research community and in the commercial sector for its real-time processing but it also supports batch processing using Trident API. Storm also supports on-line machine learning, which is of growing interest to the research community. Spouts and bolts are the two major components of Storm architecture, where the network of spouts and bolts are called as topologies. Flink [51] is another open source framework for the distributed environment. Flink can process both batch and stream processing. It can run completely independently or integrated with HDFS and YARN on Hadoop. In terms of machine learning, Flink-ML was introduced in April, 2015 and the Scalable Advance Massive Online Analysis (SAMOA) library can also be used for applying learning algorithms on stream data. a comparison between Spark and Flink has been performed by [49] and it was concluded that Spark surpasses Flink in the aspect of fault tolerance, while in terms of computation time Flink is ahead of Spark but takes a lot of time when the required resource is not available. Sparkling water (H2O) [52] is another open source framework providing machine learning libraries along with data processing and evaluation tools. The H2O platform supports a different interfaces for R, Java, Scala, Python and JSON [53]. The Graphical User Interface (GUI) has been provided by H2O while all the above mentioned platforms do not provide any GUI.

Different ML libraries are available for Big Data analytics. Some of these are presented here. Among the most well-known ML libraries, one is Mahout, providing clustering, classification and collaborative filtering, topic modelling, text vectorisation, similarity measuring etc. For classification purposes Mahout implements some basic algorithms including logistic regression, multilayer perceptron, hidden Markov models, Naive Bayes and random forest. For clustering, Mahout implements the K-Means clustering algorithm including Fuzzy K-Mean clustering, Streaming K-Means and also supports Spectral clustering. Mahout supports user-based recommendation and item-based recommendation, when it comes to collaborative filtering. The major problem associated with Mahout is its slow runtime due to the slow MapReduce. MLlib [54] is a Spark project for providing machine learning experience. MLlib is built on Apache Spark, which uses the benefits of in-memory computation for fast processing. MLlib covers all the algorithms supported by Mahout including regression models that are not included in Mahout. For classifica-

tion MLlib supports SVM, nearest neighbour, random forest etc., for K-Means clustering, spectral clustering etc., for regression SVR (support vector regression), ridge regression, Lasso and logistic regression. Additionally, MLlib also supports PCA, non-negative matrix factorization, independent component analysis etc.

H2O provides deep learning including clustering, classification, ensembles, statistical analysis, deep neural networks, data processing options and optimization tools as well [53]. Deeplearning4j [55] is another tool also providing deep learning but it is used for business purposes rather than research. SAMOA [56] was developed by Yahoo, Barcelona. SMOA is a framework and can also be used as a library [56], providing streaming algorithms for machine learning tasks like clustering, classification and regression, along with providing an opportunity for the user to develop new algorithms. Flink-ML is also a machine learning library used with the Flink processing engine, is capable of logistic regressions, K-Means clustering and DSL for linear algebra. Other machine learning libraries for Big Data include Oryx [57] and Vowpal Wabbit [58]. Oryx provides some basic machine learning tasks to perform on large scale data while Vowpal Wabbit is a fast online learner, developed by Yahoo.

2.3.2 Summary

This section has given a brief introduction to machine learning techniques and of the different technologies that support distributed machine learning.

Chapter 3

An efficient strategy for the collection and storage of large volumes of data for computation

In recent years, there has been an increasing amount of data being produced and stored. The social networks, internet of things, scientific experiments and commercial services play a significant role in generating a vast amount of data. Three main factors are important in Big Data; Volume, Velocity and Variety. One needs to consider all three factors when designing a platform to support Big Data. The velocity of the data produced by a scientific infrastructure (e.g WLCG) that are propagated will be extremely fast. Traditional methods of collecting, storing and analysing data have become insufficient in managing the rapidly growing volume of data. Therefore, it is essential to have an efficient strategy to capture these data as they are produced. In this chapter, a number of models are explored to understand what should be the best approach for collecting and storing Big Data for analytics. An evaluation of the performance of full execution cycles of these approaches on the monitoring of the WLCG infrastructure for collecting, storing and analysing data is presented. Moreover, the models discussed are applied to a community driven software solution, Apache Flume, to show how they can be integrated, seamlessly.

3.1 Introduction

The field of data science has become a widely discussed topic in recent years due to a data explosion, especially with scientific experiments such as those that are part of the LHC at CERN and commercial businesses keen to enhance their competitiveness by learning about their customers to provide tailor made products and services, dramatically increasing the usage of sensor devices. Traditional techniques of collecting (e.g. lightweight Python framework), storing (e.g. Oracle) and analysing (e.g. PL/SQL) data are no longer optimal with the overwhelming amount of data that are being generated. The challenge of handling big volumes of data has been taken on by many companies, particularly those in the Internet domain, leading to a full paradigm shift in methods of data archiving, processing and visualisation. A number of new technologies have appeared, each one targeting specific aspects of large-scale distributed data-processing. All these technologies, such as batch computation systems (e.g. Hadoop) and non-structured databases (e.g. MongoDB), can handle very large data volumes with little financial cost. Hence, it becomes necessary to have a good understanding of the currently available technologies to develop a framework which can support efficient data collection, storage and analytics.

The core aims of the presented study were the following:

- To propose and design efficient approaches for collecting and storing data for analytics that can also be integrated with other data pipelines seamlessly.
- To implement and test the performance of the approaches to evaluate their design.

3.2 Background

Over the past several years there has been a tremendous increase in the amount of data being transferred between Internet users. Escalating usage of streaming multimedia and other Internet based applications has contributed to this surge in data transmissions. Another facet of the increase is due to the expansion of Big Data, which refers to data sets that are many orders of magnitude larger than the standard files transmitted via the Internet. Big Data can range in size from hundreds of gigabytes to petabytes [59].

Within the past decade, everything from banking transactions to medical history has migrated to digital storage. This change from physical documents to digital files has necessitated the creation of large data sets and consequently the transfer of large amounts of data. There is no sign that the continued increase in the amount of data being stored and transmitted by users is slowing down. Every year Internet users are moving more and more data through their Internet connections. With the growth of Internet based applications, cloud computing, and data mining, the amount of data being stored in distributed systems around the world is growing rapidly. Depending on the connection bandwidth available and the size of the data sets being transmitted, the duration of data transfers can be measured in days or even weeks. There exists a need for an efficient transfer technique that can move large amounts of data quickly and easily without impacting other users or applications [59].

In addition to corporate and commercial data sets, academic data are also being produced in similarly large quantities [60]. To give an example of the size of the data sets utilised by some scientific research experiments, a recent study observed a particle physics experiment (DZero) taking place at the Fermi Lab research center. While observing the DZero experiment between January 2013 and May 2015, Aamnitchi et al. [60] analysed the data usage patterns of users. They found that 561 users processed more than 5 petabytes of data with 13 million file accesses to more than 1.3 million distinct data files. An individual file was requested by at most 45 different users during the entire analysed time period.

There are many scientific research facilities that have similar data demands. The most popular and well known example today is the LHC at CERN where thousands of researchers in the fields of physics and computer science are involved with the various experiments based there. The experiments being conducted at the LHC generate petabytes of data annually [61, 62]. One experiment, ALICE, can generate data at the rate of 1.25 GB/s. Figure 3.1 illustrates the growth in the size of data sets being created and stored by CERN. This graph shows the total amount of storage (both disk and tape) utilised by all of the top-level servers in the CERN organisation. The amount of data stored in the system has grown at a steady pace over the past 3 years and is expected to grow faster

now that the intensity of their experiments is increasing, which will result in more data collected per second [13].

Geographically dispersed researchers eagerly await access to the newest datasets as they become available. The task of providing and maintaining fast and efficient data access to these users is a major undertaking. Also, monitoring computing behaviours in the WLCG, such as data transfer, data access, and job processing, is crucial for efficient resource allocation. This requires the gathering of metadata which describes the data (e.g. transfer time) from geographically distributed sources and the processing of such information to extract the relevant information for the WLCG group [63]. Since the LHC experiments are so well known and many studies have been conducted on their demands and requirements, one can use the LHC experiments as a suitable case study for this research.

To meet the computing demands of experiments like those at the LHC, a specialised distributed computing environment is needed. Grid computing fits the needs of the LHC experiments and other similar research initiatives.

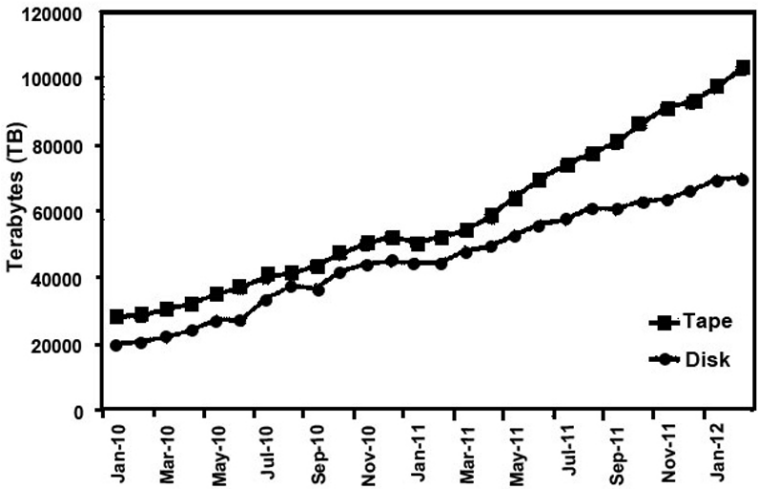


Figure 3.1: Size of CERN LHC experimental data sets over the past years. The total disk and tape storage amounts aggregated for all Tier-1 locations in the CERN grid (adapted from [13]).

The WLCG was created by CERN in 2002 in order to facilitate the access and dissem-

ination of experimental data. The goal of the WLCG is to develop, build, and maintain a distributed computing infrastructure for the storage and analysis of data from LHC experiments [64]. The WLCG is composed of over a hundred physical computing centers with more than 100,000 processors [14]. Since the datasets produced by the LHC are extremely large and highly desired, the WLCG utilises replication to help meet the demands of users. Copies of raw, processed, and simulated data are made at several locations throughout the grid.

The WLCG utilises a four-tiered model for data dissemination. The original raw data is acquired and stored in the Tier-0 center at CERN. This data is then forwarded in a highly controlled fashion on dedicated network connections to all Tier-1 sites. The Tier-1 sites are located in Canada, Germany, Spain, France, Italy, Nordic countries, Netherlands, Republic of Korea, Russian Federation, Taipei, United Kingdom and USA (Fermilab-CMS and BNL ATLAS).

The role of the Tier-1 sites varies according to the particular experiment, but in general they are responsible for managing permanent data storage (of raw, simulated, and processed data) and providing computational capacity for processing and analysis [64]. The Tier-1 centers are connected with CERN through dedicated links (Figure 3.2) to ensure high reliability and high-bandwidth data exchange, but they are also connected to many research networks and to the Internet [14]. The underlying components of a Tier-1 site consist of online (disk) storage, archival (tape) storage, computing (process farms), and structured information (database) storage. Tier-1 sites are independently managed and have pledged specific levels of service to CERN.

The Tier-2 sites were originally used for Monte Carlo event simulation and for end-user analysis. Any data generated at Tier-2 sites is forwarded back to Tier-1 centers for archival storage.

Other computing facilities in universities and research laboratories are able to retrieve data from Tier-2 sites for processing and analysis. These sites constitute the Tier-3 centers, which are outside the scope of the controlled WLCG project and are individually

maintained and governed. Tier-3 sites allow researchers to retrieve, host, and analyse specific datasets of interest. Freed from the reprocessing and simulation responsibilities of Tier-1 and Tier-2 centers, these Tier-3 sites can devote their resources to their own desired analyses and are allowed more flexibility with fewer constraints [65]. As there are thousands of researchers eagerly waiting for new data to analyse, many users will find less competition for time and resources at Tier-3 sites than at the Tier-2 sites.

It is important to note that users connecting to either Tier-2 or Tier-3 sites will use public, shared network connections, including the Internet. Grid traffic and normal World Wide Web traffic will both be present on these shared links. A user will also be sharing the site that they access with multiple other users. These factors can affect the performance of the data transfer between the selected retrieval site and the user. Retrieving these large data files also places a burden on shared resources and impacts other grid and non-grid users. When it comes to retrieving data in the WLCG, a normal user (depending on their security credentials) can access data on either Tier-2 or Tier-3 sites. Selecting a site to utilise can be a complicated task, with the performance a user obtains being dependent on the location chosen.

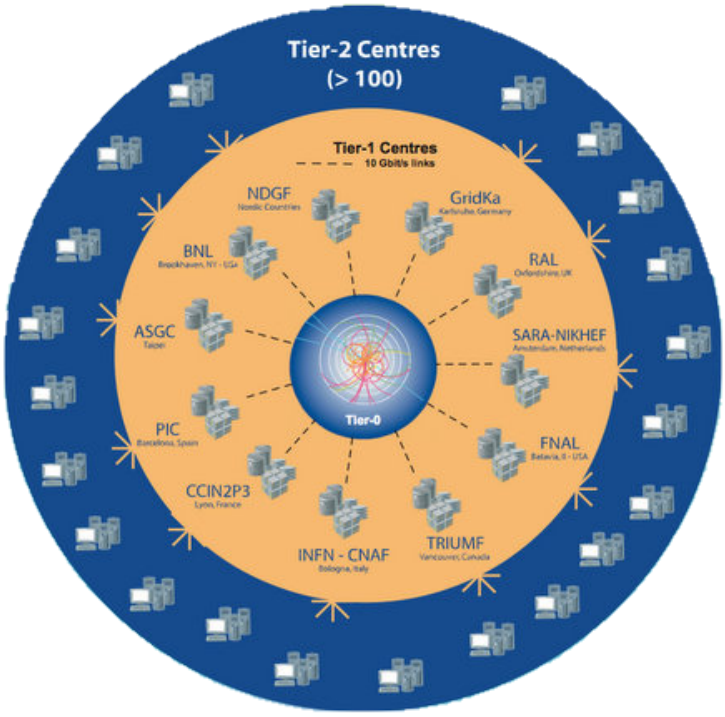


Figure 3.2: WLCG Tier-1 and Tier-2 connections [14].

Grid computing has emerged as a framework for aggregating geographically distributed, heterogeneous resources that enables secure and unified access to computing, storage and networking resources for Big Data [66]. Grid applications have vast datasets and/or carry out complex computations that require secure resource sharing among geographically distributed systems.

Grids offer coordinated resource sharing and problem solving in dynamic, multi-institutional virtual organisations [67]. A virtual organisation (VO) comprises a set of individuals and/or institutions having access to computers, software, data, and other resources for collaborative problem-solving or other purposes [68]. A grid can also be defined as a system that coordinates resources that are not subject to centralised control, using standard, open, general-purpose protocols and interfaces in order to deliver nontrivial qualities of service [69].

A number of new technologies have emerged for handling big-scale distributed data-

processing, (e.g. Hadoop), where the belief is that moving computation to where data reside is less time consuming than moving data to a different location for computation when dealing with Big Data. This is certainly true when the volume of data is very large because this approach will reduce network congestion and improve the overall performance of the system. However, a key grid principle contradicts with this as in the grid approach computing elements (CE) and storage elements (SE) should be independent, although this is changing in modern grid systems. Currently, a lot of scientific experiments are beginning to adopt the "new" Big Data technologies, in particular for metadata analytics at the LHC, hence the reason for the presented study.

Parallelising is used in order to enhance computations of Big Data. The well known MapReduce [46] framework that has been used in this chapter has been well developed in the area of Big Data science and has the parallelisation feature. Its other key features are its inherent data management and fault tolerant capabilities.

The Hadoop framework has also been employed in this chapter. It is an open-source MapReduce software framework. For its functions it relies on the HDFS [70], which is a derivative of the Google File System (GFS) [71]. In its function as a fault-tolerance and data management system, as the user provides data to the framework, the HDFS splits and replicates the input data across a number of cluster nodes.

The approaches for collecting and storing Big Data for analytics described in this chapter were implemented on a community-driven software solution, Apache Flume, in order to understand how the approaches can be integrated seamlessly in the data pipeline. Apache Flume is used for effectively gathering, aggregating, and transporting large amounts of data. It has a flexible and simple architecture, which makes it fault tolerant and robust with tunable reliability and data recovery mechanisms. It uses a simple extensible data model that allows online analytic applications [72].

3.3 Design and methodology

When data messages are consumed from a data transport layer and written into storage, there will most likely be some sort of data transformation carried out before storage in the storage layer. Such a transformation could be extracting the body from the message and removing the header as it is not required, or serialisation or compression of the data. The WLCG uses a Python agent, the Dashboard consumer, to collect infrastructure status updates, transform them, and store them in the Data Repository, which is implemented in Oracle. It uses Procedural Language/Structured Query Language (PL/SQL) procedures for analytics. This is an example of a traditional approach that is commonly used. However, these technologies and methods are no longer optimal for data collection, storage and analytics as they are not primarily designed for handling Big Data. There needs to be a strategy in place to carry out the required transformation as this will play a significant role in improving the performance of subsequent computations. In this chapter three different approaches were explored:

1. Implement the data transformation logic within the data pipeline. Therefore, the messages, M, will be read by the consumer, to apply the transformation $\langle T \rangle$ and to write the results into the storage layer, S, for analytics $\langle A \rangle$:

$$M \xrightarrow{\langle T \rangle} S \rightarrow \langle A \rangle \tag{3.1}$$

2. Write the raw messages, M, directly into the storage layer, S, without any modification. Then there is another intermediate transformation $\langle iT \rangle$ that reads the raw data from storage, transforms the data and writes the results into a new path but to the same storage layer for analytics $\langle A \rangle$:

$$M \rightarrow S \xrightarrow{\langle iT \rangle} S \rightarrow \langle A \rangle \tag{3.2}$$

3. Write the raw messages, M, into the storage layer, S, without any modification. Let

the analytics $\langle A \rangle$ carry out the transformation $\langle T \rangle$:

$$M \rightarrow S \rightarrow \langle \langle T \rangle + \langle A \rangle \rangle \tag{3.3}$$

The first approach is the traditional way of transforming, storing and computing data as has been already described for the WLCG use case. However, this method relies too much on the data pipeline. If the data pipeline is replaced then the transformation logic would need to be re-implemented. Therefore, it is an inefficient design. Nevertheless, this method needs to be tested on the technology that supports Big Data.

The second approach has two benefits as the transformation logic is moved to a centralised location and untampered raw data are stored as well as the transformed data. Therefore, in the case of any inaccuracy in the transformed data, the correct transformed data can be recreated from the raw data. This is not possible with the first method because as soon as the data are transformed the raw data are discarded. Nevertheless, the second approach is very complex as there is a requirement for a job to transform the data, rather than the consumer carrying out the transformation, and it raises the question of when and how this job should be scheduled. This approach also requires increased data storage as both raw and transformed data will be kept. A transformation job could be used here to compress the raw data and archive it to reduce the amount of storage required.

The third option is very simple and straight forward, as the raw data will be written into the storage layer without any modification. The transformation will only take place at the data analytics time. The transformation logic can be implemented in a shared library, which can be imported into any analytics jobs. Therefore, the transformation will take place as and when it is required. This way, the untempered raw data is still kept in the storage layer and no additional job or storage is needed for data transformation. This approach does add an extra execution time overhead to the analytics jobs and will repeat the data transformation every time an analytics job is carried out. This should, however, not be too much of a problem as Big Data technologies are built to enhance computation speed by parallelising jobs. Hence, this arrangement should not significantly affect the

execution time. A summary of the advantages and disadvantages of the proposed three approaches is given in Table 3.1.

3.3.1 Implementation

The data pipeline presented in this chapter uses the Dirq library that offers a queue system, using the underlying file system for storage for consuming messages, which allows concurrent read and write operations [73]. Therefore, it can support a variety of heterogeneous applications and services that can write messages and have multiple readers reading the messages simultaneously. The data pipeline was developed using the Hadoop native library that reads messages from the Dirq queue and writes them into HDFS using an appending mechanism. The Hadoop software framework was originally designed as a create-once-read-many system [18]. Therefore, appending was not available in the initial software release but later versions, 2.0 onwards, supported this mechanism. Hadoop also has the benefit of working well with a few large files but is not as efficient when working with a large number of small files. The appending method is convenient as it allows for the creation of a single large file.

For the first approach, the data will be consumed from the Dirq, transformed and written into HDFS. The implementation of the second approach is similar to the first with the exception of no transformation being carried out in the pipeline. However, it requires chained MapReduce jobs in a centralised Hadoop cluster in order to take the raw data that has not previously been processed, and apply the appropriate data transformation, merge the transformed data with previously transformed data, delete the old transformed data, update the raw data as processed and merge and compress the raw data. An issue was encountered during testing of this second approach where it was found that data that were not processed by the transformation job were not then available for analytics. The third approach is again like the second approach in that no transformation is carried out in the data pipeline, but the transformation logic is implemented in a common library and is available to be imported into any analytics jobs. Therefore, the transformation can be carried out as and when it is required. This approach does not have the issue of data

Table 3.1: Summary of advantages and disadvantages of the proposed approaches.

	Advantage	**Disadvantage**
Approach 1 Data transformation occurs within the data pipeline.	- Well tested approach: typical scenario in most data analytics platforms.	- Complex: transformation logic is kept in the data pipeline so in the case of data pipeline replacement the transformation logic needs to be re-implemented. - Lost data authenticity: the data is transformed by the data pipeline so the raw data is lost.
Approach 2 Data transformation occurs within the storage layer.	- Easy to migrate/replace: the transformation logic is moved to a centralised location so it is easier to migrate or replace the data pipeline. - Raw data is intact: meets regulatory standards of storing the raw data both before and after transformation.	- Complex: an intermediate job is required for transformation. - Large storage needed: both raw and transformed data are stored.
Approach 3 Data transformation occurs within the analytics jobs.	- Clean and simple: no complexity added to the data pipeline. - Less storage needed: only raw data is stored. - Easy to migrate or replace: the transformation logic is moved to a centralised location.	- Increased execution overhead: the analytics job will transform the data. - Repetition: transformation will take place every time an analytics job is executed

unavailability as present in the other two approaches as all written data will be picked up by the analytics jobs and the transformation will be done as and when required. All three approaches were implemented as a daemon that continuously ran on the WLCG test infrastructure checking for data every 5 minutes.

In order to decrease the data aggregation delay from the data pipeline and to evaluate how easy it is to migrate these approaches to a different data pipeline, Apache Flume was used. Apache Flume is a community driven software solution that receives messages from the transport layer and writes them into HDFS. There are three ways to flush consumed data into HDFS: periodically based on the elapsed time, the size of data or the number of events [72].

As expected, the first approach was complex as all the transformation logic was in the custom data pipeline so the transformation logic had to be re-implemented into Apache Flume. The second and third approach made the migration to Apache Flume extremely simple, as all the transformation logic was implemented within the storage layer. But, as noted before, the second approach added complexity to the storage layer, as it required a chain of actions for data transformation. The third approach was the simplest to implement, as no transformation was carried out on the Apache Flume side and no transformation was carried out in the storage layer, keeping the complexity low.

All three approaches did encounter a common problem: Apache Flume pushes the events but does not flush the file until the configured file roll time is met (e.g. every 1 hour) resulting in the data being unavailable for computation between these times. While HDFS supports appending functionality, and the custom data pipeline, Apache Flume does not support it. The analytics jobs were able to read the data that were written by the custom data pipeline but not those written by Apache Flume. Therefore, the appending functionality was taken from the custom pipeline and implemented into Apache Flume, making it a custom library (see Algorithm 3.1). With this amendment, Apache Flume was then able to write a single file and append it while at the same time analytics jobs were able to read the data while the data were being written into HDFS.

Algorithm 3.1 File appending algorithm for Apache Flume: adding a close and reopen at every push to get the required append behaviour.

```
1: procedure CREATE-GLOBAL-DATA-FILE-WRITER
2:     declare a global DataFileWriter object
3:     create a file in HDFS
4:     initialise the file to global DataFileWriter

1: procedure CONSUME-MESSAGES-AND-SYNC-FLUSH
2:     create a temp DataFileWriter refelecting(reopen) the global DataFileWriter
3:     consume all messages
4:     append messages using temp DataFileWriter
5:     close temp DataFileWriter WHEN messages <= 0

1: procedure ROLL-FILES
2:     close the global DataFileWriter
```

3.4 Results and discussion

The three approaches developed for the collection, storage and analytics of Big Data described in this chapter were evaluated on the WLCG infrastructure that provides the computing resources to store, distribute and analyse the 30 petabytes of data generated annually by the LHC and distributed to 170 computing centres around the world [74]. Furthermore, the current method used by the WLCG group for the collection and storage of data for analytics was evaluated for benchmarking the new approaches.

It was very complicated to carry out performance measurements on the proposed approaches and the current approach, as they employ different methods for consuming, writing and transforming the data in each case. Therefore, in order to get a meaningful performance measurement, a full computation cycle was carried out, including: consuming messages, writing to HDFS and carrying out a simple analytics job on those data. The full cycle comprised three segments:

1. Data ingestion with data transformation and without data transformation.
2. Intermediate data transformation using a MapReduce job.
3. A simple statistical analytic computation using a MapReduce job and a PL/SQL procedure with and without data transformation.

The configurations of the current and proposed data pipelines in the WLCG are shown in Figure 3.3 (a) and (b) respectively. For both configurations, the monitoring events are pushed as JavaScript Object Notation (JSON) records through the STOMP protocol to the ActiveMQ message broker. However, the configuration varies from the consumers in both data pipelines. The current configuration uses Python collectors for reading the monitoring events, transforming and writing them into an Oracle storage database. On the other hand, the proposed configuration uses a custom data pipeline daemon, as explained in Section 3.3.1 that reads monitoring events and writes them into a Hadoop cluster. This configuration can be modified to support the three proposed approaches, i.e. transform and serialise the messages into Avro format.

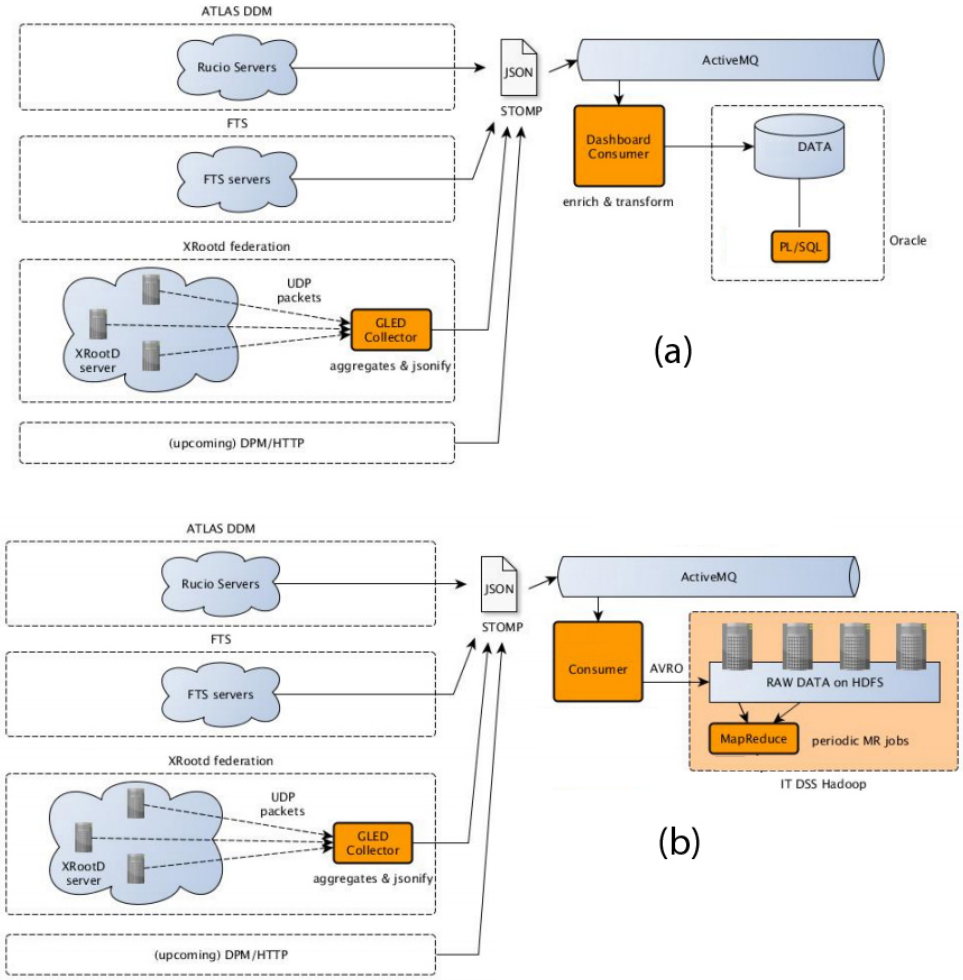


Figure 3.3: Configuration of current data pipeline in WLCG (a) and the configuration of the proposed data pipeline for WLCG (b).

In order to evaluate the proposed approaches, it was decided to push messages from the broker in batch sizes ranging from 10,000 to 100,000 messages. Data ingestion and analytics were conducted ten times for each batch of messages in order to capture an average performance time. The performance measurements were carried out on a heterogeneous Hadoop cluster that consisted of 15 nodes (8 nodes: 32 cores/64 GB, 7 nodes: 4 cores/8 GB).

3.4.1 Performance results of data ingestion with and without data transformation

The first approach had to consume all messages from Dirq, apply a simple data transformation, which involved taking the source and destination IP address from the message and using a topology mapping file to determine the domain address and replace the IP address with the domain, and finally, convert the data file into Avro format, which is a data serialisation framework that serialises the data into a compact binary format and writes the file into HDFS. As shown in Figure 3.4, this approach (pre-trans-avro) is slower than the second approach (raw-json), which just reads the raw messages in JSON, an easy-to-read format, and writes them into HDFS. The second approach is the fastest compared with the first and third approaches (raw-avro), which read raw data, convert them into Avro format and write them into HDFS. The third approach was faster than the first approach because it does not do any transformation.

The current approach (pyth-traditional-plsql) used by the WLCG is similar to the proposed first approach (pre-trans-avro) but the difference is that it uses the Python agent for collection and the Oracle database for storing the transformed data, so no serialisation is involved. Although the current approach is similar to the first of the three proposed approaches the performance of the current approach was slower than all three of the newly proposed approaches. This is due to the connection and communication limitations that occurs between the database and collectors.

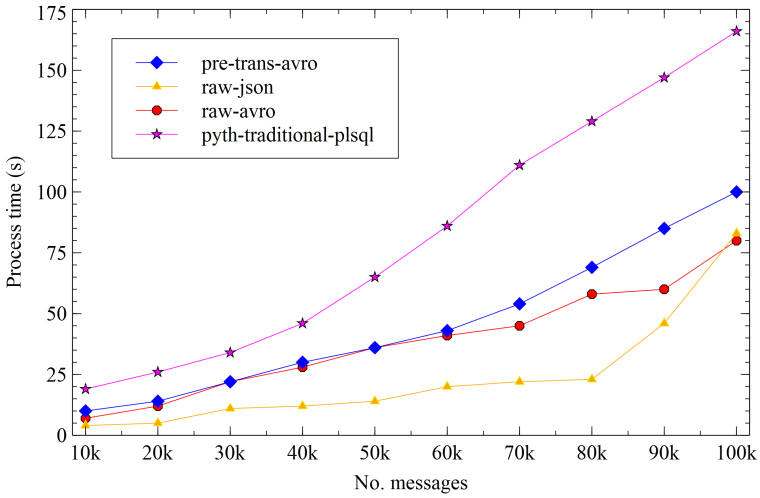


Figure 3.4: Data ingestion from message queue to HDFS with and without data transformation.

Figure 3.5 shows data representing unprocessed messages from the broker, raw JSON messages, a pre-transformed Avro and a raw Avro file written into HDFS by the custom data pipeline. The Avro files are smaller than the JSON file and contain unprocessed data because they are serialised into binary format. However, the pre-transformed Avro file is larger than the raw Avro file because transformation was applied.

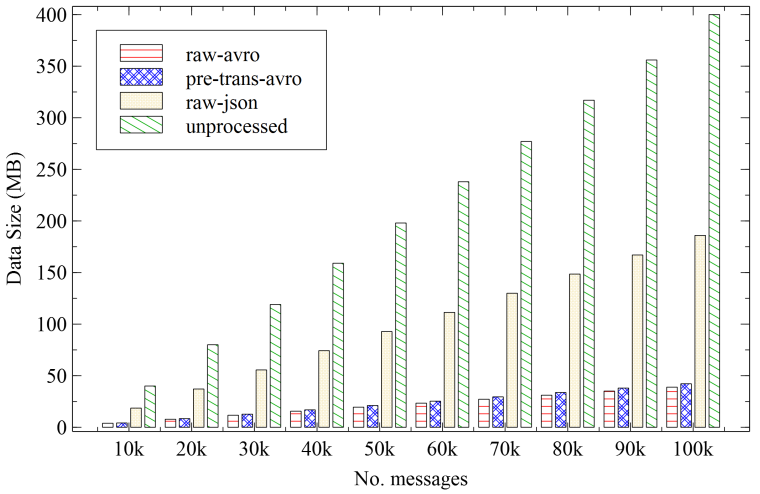


Figure 3.5: Data size of the messages that were stored into HDFS with and without data transformation.

3.4.2 Performance results of intermediate data transformation using a MapReduce job

A test was designed to measure the performance of an intermediate MapReduce transformation done on a centralised Hadoop cluster. As shown in Figure 3.6, only the raw JSON data will go through this transformation, as the pre-transformed Avro file has already been transformed at the data pipeline level and the raw Avro data will be transformed at the analytic time when it is required. Also, the data stored in the database by the Python agent does not require an intermediate transformation as it has already been performed at the data pipeline. Transforming the data using an intermediate job is very expensive in terms of execution time, as the process is carried out by chained MapReduce jobs that will transform, aggregate and merge the data. The majority of the execution time overhead was used for finding resources and submitting the chained jobs to the Hadoop cluster.

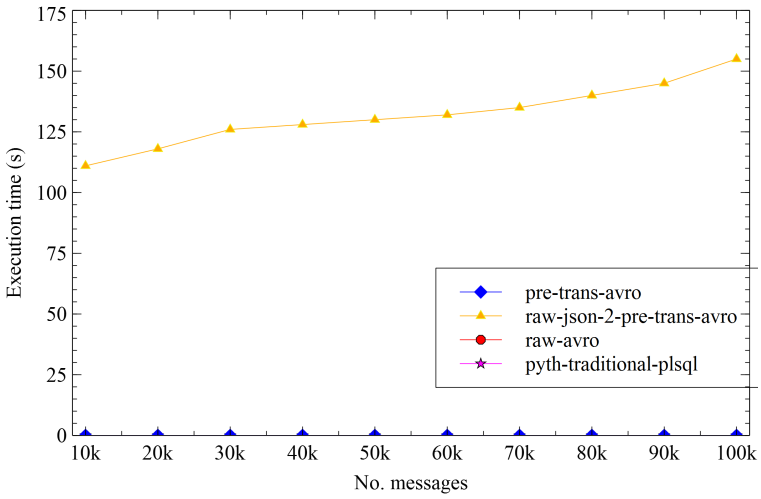


Figure 3.6: Intermediate MapReduce job for data transformation. Only the raw JSON messages are transformed with the MapReduce job.

3.4.3 Performance results of a simple analytic computation with and without data transformation

The final step of the evaluation cycle was to carry out a simple computation on the 100,000 messages dataset and measure the performance. Two sets of analytics jobs were implemented to compute a summary view of the XRootD operations, performed by the different

users for each WLCG site belonging to the XRootD federation [74]. An analytics job was modified to include the data transformation prior to the computation. The modified job was executed on the raw Avro data. As shown in Figure 3.7, an extra execution time overhead was added to the modified analytics job when compared with unmodified job that computed pre-transformed data, but the computation was seamless, as the MapReduce framework adopts a parallel programming model. Therefore, the jobs will be split into multiple tasks and will be sent to data nodes where the data reside. The current approach used by the WLCG (pyth-traditional-plsql) for analytics was very slow compared with the proposed approaches due to the constraints imposed by the database being used and its lack of scalability.

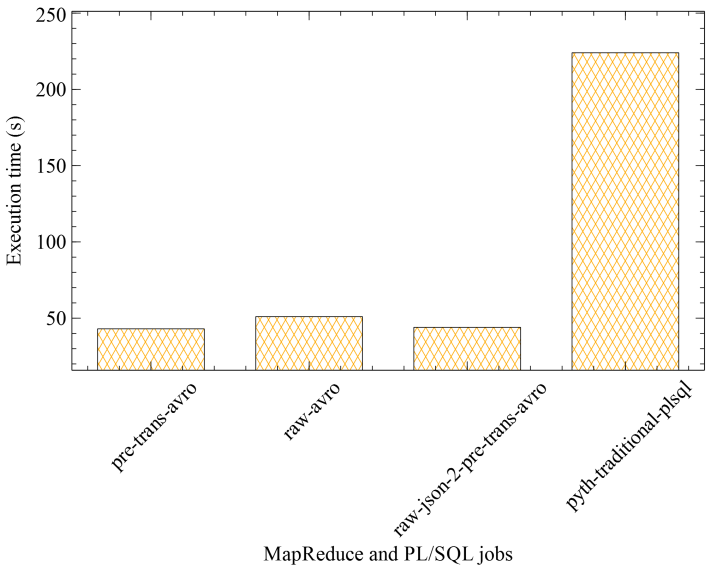


Figure 3.7: Performance measurements of the statistic computation were done on pre-transformed and the raw 100,000 messages dataset.

3.4.4 Summary of the performance results

In order to understand which approach performed better, the execution time of the largest dataset of 100,000 messages was selected from Sections 3.4.1, 3.4.2 and 3.4.3 and the total is presented in Table 3.2. It is clear that in this case writing the raw Avro data into HDFS and allowing the analytics to execute the transformation outperforms the other two proposed approaches. The slowest of the proposed approaches is the second approach where

there is an intermediate job for transformation. This is understandable as the transformation is carried out by chained MapReduce jobs, which add extra execution time overhead. The first approach is comparable in terms of performance to the third approach but it will be beneficial to keep a copy of the untempered raw data file in HDFS and let the analytics job do the transformation, which is better than carrying out transformation in the data pipeline as the authenticity is lost once the transformation is done and stored in HDFS. Although the current approach used by the WLCG employs the same pre-transformation approach, it performs inadequately compared with the new approaches presented in this chapter, primarily due to database communication and scalability constraints as the current approach cannot handle the increasing data and workload.

Table 3.2: Total sum of execution time for 100,000 messages dataset from Sections 3.4.1 , 3.4.2 and 3.4.3.

Data Transformation	**Section 3.4.1 Execution time (s)**	**Section 3.4.2 Execution time (s)**	**Section 3.4.3 Execution time (s)**	**Total Execution time (s)**
pre-trans-avro-mr	100	0	43	143
raw-json-2-pre-trans-avro-mr	83	155	44	282
raw-avro-mr	80	0	51	131
pyth-traditional	166	0	224	390

3.4.5 Evaluation of Apache Flume

During the evaluation of all three proposed approaches there was still a 5 minute delay in polling data from the message queue. In order to eliminate this polling latency, custom made Apache Flume data collectors (as explained in Section 3.3.1) that utilise an appending mechanism were put in place of the consumer shown in Figure 3.3 (b). The performance test results showed that the third approach is optimal. Therefore, Apache Flume agents were configured to consume messages and flush them into HDFS directly. Figure 3.8 shows spikes in the total number of messages propagated with a rate $>$ 1 kHz,

and it can be seen that Apache Flume seamlessly absorbs the load on its single virtual machine. Meanwhile, the current Python-Oracle based consumers used by the WLCG, running on two production virtual machines, were struggling to keep up, causing a backlog of message stored in the broker.

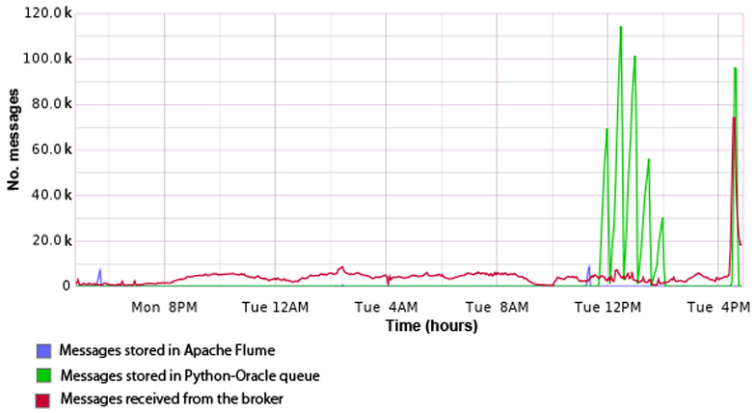


Figure 3.8: Spikes of messages with a rate greater than 1 kHz. The red line is the messages received from the broker, green denotes the messages stored in old consumers, and blue denotes the messages stored in Apache Flume.

3.5 Summary

The proposed approaches for collecting and storing Big Data for analytics presented in this chapter show how important it is to select the correct model for efficient performance and technology migration. It is clear from the study that keeping the main logic in a centralised location will simplify technological and architectural migration. The performance test results show that eliminating any transformation at the data ingestion level and moving it to the analytics layer is beneficial as the overall process time is reduced, untempered raw data are kept in the storage level for fault-tolerance, and the required transformation can be done as and when required using a framework such as MapReduce. The presented results show that this proposed approach outperformed the approach employed at the WLCG and following this work the new approach has been adopted by the WLCG and it has been used for collecting, storing, and analysing metadata at CERN since April 2015 [63]. This approach can be easily applied to other use cases (e.g. in commercial businesses

for collecting customer interest datasets) and is not restricted to scientific applications.

Chapter 4

Monitoring scientific infrastructure with the Lambda architecture

Monitoring computing activities in a scientific infrastructure, such as job processing, data access and transfers or site availability, requires the gathering of monitoring data from geographically-distributed sources and the processing of such information to extract the relevant value for scientists, computing teams and site operators. Traditional monitoring systems have proven to be a solid and reliable solution in the past for monitoring scientific infrastructure. Nevertheless, the current architecture, where relational database systems are used to store, to process and to serve monitoring data, has limitations in coping with the foreseen increase in the volume and the variety (e.g. new data-transfer protocols and new resource-types, such as cloud-computing) and velocity of monitoring events. This chapter presents a new data store and analytics platform using the Lambda approach [19], which was evaluated with the WLCG activities monitoring use case on WLCG infrastructure.

4.1 Introduction

In general, traditional architecture is used for monitoring scientific infrastructure (e.g. WLCG), which relies on an Oracle database to store, to process and to serve the monitoring

data. Raw monitoring events are archived in strctured tables for several years, periodic PL/SQL jobs run at regular interval (e.g. 10 minutes) to transform the fresh raw data into summarised time-series statistics and feed them into dedicated tables, from where they are exposed to the web-framework for user visualisation. For data intensive use cases, this approach has several limitations. Scalability is difficult to achieve, PL/SQL execution time fluctuating from tens of seconds to minutes as a consequence of the input rate spikes, affecting user interface latency. Advanced processing algorithms are complex to implement in PL/SQL within the dashboard 10 minutes time constraint, and reprocessing of the full raw data can take days or weeks. Moreover, the other components involved in data collection, pre-processing and insertion suffer from fragility and complexity, leading to higher maintenance and operational costs and increased possibility of human error. Considering the foreseen increase in the monitoring data volume, variety and velocity of monitoring events, data storage and processing technologies that scale horizontally (the capability to expand capacity by joining various hardware or software so that they work as a single unit), have low-latency and high efficiency by design, such as Hadoop, are suitable candidates for the evolution of the monitoring infrastructure. However, a careful evaluation of efficient approaches that can be employed to benefit from these Big Data technologies is required as they work totally differently to the traditional technology. The aim of the work presented in this chapter was to architect a new data store and analytics platform, able to cope with the scalability, flexibility and fault-tolerance requirements foreseen in the near future of data monitoring applications.

4.2 The Lambda Architecture

In recent years, the challenge of handling a big volume of data has been taken on by many companies, particularly in the Internet domain, leading to a full paradigm shift in data archiving, processing and visualisation. A number of new technologies have appeared, each one targeting specific aspects of big-scale distributed data-processing. All these technologies, such as batch computation systems and non-structured databases, can handle very large data volumes with little time but with serious trade-offs such as high-latency.

The Lambda Architecture, presented by Marz [19], identified three main components

needed to build a scalable and reliable data processing system:

- the **batch layer**, to store a steadily growing dataset providing the ability to compute arbitrary functions on it;
- the **serving layer**, to save the processed views, using indexing techniques to make them efficiently query-able;
- the **real-time layer** able to perform analytics on fresh data with incremental algorithms to compensate for batch-processing latency.

4.2.1 Difference between common scientific use case and the classic Lambda use case

In the classic Lambda application each monitoring event only contributes to the most recent view (e.g. a web server user access for a web analytics application only affects the user count in the last time bin). For the scientific monitoring use case, this is not true. A monitoring event, such as a completed file transfer lasting several hours from site A to site B, contributes also to several time bins in the past, so that the information about the average traffic from site A to site B has to be updated accordingly with the new monitoring information. Without this initial hypothesis, the merging of batch and real-time processing becomes more complex.

4.3 A new data store and analytics platform for monitoring scientific infrastructure

The new data store and analytics platform for monitoring is presented in Figure 4.1 and it builds on a number of existing technologies and tools. The scientific monitoring problem has been treated as a pure analytics scenario where the driving concepts, as by the Lambda principles, are to collect and to store the raw data, to minimise pre-processing and to concentrate analysis and transformation on the same framework with batch and real-time components.

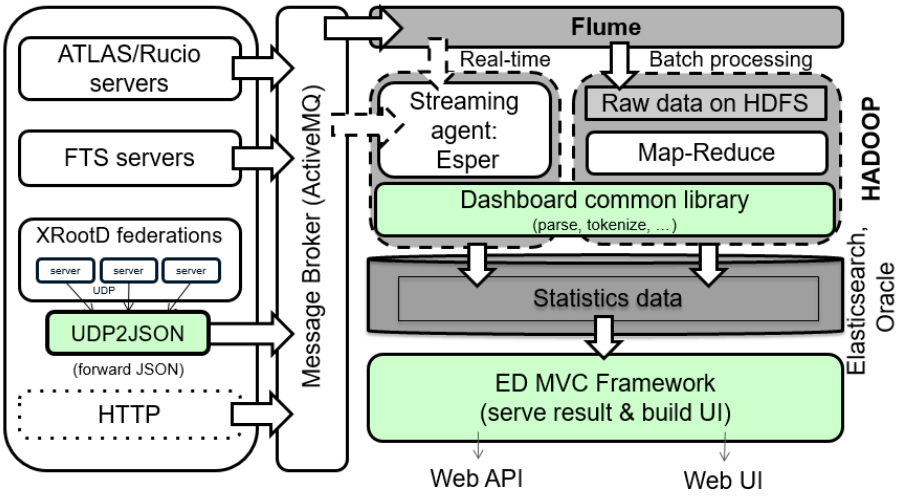


Figure 4.1: The new data analytics platform for monitoring a scientific infrastructurecture.

4.3.1 Data transport: Message Broker

The transport layer plays a key role in the new monitoring architecture. Firstly, it decouples the producer and the consumer of the monitoring data. Given that the information is produced by a variety of heterogeneous applications and services in scientific infrastructure, this is a fundamental part of the system functionality. Secondly, it allows multiple consumers to use the same data via on-demand public/subscribe API. This situation is often the case for monitoring data in a scientific environment. Thirdly, the architecture can rely on message brokers as it has durable virtual queue for recovering in the case of failure. The broker technology used was ActiveMQ and the monitoring events were reported as JSON records via the STOMP protocol.

4.3.2 Data collection: Apache Flume

Apache Flume is used as the data collector agent. It receives monitoring events from the transport layer and creates HDFS files in the archive layer for later processing. It used the custom-flume data pipeline described in Chapter 3, providing better performance and reliability. Flume connects to the brokers using the standard JMS source and writes to the storage layer via the standard HDFS sink.

4.3.3 Batch processing: Apache Hadoop

Hadoop is a distributed processing framework which allows the computation of large data sets on computer clusters built from commodity hardware. Initially focused mainly on batch-processing via MapReduce primitives [75], modern Hadoop supports multiple processing technology (e.g. Apache Samza). MapReduce is the de-facto standard for batch processing and its computation paradigm fits extremely well with the Lambda approach. The batch-processing was implemented as periodic MapReduce jobs running on the Hadoop infrastructure. The job algorithm was stateless and idempotent, the full data set which can contribute to the results (e.g. three days of data) being re-processed at each run. Job results were written into the serving layer (i.e. Elasticsearch) in the form of index (equivalent to RDBMS table) and documents (equivalent to RDBMS records), which were then used to build a web visualisation.

4.3.4 Archiving: HDFS

The Hadoop framework is built on the HDFS and executes I/O operations on it. HDFS is designed for large data files, in the order of GB in size. The data is broken into blocks and replicated across multiple hosts in the cluster. This guarantees scalability on commodity hardware, fault tolerance and high throughput. HDFS is data format independent, supporting multiple data representations, from simple text to structured binary.

4.3.5 The common data access service layer

A common drawback of the dual processing nature of the Lambda architecture is code duplication in the real-time and the batch processing layer. In order to limit this effect a Java data access service layer was developed to abstract the common functionalities. The data access service layer provides data parsing, supporting marshalling and un-marshalling in several formats, such as JSON, CSV and Avro, and also provides data validation and compression. Most importantly, it implements the algorithm to emit key-value pairs for each monitoring event received. The data access service layer played a major role in porting the jobs to different processing technologies (e.g. MapReduce, Spark) with minimal code change.

4.3.6 The serving layer: Elasticsearch

With the data archiving and processing delegated to the other components, the serving layer is solely responsible for serving the computed statistics to the web framework. In light of this simplified requirement, the serving layer can be easily implemented via non-relational technology. In the new data analytics platform the serving layer was implemented using Elasticsearch.

4.3.7 Real-time processing: Esper

The requirement for fast in-memory computation is not unique. It is needed in several fields such as financial analysis and wireless networks. Complex event processing (CEP) technologies were created in order to serve this need by processing streams of events at a high rate with low latency. The most widely adopted open source engine for this purpose is Esper, which is the main tool used for the development of the real-time layer. Esper, an open source event processing in-memory processing library [76], provides a streaming API for processing a continuous flow of events. Esper analyses data with Event Processing Language (EPL), which is SQL-like language as shown in Table 4.1, which offers much more advanced controls in processing streams over time. In contrast to the relational database model, Esper stores the query, which is continuously executed. It also stores the results if necessary but not the input data. Esper has been implemented to compute statistics on fresh monitoring events via an incremental algorithm. Being incremental hence not idempotent, special care is required in handling event duplication and multiple processing, leading to a more error prone computation. Esper was incorporated in the existing workflow to allow real-time computation and visualisation of fresh data and to speed up all the statistics generation.

4.4 Implementation of WLCG analytics on the new platform

In this section an implementation of a WLCG use case is presented.

Table 4.1: A mapping between Esper and the relational database model.

RDBMS	Esper
SQL	EPL
Rows	Events
Tables	Event Streams

4.4.1 WLCG data activities use case

The Experiment Dashboard (ED) [77] is a generic monitoring framework which provides uniform and customisable web-based interfaces for scientists and sites. Monitoring events, such as data transfers or data processing jobs reports, are collected and analysed to produce summary time-series plots used by operators and experts to evaluate WLCG computing activities. The WLCG Data acTivities (WDT) dashboards are a set of monitoring tools based on the ED framework which are used to monitor data access and transfer across WLCG sites via different protocols and services. Monitored services include the ATLAS Distributed Data Management (DDM) system, XRootD and HTTP federations and the File Transfer Service (FTS). The WDT use case is one of the most data intensive ED applications. Figure 4.2 presents the daily volume of monitoring information handled by WDT, with an overall average of more than 20 million daily monitoring events. Today, WDT dashboards are suffering from the limitation of the current processing infrastructure. For this reason, WDT was taken as a case study for the new analytics platform.

4.4.2 Implementation of the batch layer

The WLCG current monitoring architecture uses PL/SQL procedures for aggregating and computing raw data into statistics with different time period granularities and stores them into a statistics table. WLCG data servers can produce monitoring logs at 1 kHz, a rate at which that the PL/SQL procedure cannot cope with the overwhelming amount of data, which takes over ten minutes to process every ten minutes worth of data. The new analytics platform relies on Hadoop and its MapReduce framework, Elasticsearch and Esper to overcome the current latency and scalability issues. MapReduce is a programming paradigm that was designed to remove the complexity of processing data that are geo-

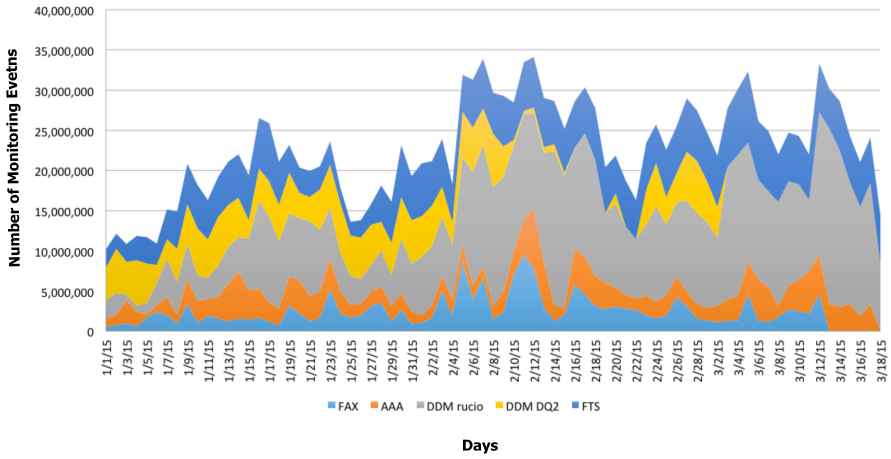


Figure 4.2: Daily volume of monitoring events from Federated ATLAS storage systems using XRootD (FAX), Anytime, Anywhere CMS storage systems using XRootD (AAA), ATLAS Distributed Data Management using Rucio (DDM rucio), ATLAS Distributed Data Management using Don Quijote (DDM DQ2) and File Transfer Service (FTS) for WDT dashboards [15].

graphically scattered around a distributed infrastructure [75]. It hides the complexity of computing in parallel, load balancing and fault tolerance over an extensive range of interconnected machines. There are two simple parallel methods, *map* and *reduce*, which are predefined in the MapReduce programming model and are user-specified methods that are used to develop the analytics platform.

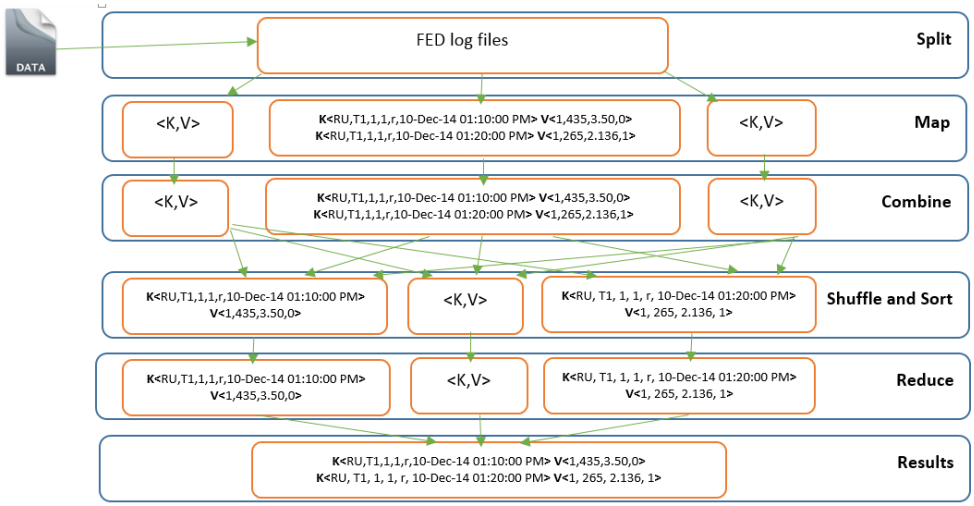


Figure 4.3: MapReduce computations diagram.

Figure 4.3 shows an how WDT MapReduce jobs are carried out by each component within the MapReduce framework:

1. A *splitter* will split the monitoring data into lines and feed them into mappers.
2. A *mapper* will process the line; breaking them into the time bins in which they belong and calculating the transfer matrices. Finally, it will emit key/value pairs for each time bin.
3. A *combiner* will run after each map task and aggregate a map output result, decreasing the number of metrics sent to the reducer.
4. The output of the combiner is then shuffled and transferred to a reducer that is responsible for processing the key and carrying out the final summing.
5. A *reducer* will aggregate and output the final results.

4.4.3 Data representation

In the current architecture the data are partitioned in HDFS, as shown in Figure 4.4, for efficient processing, as this will support the processing of data by specified date ranges.

```
|-- data
| |-- xrootd
| | |-- atlas
| | | |-- 2014
| | | | |--12
| | | | | |--01
| | | | | | | data.avro
```

Figure 4.4: HDFS data partitioning.

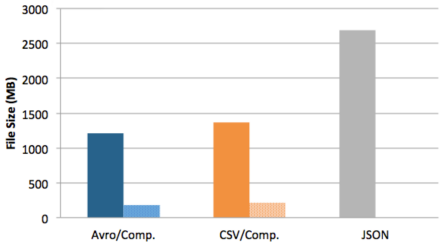


Figure 4.5: Data format comparison (Avro versus CSV versus JSON for 1 Day FAX data).

Three different data formats were evaluated for storing WDT monitoring data on HDFS:

1. Avro is a data serialisation framework that serialises the data into a compact binary format, so that it can be transferred efficiently across the network.
2. Comma-Separated Value (CSV) is a table format that maps very well to the tabular data.

3. JavaScript Object Notation (JSON) is primarily a way to store simple object trees.

Figure 4.5 shows data representing 1 (average) day of monitoring events for the ATLAS XRootD federation (FAX) on HDFS which occupies 1211 MB in Avro, 1367 MB in CSV and 2669 MB in JSON file format. As expected, the Avro format is more compact than CSV and JSON. This is because the binary version of Avro is used, whereas CSV comprises human readable comma-separated columns and JSON contains tree structured data. The JSON format was the largest because it holds both column name and data, whereas CSV format only holds the data separated by commas. This resulted in a 122.21% increase in volume for JSON data and a 12.88% increase in volume for CSV data compared with Avro, while there is a 96.85% increase in volume for JSON data compared with CSV. The data were also compressed using the Snappy compression library, which is very fast but the compaction ratio is very low compared with other libraries. Again, compressed Avro data takes up much less storage than the CSV as there is a 20.90% increase in volume. It can be seen that compressed data took over 5 times less space than uncompressed. The test results, combined with the additional benefits of being schema-based and its multi-language support, make Avro the preferred option for the WDT use case.

4.4.4 Implementation of the real-time layer

Architecture

The Esper layer architecture for monitoring scientific infrastructure is shown in Figure 4.6.

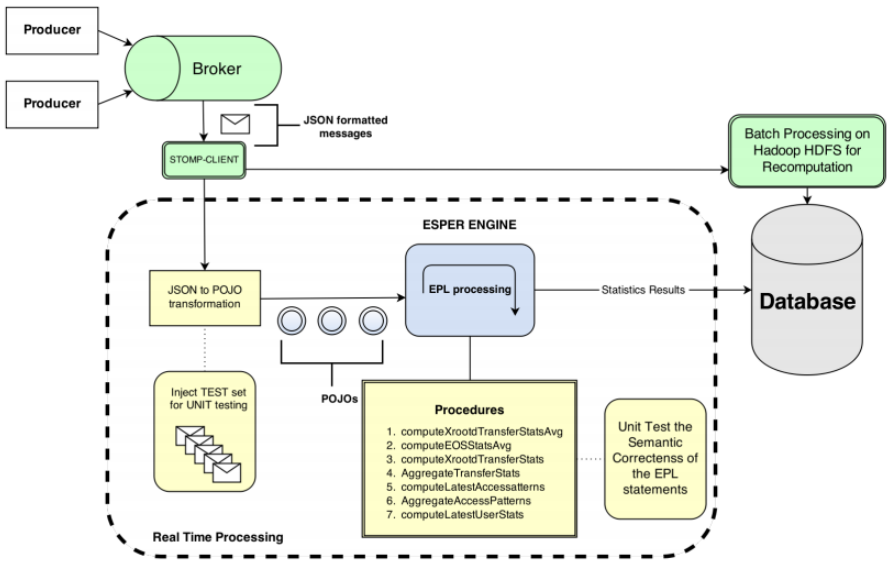


Figure 4.6: An overview of the system architecture with the Esper module [16].

JSON to POJO transformation

The data movement and job execution between different sites around collaborating countries is recorded into log messages [16]. These log messages store information such as the transaction start time, end time, source site, destination site, read bytes and write bytes to name a few [16]. Log messages distributed by the message broker were in JSON form as shown in Figure 4.6, which are not ready to be processed by Esper [16]. Therefore, some pre-processing is required before injecting them into the Esper engine [16]. The first step is to extract only the parts of the message that are needed and the second step is to transform the JSON file into a Plain Old Java Object (POJO) [16].

EPL processing

This is the module where the implemented EPL statements will continuously run. A listener is invoked periodically in order to check for incoming events and process them according to the EPL statements [16]. The implemented EPL procedures follow a Map-Reduce approach [16].

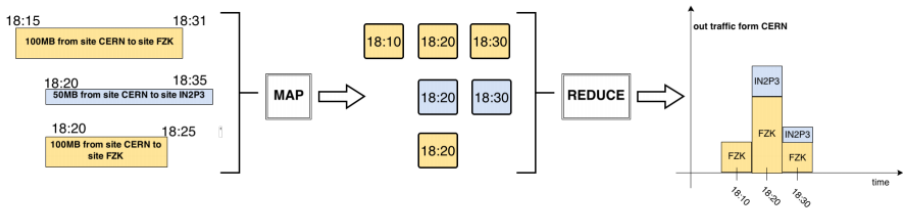


Figure 4.7: An example of the Map-Reduce approach on EPL statements implementation [16].

Figure 4.7 shows an example of the Map-Reduce implementation. Three different events, which represent transfers from the CERN site to two other sites (Karlsruhe Grid computing centre (FZK) and Institut national de physique nucleaire et de physique des particules (IN2P3) respectively), are injected into the Map statement [16]. The Map statement splits the incoming events into smaller pieces according to the time bins that they belong to [16]. For example, the first event belongs to the time bins 18:10, 18:20, 18:30, thus it is split into three different events, which are then injected into the Reduce statement, which aggregates them into the final results [16].

Example of an Transfer Statistics EPL statement

1. Inject Log Message event into Map statement.
2. Split the Log Message into several Log Map Events according to the time bins the initial event belongs.
3. Inject each of the Log Map Events into a Single Log Statistic Event and compute the following:
   (a) If writes bytes at close > 0 then we have a client domain else we have a server domain and set it as srcDomain.
   (b) If read bytes at close > 0 then we have a server domain else we have a client domain and set it as dstDomain.
   (c) If client domain=server domain set remote access 0 else set is as 1.
   (d) If writes bytes at close+read bytes at close=file size set is transfer=1 else is transfer=0.

(e) If read bytes at close > 0 then setactivity='r'.

(f) if write bytes at close > 0 then set activity='w'.
item if write bytes at close <= 0 and read bytes at close =< 0 then set activity='u'.

4. Aggregate all the single log statistics:

(a) If there is not already a time bin for the injected Single log statistic event then create it and insert:

i. srcDomain
ii. dstDOmain
iii. isRemoteAccess
iv. usrProtocol
v. isTransfer
vi. Activity
vii. periodEndTime
viii. active
ix. bytes
x. activeTime
xi. updateTime

(b) Else update the existing bin:

i. active=active+newSingleLogStatistic.active
ii. bytes=bytes+newSingleLogStatistic.bytes
iii. activeTime=activeTime+newSingleLogStatistic.bytes

4.4.5 Implementation of the serving layer

The serving layer is purely used for storing and serving statistics. The documents (records) are stored in Elasticsearch using bulk API and the UPSERT function for efficient insertion and merging of records. The merging of computed batch and real-time results is done on the client side to serve the computed statistics on the UI.

An example of a statistic document inserted into Elasticsearch:

```
{
        "_index": "dashboard_xrootd_2015",
        "_type": "transfer",
        "_id": "ATLAS|praguelcg2|n-a|1|0|r|1433251800000",
        "_score": 1,
        "_source": {
            "vo": "ATLAS",
            "src_domain": "praguelcg2",
            "dst_domain": "n/a",
            "is_remote_access": 1,
            "is_transfer": 0,
            "activity": "r",
            "period_end_time": "2015-06-02T09:30:00-04:00",
            "active": 1,
            "finished": 1,
            "bytes": 158387637,
            "active_time": 34,
            "update_time": "2015-07-13T11:41:06.762-04:00"
        }
    }
```


4.5 Performance results for WDT computation on the new platform

The Lambda approach was evaluated and used to compare the performance of different data types, data compression and data size with various cluster configurations such as scaling the nodes horizontally and parallelising the tasks. The performance was evaluated on monitoring events collected from the ATLAS FAX (which clusters Tier-1, Tier-2 and Tier-3 storage resources together into a common namespace and allows remote accessibility from geographically separated sites) and the CMS's implementation of a generic XrootD (protocol for accessing data) Any Data, Anytime, Anywhere (AAA) and EOS,

an XrootD managed disk pool. The data contained the information of dataset movement (e.g. transferred bytes).

4.5.1 Experiment setup

The performance measurements were carried out on a shared heterogeneous Hadoop cluster provided by WLCG, which consisted of 15 nodes of Intel(R) Xeon(R) CPU E5-2650 v2 @ 2.60GHz (8 nodes: 32 cores/64GB, 7 nodes: 4 cores/8GB). The analysis of the current architecture was carried on a high-performance physical setup and that was a shared service between all IT applications at WLCG, whereas the Lambda approach was run on a 15 node cluster. The Hadoop version 2.6 was configured on all machines; one of which is the Name Node and the rest are Data Nodes (the Name Node was also used as a Data Node). The data block size of the HDFS was set to 256 MB and the replication level of a given data block was set to 3. The transfer statistics computation described in the Implementation section is a CPU, Memory and IO intensive use case. Therefore, it was ideal for this evaluation. It should be noted that the cluster was being used by other members while the evaluation was carried out as it is a shared cluster as can been seen in Figure 4.8.

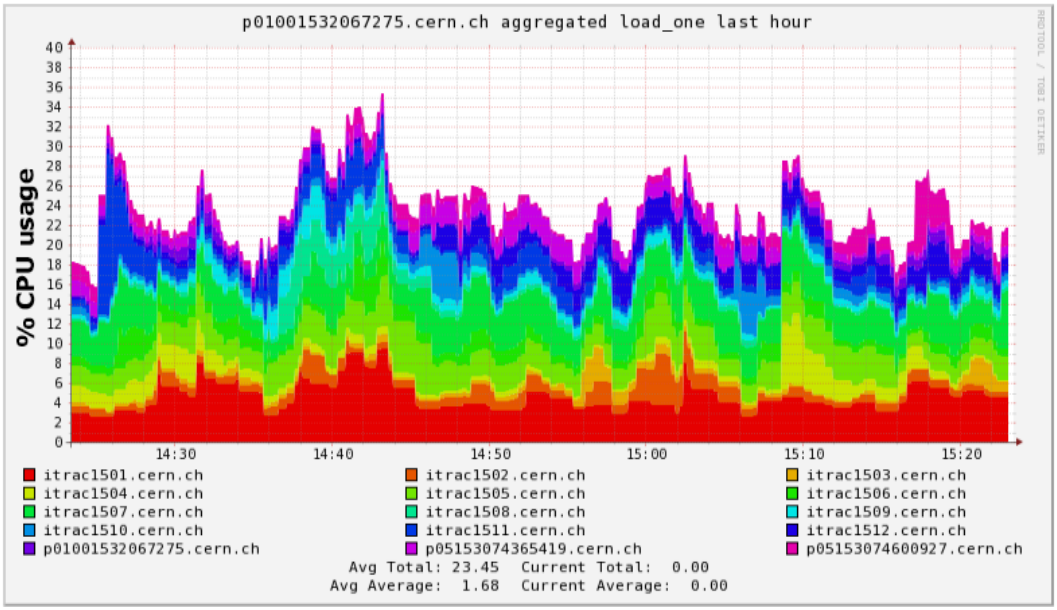


Figure 4.8: WLCG Hadoop cluster workload [15].

Metrics calculation

In this section the metrics used for the evaluation are explained.

The calculation of the percentage of time spent in Garbage Collection (GC) is shown in Equation 4.1, where P_{gc} was the calculated percentage of time spent in GC, T_{gc} was the time (ms) spent in GC, $T_{mappers}$ was the time (ms) utilised by all map tasks and $T_{reducers}$ was the time utilised by all reduce tasks:

$$P_{gc} = \frac{T_{gc}}{(T_{mappers} + T_{reducers})} \times 100 \tag{4.1}$$

Calculating the memory usage by tasks from the allocated memory was complicated. Therefore, the steps below were taken:

First, the total tasks, C_{tasks}, allocated to a job were calculated, which can be seen in Equation 4.2, where $C_{total}^{mappers}$ was the total number of allocated mapper tasks and $C_{total}^{reducers}$ was the total count of allocated reduced tasks:

$$C_{tasks} = C_{total}^{mappers} + C_{total}^{reducers} \tag{4.2}$$

Then the average task memory, M_{avg}^{tasks}, usage was calculated as shown in Equation 4.3, where $M_{physical}$ was how much RAM (in bytes) consumed by all the tasks and total tasks C_{tasks} was multiplied by 1024×1024 to get the memory in MB.

$$M_{avg}^{tasks} = \frac{M_{physical}}{C_{tasks} \times 1024 \times 1024} \tag{4.3}$$

The used memory in time, X_{total}^{used}, was calculated using Equation 4.4, where $T_{mappers}$ was the time (ms) utilised by all map tasks, $T_{reducers}$ was the time utilised by all reduce tasks and M_{avg}^{tasks}, was the average task memory calculation from Equation 4.3:

$$X_{total}^{used} = (T_{mappers} + T_{reducers}) \times M_{avg}^{tasks} \tag{4.4}$$

The total allocated memory in time, $X_{total}^{allocated}$, was calculated using Equation 4.5, where X_{maps} was the memory time of RAM (MB × ms) allocated to map tasks and $X_{reducers}$ was the memory time of RAM (MB × ms) allocated to reduce tasks:

$$X_{total}^{allocated} = (X_{maps} + X_{reducers}) \tag{4.5}$$

Finally, the percentage of memory allocation used was calculated as shown in Equation 4.6:

$$P_{used}^{allocated} = \frac{X_{total}^{used}}{X_{total}^{allocated}} \times 100 \tag{4.6}$$

The CPU time used versus allocated time was calculated using Equation 4.7, where T_{used}^{cpu} was the CPU time used, T_{total}^{cpu} was the total CPU time used by all tasks, $T_{mappers}^{cpu}$ was the total virtual core time allocated for map tasks and $\mathrm{T}_{reducers}^{cpu}$ was the total virtual core time allocated for reduce tasks:

$$T_{used}^{cpu} = \frac{T_{total}^{cpu}}{(T_{mappers}^{cpu} + T_{reducers}^{cpu})} \tag{4.7}$$

4.5.2 The performance of batch computations with scaling dataset

In order to evaluate the execution time, RAM memory, CPU, DISK and network utilisation of the job on the WLCG platform using the Lambda approach, it was decided to use the dataset from November 2015, ranging from 1 day to 30 days, which were added and incremented at each test to scale the dataset. Jobs were submitted five times for each dataset in order to capture an average performance time.

Analysis of uncompressed dataset

The main computation result, as presented in the plot in Figure 4.9, demonstrates that the jobs were able to successfully process all data ranges in at most just a few minutes. This result alone satisfies the requirement of a monitoring system, in particular for the

WDT use case, where it can process 24 hours of data in under a minute when the Avro and CSV data format was used as the traditional architecture cannot match the performance of the proposed architecture. Moreover, it demonstrates the scalability of the system over different data volumes as it can be seen that performance is relatively stable as the dataset is increased beyond six days. It should be noted that difference between the execution time on the one day dataset compared with the incremented 30 days dataset is by only a factor of 2, while being many times bigger in size. This demonstrates the scalability of the Hadoop system, which distributes the computation across several nodes. Nevertheless, there is always a fixed overhead added to the job for finding appropriate resources and submitting them.

Although Avro and CSV jobs were processed faster than JSON jobs, the Avro format is processed fastest because the data are serialised in binary format so it is quicker to read the data. Although the JSON data were processed the slowest, the execution time was still better than that of the current architecture used by the WLCG group for monitoring.

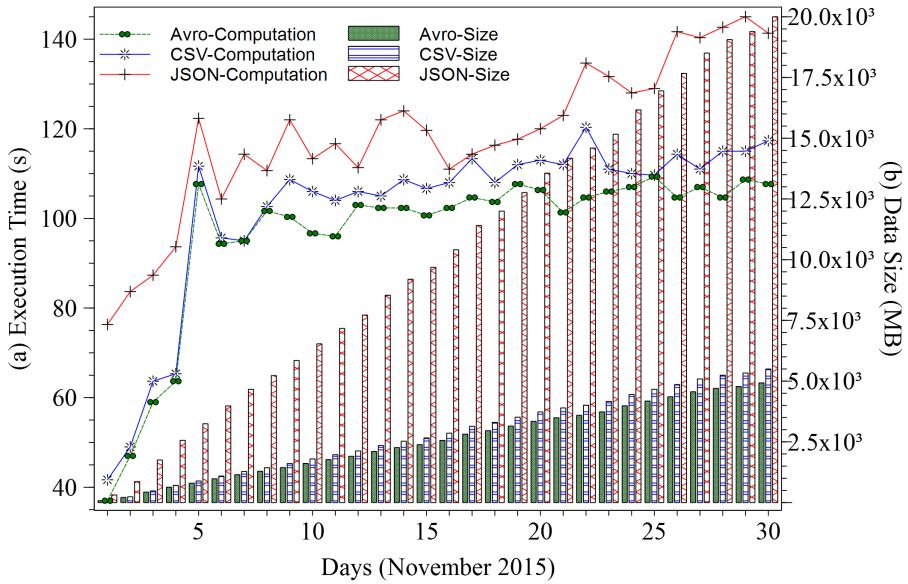
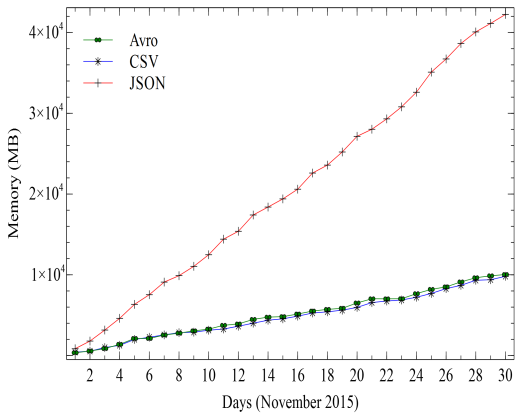
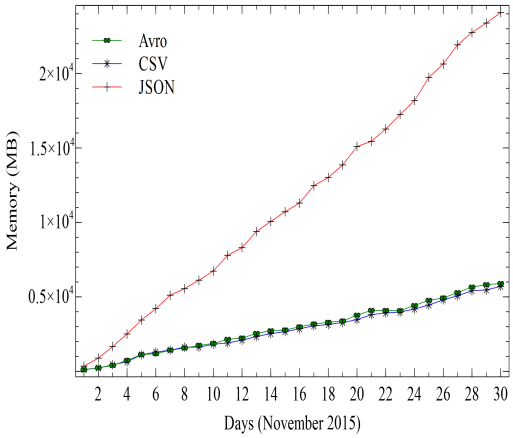


Figure 4.9: Computation of uncompressed Avro, CSV and JSON files over different date ranges. The primary axis (a) shows the execution time that is represented by lines, whereas the secondary axis (b) represents the input data size in MB is represented by bars.

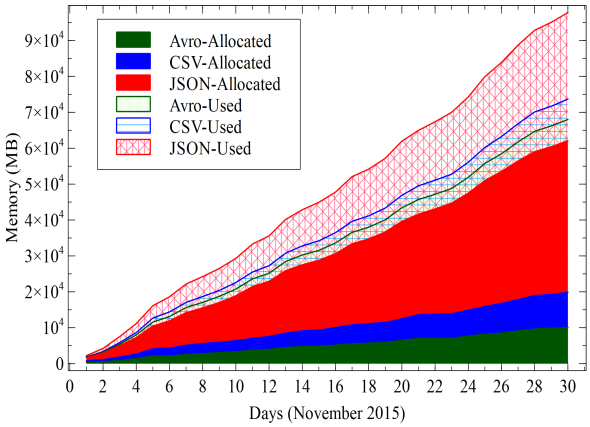
Figure 4.10 illustrates the memory allocation and the memory usage of the jobs. Figure 4.10 (a) shows memory allocated to tasks for each job (days). The resource manager has allocated in total an average of 307% more memory for JSON compared with Avro. However, Avro jobs on average were allocated 5% more memory than CSV, which is not significant compared with JSON jobs. It is very important to understand how the memory is used in an infrastructure by an application because it will play a huge role in efficient and effective computations. An over-allocated memory means the resource manager does not allocate as many tasks onto nodes as it can handle; hence, the cluster is underutilised. Figure 4.10 (b) shows that JSON used 294% more memory than Avro and Avro used 5% more memory than CSV. On the other hand, JSON has used 45% less of the allocated memory and both Avro and CSV have used 44% less memory than was allocated by the resource manager, which can be seen in the stacked plot shown in Figure 4.10 (c). All jobs have used a lot less memory than was allocated but what is clear is that the JSON job requires more memory compared with the other two jobs.



(a) The plot shows the allocated memory.



(b) The plot represents the used memory.



(c) The plot represents the stacked up used memory versus allocated memory.

Figure 4.10: Memory allocated/used for computing Avro, CSV and JSON files over scaling dataset.

The average memory allocated to each task for all jobs was very consistent, which is between 500 MB and 600 MB as seen in Figure 4.11. The JSON task was allocated less memory on average when compared with the other two jobs but this can be explained by the number of tasks allocated for the JSON job as 4 times more tasks were allocated so the memory is spread out. This also explains why, overall, a lot of memory is allocated to the JSON job as shown in Figure 4.10. The resource manager assigns a task to each HDFS data block; each block is 256 MB so more data mean more blocks, which will result

in more tasks which is the case with the JSON job.

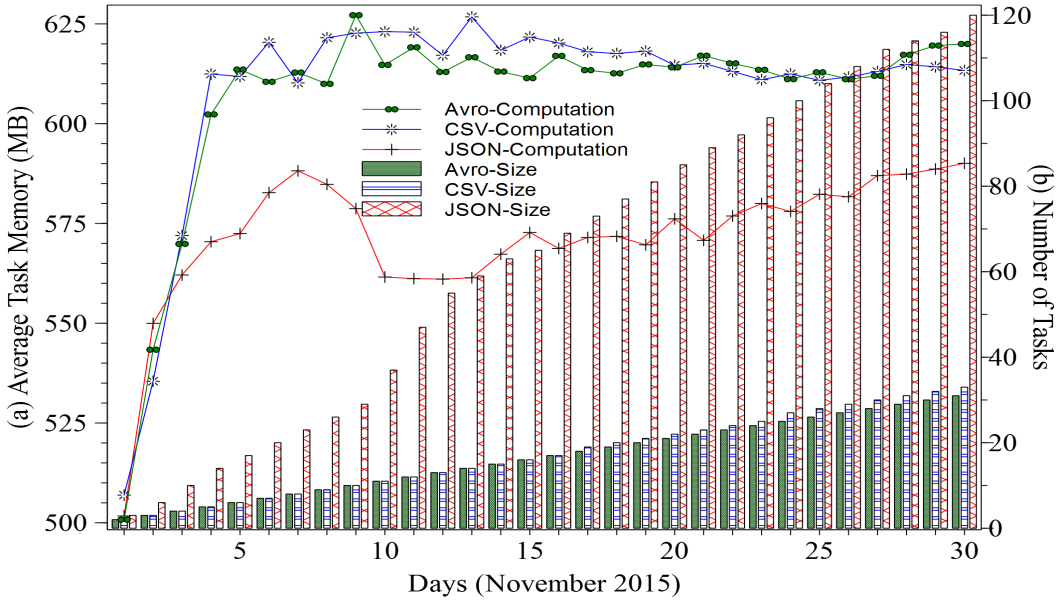


Figure 4.11: Average memory allocated for each task for computing Avro, CSV and JSON files over scaling dataset. The primary axis (a) shows the average allocated memory in MB that represented by lines, whereas the secondary axis (b) represents the number of allocated tasks represented by bars.

In contrast to over-allocated memory, if the memory is under-allocated it causes issues such as the distinct Out Of Memory (OOM) errors, although this failure is usually recoverable (supported by fault-tolerance mechanisms, however, it does have a limit on the number of retries) and the second but serious issue caused by the lack of memory is GC. When a Java application approaches the full heap (runtime data storage area in memory) utilisation, often the Java Virtual Machine (JVM) will run a full GC, which will block the other tasks in order to use the CPU (it is a CPU-intensive task). Figure 4.12 show the percentage of memory used for GC from the total used memory by Avro, JSON and CSV jobs respectively. All the jobs have used a tiny portion of the memory for GC. However, it appears that the Avro job used more memory for garbage collection than the other two, which may be to do with cleaning up of the memory footprint left by the serialisation.

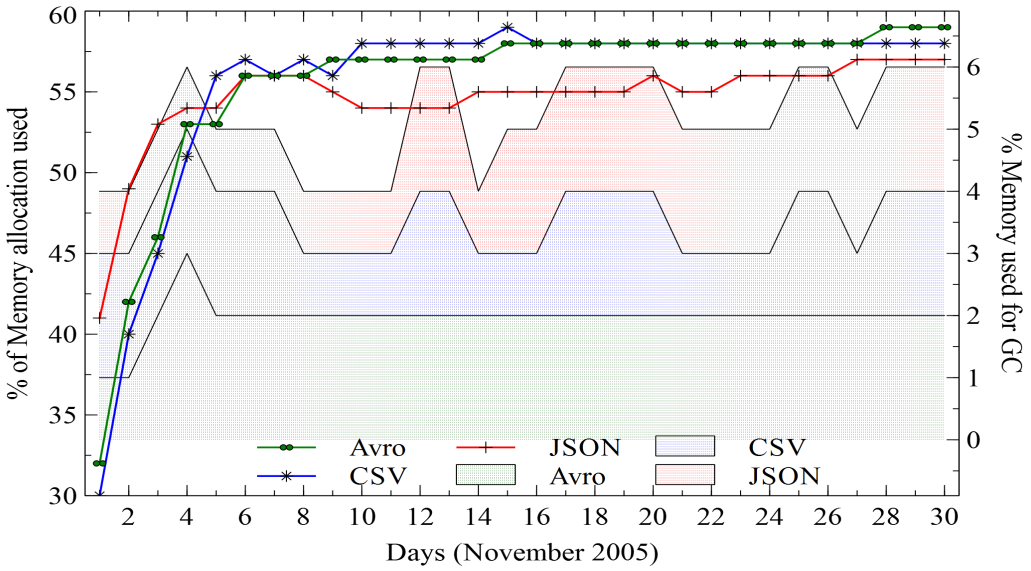
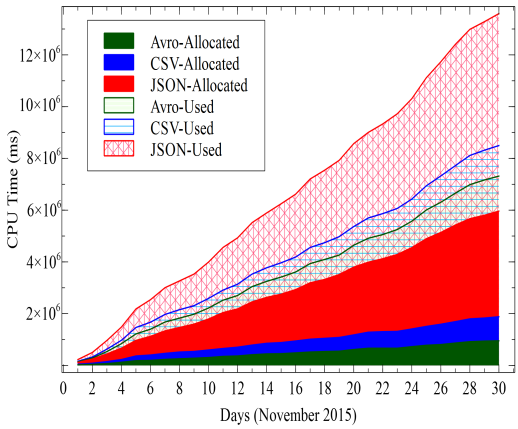
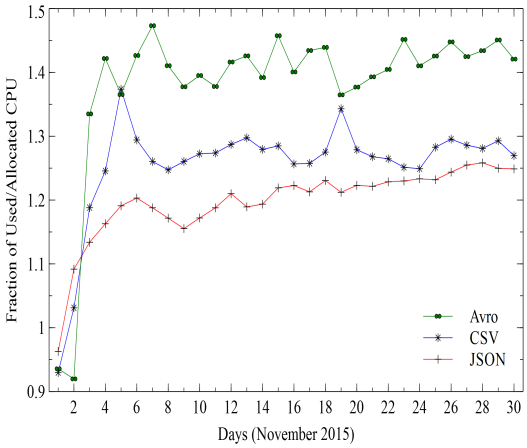


Figure 4.12: The percentage of memory used for GC from the overall used memory by the tasks. The primary axis (a) shows the percentage of allocated memory for the job that are represented by lines, whereas the secondary axis (b) represents the percentage of memory used for GC represented by stacked bars.

While the resource manager strongly enforces memory limits, the CPU limit is not affected, although this is useful in maximising total CPU utilisation. The GC process is multi-threaded; tasks requiring fewer cores could end up using more cores due to the dynamic method used in allocating the cores. Therefore, a rogue job can degrade the performance of every other application on the cluster. Figure 4.13 (a) illustrates that in general more CPU time is used than allocated. Again, JSON has been allocated 319% more CPU time than Avro. Nevertheless, Avro has been allocated 5% more CPU time than CSV. Figure 4.13 (b) shows that on average Avro has used 140% of the allocated CPU time, CSV has used 130% of the allocated CPU time and JSON has used 120% of the allocated CPU time. However, JSON has used 294% more CPU time than Avro, which, on the other hand, used 16% more CPU than CSV.



(a) The plot represents the stacked up used and allocated CPU time.



(b) The plot represents used/allocated CPU time.

Figure 4.13: CPU time used/allocated for computing Avro, CSV and JSON files over 30 day dataset.

Even though the job is split into multiple map tasks and sent to data nodes where the data reside in order to reduce the movement of large data files over the network, the intermediate results of these tasks still need to be shuffled and shifted around to reducers (most likely to different nodes). Figure 4.14 indicates how much of the intermediate results (MB) from the mappers were transferred to the reducers. JSON appears to have shuffled more data. A greater data shuffle will make the job go slower as the shuffle process utilises network connection bandwidth in order to transfer the intermediate results from mapper nodes to reducer nodes.

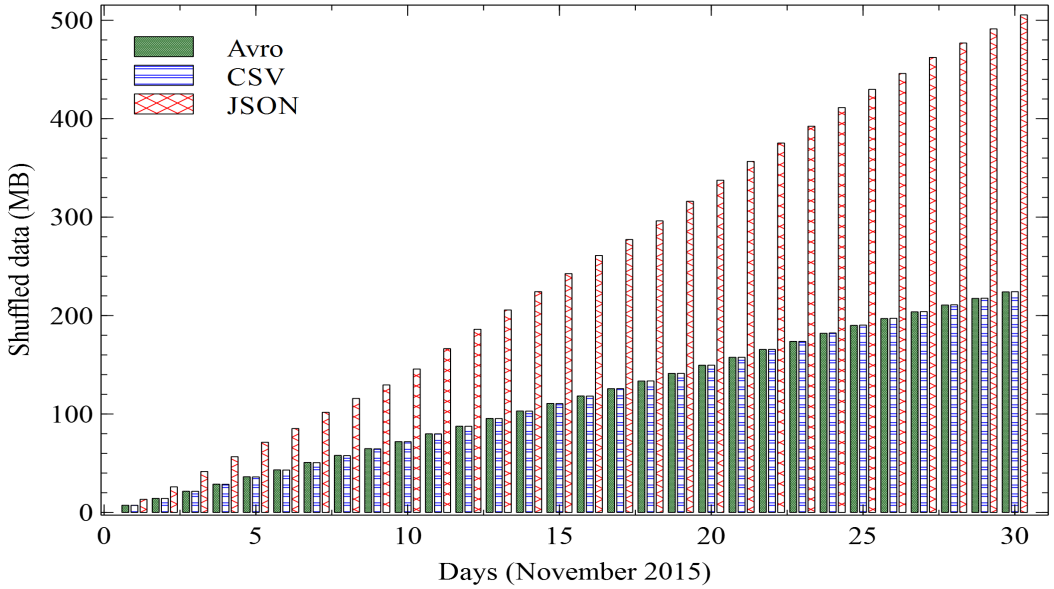


Figure 4.14: Shuffled intermediate results from mappers to reducer nodes.

Analysis of compressed dataset

Compressed data using the Snappy compressor were evaluated against their uncompressed counterparts as shown in Figure 4.15. The test was identical to the one that was carried out previously on the uncompressed data. In general, the computation time of compressed data was slower than for the uncompressed data. It is understandable why compressed data were slow to process as the data will need to be uncompressed before processing and this will therefore add additional overhead to the computation time. Although uncompressed Avro and CSV jobs were fast, the CSV appears to be the fastest when compared with the uncompressed data; it is possible that processing serialised and compressed data adds slightly more overhead than parsing a CSV dataset. Processing the compressed JSON dataset took on average 94% more execution time than the uncompressed JSON. In contrast, the compressed Avro dataset took 21% more execution time and CSV took 5% more execution time compared with the corresponding uncompressed Avro and CSV respectively. However, CSV has the smallest deviation between compressed and uncompressed in comparison to other two jobs.

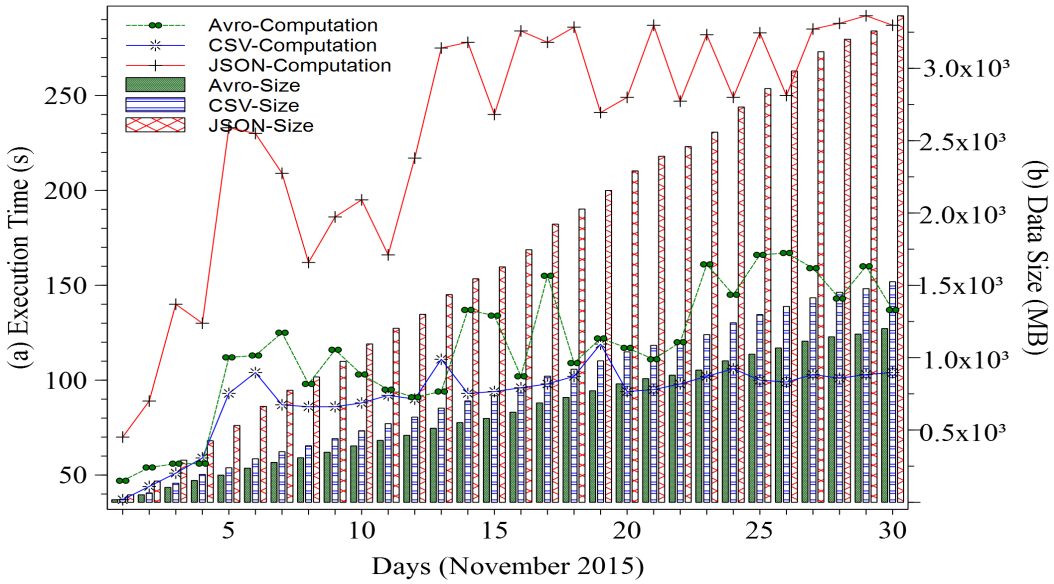
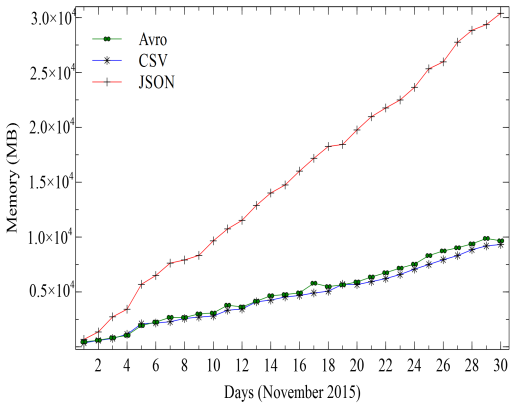
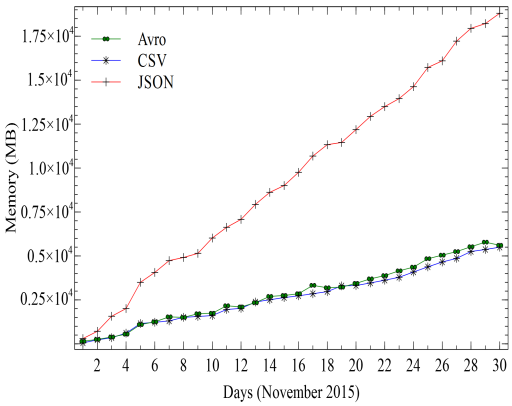


Figure 4.15: Computation of compressed (Snappy) Avro, CSV and JSON files over scaling dataset. The primary axis (a) shows the execution time represented by lines, whereas the secondary axis (b) represents the input data size in MB represented by bars.

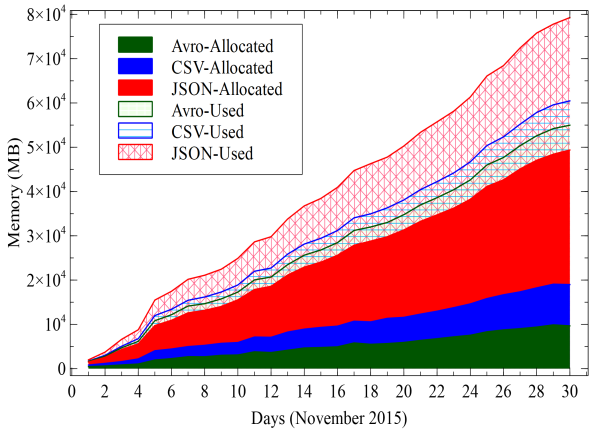
The resource manager has allocated less memory for computing the compressed data than the uncompressed data. It has allocated 2% less memory for Avro, 26% less memory for JSON and 4% less memory for CSV for computing the compressed dataset when compared with the uncompressed dataset as shown in Figure 4.16 (a). It has used 2% less memory for Avro, 18% less memory for JSON and 4% less memory for CSV for computing the compressed dataset when compared with the uncompressed dataset as shown in Figure 4.16 (b). Figure 4.16 (c) represents stacked used versus allocated memory for the compressed dataset and it can be observed that less memory is used than allocated, just like the uncompressed dataset. Overall less memory was allocated and used for computing a compressed dataset compared with the corresponding uncompressed dataset.



(a) The plot shows the allocated memory.



(b) The plot represents the used memory.



(c) The plot represents the stacked up used memory versus allocated memory.

Figure 4.16: Memory allocated and memory used for computing compressed (Snappy) Avro, CSV and JSON files over scaling dataset.

On average, all three jobs have been allocated with a comparable amount of memory as can been see in Figure 4.17. However, the JSON job has been allocated 12% more memory, but 286% fewer tasks for the compressed dataset than the uncompressed dataset, which is due to the decrease in the data size, hence the average memory is balanced out. The number of tasks was identical for all three jobs because a task was allocated to each partition (i.e. each day) of the dataset, which was compressed, so it is smaller than the default 256 MB block size. Therefore, all three jobs had identical tasks, which were allocated on smaller-sized partitions for processing.

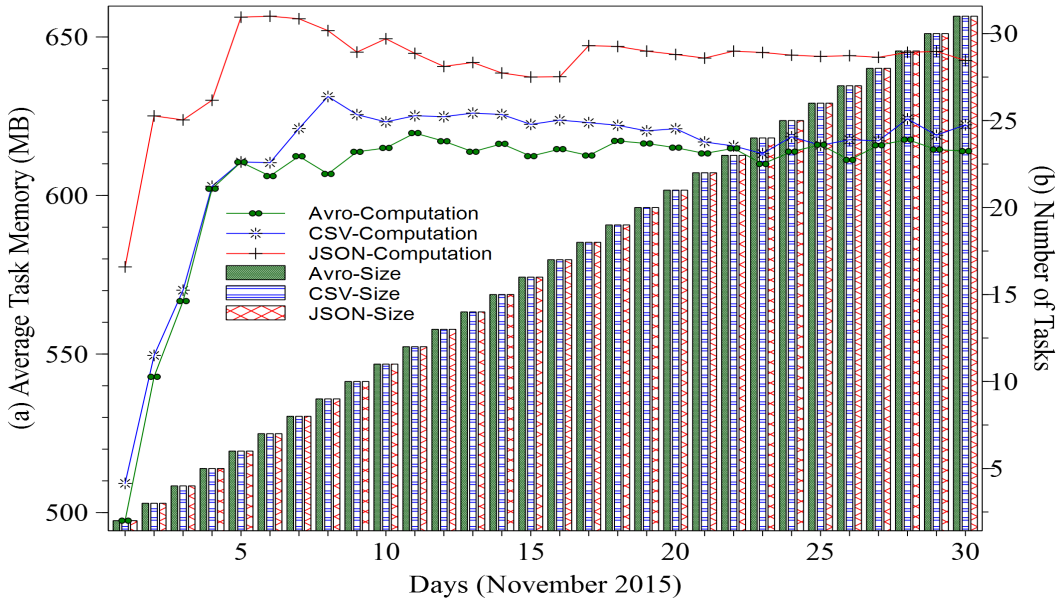


Figure 4.17: Average memory allocated for each task for computing Avro, CSV and JSON files over scaling dataset. The primary axis (a) shows the average allocated memory in MB that represented by lines, whereas the secondary axis (b) represents the number of allocated tasks represented by bars.

The compressed tasks for Avro, JSON and CSV have used a very tiny portion of the memory for GC from the used memory, which is consistent with the uncompressed tasks as can been see in Figure 4.18.

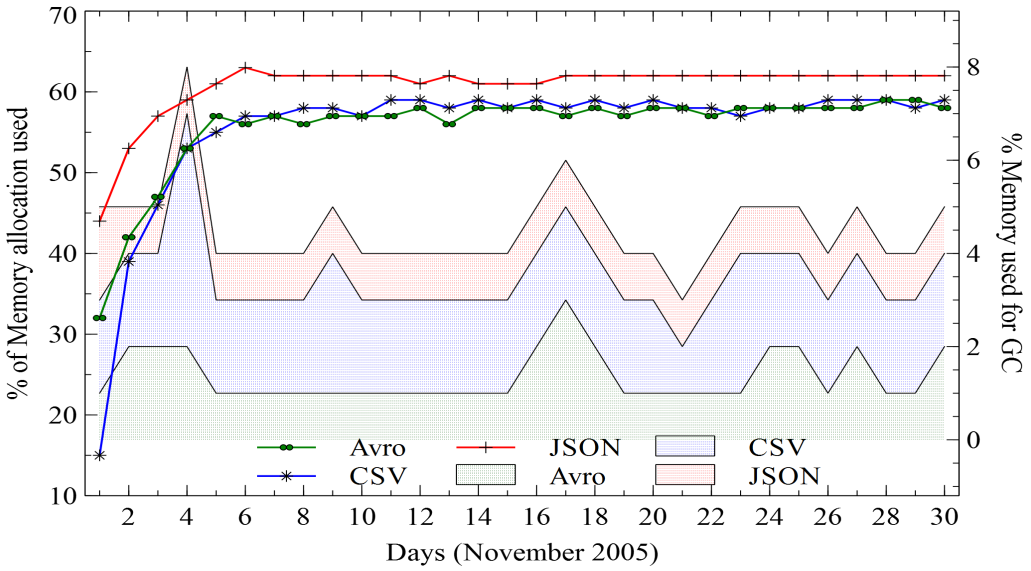
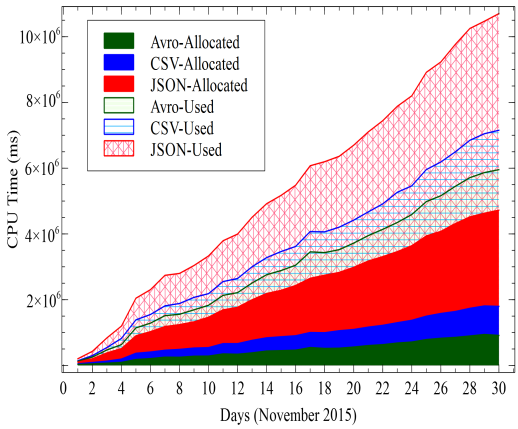
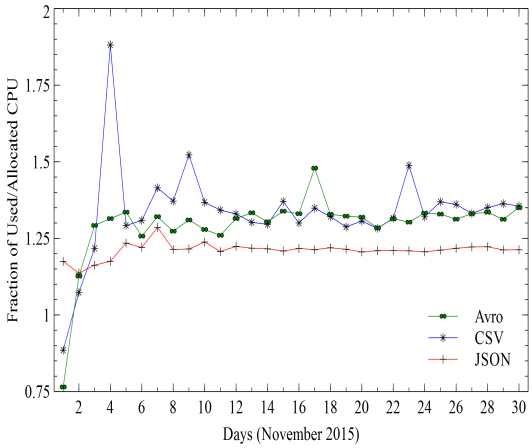


Figure 4.18: The percentage of memory used for GC from the overall used memory by the tasks for computing compressed (Snappy) dataset. The primary axis (a) shows the percentage of allocated memory that are represented by lines, whereas the secondary axis (b) represents the percentage of memory used for GC represented by stacked bars.

On average, uncompressed and compressed jobs have been allocated and used a comparable amount of CPU time with the exception of CSV, which appeared to have slightly more CPU time as can be seen in Figure 4.19 (a) and (b).



(a) The plot represents the stacked up used and allocated CPU time.



(b) The plot represents used/allocated CPU time.

Figure 4.19: CPU time used/allocated for computing compressed (Snappy) Avro, CSV and JSON files over scaling dataset.

Figure 4.20 shows the intermediate data transferred from the mappers to the reducer nodes. The compressed Avro and CSV job shuffled the same amount of intermediate results to reducers, but JSON has decreased the data shuffle by 23% which is due to the decrease in mapper tasks as a result of the smaller input dataset.

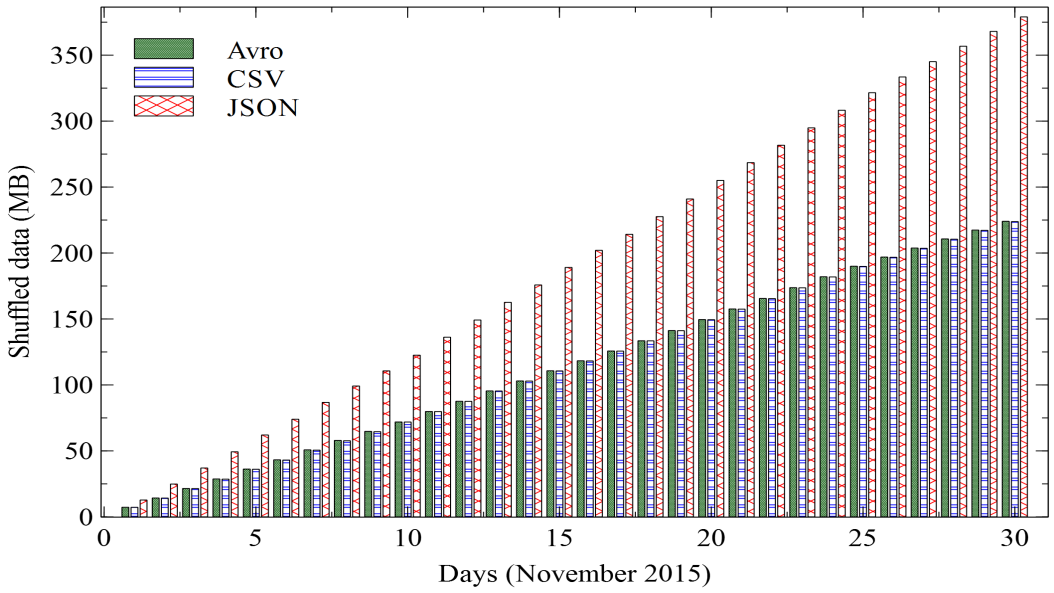


Figure 4.20: Shuffled intermediate results from mappers to reducer nodes

4.5.3 The performance of batch computations with scaling nodes

In order to evaluate the scalability of the architecture, it was necessary to see whether the workload is balanced out as the number of worker nodes were increased. However, there were difficulties in evaluating the scalability on the WLCG cluster, due to a lack of administrator privilege as this task requires decommissioning nodes and activating the nodes incrementally. Therefore, this evaluation was conducted in the Azureus Cloud infrastructure. Each node was configured to run Ubuntu version 14, with 13.5 GB of memory and with a 4 cores CPU, which is less powerful than the WLCG cluster. However, there was no other workload on the cluster unlike the shared cluster provided by the WLCG. A node was incrementally activated by one at each evaluation starting from a single node cluster. The evaluation was carried out on the same number of monitoring events as used on the WLCG cluster (3.3 million events) but the file size varied due to the difference in data format. Figure 4.21 shows that, in general, the execution time was high when there was a lower number of nodes but the performance improved as the number of nodes was increased. The execution time became stable after a certain point as the resource manager did not allocate the job to all the nodes because it would have to break the primary principle of Hadoop of sending the computing to where data reside (data locality) by moving the data to the computing node. The performance would have degraded if the number of nodes was increased further and if the resource manager decided to use those nodes as it would have to move large datasets through the network. The Avro and CSV jobs did not see a huge improvement in performance despite increasing the number of nodes; however, they were performing much better when compared with the current architecture used by the WLCG. The JSON job shows a large improvement as the number of nodes was increased, which is due to large file size being distributed across the computing nodes.

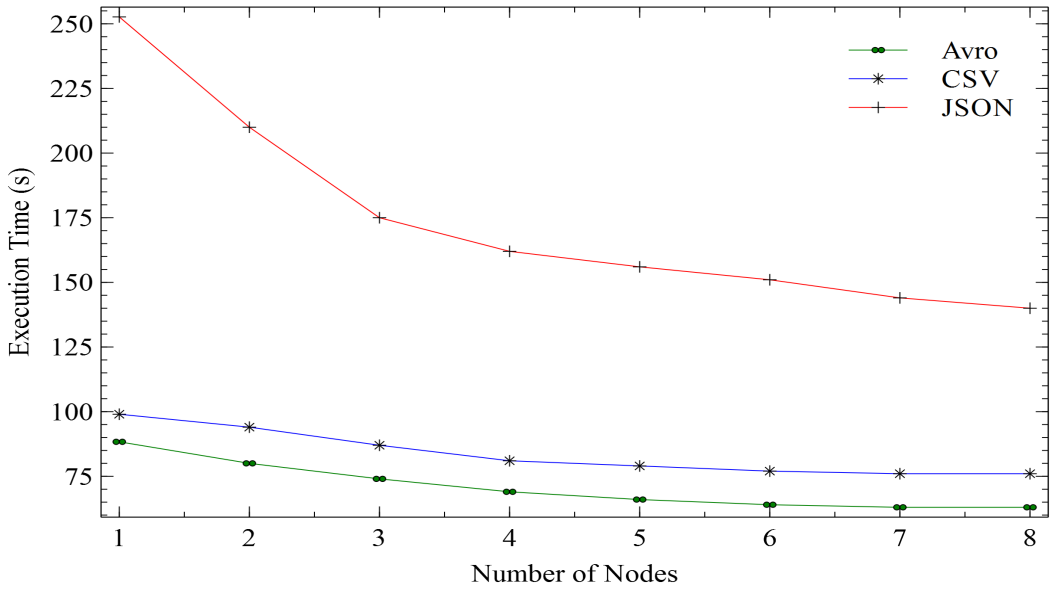


Figure 4.21: Execution time versus the number of worker nodes.

4.5.4 The performance of batch computations with parallelisation

An evaluation of the parallelisation available using Lambda approach was carried out on the reducer tasks to understand how it impacts the performance. The evaluation was carried out on a 5 GB dataset for each data type (Avro, CSV and JSON), but the number of monitoring events in the dataset varied for each data type. The parallelisation of the reducer was incremented by two at each evaluation and the results can be seen in Figure 4.22. Initially, the performance improved as the parallelisation was increased but gradually the performance degraded due to a lot of data movement of the intermediate results from the mapper tasks to reducer tasks. Also, additional time was added due to the resource manager needing to find appropriate resources as the parallelisation was increased. In any case, Avro and CSV performed better as the intermediate data size was a lot smaller than JSON so it would have been quicker to transfer the data to the reducers.

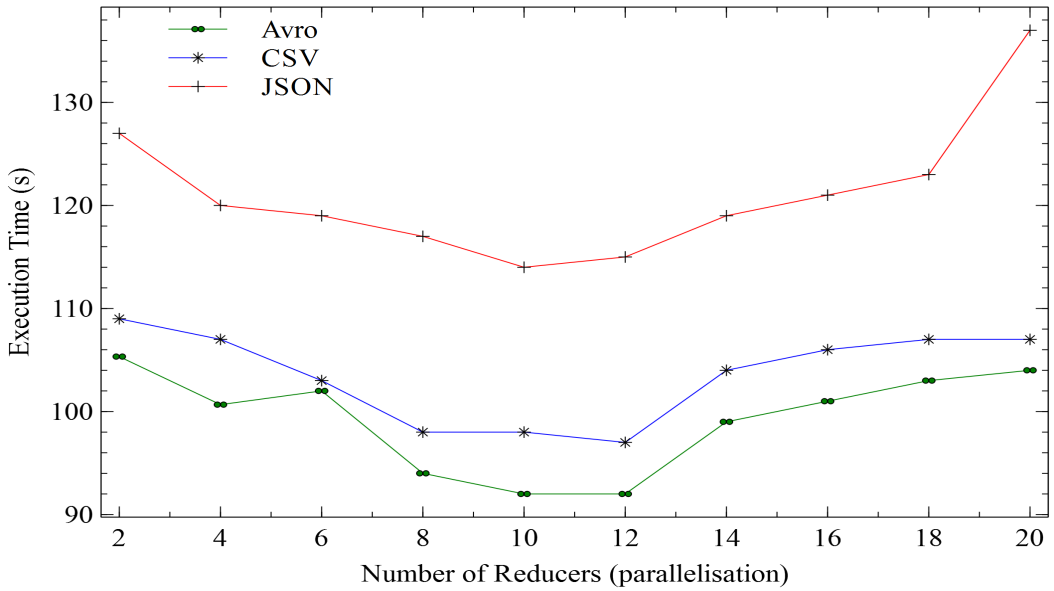


Figure 4.22: Evaluation of parallelisation of reducer tasks when computing 5 GB dataset. Execution time versus the number of reducers.

Figure 4.23 shows the result of parallelisation evaluation, which is the same as the above test but this time it was evaluated on same dataset (4 million monitoring events) for each data types (i.e. Avro, CSV and JSON) so the dataset size varied due to different data format. Again, the reducer was incremented by two at each evaluation. There was a large difference in execution time between JSON and the other two data types even though they were computing on the same dataset. However, regarding the performance of parallelisation, there was a minor improvement at the beginning but the performance degraded as the paratitions was increased further. In order to minimise moving larger datasets over the network, the intermediate results were compressed, which improved the performance slightly.

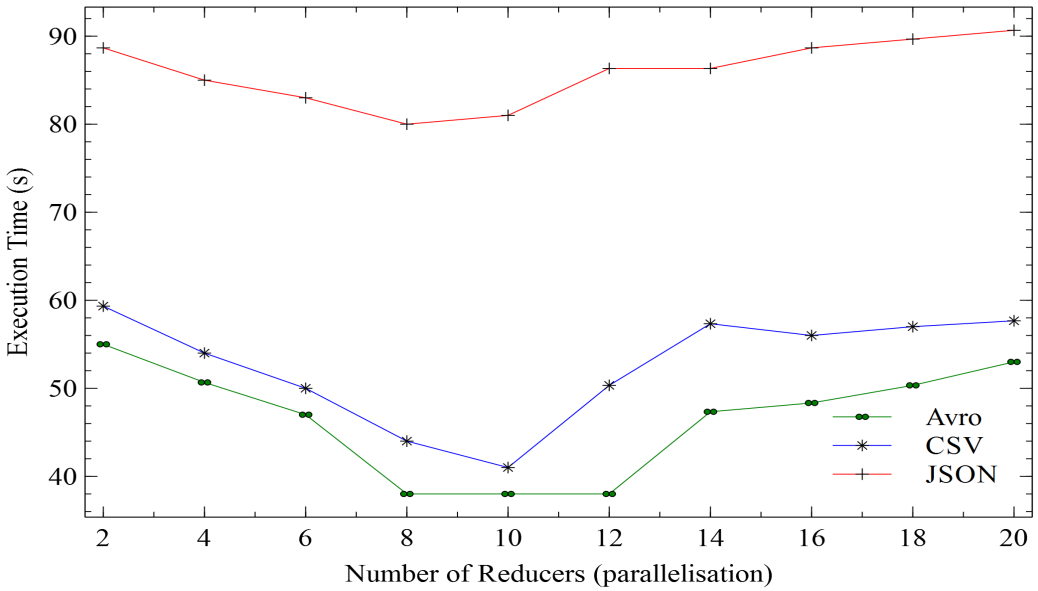


Figure 4.23: Evaluation of parallelisation of reducer tasks when computing 4 million monitoring events. Execution time versus the number of reducers.

4.5.5 The performance of the serving layer

The Elasticsearch serving layer was evaluated in the Azureus Cloud infrastructure. The layer was configured with three nodes and two replication factors. Each node was configured to run Ubuntu version 14, with 13.5 GB memory and with a 4 core CPU. The Elasticsearch-Hadoop library was used for reading and creating index and documents in the Elasticsearch serving layer. A small test was performed to see how long it would take to query and create 500,000 documents (records) in the serving layer, the results of which can be seen in Table 4.2. Better performance was achieved with Elasticsearch than with the current Oracle storage (with a 4 core CPU and with 32 GB memory).

Table 4.2: Performance of querying and creating documents (records) in Elasticsearch and Oracle

Serving Layer	No.documents (records)	Query time (s)	Create time (s)
Elasticsearch	500,000	18	47
Oracle	500,000	93	485

4.5.6 The performance of the real-time processing

The streaming version of the transfer statistics algorithm described in the implementation section was tested for the job throughput, in order to test the number of input events processed per second by the Esper real-time layer. A single Esper node was configured to read events from the JMS. The node was configured with 8 GB of memory and a 4 core CPU. Many datasets of test messages were created and injected into the Esper engine. The throughput results obtained from evaluating the real-time layer are shown in Table 4.3. The current architecture does not support real-time computation so the result observed from the Esper cannot be compared with it. However, the Esper performance is reasonable for the WDT use case as it was estimated that message queues propagate on average 250 events per second but the real-time layer was able to process ∼600 events per seconds (simulated).

Table 4.3: Throughput results obtained from evaluating the real-time layer.

Events (throughput)	Processed per second (s)
~ 600	1

4.6 Summary

The new data store and analytics platform presented in this chapter has been shown to be a scalable, low-latency and effective solution. The performance of the serving layer (Elasticsearch) and real-time layer also performed better in supporting low-latency in monitoring the infrastructure, which was not viable with the existing system used by the WLCG. The advantage of using Esper is that it can carry out incremental updates, providing low-latency statistics computation when compared with full re-computation, which causes high-latency. Esper by design is a single node engine, something that raises issue regarding the scalability and potential application of this solution. Nevertheless, Esper is designed to process thousands of events per second and in this specific use case it processed 600 events per second (simulated), but in real case scenario this use case receives on average 250 events per second. The Esper processor was very sluggish when several

thousands of messages were pushed in at one time.

Chapter 5

Optimised Lambda Architecture using Apache Spark technology

Within scientific infrastructure scientists execute millions of computational jobs daily, resulting in the movement of petabytes of data over the heterogeneous infrastructure. Monitoring the computing and user activities over such a complex infrastructure is incredibly demanding. Whereas present solutions are traditionally based on the Oracle RDBMS for data storage and processing, recent developments evaluate the LA as shown in Chapter 4. In particular these studies have evaluated data storage and batch processing for processing large-scale monitoring datasets using Hadoop and its MapReduce framework. Although LA performed better than the RDBMS-based architecture, it was fairly complex to implement and maintain. This chapter presents an Optimised Lambda Architecture (OLA) using the Apache Spark ecosystem, which involves modelling an efficient way of joining batch computation and real-time computation transparently without the need to add complexity. A few models were explored: pure streaming, pure batch computation, and the combination of both batch and streaming. An evaluation of the OLA on the CERN IT on-premises Hadoop cluster and the public Amazon cloud infrastructure for the monitoring WDT use case are both presented, demonstrating how the new architecture can offer benefits by combining both batch and real-time processing to compensate for batch-processing latency.

5.1 Introduction

Monitoring a scientific experiment requires the gathering of a large volume of data that is produced at a rapid rate. This is illustrated in Figure 5.1 that shows dataset size produced over various days.

Scientific infrastructures can be highly distributed and heterogeneous platforms with various middleware characteristics, job submission and execution tools, and diverse methods of transferring and accessing datasets. The high computation activity and distributed nature of such infrastructures makes the system extremely complex. Efficient monitoring is necessary in order to recognise and resolve any potential issues within the infrastructure that may cause failures or inefficiencies. This is also an important determinant in the overall effective utilisation of the resources.

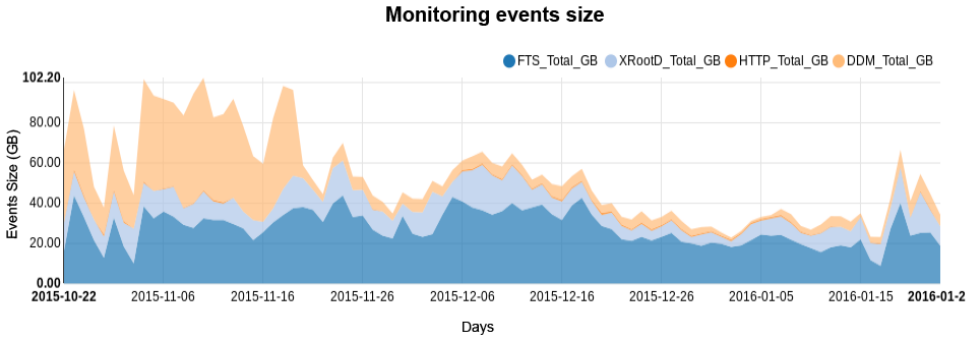


Figure 5.1: WDT dataset size [14].

There are already a few architectures that have been introduced to support Big Data as reviewed in Chapter 2. However, the main objective of the scientific use case is the need to process an arbitrary set of historical data, and handle recomputation or an old backlog of data injected by producers. In the scientific realm it is normal to have jobs running for a long period of time. Old backlog injection is therefore very common when a long running job is completed. It is necessary to have both a batch layer as well as a streaming layer (real-time) as presented in Chapter 4. However, this requires better mechanisms in place to simplify the system. In this current chapter an OLA has been presented, and evaluated.

5.2 Background

Monitoring events, metadata about the jobs, and information on data transfers are collected and analysed to produce summary plots used by operators and experts to evaluate computing activities [77]. Due to the high volume and velocity of the events that are produced, the traditional methods are not optimal. Therefore, the LA [19], an architecture leveraging many different technologies for supporting Big Data, was employed to support the WDT use case. However, by combining and synchronising many technologies, the issue of high complexity became a significant concern.

The LA that is presented in Chapter 4 demonstrated that it has the ability to work well for monitoring. Most notably, the WDT use case has shown that it outperforms the traditional architecture. However, with the complexity of a three-layer structure that includes various technologies, comes a price when integrating all three layers together to serve several main goals (monitoring infrastructure in real-time, supporting scalability, ease of implementation, maintenance and migration). Having different technologies for each layer would be difficult to integrate, implement, and maintain. There is a pressing need to identify a single solution that can accommodate and integrate the batch layer as well as the streaming layer for monitoring events seamlessly. Apache Spark [38] is a new parallel processing paradigm similar to MapReduce [18], but with improved analytical performance. By exercising in-memory computation, it has the ability to support iterative computation [39][48]. It can also support data streaming, which is useful in optimising the LA to limit code differences between the batch and streaming layers. It can also support SQL-like commands, interactive command line, machine learning, and Graphx [78]. Having Spark batch and streaming under a stack is useful in optimising the LA. The Spark streaming and batch computations adapt the RDD, an abstract data collection that is distributed across nodes for parallel processing [78],[79]. Transformation and computation logics can therefore be reused between batch and streaming layers.

Spark processes are 'lazy' [78], and no action is carried out until it is required. An example would be the RDD, which does not physically hold data. It contains instructions on what to do when an action is called. The RDDs support two types of operations: trans-

formations, which create a new dataset from an existing one, and actions, which return values to the driver program after running a computation on the dataset. For example; the *mapPartition* is a transformation that passes each dataset element to a partition level through a function and returns a new RDD representing the results. Counter to this, *reduceByKey* is an action that aggregates all the items of the RDD using a function and returns the final result to the driver program.

By default, each transformed RDD may be recomputed every time it is put into action. It is also possible to persist an RDD in memory using the persist (or cache) method, in which case Spark would keep the computed data in memory for expedited access the next time it is queried. There is also support for persisting RDDs on disk, or replicated across multiple nodes [79]. When monitoring a scientific infrastructure it is typical that various statistics are derived from the same monitoring events. In-memory storage and computations are profitable as multiple yet distinct computations can be carried out by a job on the cached data. This will make it easier to maintain the job as well. The LA evaluated in Chapter 4 employs MapReduce framework which does not support in-memory persistence [18], so data cannot be shared. In order to implement a complex algorithm in MapReduce framework it requires the creation of chained MapReduce jobs. Essentially, the output of a job will need to be directly connected to the input of the next job. Spark does not require this due to in-memory processing. Spark can also support global data sharing using Tachyon (licensed under Apache), which is a memory-centric distributed storage system that can be used for data sharing.

Spark Streaming supports three notable functions:

1. **Cumulative Computations**, which supports cumulative statistics computation while streaming in new data (incremental calculations). Spark Streaming supports maintenance of the state (which is stored information at a given instant in time) for those statistics. The Spark Streaming library has a function called *updateStateByKey* for maintaining and manipulating the state [78].

2. **Windowed Computations**, which is useful when the data received in the last n amount of time is non-trivial. Spark Streaming readily splits the input data into the

desired time windows for easy processing, using the window function of the streaming library [78]. A function such as *forEachRDD* allows access to the RDDs created at each time interval.

3. **Transformation**, which returns a new DStream (stream of events) by applying an RDD to RDD function for every RDD of the source DStream [78]. This is where the code can be reused between batch and streaming layers using the *transform()* function as both frameworks support RDD as the core component. This feature also supports merging (i.e. joining) the batch RDDs with the streaming RDDs, which optimises the LA.

5.3 Architecture and design

The core part of the OLA inherits the technologies and approaches from Chapters 3 and 4 such as a message broker, data pipeline (Flume), storage (HDFS), and serving layer (Elasticsearch). This is outlined in Figures 5.2 and 5.3.

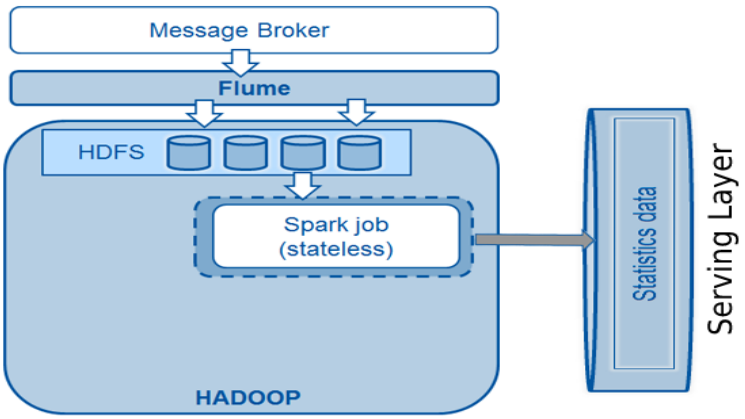


Figure 5.2: Pure stateless batch computation. Monitoring events were sent to the HDFS for batch computation, which can be scheduled to run at any preferred time interval.

The main requirement of any monitoring architecture is that it be able to provide information about the infrastructure in near-real-time so appropriate action can be taken. Therefore, the following approaches were designed and implemented:

1. Pure stateless batch computation as seen in Figure 5.2, which can be scheduled to

run at a preferred interval. The system will not have any knowledge of the previous jobs. This does not support real-time computation, but Spark framework provides in-memory computations. Therefore, the execution time can be compared with the MapReduce framework that was used in Chapter 4. The batch computation can also be used for historical computation (i.e. high-latency).

2. Pure stateful streaming computation as seen in Figure 5.3, will carry out incremental computation on continuously streaming data 24 hours/7 days a week. From this, it can maintain the state of the computed statistics. It also has a checkpoint mechanism to dump the state to the disk; in case of job failure it can pick up from where it stopped. This method on it is own is enough for real-time computation. This allows the complexities of merging multiple technologies, as in the LA, to be eliminated.

3. A combination of batch and streaming computation is also shown in Figure 5.3. Pure streaming is enough, but the potential of getting duplicate events from the message brokers due to failure is prevalent. Having pure streaming computation cannot address this issue, as the raw events are dropped once they are processed. The state of the streaming job cannot keep the unique ID of the events once they are aggregated by a key (e.g. sites). Incorporating batch computation can correct the inaccurate statistics as it will recompute whole datasets from the storage layer, eliminating duplicate events. Having a streaming layer do continuous calculation, while scheduling the batch layer to run at specified intervals in order to override the results ultimately validating the statistics seems most appropriate. As pointed out previously, historical computation is necessary in scientific domains, so it is important to incorporate a batch layer in the architecture. To support this approach, the monitoring events were duplicated with one being sent to the HDFS for batch computation, while the other was streamed straight into the streaming receiver. However, there are a variety of complexities that need to be addressed in synchronising these approaches together including: informing the streaming job about newly available data (computed by batch job) so that it may utilise it to override the streaming state as well as the serving layer that is used for storing computed statistics for serving the UI, and a mechanism to eliminate the network communication bottleneck at the serving layer to make sure only the newly streamed data are updated/inserted into

the serving layer. This is discussed further in Section 5.3.1.

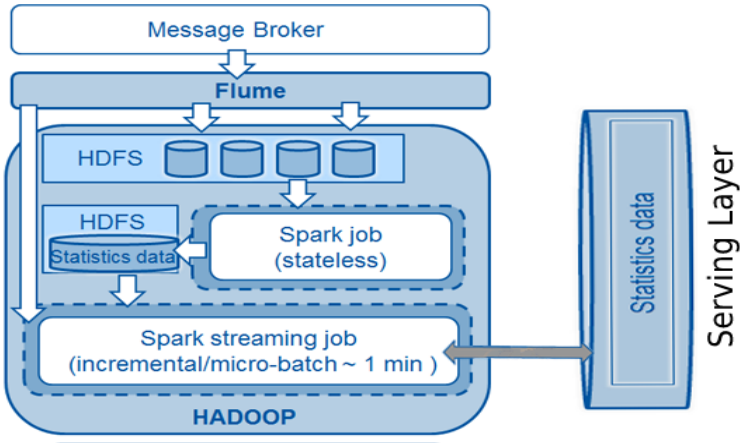


Figure 5.3: Pure stateful streaming and combination of both batch and streaming computations. Monitoring events were duplicated with one sent to the HDFS for batch computation, while the other streamed straight into the streaming receiver for incremental computation.

5.3.1 Merging and synchronising Optimised Lambda Architecture layers

This section explores how the batch, serving, and streaming layers are merged and synchronised. Table 5.1 denes the variables used in the equations below.

Table 5.1: Used variables and their corresponding expressions in this section.

Variables	**Expressions**
B	Batch
D	Dataset
F	Function
K	Key
M	Memory
S	Storage
T	Time
V	Value

Batch Layer

The batch layer is a high-latency mechanism, so computing an enormous volume of data would result in a delay which would be reflected in monitoring statistics. It is important that the batch should discard some input data, a certain amount of data which is linked to how often the batch process is executed and how big the dataset is. These 'missing statistics' can be accommodated by the streaming layer.

$$D_{filtered} = F_{discard}\left(D_{raw} \exists \left(B_{time} \ominus B_{interval}\right)\right) \tag{5.1}$$

In Equation 5.1, how the monitoring events should be discarded from the computation is represented by a formula expression. In this equation $F_{discard}()$ is the function for discarding events, D_{raw} is the number of raw events prior to event selection (filter), B_{time} is the batch execution time, $B_{interval}$ is the time interval for discarding events from the batch and $(B_{time} \ominus B_{interval})$ calculates the time frame for selecting events and for emitting all existing, $\exists$, events that match the condition. Assuming a batch job runs at B_{time}, specified $B_{interval}$ 1 hour, a batch should discard all the events in a time $>$ $(B_{time} \ominus B_{interval})$. This will prevent having partially computed results, which will be compensated by the streaming layer.

$$D_{batch} = D_{filtered} \overset{F_{map}(K,V)}{\rightarrow} \overset{F_{reduce}(K,V)}{\rightarrow} \rightarrow S_{batch}^{data} \tag{5.2}$$

Equation 5.2 describes how the selected (filtered) events, $D_{filtered}$, will go through a mapping process, $F_{map}()$, to generate (K)ey (a unique ID for the statistics)/(V)alues (matrices values associated with the key) pairs. Subsequently, it will go through the reduce process, $F_{reduce}()$, to aggregate the values by the key from all distributed nodes, which are then stored in a designated storage, S_{batch}^{data}, folder. The batch process will write the result (i.e. a new file) into a known folder on HDFS (this can be replaced by any storage layer, e.g. Tachyon).

Streaming Layer

In the consolidated streaming and batch layers, previously computed statistics (if they exist) need to be loaded from the serving layer, which can be represented by Equation 5.3.

Serving Layer process

$$\begin{aligned} D_{stats}^{storage} &= F_{load}^{storage}(T_{current}, T_{from}) \\ &= \left\{ D_{filtered}^{storage} = \left(D_{input}^{storage} > (T_{current} \ominus T_{from}) \right) \right. \end{aligned} \tag{5.3}$$

where $D_{stats}^{storage}$ are the loaded pre-computed monitoring statistics from the serving layer, $T_{current}$ is current the timestamp, T_{from} is the timestamp that statistics will be loaded from and $F_{load}^{storage}()$ is the loading function for loading data from the serving layer. If input data, $D_{input}^{storage}$, which are all statistics from the serving layer to the DB load function, are greater than $(T_{current} \ominus T_{from})$ then select, and return the statistics $D_{filtered}^{storage}$.

$$D_{processed}^{storage} = D_{stats}^{storage} \overset{F_{map}(K,V)}{\rightarrow} M_{stats}^{storage} \tag{5.4}$$

Equation 5.4 is an expression for $D_{processed}^{storage}$, the mapped and stored statistics selected from the serving layer $D_{stats}^{storage}$ into the memory, which goes through a mapping process, $F_{map}()$, generates (K)ey/(V)alues pairs, which are then stored into memory and/or disk, $M_{stats}^{storage}$ for later usage (i.e. for merging with other layers).

Streaming Layer process

In the streaming layer, the computation is defined as:

$$D_{processed}^{stream} = F_{transformation}^{data}(D_{stream}) \overset{F_{map}(K,V)}{\rightarrow} \overset{F_{reduce}(K,V)}{\rightarrow} \tag{5.5}$$

where $D^{stream}_{processed}$ is the mapped, aggregated, and computed statistics from streaming monitoring events, D_{stream} is the number of streaming monitoring events, $F^{data}_{transformation}()$ is the function filtering and transforming events, wich then go through the mapping process, $F_{map}()$, which generates (K)ey/(V)alues pairs. Finally, it goes through the reduce process, $F_{reduce}()$, to aggregate the values by the key.

Batch Layer process

The batch reading implementation is defined as:

$$D^{batch}_{loaded} = F^{load}_{batch}\left(D_{batch}\right) \overset{F_{map}(K,V)}{\rightarrow} \tag{5.6}$$

where D^{batch}_{loaded} are the statistics read from storage and mapped, D_{batch} is the pre-computed statistics from Equations 5.1 and 5.2, $F^{load}_{batch}()$ is a function to load only the "new" pre-computed batch statistics and flag the file as "old" once it is loaded successfully which then goes through mapping process, $F_{map}()$. The mapping process does not require any reduction in the statistics as it has already been done by the batch process.

Synchronise and update

The implementation of joining, merging and synchronising statistics from all three layers is defined as:

$$D_{joined} = \left(D^{storage}_{processed} \bigcup D^{batch}_{loaded} \bigcup D^{stream}_{processed}\right) \tag{5.7}$$

where $D^{storage}_{processed}$ are the statistics loaded from the serving layer, D^{batch}_{loaded} are the data loaded from batch computations, $D^{stream}_{processed}$ are the data computed from streaming data, which are unioned (joined) and returned as a new dataset D_{joined}.

The implementation of the statistics state is defined as:

$$D_{state}^{memory} = F_{state}^{update}(D_{joined}) = \begin{cases} insert, & \text{if storage=1} \wedge \ state' \quad (5.8) \\ overwrite, & \text{if batch=1} \quad (5.9) \\ update \vee insert, & \text{if storage}' \vee \ batch' \quad (5.10) \end{cases}$$

where D_{state}^{memory} is where the state of new and old statistics are kept in the memory for incremental calculation, D_{joined} are the joined $D_{processed}^{storage}$, D_{loaded}^{batch}, and $D_{processed}^{stream}$ statistics. The $F_{state}^{update}()$ is the state update function for updating the statistics and keeping them in the memory. If the statistics are from the serving layer, *storage*, and if it is not already in the state, *state*$'$, then it should insert the statistics into state memory. If the data are from the batch layer, *batch*, then it should over-write the state memory with the batch statistics. If the statistics are not from serving layer, *storage*$'$, or batch layer, *batch*$'$, then they are from the streaming layer (relatively new statistics) so they should be aggregated with the statistics in the state memory, and updated if they already exists (or it should insert the statistics into state memory if they do not exist (totally fresh statistics)).

Update serving layer

Only the new and altered statistics are inserted/updated into the serving layer which is defined as:

$$\left(D_{processed}^{stream} \bigcup D_{loaded}^{batch}\right) \rtimes D_{state}^{memory} \ \forall \ F_{serving\_layer}^{upsert} \quad (5.11)$$

The expression in Equation 5.11 says join the $D_{processed}^{stream}$ and D_{loaded}^{batch} and then leftjoin, $\rtimes$, with the D_{state}^{memory}, to insert/update only the new and updated statistics from the batch (if D_{batch} exists in the spooling location) and streamed statistics into the serving layer. Statistics from $D_{processed}^{storage}$ are not required because they are already in the serving layer. For each, $\forall$, statistics partition, establish a connection to the serving layer and bulk upsert, update the records if it already exists in the serving layer, otherwise insert new records.

Finally, set up a checkpoint at a specified interval for recovery in case of any failure.

Summary

In short, the functions explained above are:

1. The batch layer will write the result in a known folder on HDFS (this can be replaced by any storage layer e.g. Tachyon). The streaming layer will initially load specified data from the serving layer to start incremental calculations from old statistics. Then, at each micro-batch loop, and at the end of the statistics computation, it will check if there are any data in the "batch" folder. If yes, load the computed data, join with its last computed results from history, and insert the newly computed results into the serving layer while updating the history.

2. The batch layer should discard some statistics, certain data which are linked to how often the batch process is executed and the expected delay in processing the ever growing dataset. Assuming a batch run at time t, discard the last hour of data, the batch should discard all the statistics referring to time interval $> t - 1$ hour, this will prevent having a partially computed result.

3. The streaming layer will run forever, and the batch process can be executed regularly, or on-demand. Broker queues can be used for ingesting messages from the data pipeline so that if the streaming fails, the data will be retained on the broker indefinitely.

4. The serving layer insertion time will be reasonably short due to micro-batch computation. When the streaming iteration reads the full batch, and inserts it into the serving layer, it will stop processing new data. This has the potential of being noticeable on the UI. This would only be a short-lived temporary glitch. Data would still be present and it would quickly (scaling nodes and paralleling the tasks would improve performance) recover when insertion is over.

5.4 Performance evaluation of the Optimised Lambda Architecture

5.4.1 Experiment setup

For the evaluation of the OLA, the same CERN IT on-premises Hadoop cluster that was used in Chapter 4 for evaluating LA was used. The cluster consisted of 15 nodes of Intel(R) Xeon(R) CPU E5-2650 v2 @ 2.60 GHz (8 nodes: 32 cores/64 GB, 7 nodes: 4 cores/8 GB). Hadoop-2.6 and Spark-1.6.0 were configured on all machines. The OLA was also evaluated on the EC2 cloud infrastructure [80], as described in Section 5.4.5.

Three different computing and data intensive algorithms from the WDT use case were used for evaluating the OLA:

Access Pattern - This algorithm works out what are the hot (popular) and cold (unpopular) data. Hot data is very popular among physicists, so they need to be replicated and distributed across many nodes for load balancing, and better accessibility.

1. Inject Log Message event into Map statement.
2. Split the Log Message into several Log Map Events according to the time bins the initial event belongs.
3. Inject each of the Log Map Events into a single Log Statistic Event and compute the following:
   (a) If (client domain == server domain) then remote access = 1 else 0
   (b) If (read bytes + write bytes == file size) then is transfer = 1 else 0
4. Aggregate all the single log statistics:
   (a) If user domain == null then replace it with server username. In case that server username is also null replace it with “n/a”
   (b) If file name == null then replace it with “n/a”
   (c) AVG(file size)
   (d) If (read bytes > 0) then number of read = 1 else is 0

(e) SUM(read bytes)

(f) If (read bytes > 0) then sum(end time - start time) else read time $= 0$

(g) If (write bytes > 0) then number of write $= 1$ else is 0

(h) SUM(write bytes)

(i) If (write bytes > 0) then sum(end time - start time) else write time $= 0$.

5. Aggregate and Reduce all the log statistics:

   (a) If there is not already a time bin for the injected log statistic events then create it and insert (establish connection to Elasticsearch and Bulk insert):

   (b) Else update the existing bin (update the Elasticsearch document version):

Transfer Statistics - This algorithm has already been used in LA evaluation (Section 4.4.4) and works out the average data transfer rate from site A to B. A completed file transfer lasting several hours from site A to site B, also contributes to several time bins in the past. Information about the average traffic from site A to site B has to be updated.

User Statistics - This algorithm works out the number of active users, and how much data they have downloaded within a specified time interval.

1. Inject Log Message event into Map statement.

2. Split the Log Message into several Log Map Events according to the time bins the initial event belongs.

3. Inject each of the Log Map Events into a Single Log Statistic Event and compute the following:

   (a) If (client domain == server domain) then remote access $= 1$ else 0

   (b) If (read bytes + write bytes == file size) then is transfer $= 1$ else 0

   (c) If if user domain == null then replace it with server username

4. Aggregate all the single log statistics:

   (a) SUM(read single bytes)

   (b) SUM(read vector bytes)

(c) SUM(file size)

5. Aggregate and Reduce all the log statistics:

(a) If there is not already a time bin for the injected log statistic events then create it and insert (establish connection to Elasticsearch and Bulk insert):

(b) Else update the existing bin (update the Elasticsearch document version):

5.4.2 Illustration of the workflow

An evaluation of the workflow in the OLA is presented in this section.

The timeline in Figure 5.4 shows sequential job execution, where jobs were performed one at a time. The next job will only be initiated once the previous job has been completed. This workflow is useful when the later job is dependent on the previous job, e.g. when the second job relies on the results computed by the previous (similar to the MapReduce framework).

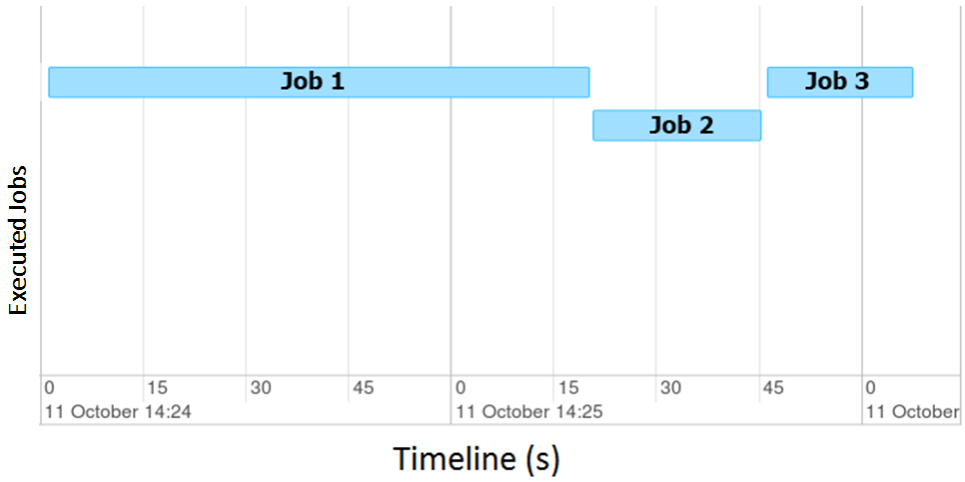


Figure 5.4: Sequential jobs execution. Jobs were executed one at a time.

Figure 5.5 shows parallel job execution, where multiple concurrent jobs are executed at the same time. This is not achievable with the MapReduce framework presented in Chapter 4. This is the workflow that is beneficial for carrying out in-memory computation, meaning data can be loaded into memory, and used by concurrent jobs rather than having each job load data from the storage layer.

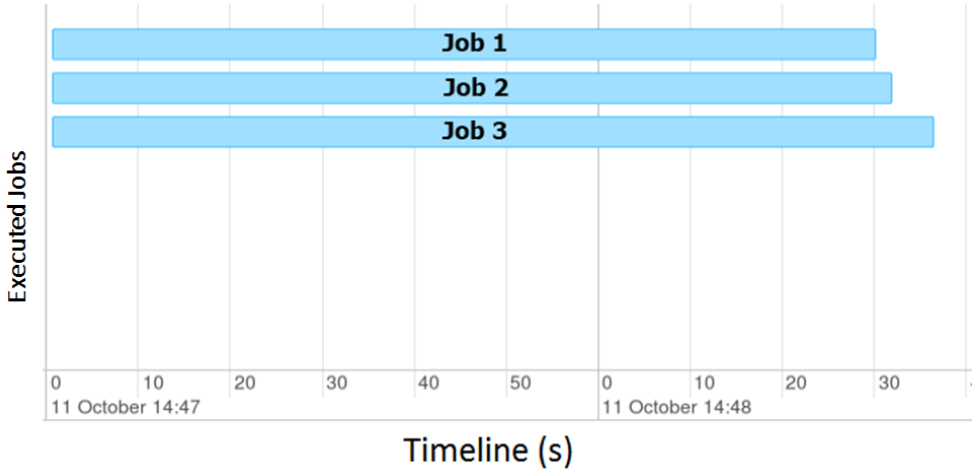


Figure 5.5: Parallel jobs execution. Multiple jobs were executed at a time.

The course of action shown in Figure 5.6 is reasonably straight forward. After all executors required for a job have been registered, the job commences execution. The executors were removed when the job was complete, in order to make resources available for other jobs.

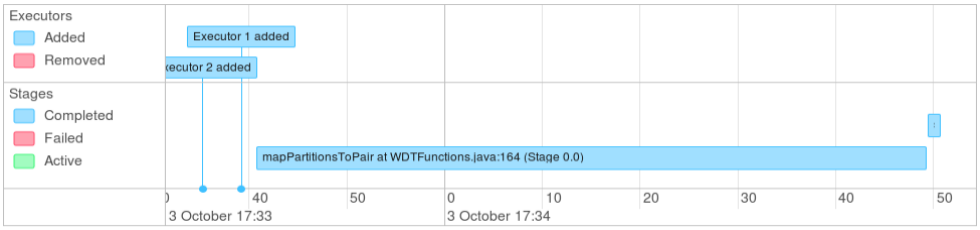


Figure 5.6: Sequential tasks execution.

In Spark, a job is joined with a chain of RDD dependencies arranged in a direct acyclic graph (DAG) as can be seen in Figure 5.7. From the DAG, it can be seen that the evaluated WDT use case (i.e. transfer statistics) first executed a textFile operation to read data from the HDFS, then called the mapPartitions operation to transform the data into Java Objects, calling another mapPartitions operation to extract the required data and to carry out an initial transformation. Subsequently, it then called a reduceByKey function (in the second stage, which is dependent on the first stage) to aggregate the final results, and finally the saveAsTextFile operation was used to save the data into HDFS. It can be seen that each executor immediately applied the subsequent mapPartitions action to the dataset partition after reading it from HDFS in the same task, minimising the

data shuffles between nodes. The black dots in the boxes represent RDDs created by the corresponding actions [81].

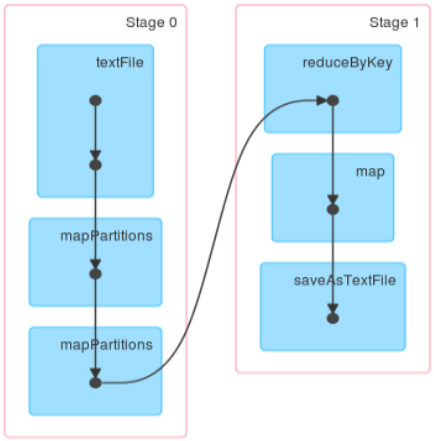


Figure 5.7: Overview of job stages.

Spark supports caching stages into memory so that data may be reused, rather than recomputed. Figure 5.8 illustrates that Stage 3 reads data from HDFS and carries out initial transformation as discussed in Chapter 3, caching into memory (shown greyed out). The subsequent jobs can easily recover the stage from memory, therefore, reducing re-computation time.

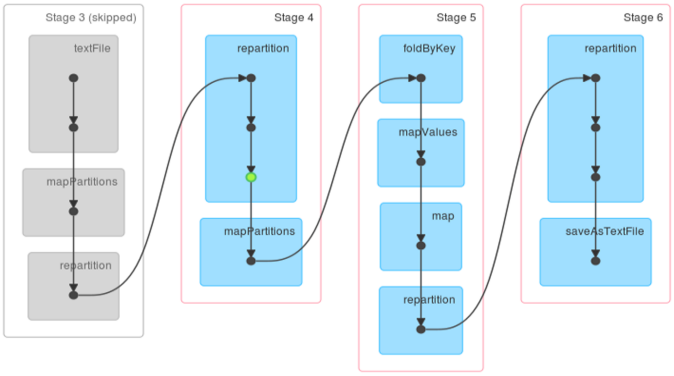


Figure 5.8: Cached stages were reused by parallel jobs. The green circle denotes that an RDD is cached from the previous stage. The greyed stage (cached) was skipped by the following concurrent jobs.

Figure 8 shows an insight into Stage 3 of the event timeline from Figure 6.

Figure 5.9 shows an insight into Stage 3 of the event timeline from Figure 5.8. It shows that tasks are distributed to two worker nodes. Most of the execution time was

spent on computing the statistics rather than scheduler delay, network or I/O overheads. This is not unexpected since the job involves shuffling very little data. Each executor is performing three tasks concurrently, due to the CPU cores which are explicitly configured with the job submission. The parallelism can be increased or decreased in direct relation to the number of cores, which would have an effect on performance.

Figure 5.9: Concurrent tasks execution. A job was split into multiple tasks and executed in each executor CPU core concurrently.

Figure 5.10 shows an insight into the Stage 1 event timeline which aggregates the results, and writes them into the HDFS. It can be seen that there is only one task in Stage 1, which spent most time on computation. There is a noticeable delay in task deserialisation, which is used for deserialising the result sent from the mapper node.

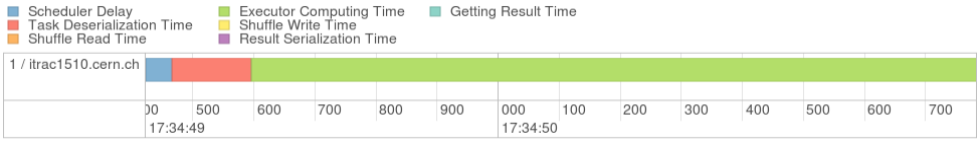


Figure 5.10: Insight into Stage 2 timeline.

5.4.3 Performance evaluation of WLCG environment and WDT use case

In this section the batch computation, as well as the real-time computation of the OLA, were evaluated.

Evaluation of Spark's batch computation

In this section, an evaluation of Spark batch computation over increasing dataset size was carried out that was similar to the evaluation detailed in Chapter 4. This evaluation was carried out on the same dataset so that it could be compared with the MapReduce framework computation. Although Spark supports in-memory computation, it was not used in this evaluation as the job consisted of a single algorithm (transfer statistics). It was unnecessary to persist the dataset into memory as there were no follow-up jobs that could benefit from it. The evaluation results are shown in Figure 5.11. It can be seen that computing 30 days of the dataset overall was completed in ∼2 minutes by the Avro, CSV and JSON jobs. It can also be seen that execution time linearly increases as the dataset size is increased. Nevertheless, the performance was improved when compared with the current approach used by the WDT, which occasionally took longer than 10 minutes. Again the performance pattern of the data types are similar to the MapReduce job, as Avro performed better overall compared to the other jobs. On the other hand, JSON performed poorly when compared with the other jobs. In total, the JSON and CSV jobs took an average of 64% and 14% more execution time compared with Avro, respectively.

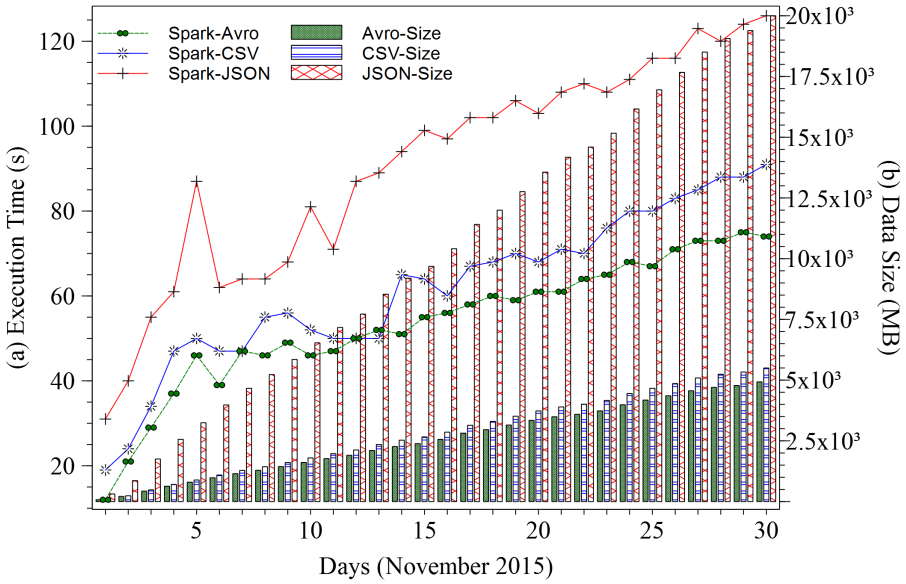


Figure 5.11: Computation of Avro, CSV and JSON files over augmented dataset (day 1 to 30 days). The primary axis (a) shows the execution time that is being represented by lines, whereas the secondary axis (b) represents the input data size in Megabytes (MB) which is represented by bars.

A comparison of the job execution time using MapReduce framework in Chapter 4 and using Spark is presented in Figure 5.12. It can be observed that Spark jobs performed much better when compared with the MapReduce jobs, although data persistence was not used in both frameworks. The first day dataset (smallest) took a lot less time for computing using Spark, whereas computing using MapReduce took significantly more. From this observation it can be concluded that Spark took less overhead time in allocating resources when compared with the MapReduce. Nevertheless, MapReduce job execution time stablised as the dataset size was were increased further as only minor oscillations were seen. Comparatively the Spark execution time increased linearly when the dataset size was increased. This can be explained by the computing limitation of the cluster due to its heterogeneous setup. The Spark-Avro job total execution time on average was 43% less than the MR-Avro job, whereas the Spark-CSV performance improved by 38% when compared to the counterpart, and the Spark-JSON improved 23% compared to its counterpart. All Spark jobs appeared to have performed better than their counterpart, but the most improvement can be seen with the Spark-Avro computation.

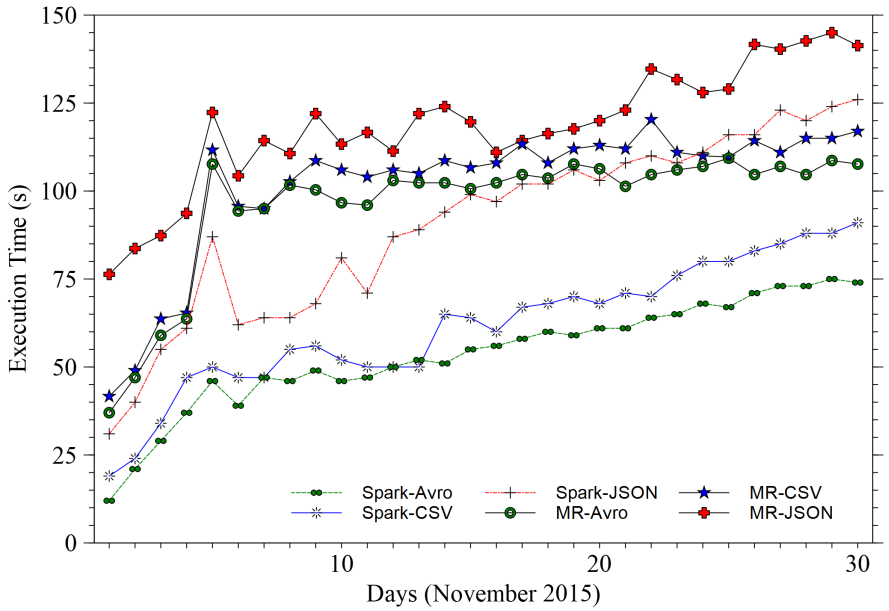


Figure 5.12: Comparison of the MapReduce versus the Spark framework against various data types.

Figure 5.13 shows the execution time over various data partition sizes (i.e. parallelisation, which is the process of splitting the dataset into a number of partitions, and allocating tasks to process each of those split portions). It can be seen that the execution time improved as the number of partitions was increased (execution time decreased). It can also be observed that after the job met a certain number of partitions, the execution time stabilised. This can be explained by the fact that more partitions would require an equal share of tasks, requiring finding resources, allocating, and garbage collections. This would also require shuffling data over the network. It can be observed that the Avro job performed better compared to the other two jobs.

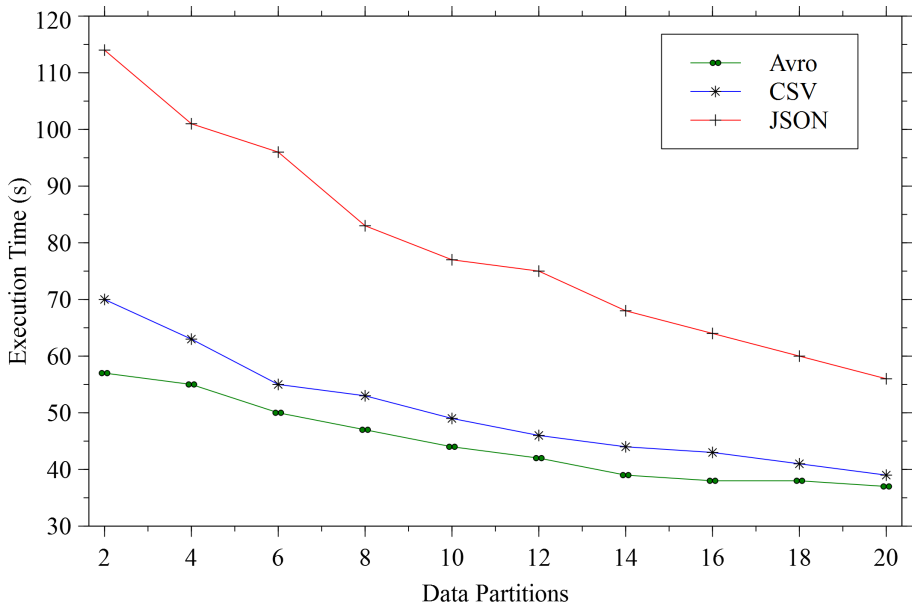


Figure 5.13: Execution time versus the number of partitions of various data types.

In the previous evaluation, only a single job (algorithm) was evaluated, so persisting the dataset into memory was not trivial. In order to benefit from the in-memory computation, it was necessary to evaluate multiple jobs (multiple statistics algorithms) on the same dataset (i.e. derive various statistics from the same dataset). The single job assessed previously only parallelises the tasks, but as multiple jobs may be deployed to profit from in-memory computation, it was essential to evaluate parallel job execution versus sequential job execution. Figure 5.14 illustrates parallel jobs performed better than the sequential jobs; in particular, the cached job performed exceptionally well. However, when comparing the uncached parallel job with the cached sequential job, it is evident that the parallel job performed better, which could only be explained by the simultaneous job execution. The sequential job requires submitting one job at a time so that the next one in the queue can only be submitted when the previous job has been completed. This is not the case for the parallel job as it would submit all jobs at one time. This should not be a problem in the OLA, as it can scale dynamically when there are more demands for resources.

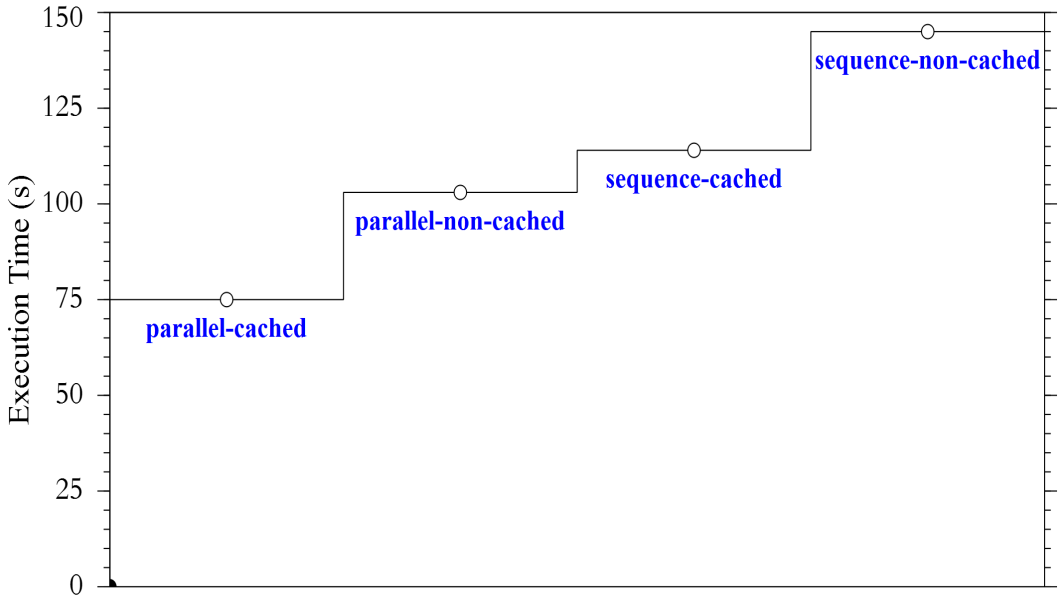


Figure 5.14: Comparison of parallel and sequential jobs with cached and uncached datasets. Execution time versus parallel, sequential cached and uncached jobs.

Figure 5.15 shows the evaluation of execution time over various types of data persistence used in parallel jobs submission. The persistence options are:

- Memory only (MEMORY_ONLY), which only uses the memory for caching the dataset. In the case of a dataset being larger than the memory capacity, it will use the disk for dumping the remaining dataset.
- Memory only with two replications (MEMORY_ONLY_2), which is similar to MEMORY_ONLY but it replicates the dataset two times for improved data availability.
- Memory only with serialisation (MEMORY_ONLY_SER), which is similar to MEMORY_ONLY, but it uses serialisation to compact the data so that more information can be stored into memory as memory spaces are very limited. However, serialising and deserialisation will add computation overhead to the job.
- Memory only with two serialised replications (MEMORY_ONLY_SER_2), which is similar to MEMORY_ONLY_SER but replicates the dataset two times.
- Disk only (DISK_ONLY), which spills the dataset onto the disk.

- Disk only with two replications (DISK_ONLY_2), which is similar to DISK_ONLY but it replicate the dataset two times.
- Memory and disk (MEMORY_AND_DISK), which uses both memory and disk for storage, but some data that need to be persisted into memory are configurable at execution time.
- Memory and disk with two replications (MEMORY_AND_DISK_2), which is similar to the MEMORY_AND_DISK but with two replications of the dataset.
- Memory and disk with serialisation (MEMORY_AND_DISK_SER), which is analogous to the MEMORY_AND_DISK but it uses serialisation to compact the data so that more data can be stored in memory.
- Memory and disk with two serialised replications (MEMORY_AND_DISK_SER_2), which is similar to MEMORY_AND_DISK_SER but with two replications of the dataset.

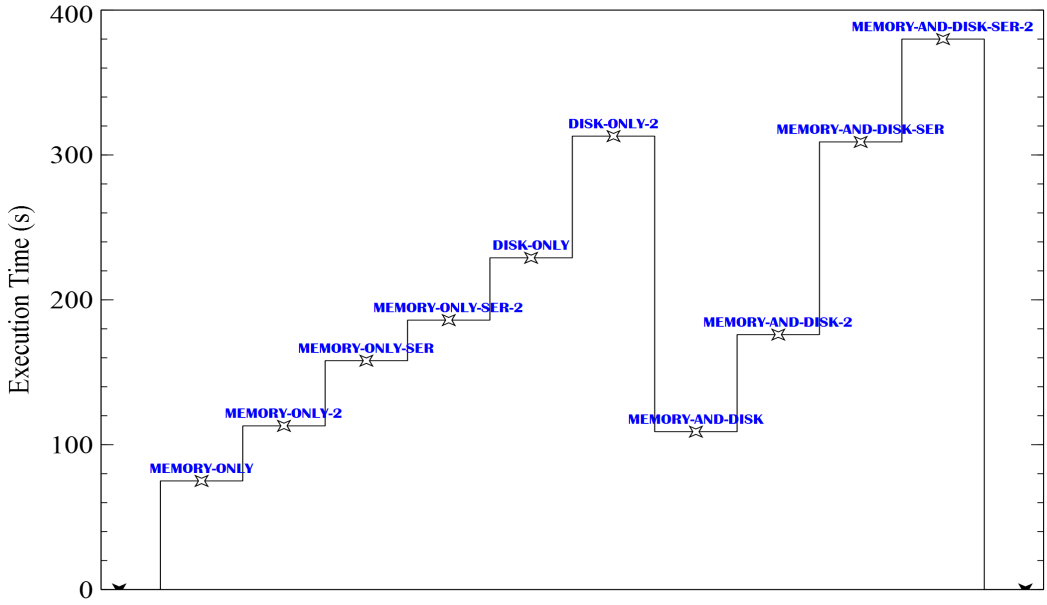


Figure 5.15: Comparison of various cache types. Execution time versus computation of data cached in memory, disk, and memory and disk (also a combination of replicated and serialised dataset).

As shown in Figure 5.15, it is evident that in-memory persistence outperformed the

other methods. Having two replications of the dataset into memory did not improve the performance compared to the single dataset. In general, serialisation did not perform well, which is understandable as extra overhead is required for serialising and deserialising the dataset. Using memory and disk performed better than the pure disk option. It was better to recompute from the source rather than reading the cached data from disk. When clustered and compared the execution time of all memory only, disk only, and disk and memory, the disk only options took 104% more execution time than the memory only options, whereas the memory and disk options took 83% more execution time than the memory only options.

The scalability was evaluated by incrementing the number of executor nodes. Executor were incremented one at a time. The memory size was fixed to 1024 MB. The evaluated dataset size was 7.5 GB, which was used for the following evaluations unless otherwise stated. The total amount of memory allocated for the jobs can be calculated by multiplying the number of executors by amount of allocated memory for each executor (i.e. 1024 MB). The previous evaluations showed that the Avro job performed better compared with the CSV and JSON jobs. Therefore, it was used for the node scalability analysis. Figure 5.16 shows that execution time improved as the number of executors was increased. However, there was a dramatic improvement in performance in increasing up to three executors. With more than three executors, there was not a significant further improvement. The execution time decreased by 64% when three executors were used compared to the initial single executor execution time. However, when nine more executors were used, compared with the single executor, the performance improved by 84%. This shows an only 20% improvement using nine executors over three executors. Over allocating resources (in this case executors) can be wasteful, displacing resources that could have been used for other jobs.

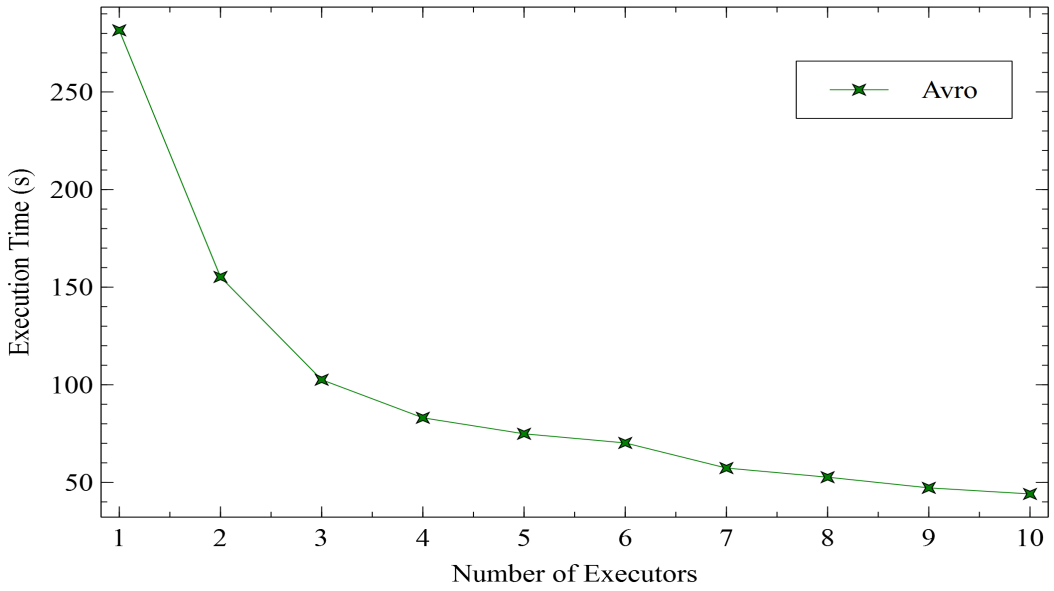


Figure 5.16: Execution time versus the number of executors.

For the evaluation of memory usage, the number of executors was fixed at four (for comparison to the former analysis), the number of CPU cores was fixed at one, and memory was increased by 1024 MB at each evaluation. The total amount of memory allocated for the jobs can be calculated by multiplying the amount of allocated memory for each executor by the number of executors (i.e. 4). The allocated memory for each core would be the same as the executors as the core was fixed at 1. Therefore, it does not need to share the memory. The dataset size was the same as in the previous evaluation, which was 7.5 GB. The performance improved rapidly as the memory was increased, as seen in Figure 5.17. What was indisputable from the results, was that memory plays a significant role in performance. With four executors a better result was achieved by just increasing the memory, rather than using ten executors as can be seen from the previous analysis.

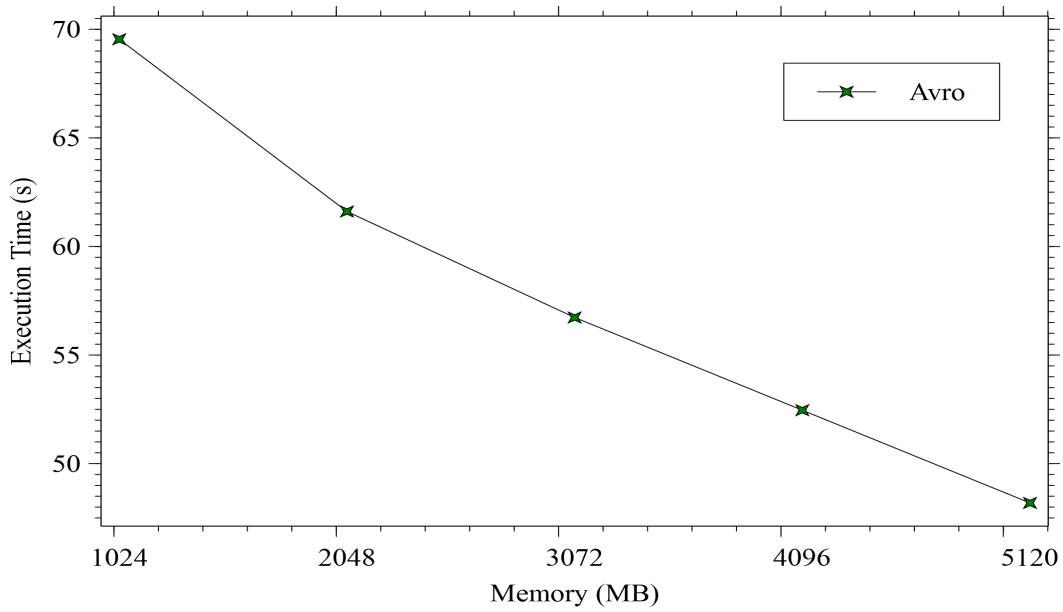


Figure 5.17: Execution time versus the amount of memory size.

In order to evaluate the CPU core utilisation in optimising the performance, the number of executors was fixed at four, and the memory was fixed at 2048 MB. The number of cores was increased by one at each execution. Each executor was allocated 2048 MB memory, but there were four executors so the total amount of memory allocated to these jobs was 8192 MB. The amount of memory allocated to each core was calculated by dividing the memory allocated to each executor (i.e. 2048 MB), by the total number of cores allocated to each executor. Again, the performance was improved as the number of cores was increased as seen in Figure 5.18. The performance improved steadily as the number of cores increased. The observed improvement in the performance was caused by the parallelisation of the tasks.

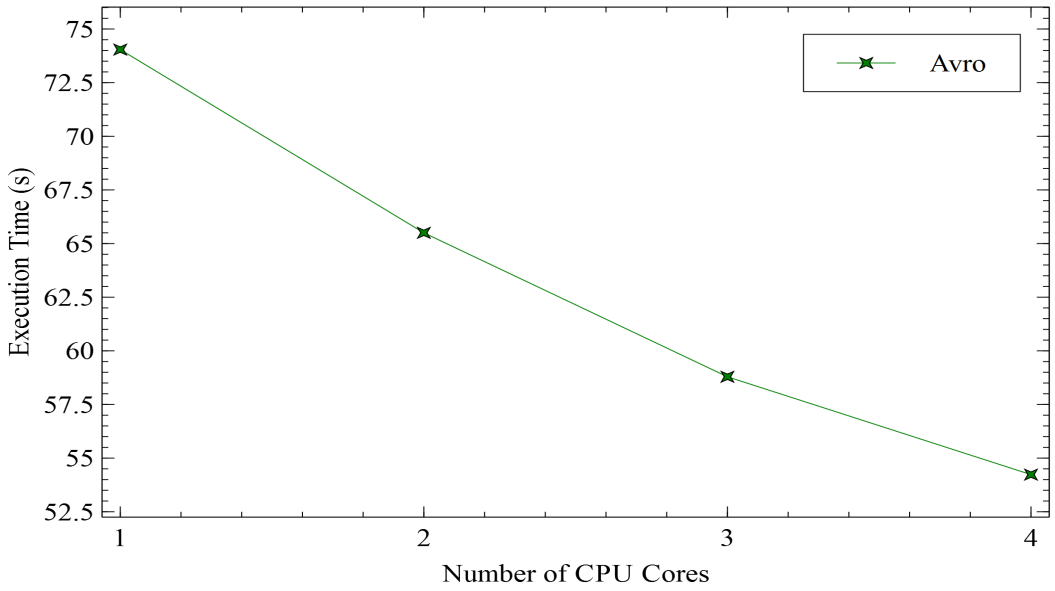


Figure 5.18: Execution time versus the number of CPU cores.

Evaluation of Spark's Streaming computation

In Chapter 4, Esper was used for carrying out real-time computation, which did not support scalability. However, Spark Streaming supports scalability just as it supports batch computation. The performance observed with Esper was reasonable for the WDT as it computed the events as soon as they were received. Nevertheless, to support the foreseen explosion of volume and speed of the data, scalability is required, so Spark Streaming was investigated. Despite this, it needs to be evaluated to see how it performs on a real life scientific application, which was the same algorithm that was used for evaluating batch computation (i.e. transfer statistics). A few metrics are important in evaluating the streaming layer. One such is the event input rate at which data is being received, while the other is the processing time of each micro-batch. The streaming layer was deployed with three executors, each with 2048 MB memory and three cores. The streaming layer was evaluated on the last 1000 batches of streamed data. The streaming layer was run for ∼15 hours at a two seconds batch interval prior to the evaluation. At the time of the evaluation, the streaming layer had completed ∼27 thousand batches and computed ∼5 million records.

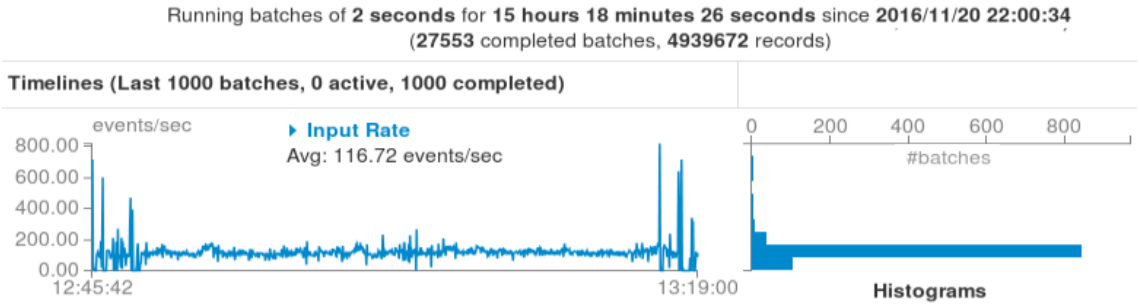


Figure 5.19: Streaming data input rate. Streaming job receiving data at a rate of 116 events/second on average.

Figure 5.19 shows that the streaming layer was receiving data at a rate of about 116 events/second on average across all its sources. The streaming layer is capable of handling a much larger rate than the one shown in Figure 5.19. However, the source was sending a relatively low load of events at the time of evaluation.

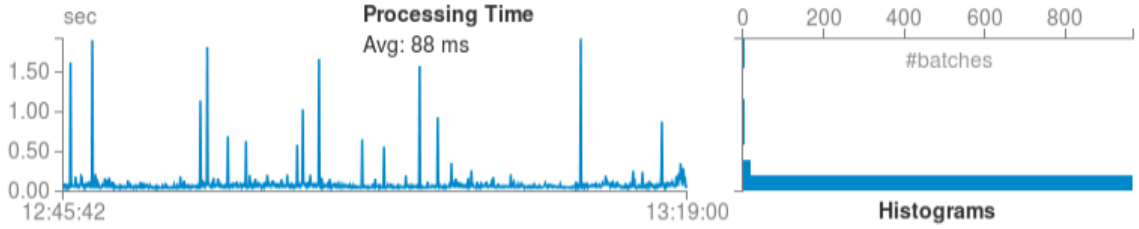


Figure 5.20: Streaming data processing time. Processing time shows that these batches have been processed within 88 ms on average.

Figure 5.20 presents processing time which shows that these micro-batches were processed within 88 ms of being received on average. Displaying a reduced processing time compared to the batch interval means that the scheduling delay (which is the time a batch waits for previous batches to complete [81]) was almost zero as seen in Figure 5.21. It can also be noted that there were a few spikes on the schedule delay, including when there was a sudden peak in data input rate which increased the schedule delay by 16 ms. The scheduling delay is the key indicator of whether the streaming layer is stable or not [81]. In this particular evaluation it indicated the streaming layer was very steady.

Figure 5.21: Schedule delay in processing next batch.

Figure 5.22 shows that the total delay in scheduling and processing the batches was 105 ms on average. This means the transfer statistics can be presented to the end user within a second.

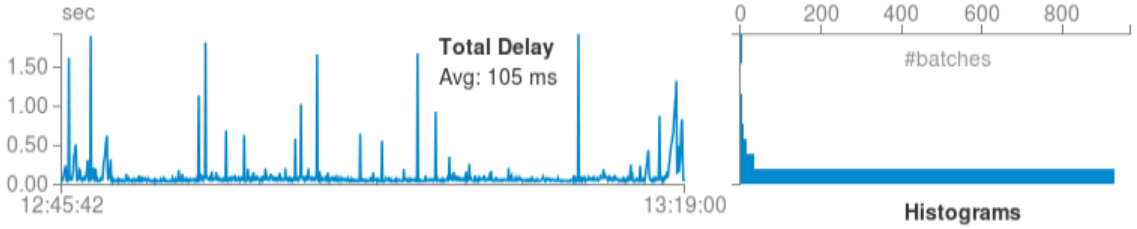


Figure 5.22: Total delay in scheduling and processing streaming data.

5.4.4 Evaluating the accuracy of monitoring computations

To evaluate how accurately the architecture was able to compute the WLCG sites throughput in time-series, all three OLA approaches were tested. As shown in Figure 5.23, the stateless batch job was scheduled to run every five minutes and carry out batch computations on the data stored in HDFS. However, the plot highlighted in Figure 5.23 shows that some data is missing. This is due to the latency of the batch computation and the unavailability of the data when the job started.

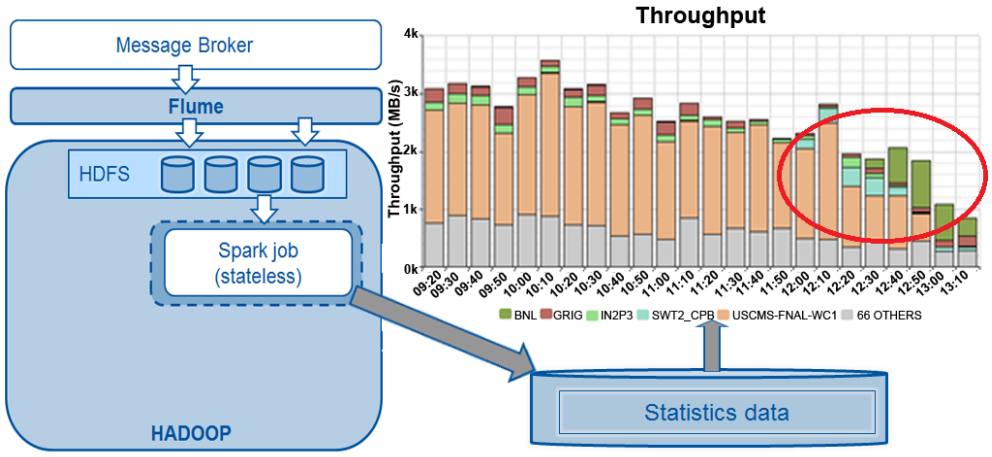


Figure 5.23: The Spark batch computations for WLCG monitoring (some statistics are missing as highlighted).

Figure 5.24 represents the combination of batch and streaming approach. This approach shows the computation in real-time as highlighted in the plot. This shows that the combination of batch and streaming approach is capable of providing up-to-date statistics that are beneficial to the users in comparison with the pure batch computation approach. The Spark batch computation performed better than the MapReduce job presented in Chapter 4 due to the use of in-memory processing. The intermediate results were cached into memory in comparison with the former approach, which utilises the disk for reading and writing.

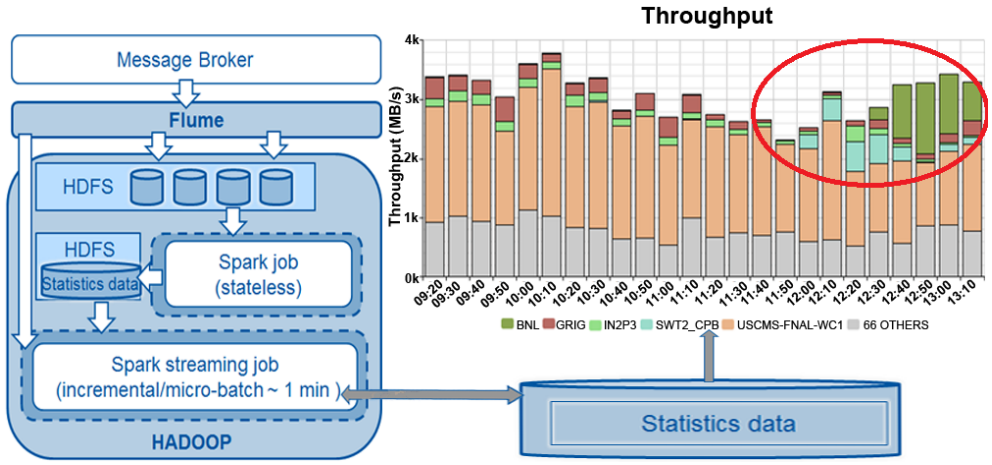


Figure 5.24: The Spark batch and streaming computations for WLCG monitoring (statistical data are in near real-time as highlighted).

5.4.5 Evaluation of scalability, on the Amazon EC2 cloud cluster

In the previous section, the OLA was evaluated on the CERN IT on-premises Hadoop analytics infrastructure (shared). The architecture was also evaluated on a public cloud infrastructure to understand how portable the architecture is. The purpose of the evaluation was not to compare the performance of the on-premises Hadoop analytics infrastructure and the cloud infrastructure, but solely to understand the flexibility of the OLA model. The metadata from the ATLAS datasets were used for the evaluation of the on-premises Hadoop analytics infrastructure, whereas metadata from the CMS datasets were used for the evaluation of cloud infrastructure. In this section, various scalability properties were evaluated on the Amazon cloud cluster such as the number of cores, memory size, and the number of executors. All three algorithms discussed in the previous sections of this Chapter were used to evaluate the parallelism. Taking this into account, the performance on the cloud may vary in comparison with the CERN IT Hadoop cluster.

Cloud computing is an approach that enables sharing of computer resources over the Internet. The resources are shared among the consumers, and they can be dynamically allocated and de-allocated on demand. Consumers obtain the resources from cloud service providers based on a metering system. The consumers only will be charged for the resources that are utilised (pay-as-you-go model). To assess the portability aspect of the OLA presented in this chapter, a virtual cluster was created in the Amazon Elastic Cloud (EC2) using a general purpose instance "m4.2xlarge" that had eight virtual CPUs, 32 GB of memory, and 20 GB of storage per instance. The cluster was configured with four nodes, one name node and three data nodes. The same Hadoop, Flume and Spark versions, and operating system were installed on each instance. The OLA was ported into the EC2 cluster with ease as it is a software architecture that is happy to run on any cluster that supports Hadoop. For conducting the tests, the job was submitted with various scalability properties. At each execution, a parameter was changed, and the rest remained fixed.

Executor memory

Although there were 32 GB of memory available in each node, it is possible to limit how much memory should be allocated to a job. For this evaluation, the number of executors

was fixed at four. Then, the jobs were submitted with varying memory sizes, such as 2 GB, 4 GB, 6 GB, 8 GB, and 20 GB for each executor. Since there were four executors running, the total memory used for each test was 8 GB, 16 GB, 24 GB, 32 GB and 80 GB. In general, the performance was improved as the memory was increased. In particular, the performance from 2 GB to 8 GB in the execution time was improved by 48%. What is evident from the Figure 5.25 was that the difference in execution time is not substantial when increasing from 8 GB to 20 GB; in fact, it varies by just 10 seconds. This difference can be explained by the fact that when 8 GB per node is used, the total available memory for the jobs is 32 GB; more than enough to accommodate 24 GB of data that is required to be processed. Any additional memory would not have a huge impact on job performance, as it would mostly remain unused.

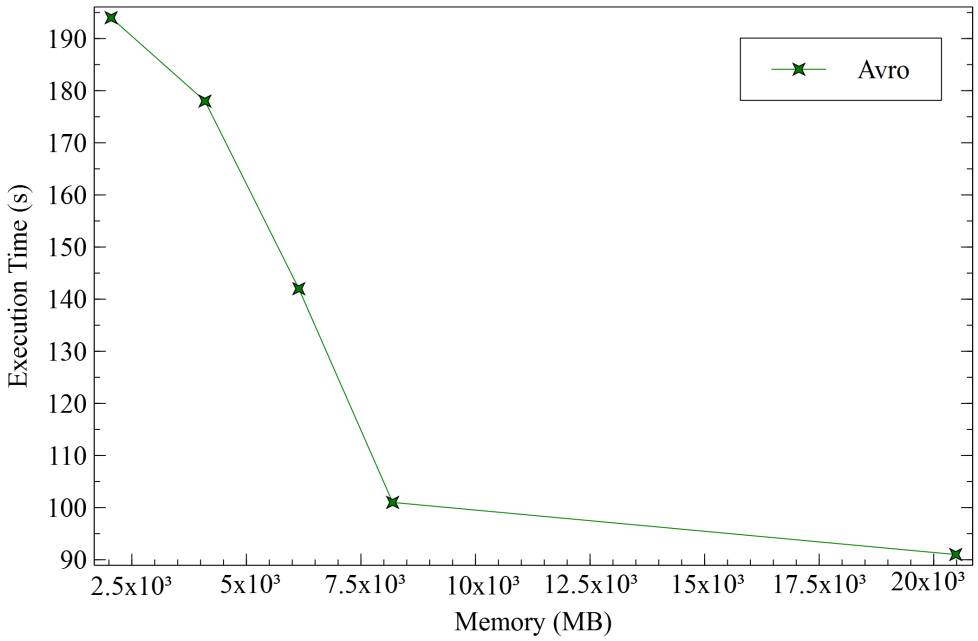


Figure 5.25: Execution time versus the memory size on the cloud infrastructure.

Executor instances

The evaluation of the executors was carried out in order to measure how performance would be impacted when changing the number of executors. For this test, the amount of memory used for each executor was fixed at 4 GB. The number of cores per executors was fixed at two. As seen in Figure 5.26, the execution time was improved by 76% using four

executors, when compared with just one. The execution time was seven seconds slower when five executors were used in comparison to four. This was in part due to there only being four virtual nodes available in the cluster. When there were five executors, one of the nodes would run more than one executor, contributing to an uneven distribution of the job. Ultimately, this would cause an overloaded node.

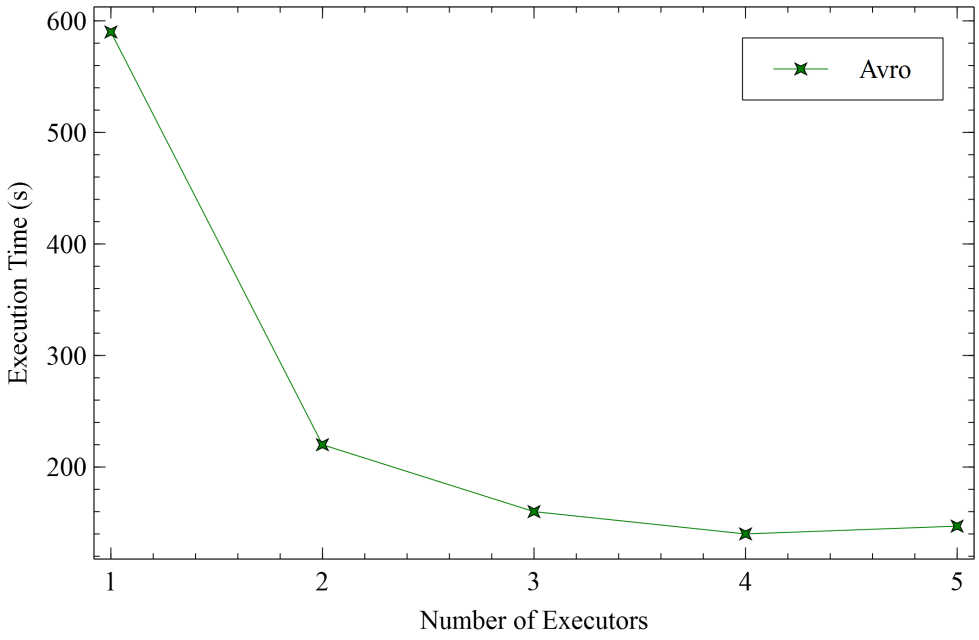


Figure 5.26: Execution time versus the number of executors on the cloud infrastructure.

Executor cores

The executor cores parameter defines the number of tasks that each executor can run concurrently. In this test, the number of cores per executor was analysed as shown in Figure 5.27. The amount of memory used for each executor was fixed at 4 GB, and the number of executors was fixed at four, so all nodes would be utilised. The performance improvement of using eight cores over 2 cores was 69%. No difference was observed between using eight or ten cores. This is due to the fact that the maximum number of virtual CPU cores available in each node is eight.

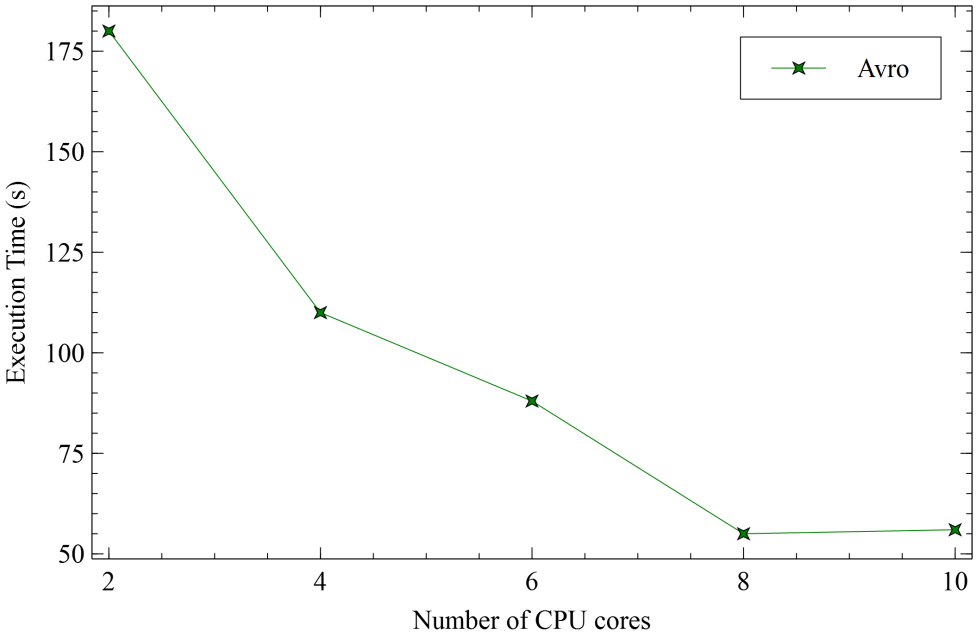


Figure 5.27: Execution time versus the number of cores on the cloud infrastructure.

5.5 Summary

The three data monitoring approaches presented in this chapter outperform the RDBMS based system and the Lambda Architecture that is used by the WDT in terms of execution time, low-latency, maintenance, and scalability. In particular, the streaming approach provides the up-to-date state of the infrastructure. The evaluation also shows that Optimised Lambda Architecture can be ported into other computing infrastructures with ease, as it was demonstrated in the CERN IT on-premises Hadoop cluster and a cloud infrastructure. On completion of the work described in this chapter the WLCG monitoring group has adopted the Optimised Lambda Architecture, a combined batch and streaming approach, and has been using this approach for monitoring the monitoring the WLCG data activities since October 2015 [82]. Since the deployment of the Optimised Lambda Architecture, the WDT tools have been able to monitor infrastructures (e.g. EOS data storage) that it was once assumed would have been impractical. It has also saved operational time as well as computation time in comparison to the traditional architecture formerly used. With the traditional workflow consisting of local filesystems (dirq) and local collectors, PL/SQL computations had several hours of operational time per week dedicated to

cleaning up the partition of the machines and maintaining services. With the Optimised Lambda Architecture now implemented the corresponding operational time has reduced to almost zero. An estimated 0.5 days/week is saved through use of the Optimised Lambda Architecture. In terms of computation time, the Optimised Lambda Architecture utilises real-time computation, whereas the traditional architecture required recomputation at a regular intervals. The Optimised Lambda Architecture batch layer reduced computation time by a factor of five when compared with the traditional PL/SQL system, and by a factor of two when compared with the original Lambda architecture.

Chapter 6

Real-time processing and Machine Learning for forecasting data access pattern

In most data-intensive experiments it is normal to collect, replicate, distribute, and store petabytes of data on a heterogeneous computing infrastructure. Data management systems are designed to handle the demand of users for a wide range of scientific analyses. Typically, data movement and accesses are recorded in the form of metadata by a collection system. Using this information, a model can be built to forecast data access patterns for efficient data management and placement. This chapter presents some insight into studies on data popularity and access pattern evaluating the Machine Learning Library from the Apache Spark stack using a supervised K-Nearest Neighbours algorithms for forecasting data access pattern. In recent years there has been an explosion of Deep Learning techniques, and there has been a lot interest from the scientific community as well as the commercial sectors. Based on the data access pattern prediction, the number of data replicas required for efficient resource utilisation can be calculated, and a decision can be made as to whether enough (do nothing), too few (add), or too many (delete) replicas have been found. This chapter presents both a batch learning technique, which makes predictions based on historical training data using Spark, and a technique combination of a batch and online learning technique, actively adapting and updating the model as new data are streamed in from Spark Streaming for prediction. The techniques will be evaluated as an

additional layer of the OLA, which was presented in Chapter 5. This intelligence layer (data access pattern model) is the final addition required for the building of a modern state-of-the-art monitoring system for scientific infrastructure.

6.1 Introduction

Many things are vital for monitoring scientific infrastructure. This includes following specific analysis jobs and tasks, identifying and investigating inefficiencies and failures, and identifying trends while predicting future requirements. This information can be used for efficient resource allocation. Although real-time monitoring of scientific infrastructure is a stepping stone towards a more effective monitoring system, it is not entirely a complete solution. Without an auto-adaptive mechanism human interference is required, which is not ideal as outlined in this section. The OLA presented in Chapter 5 has demonstrated that batch and streaming layers can be synchronised to perform real-time analytics on monitoring events. It will be beneficial to add an intelligence mechanism into this architecture to understand how easily an adaptive model could be implemented. Various use cases can be studied for an adaptive model, such as adopting a classical pattern matching approach to promptly detect errors and failures in the streaming of monitoring events including unauthorised access detection. However, since the research presented in this thesis is related to Big Data, the focus will be on managing the large volume of data. In scientific experiments, large volumes of data are generated, transferred, stored, and analysed. Efficient data placement is critical when dealing with large scale infrastructure as it plays a significant role in efficient resource allocation and deployment. An efficient data placement strategy, also known as Data Access Pattern (DAP), was implemented on the batch and streaming layers. This is of great importance, given that the scale of the computing problem will increase far faster than the resources available to the experiments.

In the modern distributed scientic community it is common that accessing raw data and simulated data via analysis jobs takes place at computing sites. The details of this can be collected and saved for analytics (e.g. the timestamp of the access, the name of the site hosting the data, the number of existing replicas, the location of such replicas on

the computing infrastructure, and the amount of CPU time used to access such datasets). The basic idea is to profit from the experience that can be gained from studying the DAP in the past and present to perform prediction on data accesses in the future.

To date, no studies has been carried out to understand the use of DAP on streaming data. However, there are a few studies already exists with the DAP using traditional data mining techniques [83], which are single node models. These patterns, if understood, can be used as input to the simulation of computing models to optimise existing systems, and to explore next-generation CPU/storage/network co-scheduling solutions. This study aims to use the information coming from DAP studies to improve the use of computing resources (i.e. optimising disk occupancy, minimising the number of dataset replicas, etc.). The motive behind this modelling is to be proactive, as well as adaptive since past models cannot be applied to future prediction for an extended amount of time. Only adaptive models would equip scientific experiments with up-to-date predictive power in the long term. The DAP, despite being relatively simple in definition, is a complex use case once it is deeply investigated.

6.2 Data Analysis and Data Modelling

The DAP study was conducted on a data-intensive data management system known as the ATLAS Rucio Distributed Data Management (RDDM) system [84]. The RDDM operates on files, which usually contain many events. Events taken under the same detector conditions can be distributed over multiple files, necessitating some sort of aggregation. For this, RDDM has the concept of datasets. Datasets consists of one or more files [84] and they are the operational unit for the RDDM. Only datasets can be transferred between sites, files cannot.

The copies of a dataset on a given site are called replicas. To create a new replica all files in a dataset are copied to a site in the grid network and registered as a new replica in the system. A dataset in the RDDM can have multiple physical replicas. The distribution of datasets is based on policies defined by the Computing Resource Management system [84], guided by the ATLAS computing model and operational constraints. For each type

of dataset, a minimum number of replicas is defined. To run an analysis on this data, users send their jobs to a workload management system called PanDA [85]. PanDA takes the user jobs and schedules them to run on a site that hosts a replica of the dataset needed for analysis, based on availability and the workload of the site as well as the jobs' priority. Each time PanDA accesses a file in the dataset it sends a report to the RDDM tracer system [84]. This report includes information about the corresponding dataset, the site it was accessed on, the user, the starting and ending time, and the download status.

6.2.1 Data Acquisition

The raw data for the DAP study was acquired from the ATLAS experiment that was used for both training and testing the DAP model. As the dataset was in a raw form it needed to be pre-processed.

6.2.2 Data Pre-Processing

A number of pre-processing methods were applied to handle noisy and missing data which might disrupt the decision-making process. The pre-processing method involved converting raw data into a machine-understandable vector format and removing data that did not have required features. Low-quality data will produce poor quality results. It is important that the quality of the training data could not be compromised.

6.2.3 Initial DAP dataset analysis

The aim of this study was to demonstrate and evaluate how an intelligence layer can be deployed on the OLA. However, the OLA is a distributed architecture leading to a complex and time-consuming process to build a DAP prediction model without understanding the DAP dataset. It was first evaluated on a traditional machine learning library (Orange Canvas [86] and R [87]) in place of a distributed machine learning library. This analysis assisted in determining the features for independent and dependent variables for the DAP model.

Figure 6.1 illustrates the most popular datasets each month in 2016 from January to

June. It is notable that most of the popular data consistently remains as top accessed data at each month. Figure 6.2 details dataset access in January. It is apparent that many datasets were not accessed frequently, but would become popular due to seasonal or sudden interest which happens consistently in the scientific domain. This plot is useful in determining which datasets are popular, but does not assist in determining how popular these datasets will become. It is important to use a regression algorithm to forecast the DAP. Therefore, the prediction output would be a continuous number instead of a binary option such as popular or unpopular.

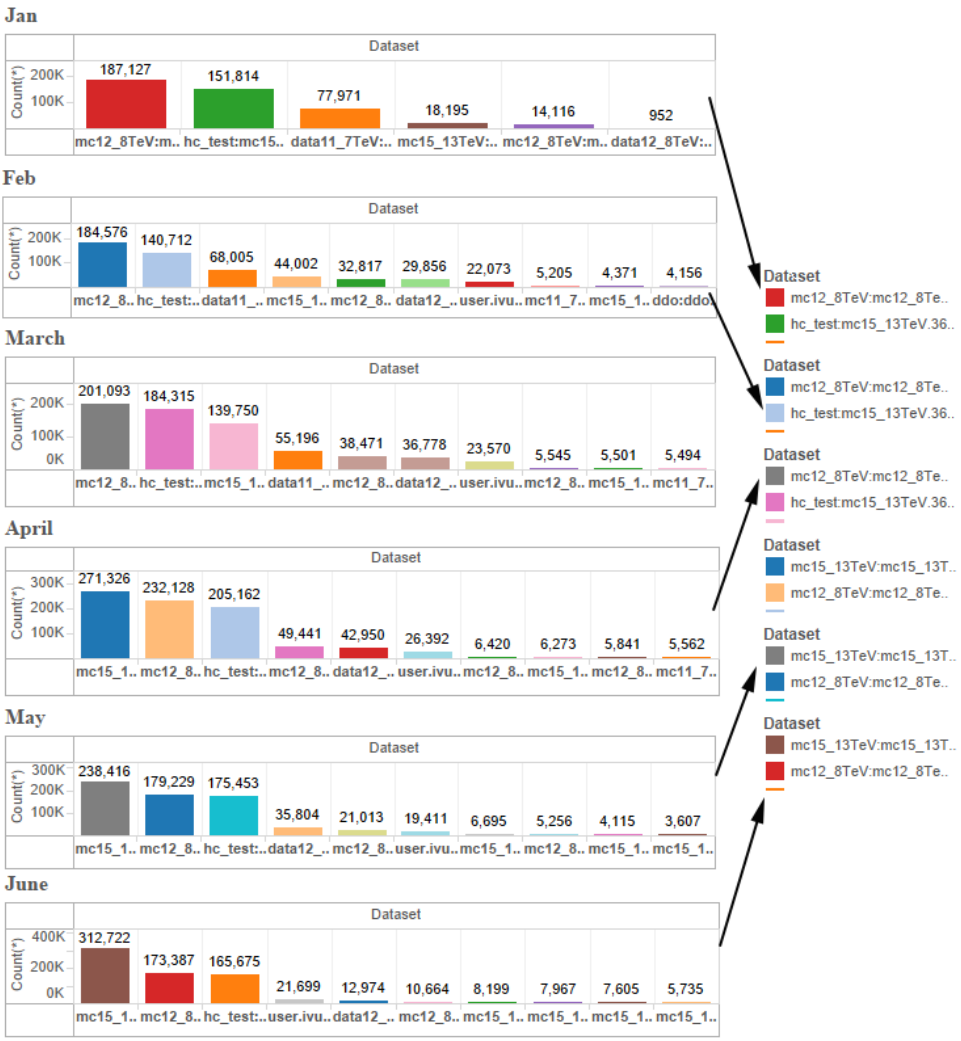


Figure 6.1: Monthly popular dataset trends.

The predicted DAP gained from the analysis can be used to add and remove dataset

replicas to enhance data distribution. Normally, data arrangements consist of two chains that have to operate hand-in-hand; the reclaiming of disk space at sites (i.e. clear disk space), and the creation of new replicas. For the creation of replica datasets, the system must rely on the results of the dataset access prediction, which can be used to place replicas evenly across the available computing resources. This would enable jobs to be executed simultaneously on dierent data nodes instead of having to wait for the other occupied job to be completed. By adding new replicas the number of job slots able to process a particular dataset becomes manageable, and the workload management system has more options for where to put jobs. Ideally, this would lead to lower overall waiting times for the user and better resource utilisation.

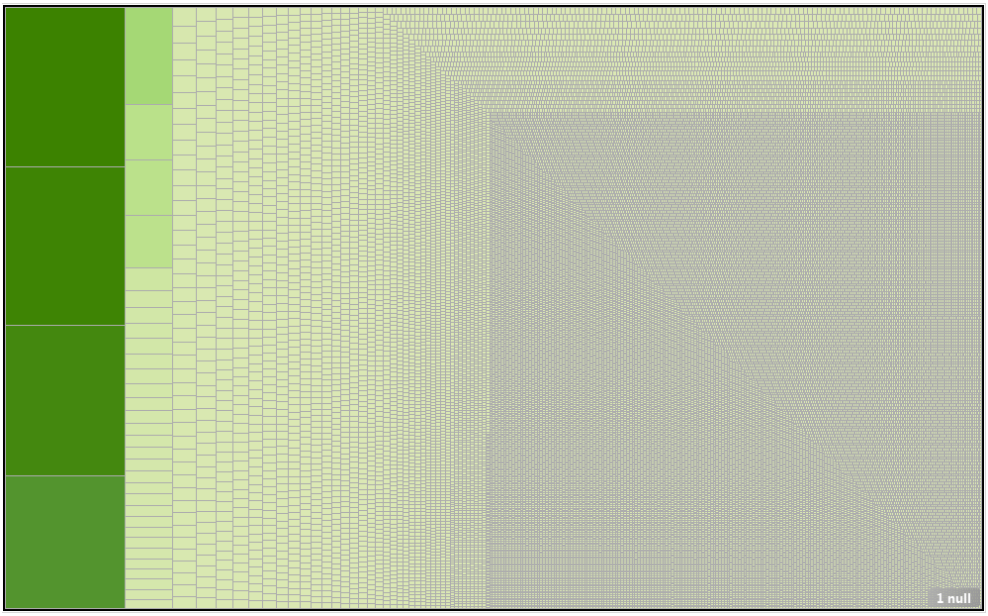


Figure 6.2: Popular dataset trends in January. It shows all datasets accessed in January and each boxed scaled by the number of access.

In order to understand the DAP dataset and trends, the following studies were conducted: using mean calculation on dataset access history, and a simple auto-regression model.

Mean Analysis: Five Days of Data Points

In order to find the top ten maximum requested datasets (indexes) for each day an aggregation was carried out which is shown in Table 6.1.

Table 6.1: Index numbers of top ten accessed datasets each day.

Day 1	259953	179873	232440	47193	101738	52285	164062	30944	174116	234065
Day 2	259953	179873	101738	232440	47193	52285	164062	30944	174116	140942
Day 3	259953	179873	101738	232440	47193	52285	164062	30944	174116	140942
Day 4	259953	179873	232440	101738	47193	52285	164062	30944	174116	140942
Day 5	259953	179873	232440	101738	47193	52285	164062	30944	174116	140942

A pattern can be identified in the frequency of dataset access, though some of them were in different orders. The plot in Figure 6.3 shows the evaluation of frequency of data access for the top five maximally requested datasets for five days from the list found above. These datasets were mostly stable with little oscillation.

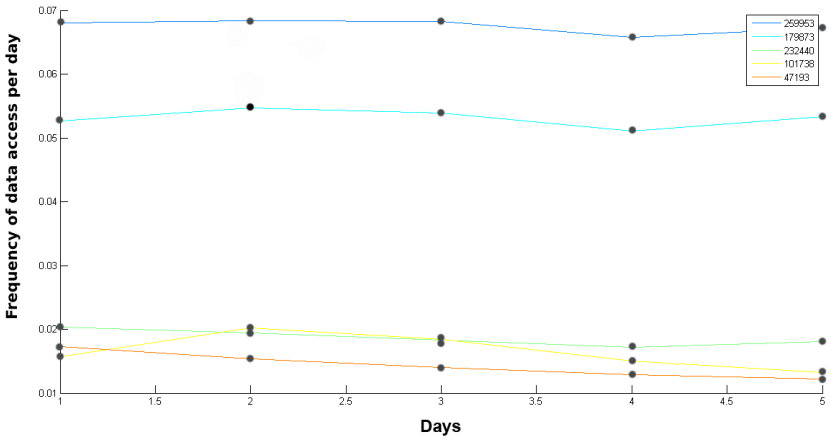


Figure 6.3: Evaluation of frequency for top five maximally requested datasets.

It cannot be concluded that there was a stable frequency of data access by just inspecting the top ten datasets due to the small amount of data points (five days). Possibly a wider time range (more days data points) analysis was required to extract further information about the patterns which can be found in Section 6.2.3. If it were to be assumed that

behaviour will be the same for other days (small oscillation upon some stable level) then these "stable datasets" would not need a prognostic model as frequency of data access is constant.

Mean Analysis: Thirty Days of Data Points

Stability across a wider time range (30 days) was analysed in order to further understand the DAP. To assess the quality of the prediction the Sum of the Squared Errors (SSE) was used.

SSE equation:

$$SSE = \sum_{i=1}^{N} (x_i - \hat{\mathbf{x}}_i)^2 \tag{6.1}$$

where x_i is the actual observations and $\hat{\mathbf{x}}_i$ is the forecasted values.

The mean was calculated for the first 15 days, while the SSE for the remaining days was predicted/calculated as well. The plot in Figure 6.4 shows the SSE for the mean prediction for 100 datasets. The idea was that if the behaviour was the same, then the use of mean frequency for forecasting would be appropriate. However, the plot demonstrates that the predictions were not great as there were high error rates.

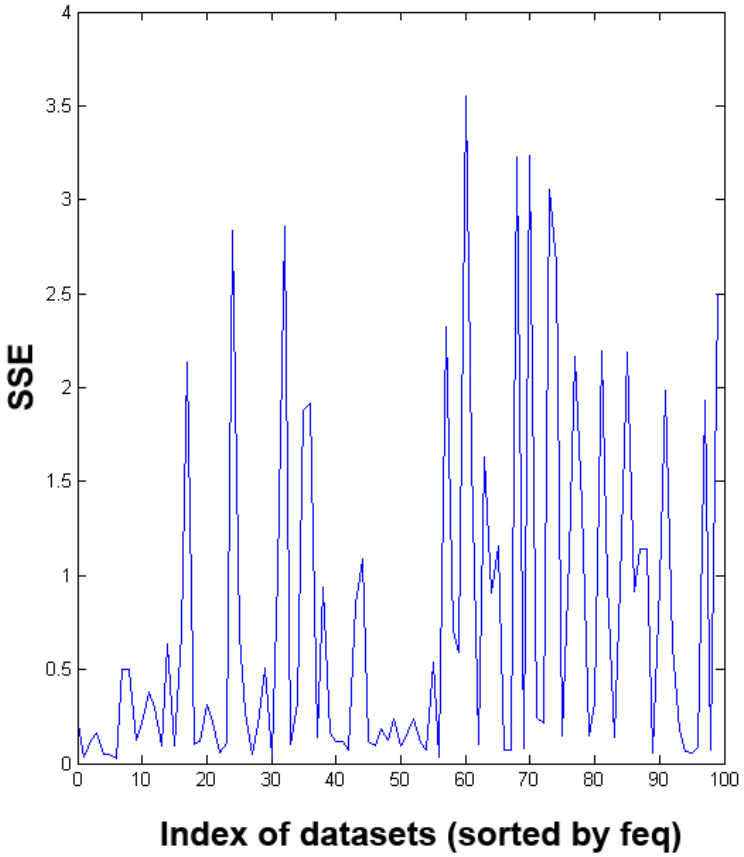


Figure 6.4: Mean forecasting the top 100 datasets.

6.2.4 Auto-regression analysis

An auto-regression model functions under the assumption that past values have an effect on current values. This suggests that the past DAP might predict the current DAP. In auto-regression model, the future values are estimated based on a weighted sum of past values as shown in Equation 6.2. The $\phi_1 y_{t-1}, \ldots, \phi_p y_{t-p}$ are the past observation parameters of the model, μ is a constant, and u_t is the error terms.

$$Y_t = \mu + \sum_{i=1}^{p} \phi_1 y_{t-1} + u_t \tag{6.2}$$

An auto-regression analysis was performed on the DAP dataset. Results of auto-regression for 100, 10,000 and 100,000 most accessed datasets are shown in Figures 6.5, 6.6 and 6.7 respectively. The left plot in each figure highlights the application of the auto-regression model to the study set (the set of data that was used to study the model),

while the right plot in each figure shows the application of the auto-regression model to the testing set (the set of data that was not used to study the model). It is notable that some of the data forecasting is good as goodness of fit of the auto-regression model in Figures 6.5 and 6.7 shows it fits well on the test datsets based on the model built by the training datasets, while Figure 6.6 shows a poor goodness of fit.

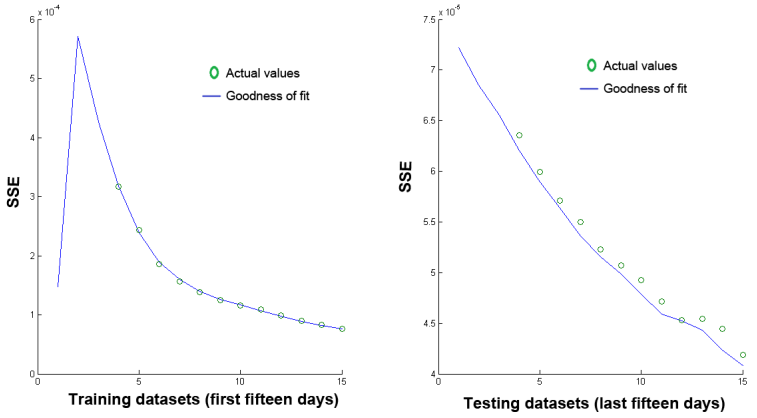


Figure 6.5: Auto-regression on 100 datasets.

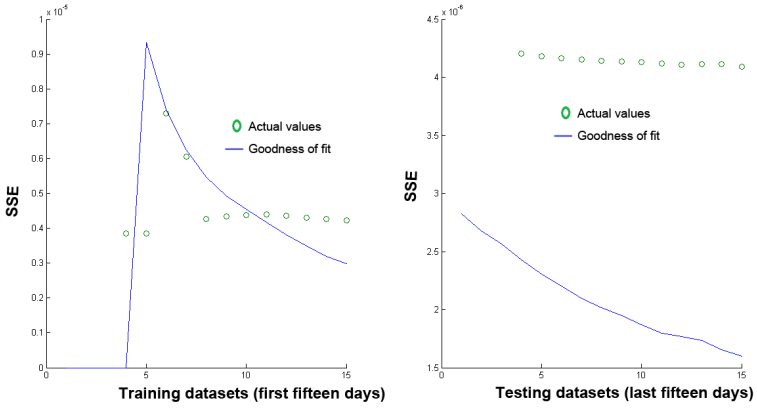


Figure 6.6: Auto-regression on 10,000 datasets.

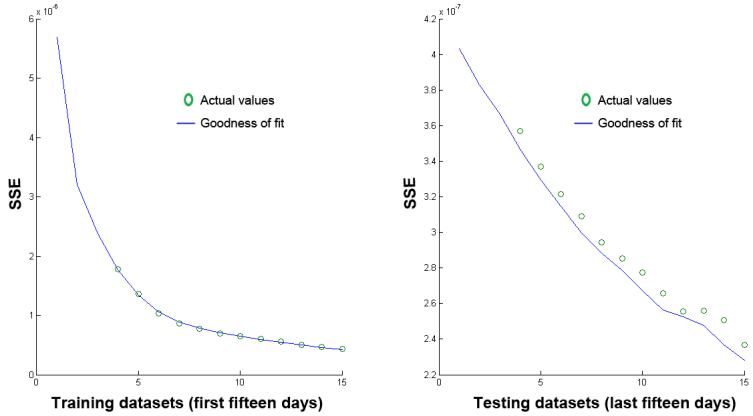


Figure 6.7: Auto-regression on 100,000 datasets.

6.2.5 Comparison of auto-regression and mean forecasting model

To compare the results of the mean forecasting model and auto-regression model, the SSE for all the datasets presented in Figure 6.1-6.2 are presented in Tables 6.2 and 6.3. From the results, it can be noted that both prediction models were poor as the error rate was very high.

Table 6.2: Summed error for mean model.

Dataset index	**SUM(err_d_mean)**
1-100	74.3357
10000-10100	134.2859
1000000-1000100	118.5729

Table 6.3: Summed error for auto-regression model.

Dataset index	**SUM(err_d_mean)**
1-100	58.0791
10000-10100	203.3239
1000000-1000100	324.8026

From this analysis it was apparent that time-series analysis was required. The idea

for the DAP study was to aggregate accesses for each dataset by year, month, week, day, or hour (time-series) and subsequently feed this to a machine learning algorithm in order to get an access prediction for the next year, month, week, day, or hour. A regression algorithm was required for forecasting the DAP. However, an appropriate machine learning technique was also required, as the mean and auto-regression models did not provide good predictions. A suitable algorithm needed to be selected that could identify deep connections with independent variables, which was not apparent in the analysis presented in Section 6.2. Additional independent variables were required for finding coherent pattern.

6.3 Machine Learning Techniques and Algorithms

It was important that chosen Machine Learning algorithms supported the underlying technology used by the OLA, which was Apache Spark. Therefore, the Machine Learning Library (MLlib) from the Apache Spark ecosystem [54] and the Sparkling Water (H2O) [52] framework were used in the DAP study. These frameworks were expected to reduce DAP training time significantly while assuring the needed level of accuracy. The adaptive model does not require retraining as the model will be actively updated. The Spark batch offline learning of DAP should reduce the training time and handle the large volume of data, which has been an issue with the traditional solution as it was limited by the machine specification (disk space, memory, and CPU). Online training for the DAP, on the other hand, introduces the concept of incremental prediction and updates of the model. The main distinction between the two approaches is that while the batch approach supports historical DAP analysis, the streaming approach supports current trend study which has reduced training time requirements. The online method avoids retraining, which is accomplished by utilising an incremental update approach by using new training samples as they become available. Newly acquired training samples are used in conjunction with the batch model for presenting an adaptive model which not only looks at the past, but also the present in order to create predictions. This approach fits harmoniously with the OLA, while maintaining an up-to-date model.

Many distributed machine learning methods exist that can be utilised for the DAP study. Supervised machine learning techniques such as Naive Bayesian, Random Forest,

Gradient-Boosted Trees, and Decision Trees are well-used regression techniques [88]. However, the supervised K-Nearest Neighbour (KNN) algorithm is a simple technique which does not generalise the model, but instead uses available data to make a decision [89]. The Deep Learning (DL) technique was also utilised in the DAP study as it is capable of identifying hidden interconnection between variables [17]. These qualities justified the decision to use it as a supervised machine learning regression technique for this research work, especially, in terms of guaranteeing that the volume and velocity perspectives of the data access pattern challenge were tackled.

6.3.1 Sparks K-Nearest Neighbours (KNN)

The Spark KNN is very simple yet works well in practice. The KNN algorithm is a learning algorithm that is non-parametric (it does not involve any assumptions about the underlying data distribution) and lazy (it does not use the training dataset to make any generalisations) [89]. The non-parametric characteristic is very useful in real world applications as frequently the practical data do not obey typical theoretical assumptions [89]. The KNN algorithm makes a decision based on the entire training data set. Therefore, it needs more processing time to predict as it must go through whole datasets and more memory is required for storing whole datasets for processing. Each of the training datasets consists of a set of vectors (features) and their corresponding labels (expected prediction). There is also the "K" parameter (numeric value) in KNN which needs to be passed. This parameter decides how many neighbours (neighbours is defined based on a distance metric which is used to identify nearest neighbors to the test datapoint) influence the regression [89].

6.3.2 Sparkling Water (H2O) and Deep Learning

DL is a evolving technique that models high-level patterns in data as complex multi-layered networks. DL can resolve most challenging prediction difficulties as it employs general modelling [17]. Alternative deep neural network techniques apply artificial neural network as the baseline with multiple hidden layers [17]. Traditional neural networks are very challenging to train and usually do not perform better than other machine learning

techniques [17]. DL, on the other hand, provides an optimisation framework to improve performance and accuracy as compared with the other available techniques as shown in Figure 6.8.

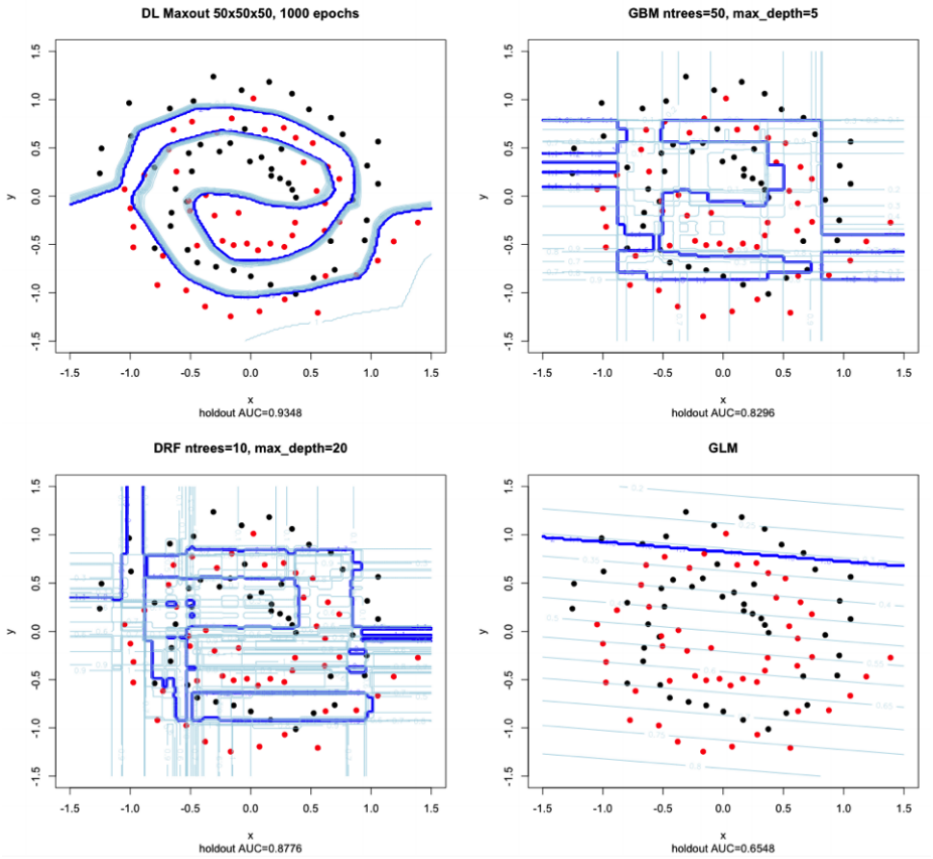


Figure 6.8: Deep Learning versus other Machine Learning techniques [17].

The four diagrams in Figure 6.8 illustrate how different techniques were used to model a complicated scenario. The Generalised Linear Model (GLM) fits a straight line through the data. Tree-based ensemble methods such as Distributed Random Forests (DRF) and Gradient Boosted Machines (GBM), perform better than GLM as these methods fit many straight lines through the data, improving model "fit". DL creates complex curves to the data, delivering the most accurate model [17].

In DL the essential unit in the model is the neuron as shown in Figure 6.9 [17]. In this model, the weighted combination $\alpha = \sum_{i=1}^{n} w_i x_i + b$ of the input is aggregated, and then

an output *f(α)* is transmitted by the connected neuron. Each neuron has a set of inputs x_i, each of which is given a specific weight w_i. The function *f()* represents the non-linear activation function (defined as an output of a neuron given an input or set of inputs) utilised throughout the neuron network, and the bias b depicts the neuron's activation threshold (i.e. error threshold) [17].

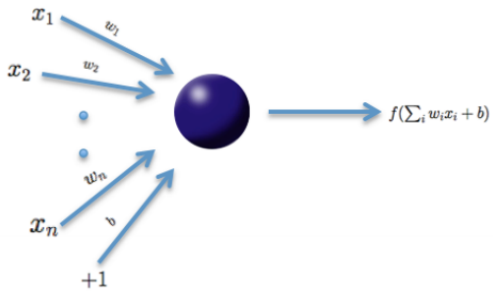


Figure 6.9: Single neuron unit [17].

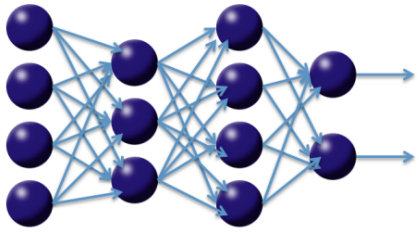


Figure 6.10: Multilayer of interconnected neuron units [17].

Multilayer, feed-forward, neural networks contain many layers of interconnected neurone units as shown in Figure 6.10 [17]. They consist of an input layer, multiple layers of non-linearity, and a linear regression (or classification layer) to match the output. The inputs and outputs of the Multilayer model's units obey the fundamental logic of the single neuron shown in Figure 6.9.

DL can recognise hidden intercommunication between features, study low-level features from basic processed raw data, work with a lot of features and unlabelled data. Therefore, it can be used to build more accurate predictive models. DL detects interactions among features (two or more variables acting in combination) automatically, so it improves the model. Traditional predictive modelling techniques can detect these interactions, but only with manual interference. However, a DL model is very complex to understand as it can have many layers and nodes and utilise a computationally intensive technique. Due to the high volume of data and computing requirements however, machine learning techniques are being developed to support distributed computing frameworks. The computing issue with DL can be handled with such a framework. Sparking Water (H2O) is such a framework that supports scalable machine learning and DL techniques on a distributed computing cluster. H2O supports in-memory computation and is designed to run on a Hadoop cluster (i.e. it supports Spark Cluster), hence it can be deployed on the OLA architecture. H2O's DL

is based on a multilayer feed-forward artificial neural network, the data flows in the only forward direction, is trained with stochastic gradient descent (an iterative optimisation algorithm) using backpropagation, so it repeatedly processes data to enhance the efficiency of predictions [17].

6.4 Design of the Data Access Pattern Intelligence Layer

This section demonstrates a few important components of the intelligence layer for the DAP study.

6.4.1 Deep Learning Model for the Data Access Pattern Study

Figure 6.11 shows the DL model for the DAP. The method adopted was to train the DL model with each dataset individually. Each variable (i.e. day, week, month or year) will be represented by a neuron for the selected dataset for prediction. The following features will be passed into each input neurons; *eventType, userID, typeOfDay, dayGroup, day, month, year and timeInTerms*. Providing as many input neurons as possible will improve the knowledge of each dataset in the model. The hidden layers are used for learning the interconnections between neurons that are not visible at high-level. A prediction will be made based on the learned patterns. This process will be carried out on each individual input dataset. The training time will be very long if the model were to train each dataset individually. With the help of streaming online learning, the training time can be reduced as the model will be updated when new data become available.

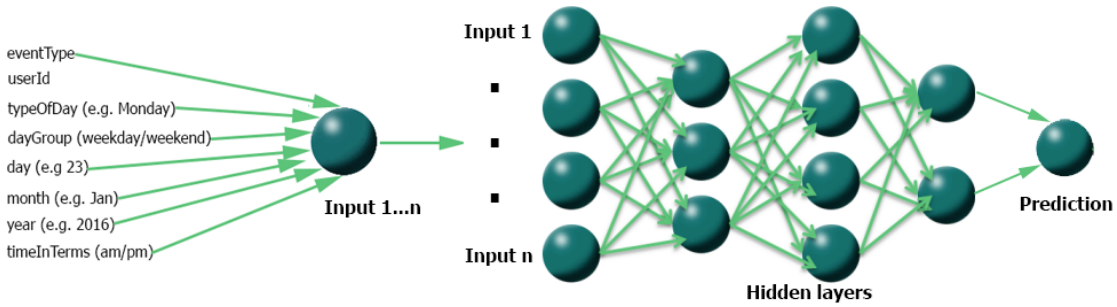


Figure 6.11: Deep Learning model for the DAP study.

6.4.2 KNN Model for the Data Access Pattern Study

Figure 6.12 shows a higher overview of the KNN model for DAP. The selected and pre-processed training data instances are passed into the KNN algorithm, and sample data are extracted and used for prediction and model validation.

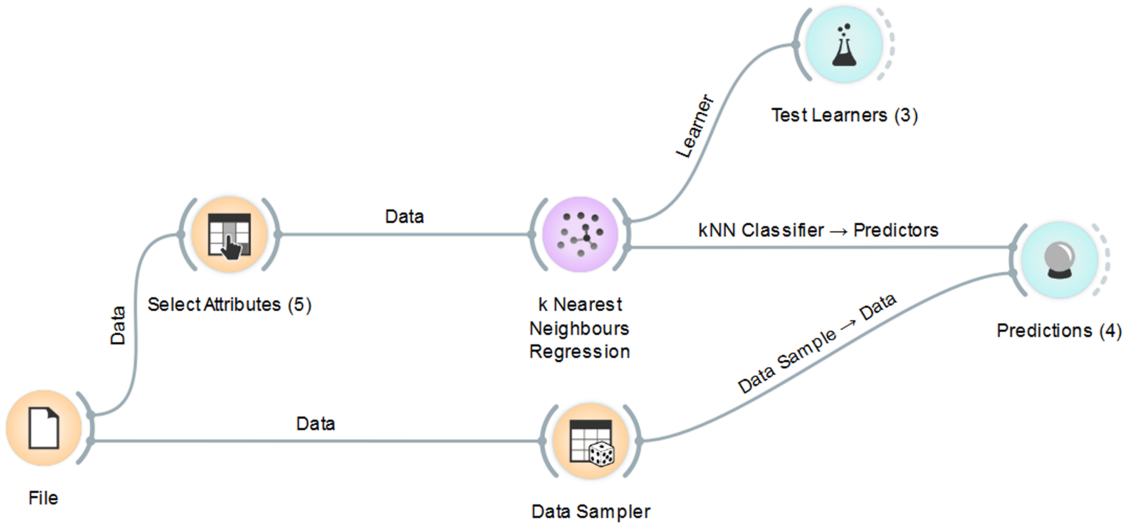


Figure 6.12: Traditional KNN model on Orange Canvas.

Table 6.4 shows the raw data and their corresponding ML features, which was converted into a format supported by the KNN model. The KNN model uses the following features; *user ID (usrIn), Event Type (eventTypeIn)* and *dataset ID (datasetIn)* in training and prediction as shown in Table 6.4. The *count* column shows the number of access for each dataset, which was used as a labeled point in supervised training.

Table 6.4: An example of raw data and prepared data (in vector form) for KNN training and prediction.

Raw data			**labeled point**	**ML features corresponding to the raw data**			**Features vector**
dataset	***eventType***	***usr***	***count***	***datasetIn***	***eventTypeIn***	***usrIn***	***features***
panda.um.user.tik...	put_sm_a	b30926ba0a47ea7fb...	1.0	199.0	0.0	14.0	[199.0 0.0 14.0]
user.dfreebor.117...	put_sm_a	d41d8cd98f00b204e...	1.0	156.0	0.0	0.0	[156.0 0.0 0.0]
hc_test.gangarbt....	put_sm	d41d8cd98f00b204e...	1.0	121.0	2.0	0.0	[121.0 2.0 0.0]
mc12_8TeV.206751....	put_sm	d41d8cd98f00b204e...	1.0	42.0	2.0	0.0	[42.0 2.0 0.0]
mc15_13TeV:mc15_1...	get_sm	f876cbbf7bd93659a...	1.0	10.0	3.0	10.0	[10.0 3.0 10.0]
panda.um.user.mak...	put_sm_a	d41d8cd98f00b204e...	1.0	5.0	0.0	0.0	[5.0 0.0 0.0]
panda.um.user.mak...	put_sm_a	a42a65c5e4dff8422...	1.0	111.0	0.0	1.0	[111.0 0.0 1.0]
data11_7TeV:data1...	get_sm_a	fc987d5e531046f1a...	1.0	248.0	1.0	2.0	[248.0 1.0 2.0]
user.gangarbt.hc2...	put_sm_a	d41d8cd98f00b204e...	1.0	219.0	0.0	0.0	[219.0 0.0 0.0]
data15_13TeV:data...	get_sm_a	76bcbb8d6fed0b7e1...	1.0	200.0	1.0	33.0	[200.0 1.0 33.0]
panda.0228214916....	get_sm_a	a42a65c5e4dff8422...	1.0	8.0	1.0	1.0	[8.0 1.0 1.0]
mc15_13TeV.361824...	put_sm	d41d8cd98f00b204e...	1.0	208.0	2.0	0.0	[208.0 2.0 0.0]
mc12_8TeV:mc12_8T...	get_sm_a	6284a2a2697608593...	55.0	239.0	1.0	16.0	[239.0 1.0 16.0]
mc15_13TeV:mc15_1...	get_sm	84850d67f30bee551...	4.0	187.0	3.0	4.0	[187.0 3.0 4.0]
mc15_13TeV:mc15_1...	get_sm	84850d67f30bee551...	25.0	10.0	3.0	4.0	[10.0 3.0 4.0]
panda.0228154359....	get_sm_a	a42a65c5e4dff8422...	7.0	43.0	1.0	1.0	[43.0 1.0 1.0]
user.dfreebor.117...	put_sm_a	d41d8cd98f00b204e...	1.0	236.0	0.0	0.0	[236.0 0.0 0.0]
mc15_13TeV.361800...	put_sm	d41d8cd98f00b204e...	1.0	21.0	2.0	0.0	[21.0 2.0 0.0]
panda.0227074942....	get_sm_a	6284a2a2697608593...	5.0	215.0	1.0	16.0	[215.0 1.0 16.0]

6.4.3 Batch and Online Models for the Data Access Pattern Study

As the intelligence layer adapts the OLA, it inherits the pure streaming, pure batch, and combination of batch and streaming layers. Therefore, streaming was used for developing a pure online model for access prediction. For example, Figure 6.13 (a) shows that online model uses the previous seven days of data points (D1...Dn) for training and predicting the eighth day. In order to be an active and adaptive online model, the streaming job incrementally moves each new day to the model while removing the oldest day from the model as shown in Figure 6.13 (b) at the end of each day. The principle of this model is that it be a long lived online prediction which will continuously train, predict, and produce reports on streaming events. Although used for daily prediction in this study, this model can be used for hourly prediction on a rapidly changing environment or monthly prediction which is a more slow changing environment.

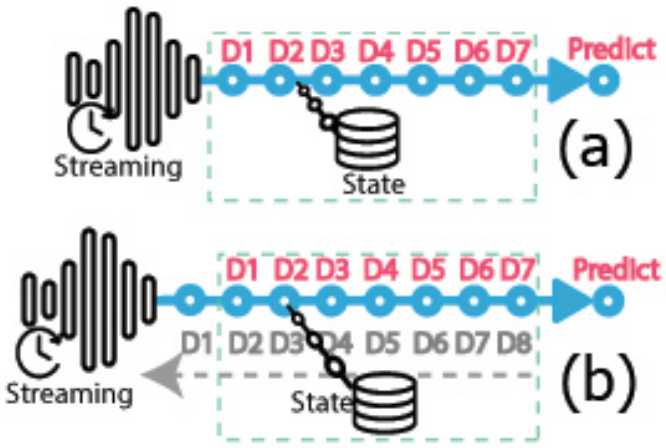


Figure 6.13: Streaming online model. It shows that online model uses the previous seven days of data point for prediction (a), whereas (b) shows the incremental movement of each new day data points to the model.

Figure 6.14 shows the combination of batch and online active adaptive prediction models. The batch training uses historical data to build a model. Then the streaming online prediction job loads the batch model, stores it into the state, updates the model as the new events are streamed, and then makes prediction. Regular batch training is not required as the model is actively shaped as new events arrive eliminating the time required for retraining. This model was built in such a way that it can support year, month, week, day, or hour predictions, and the *D1...Dn* range can be anything based on the application scenario.

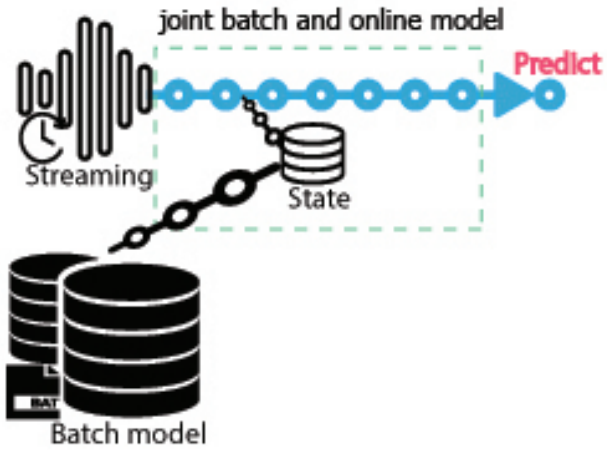


Figure 6.14: Batch and Streaming online model.

6.4.4 Active and Adaptive Learning

An active and adaptive learning algorithm collects new examples streamed in to update the model rather than only using the initial batch processing training model. This model takes raw data as input and preprocesses it to remove unnecessary parts of the data (e.g. IP address) as there are more than 50 elements in each instance of the DAP metadata. Then it converts the selected features into a vector with a labelled point. At a regular interval, it updates the access pattern model using an online process.

6.4.5 Dataset Access Pattern Training Algorithm

Each compute node in the cluster trains a copy of the global model parameters on its local DAP dataset concurrently, and regularly shares the weights to the global model through model averaging across all the nodes in the network (as shown in Algorithm 6.1).

Algorithm 6.1 Training the DAP model

1: Initialise a global model
2: Access the DAP training dataset from distributed nodes
3: Iterate over the DAP dataset in parallel and do:
4: Get a copy of the global model
5: Select sample data from each iteration
6: train and update all weights locally
7: Update the global model by combining and average the local weights across all nodes

6.5 Evaluation of the Adaptive Data Access Pattern Model

In this section an evaluation of the proposed adaptive DAP model is presented.

6.5.1 Experiment setup

The following quality matrices were used to evaluate the prediction model:

1. The Root Mean Squared Error (RMSE) show how close a fitted line is to a set of data points. The calculation is done by taking the distances from the data point to the regression line and squaring them. These distances are referred to as "errors". The squaring is done to prevent negative values cancelling positive values. It then finds the average of the set of errors and calculate the square root of it. The smaller the RMSE, the closer the fit is to the data [90]. In equation 6.3 the $\mathbf{y}_i$ is the actual value for a given input i, the N is the total number of fitted data points, and $\hat{\mathbf{y}}_i$ is the fitted forecast value for the input i.

$$RMSE = \sqrt{\frac{\sum_{i=0}^{N-1}(\mathbf{y}_i - \hat{\mathbf{y}}_i)^2}{N}} \tag{6.3}$$

2. The coefficient of determination (R^2) is applied in order to analyse how variations in one variable can be described by a difference in a second variable. The coefficient of determination is similar to the correlation coefficient (R), which will illustrate how strong the linear relationship between two variables. The R^2 is the square of the correlation coefficient [91]. It can be used estimate how many data points fall within

the fitted regression line by the model. In equation 6.4, the $\mathbf{y}_i$ is the actual value for a given input i and the $\bar{\mathbf{y}}$ is the mean value of the actual values.

$$R^2 = 1 - \frac{\sum_{i=0}^{N-1}(\mathbf{y}_i - \hat{\mathbf{y}}_i)^2}{\sum_{i=0}^{N-1}(\mathbf{y}_i - \bar{\mathbf{y}})^2} \tag{6.4}$$

6.5.2 Data Preparation for Evaluation

On average five million data access event logs are recorded each day (approximately 1.5 GB data size after compression). Due to this volume, using the whole dataset for traning with the traditional data mining tool is impossible, and sampling of the dataset is necessary. However, sampling is not required with the proposed intelligence layer as it inherits the OLA, so it can support processing of large volumes of data by splitting the dataset and distributing it to multiple nodes while executing the training algorithm on that portion of the data concurrently. The formulated dataset has a series of independent variables such as *dataset ID* and *event Type*, etc. It also contains a continuous label (actual access value i.e. access count) which is a dependent variable.

Many experiments were carried out to identify the accuracy and efficiency of the DAP prediction. The Orange Canvas [86], a data mining toolbox, was used to evaluate the DAP on a traditional model. For this test the training dataset had to be sampled, so a dataset for training as well as a separate dataset for testing and validating the technique was required. A random sampling of the data was carried out for the traditional model training and prediction. The same training dataset was used for predictions on both the traditional model and on the distributed intelligence layer.

6.5.3 Performance Evaluation of the DAP on a Traditional Model

Orange Canvas's KNN regression algorithm was employed for the performance and accuracy evaluation of the traditional model. The same dataset used for training the model was used to test the prediction, however, the label used in supervised learning was removed. Figure 6.15 shows the training and testing times of the traditional KNN regression algorithm. It can be observed that the number of training data instances varied from 1000 to 150,000 with the training time ranging from one second to 2750 seconds. The traditional

machine was struggling (freezing) as the number of training and testing instances was increased.

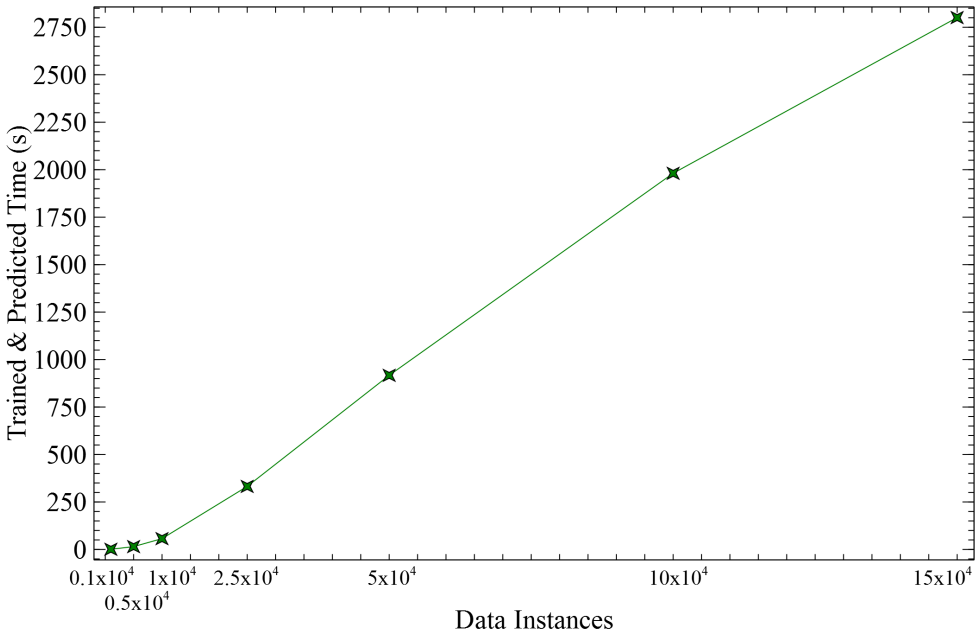


Figure 6.15: Traditional KNN training and prediction time over various data instances.

6.5.4 Evaluation of Accuracy of DAP on a Traditional System

Figure 6.16 shows the prediction error rate (RMSE) ranging from a lower error rate of 0.26 to a higher error rate of 0.37. It was evaluated on the same dataset used for the performance evaluation. It also shows that the error rate dropped as a high coefficient of determination (R^2) was identified.

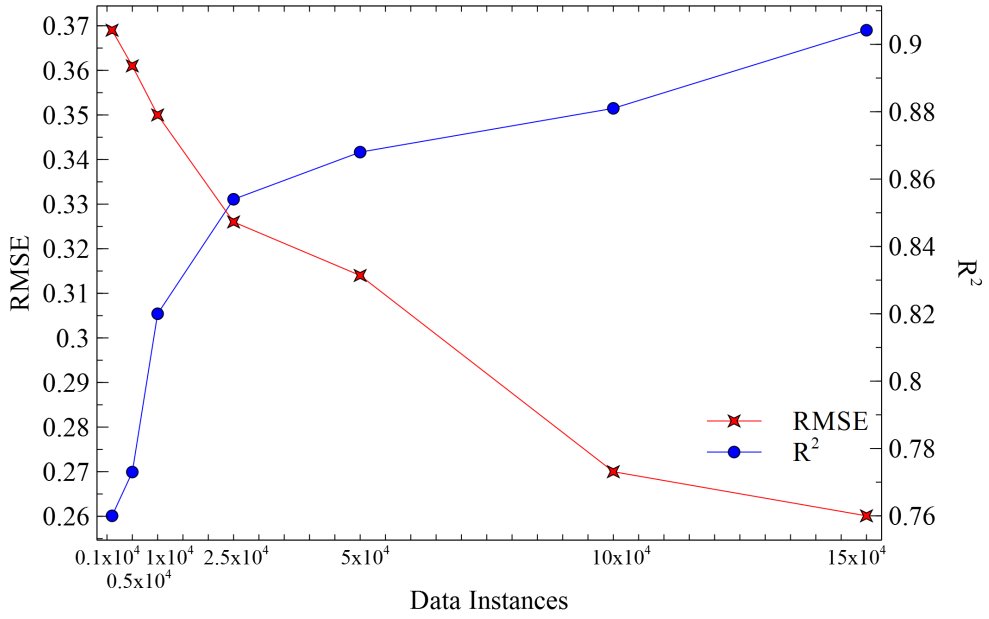


Figure 6.16: Traditional KNN prediction error rate.

6.5.5 Performance Evaluation of the Intelligence Layer for the DAP Study

This section presents a performance evaluation of the pure batch model, and the combination of batch and online model.

Evaluation of the batch training

A KNN algorithm and DL algorithm for the DAP study were developed as described in Section 6.4 to support the distributed nature of the OLA. Figure 6.17 shows the training and prediction time of the distributed batch KNN. The amount of time it took to train and test the DAP on the traditional KNN model using 100,000 data instances on a single node was 1981 seconds, while the distributed batch KNN took ∼541 seconds using four executors, two cores and 4096 MB memory for each executor (i.e. $4096 \times 4 = 16384$ MB). The distributed batch KNN was able to train and predict 3x faster than the traditional KNN. However, the traditional KNN took less time to train and predict compared to distributed batch KNN when the number of data instances was less than 5,000. This is due to the overhead added to the distributed batch KNN in finding, and allocating,

resources.

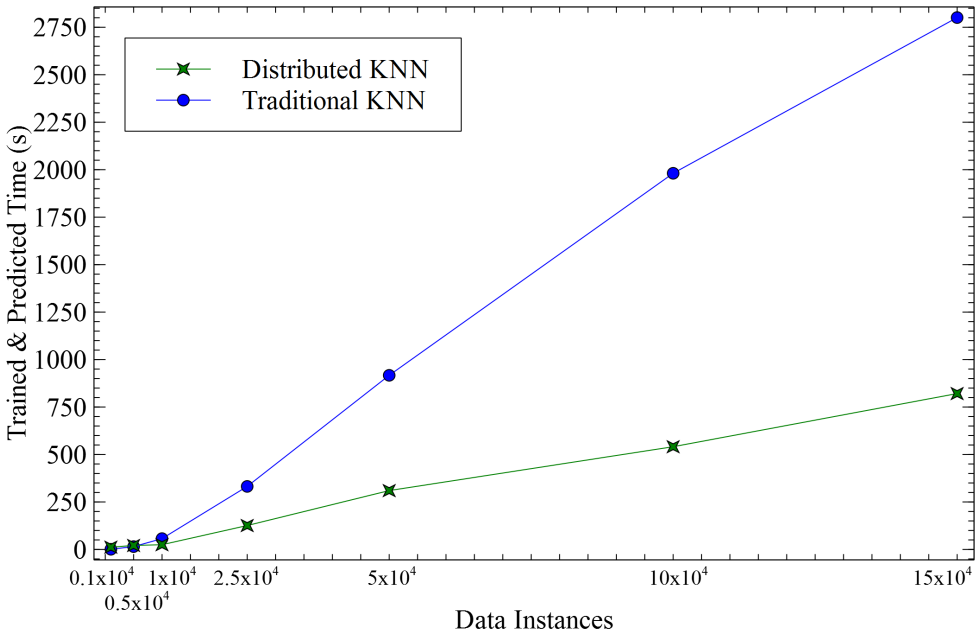


Figure 6.17: Distributed batch KNN and Traditional KNN training and prediction time over various data instances.

In order to evaluate the DL approach, the training data was split by day for each dataset and subsequently fed into the model. The training data ranged from seven days to 48 days. For example, for the seven day evaluation the DL model was trained with seven days, and predicted the 8th day. As expected 48 days of events was large (62 GB), and it took a long time to pre-process the data for input to the DL model. The data were pre-processed before conducting the performance evaluation (so pre-processing time has been discarded in this evaluation). The prepared training data of 48 days was reduced to 1.1 GB (2.8 million instances). Figure 6.18 shows the performance of the training. Eight executors were used to train the DL model and each executor was allocated with 12 GB memory, and with four CPU cores. The DL model has to train each dataset individually, and it must train many instances so it is a resource depleting process. Training the DL model was expensive in terms of execution time although large computing resources were allocated.

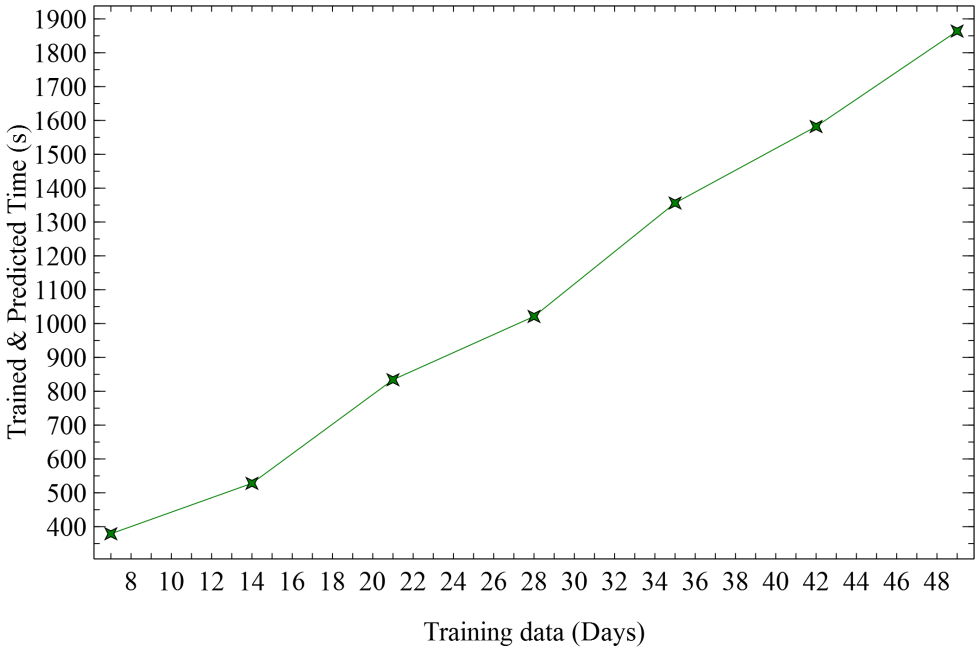


Figure 6.18: Distributed batch DL training and prediction for different sized training datasets.

Assessment of the adaptive model training

The training of the distributed batch KNN and DL model was faster than the traditional model, particularly, in the case of KNN. However, distributed batch training requires human interaction for updating the model. It is a time consuming process as all historical data are needed to be used for retraining. With the adaptation of the OLA, the streaming layer was able to process the incoming events and update the model as it was streaming, eliminating the retraining time.

6.5.6 An Accuracy Evaluation of the Intelligence Layer for the DAP Study

This section presents an evaluation of the accuracy of the pure batch and the combination of batch and online models. Accuracy comparisons were made between the simplest KNN method and the advanced DL approach.

Evaluation of the batch training

Figure 6.19 demonstrates that the accuracy of the distributed batch KNN is comparable to that of the traditional KNN. However, the distributed batch KNN has a slightly higher error rate (lowest: 0.29 and highest: 0.39) and lower variance than that of traditional model. It also shows that the error rate (RMSE) dropped as high coefficient of determination R^2 was identified.

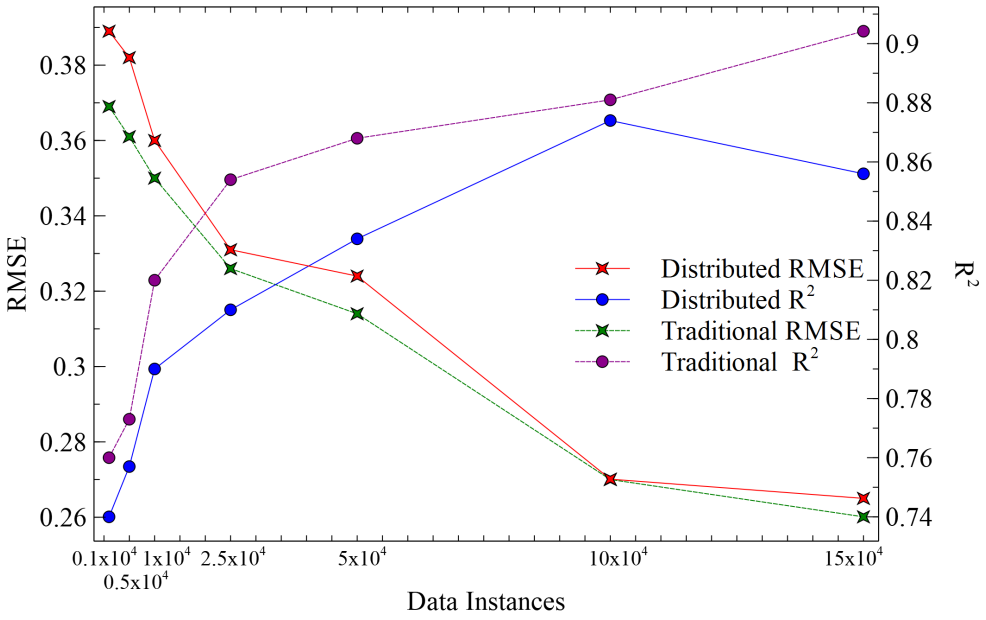


Figure 6.19: Distributed batch KNN and Traditional KNN prediction error rate.

Figure 6.20 shows the prediction accuracy of the DL model. In general, the DL model predicted well with a lower error rate of .095 in comparison to other models though it is expensive in terms of resources and time. The DL model also found a high variance between variables. It is understandable why this model had the lowest error rate, as each dataset was trained and predicted individually.

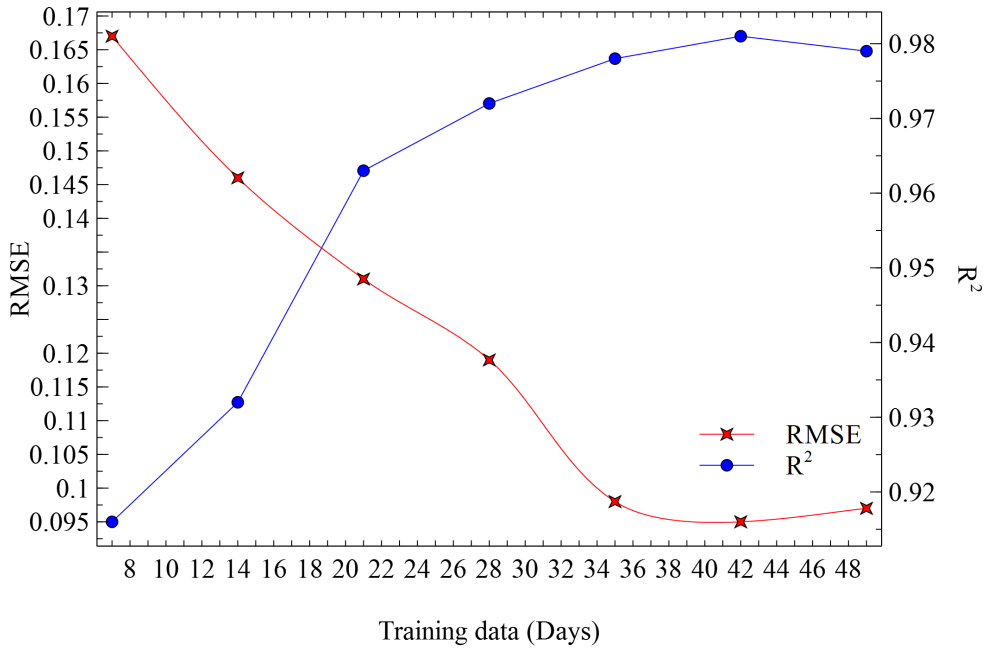


Figure 6.20: Distributed batch DL prediction error rate.

In addition to the KNN and DL techniques, the Linear Regression (LR), Decision Tree (DT), and Random Forest (RF) methods were evaluated in batch processing for the DAP study, using 100,000 training data instances.

LR is a very basic method used for predictive analysis. Regression ratings are applied to describe data, and to describe the relationship between a dependent variable and one or more independent variables. The LR analysis consists of three main stages; analysing the correlation and directionality of the data, computing the model (i.e. fitting the line) and evaluating the validity and usefulness of the model [88]. Figure 6.21 shows the actual number of data accesses against the predicted number of data accesses for a random sample of data using LR. The RMSE for the LR prediction was 0.51 which clearly has a high error rate.

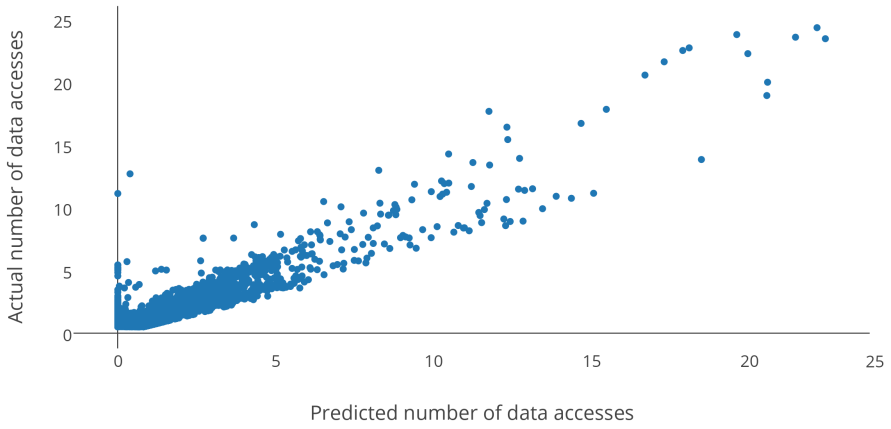


Figure 6.21: Plot showing the actual against the predicted dataset accesses for Linear Regression.

A DT is a structure that uses a branching approach in order to demonstrate every potential outcome of a decision including a root node, branches, and leaf nodes. In a DT, each regional node signifies a test on an attribute, each branch signifies the outcome of a test and each leaf node holds a continuous value (regression) or class label (classification) [88]. Figure 6.22 shows the actual data access against the predicted access for a random sample of data using DT. The RMSE for the DT prediction was 0.32 which has a better error compared to the LR. The plot presents that the predicted access pattern is much better than the LR prediction.

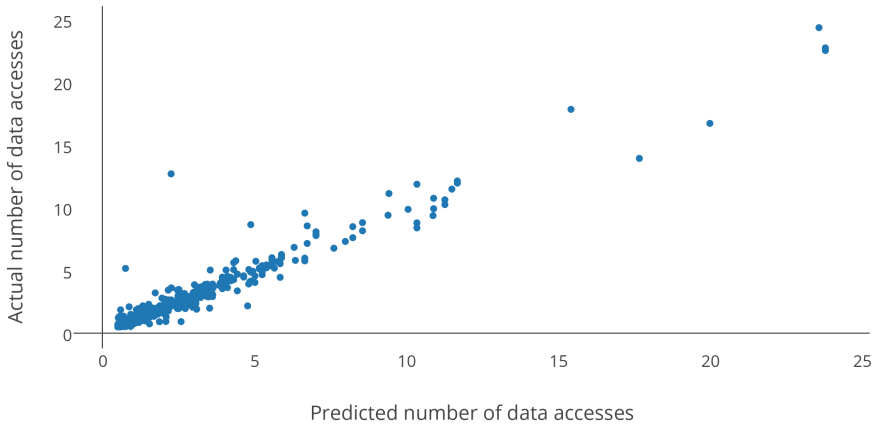


Figure 6.22: Plot showing the actual against the predicted dataset accesses for Decision Tree.

A RF approach consists of a great number of DTs and is known as an ensemble approach. All information is supplied to every DT. The most frequent outcome for each piece of information is accepted as the final output. The RF corrects the DTs overfitting issue by discarding random noises in the data and by picking up the underlying relationship between the data. The principal behind an ensemble approach is that a collection of weak models can put together to form a strong model [88]. Figure 6.23 shows the actual data access against the predicted access for a random sample of data using RF. The RMSE for the RF prediction was 0.26 which is a better error compared to the LR and DT. The plot shows that the predicted access pattern is much better than the LR and DT predictions. The prediction is well aligned with the actual number of data accesses.

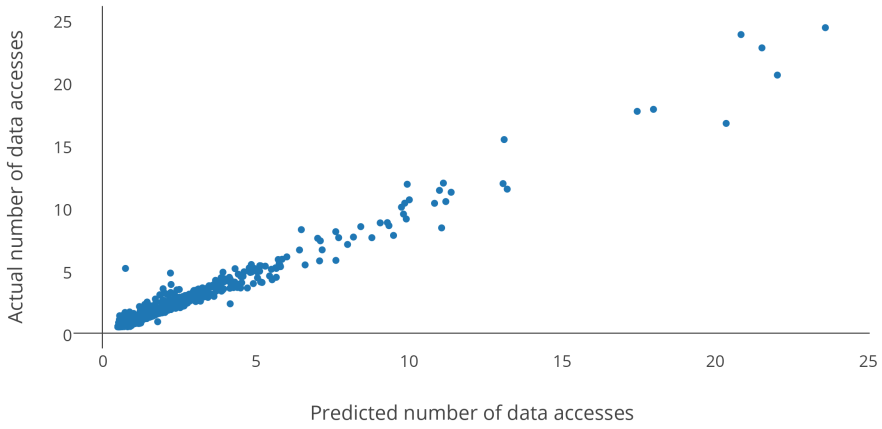


Figure 6.23: Plot showing the actual against the predicted dataset accesses for Random Forest.

Assessment of the adaptive model training

The pure streaming model of both KNN and DL did not give a good prediction as it had to build a model purely based on streaming events. With time, however, it was able to improve the model. A pure online model can be used when there is no historical data available. It is best to use the OLA model for prediction, which is the combination of both batch and streaming layers. As already demonstrated, a distributed batch model provides good predictions, particularly when using the DL technique. Therefore, the role of the streaming layer was to load the batch model and to actively adapt the model as new events were streamed in. The observation of the OLA based model prediction showed it was comparable to the pure batch based model, though minor difference were present due to dynamic training.

6.5.7 Scalability Evaluation of the Intelligence Layer for Studying DAP

The OLA evaluation presented in Chapter 5 has already demonstrated that it scales well. Therefore, deploying distributed batch algorithms will inherit the characteristics of the OLA. Allocating an appropriate number of processing executors, parallel tasks, and memory, reduces the training and prediction time considerably for the proposed approaches.

Figure 6.24 shows the basic throughput of the distributed batch KNN, in terms of the number of data instances trained and predicted per second and the total execution time as the number of processing executors was increased. The evaluation was carried out with 100,000 data instances.

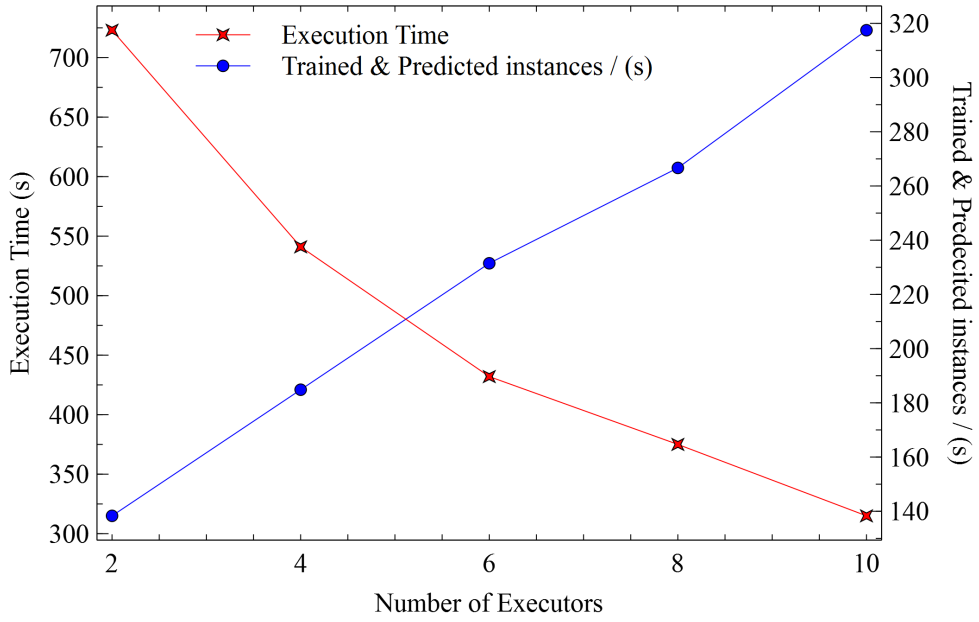


Figure 6.24: Scalability of the distributed batch KNN training and prediction.

Figure 6.25 shows the scalability of the DL training. The plot shows how changing the number of executors impacts the performance. For this test, the amount of memory used for each executor was fixed at 4 GB and with 2 CPU cores. The evaluation was carried out using 14 days of training data. The plot demonstrates that performance improves roughly linearly as the number of executors was increased. However, between 8 nodes and 10 nodes the performance stabilised due to resource allocation and intermediate data distribution between executors.

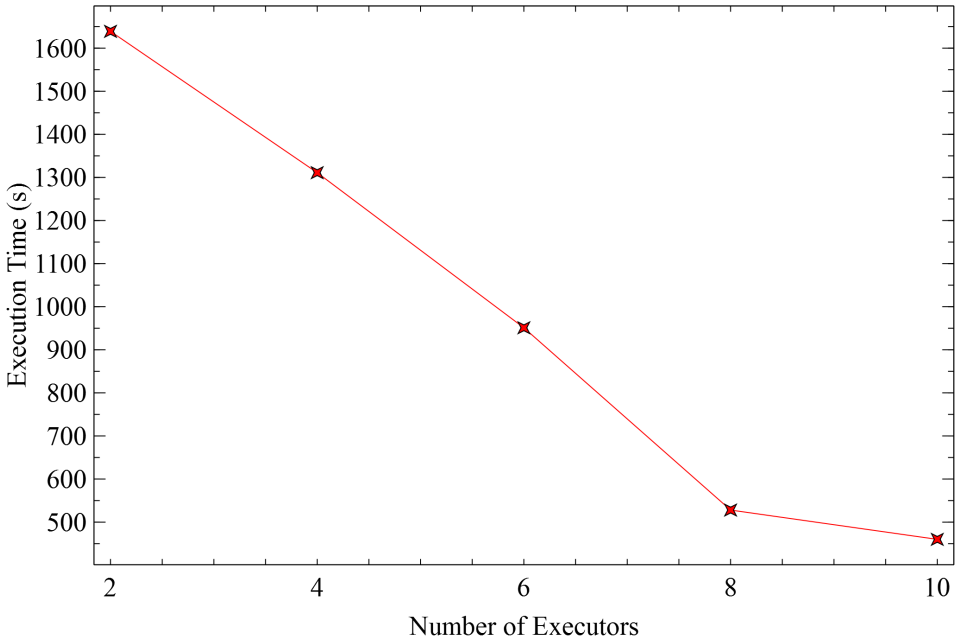


Figure 6.25: Scalability of the distributed batch DL training and prediction.

6.6 Summary

To distribute data more efficiently across a network for analysis prediction of the future data access pattern is needed. The underlying assumption was that historic user behaviour could be used to make predictions about future data access patterns. This chapter presented a data access pattern study using an intelligence layer which employs an active and adaptive model for prediction. The batch model was built using historical training data and was then actively adapted and updated using an online model as new events were streamed in. The built model was deployed on the Optimised Lambda Architecture for evaluating data from the ATLAS distributed data management system. The model was explored with a simple algorithm, K-Nearest Neighbour, and an advanced technique, Deep Learning. The data access pattern model was developed with the Spark and Spark Streaming stack with the intention of inheriting the parallelism, scalability, low-latency computation, and performance properties. The performance and accuracy of the proposed prediction models show that they can be used to provide efficient data distribution decisions. Deep Learning prediction presented a lower error rate in comparison to the other models studied. The ATLAS group's current prediction model error rate is 0.24 (RMSE)

[83], whereas the proposed Deep Learning model demonstrated that it was able to predict with 0.095 error rate. However, the proposed model did not use the same training dataset as that used by the ATLAS group.

Chapter 7

Conclusions and Future Work

In this chapter, a summary of the foremost contributions and limitations of the research carried out for this thesis is presented. Opportunities for future work are highlighted.

7.1 Conclusions

Scientific infrastructures are highly distributed and heterogeneous platforms with various middleware characteristics, job submissions, and execution tools, and have diverse methods of transferring and accessing datasets. The high level of computation activity and the distributed nature of the infrastructure makes the system extremely complex. Taking this into consideration, the probability of failures or inefficiencies is high compared with more traditional systems. Efficient monitoring is necessary in order to present a comprehensive strategy to recognise and resolve any issues within the infrastructure. This is an important determinant in the overall effective utilisation of resources.

In this research an investigation into evolving approaches in architecting a scalable data store and analytics platform for real-time monitoring of data intensive scientific infrastructure was conducted. Various methods were studied and developed into a state-of-the-art monitoring system. While numerous techniques exist, Lambda Architecture drew substantial interest and was adopted on a smaller scale.

In real world scientific settings monitoring systems traditionally use a relational database

for storage and analytics. However, the traditional method is no longer optimal due to foreseen characteristics of the Big Data. The use of Big Data technologies and approaches, such as distributed computing and parallel frameworks for a large scale real-time monitoring system for scientific infrastructure, in these scenarios is considered an innovative research area.

In this thesis, the Lambda Architecture (LA) has been recognised as an empowering approach for a monitoring system that supports scalability and real-time operation. This was deemed an essential baseline for research work performed towards optimising, establishing, presenting, and evaluating, a flexible, collaborative, distributed, scalable, real-time monitoring system. This architecture is designed to be able to scale at the rate mandated by the continuously evolving volume, velocity, and variety of monitoring events. However, the LA was complex to implement, maintain, and synchronise the layers in order to serve the computed statistics. To mitigate this complexity, the presented thesis proposes and evaluates an Optimised Lambda Architecture (OLA); a scalable distributed monitoring system supporting the real-time Spark ecosystem. This was achieved by utilising Spark batch computation and a Spark streaming layer, which enables reusable codes between each layer. Streaming was used for incremental calculation to serve the real-time state of the infrastructure, while batch computation was scheduled to run to correct any inaccuracies caused by the real-time computations. The performance improvements of the OLA, when compared to the traditional counterpart, included a x5 improvement in speed and x2 improvement in speed compared with the LA.

Parts of the infrastructure could not be monitored by the conventional system due to the high volume and velocity of the data. However, the OLA demonstrated that it had no issue with accommodating and processing large datasets at high speed while the data were propagated. While monitoring scientific infrastructure is a computationally intensive process, the parallel processing framework and in-memory computation of the OLA allows for faster computation, with execution times that were not feasible with the traditional system.

The OLA is highly efficient in the case of supporting a data-intensive use case, and it

scales well with increasing dataset size. The OLA has saved operational time as well as computation time in comparison to the traditional architecture used by the WLCG. With the traditional workflow consisting of local filesystems (dirq), local collectors, PL/SQL computations had several hours per week dedicated to clean up the partition of the machines and to maintain services. However, with the OLA resiliency, stability, low-latency and ease of recovering and recomputing it has reduced the operation time to almost zero. An estimated 0.5 days/week is saved with the OLA now implemented by the WLCG. In terms of performance, the OLA does not require recomputation by utilising real-time computation, whereas the traditional architecture recomputes at a regular interval. The OLA batch layer reduced computation time by a factor of 5 in comparison with the traditional PL/SQL system.

In order to further improve the OLA, with the intention of coming up with an intelligence layer, a data access pattern prediction study utilising an active and adaptive model was presented. A data access pattern study was selected solely based on its ability to complement Big Data characteristics. This study should open avenues to automating other use cases such as the detection of failures and anomalous accesses. The intelligence layer for studying the data access pattern provides a heterogeneous machine learning technique on an offline dataset (batch layer) and an online dataset (streaming layer). The intelligence layer has been incorporated into the OLA and it tries to achieve a balance between maximising resources, minimising economical costs, and performance for predicting the dataset access pattern with high accuracy.

The combination of the proposed efficient data acquisition techniques, the OLA, and the intelligence layer designed and evaluated are believed to contribute a unified approach towards a scalable, low-latency, intelligent, high-performance, and commodity-based monitoring architecture for data-intensive scientific infrastructures. The outputs of this research will be applicable to any scientific infrastructure as well as any commercial infrastructure, as it was demonstrated with the Amazon Cloud System.

7.2 Future Work

In this thesis, a single scientific infrastructure was evaluated, though the evaluation was carried out on two independent experiment datasets (CMS and ATLAS). For a further understanding of the proposed architecture, it would be interesting to evaluate another scientific or commercial infrastructure, for example, a smart grid for which Big Data has become a significant concern due to the rapid deployment and use of digital devices such as smart meters (electronic devices that register consumption of electric energy) and phasor measurement units (devices that measure electrical waves).

Further research opportunities also exists in terms of the financial perspective of the architecture, especially with regard to the use of a cloud system where a pay per use metering system could be employed. Evaluation of the architecture to see whether deploying the architecture into a cloud is better in comparison to building the architecture on an in-house cluster could be carried out. This study would require detailed evaluation of current requirements as well as future needs.

The intelligence layer could be enhanced by using an alerting mechanism to notify users when it predicts there will be a sudden demand for a dataset, automatically replicating the dataset in demand.

Bibliography

[1] Amir Gandomi and Murtaza Haider. Beyond the hype: Big data concepts, methods, and analytics. *International Journal of Information Management*, 35(2):137–144, 2015.

[2] Julia Andreeva, Max Boehm, Benjamin Gaidioz, Edward Karavakis, Lukasz Kokoszkiewicz, Elisa Lanciotti, Gerhild Maier, William Ollivier, Ricardo Rocha, Pablo Saiz, et al. Experiment dashboard for monitoring computing activities of the lhc virtual organizations. *Journal of Grid Computing*, 8(2):323–339, 2010.

[3] Michael Hausenblas and Nathan Bijnens. Lambda architecture. *URL: http://lambda-architecture. net/. Luettu*, 6:2015, 2014.

[4] J Forgeat. Data processing architectures lambda and kappa. http://www.ericsson.com/research-blog/data-knowledge/data-processing-architectures-lambda-and-kappa, 2015. [Online; accessed 25-05-2016].

[5] Apache. Drill. http://drill.apache.org, 2015. [Online; accessed 25-05-2014].

[6] Cloudera. Impala. http://www.cloudera.com/content/cloudera/en/products-and-services/cdh/impala.html, 2015. [Online; accessed 25-05-2014].

[7] Facebook. Presto. http://prestodb.io, 2015. [Online; accessed 25-05-2014].

[8] Apache. Apache storm. https://storm.apache.org, 2012. [Online; accessed 12-04-2016].

[9] Leonardo Neumeyer, Bruce Robbins, Anish Nair, and Anand Kesari. S4: Distributed stream computing platform. In *Data Mining Workshops (ICDMW), 2010 IEEE International Conference on*, pages 170–177. IEEE, 2010.

[10] Amazon. Kinesis. 2014. [Online; accessed 25-05-2014].

[11] Apache. Samza. 2014. [Online; accessed 25-05-2014].

[12] Apache. Spark streaming. 2014. [Online; accessed 25-05-2014].

[13] CERN. LHC physics data taking gets underway at new record collision energy of 8TeV. http://press.web.cern.ch, 2012. [Online; accessed 18-12-2015].

[14] WLCG. Worldwide LHC Computing Grid. http://wlcg.web.cern.ch. [Online; accessed 20-12-2015].

[15] WLCG. WLCG Experiment Dashboard. http://dashboard.cern.ch, 2012. [Online; accessed 01-10-2016].

[16] M.V Georgiou and L. Magnoni. Real-time statistic analytics for the WLCG Transfers Dashboard with Esper. Technical report, CERN, Geneva, Sept 2014.

[17] Arno Candel, Viraj Parmar, Erin LeDell, and Anisha Arora. Deep learning with h2o. *H2O. ai Inc.,*, 2016.

[18] T. White. *Hadoop: The Definitive Guide, 3rd ed.* Yahoo press, 2012.

[19] Nathan Marz and James Warren. *Big Data. Principles and best practices of scalable realtime data systems.* Manning Publications, 2015.

[20] Joseph M Hellerstein, Michael Stonebraker, and James Hamilton. *Architecture of a database system.* Now Publishers Inc, 2007.

[21] Jim Brandt, Karen Devine, and Ann Gentile. Infrastructure for in situ system monitoring and application data analysis. In *Proceedings of the First Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization*, pages 36–40. ACM, 2015.

[22] Vladimír Marík, JL Martinez Lastra, and Petr Skobelev. Industrial applications of holonic and multi-agent systems. In *6th International Conference, HoloMAS*. Springer, 2013.

[23] Fernando P. Garcia Martinez, Antonio R and Rafael Marin-Lopez. Architectures and protocols for secure information technology infrastructures, 2014.

[24] J Andreeva, A Beche, S Belov, I Dzhunov, I Kadochnikov, E Karavakis, P Saiz, J Schovancova, and D Tuckett. Processing of the wlcg monitoring data using nosql. In *Journal of Physics: Conference Series*, volume 513, page 032048. IOP Publishing, 2014.

[25] Fangjin Yang, Eric Tschetter, Xavier Léauté, Nelson Ray, Gian Merlino, and Deep Ganguli. Druid: a real-time analytical data store. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 157–168. ACM, 2014.

[26] Subhas Chandra Mukhopadhyay. *New Developments in Sensing Technology for Structural Health Monitoring*. Springer, 2011.

[27] S Uesugi. What is kappa architecture? http://www.ericsson.com/research-blog/data-knowledge/data-processing-architectures-lambda-and-kappa, 2015. [Online; accessed 25-05-2016].

[28] Who's who in technology, 2005.

[29] Byron Ellis. *Real-time analytics: Techniques to analyze and visualize streaming data*. John Wiley & Sons, 2014.

[30] Hesheng Chen. *Large Research Infrastructures Development in China: A Roadmap to 2050*. Springer, 2011.

[31] Karan Vahi, Ian Harvey, Taghrid Samak, Dan Gunter, Kim Evans, David Rogers, Ian Taylor, Monte Goode, Francisco Silva, Eddie Al-Shakarchi, et al. A general approach to real-time workflow monitoring. In *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, pages 108–118. IEEE, 2012.

[32] Pethuru Raj. *Handbook of research on cloud infrastructures for big data analytics*. IGI Global, 2014.

[33] Bent J Mackey G, Sehrish S. Introducing map-reduce to high end computing. petascale data storage. *PDSW*, 2008.

[34] Ronald C Taylor. An overview of the hadoop/mapreduce/hbase framework and its current applications in bioinformatics. *BMC bioinformatics*, 11(12):S1, 2010.

[35] Gang-Hoon Kim, Silvana Trimi, and Ji-Hyong Chung. Big-data applications in the government sector. *Communications of the ACM*, 57(3):78–85, 2014.

[36] Fox G. Ekanayake J, Pallickara S. Mapreduce for data intensive scientific analyses. *IEEE*, 2008.

[37] et al. Attebury G, Baranovski A. Hadoop distributed file system for the grid. *IEEE*, 2009.

[38] Apache. Apache spark. https://spark.apache.org. [Online; accessed 12-04-2016].

[39] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. *HotCloud*, 10:10–10, 2010.

[40] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. Dremel: interactive analysis of web-scale datasets. *Communications of the ACM*, 54(6):114–123, 2011.

[41] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized streams: A fault-tolerant model for scalable stream processing. Technical report, DTIC Document, 2012.

[42] T. M. Mitchell. The Discipline of Machine Learning. Technical Report 9, Carnegie Mellon University, School of Computer Science, Machine Learning Department, 2006.

[43] S. J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach, 3rd edition*, volume 9. Pearson, 2009.

[44] EMC Digital Universe with Research Analysis by IDC. The digital universe of opportunities: Rich data and the increasing value of the internet of things. http://www.emc.com/leadership/digital-universe/2014iview/executive-summary.htm, 2014. [Online; accessed 11-04-2016].

[45] Apache. Apache hadoop. https://hadoop.apache.org, 2008. [Online; accessed 11-04-2016].

[46] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Communications of the ACM*, 51, 1:107–113, 2008.

[47] S. Shenker, I. Stoica, T. Hunter, M. Zaharia, T. Das, and H. Li. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. *Usenix*, 37, 4:45–51, 2012.

[48] M. Zaharia, M. Chowdhury, T. Das, and A. Dave. Fast and interactive analytics over hadoop data with spark. *Usenix*, 37, 4:45–51, 2012.

[49] Z. Ni. Comparative evaluation of spark and stratosphere. *KTH Information and Communication Technology*, 2013.

[50] S. Shenker and I. Stoica and T. Hunter and M. Zaharia and T. Das and H. Li. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. Technical Report UCB/EECS-2012-259, Univ. California, Berkeley, Apr 2012.

[51] Apache. Apache flink. https://flink.apache.org, 2014. [Online; accessed 12-04-2016].

[52] Apache. Apache h2o. http://www.h2o.ai, 2014. [Online; accessed 12-04-2016].

[53] Anisha Arora, Arno Candel, Jessica Lanford, Erin LeDell, and Viraj Parmar. Deep learning with h2o, 2015.

[54] Apache. Apache spark machine learning library. http://spark.apache.org/mllib, 2014. [Online; accessed 12-05-2016].

[55] skymind. Deeplearning4j. http://deeplearning4j.org, 2014. [Online; accessed 14-05-2016].

[56] Gianmarco De Francisci Morales and Albert Bifet. Samoa: Scalable advanced massive online analysis. *The Journal of Machine Learning Research*, 16(1):149–153, 2015.

[57] Cloudera. Cloudera oryx. https://github.com/cloudera/oryx, 2014. [Online; accessed 18-05-2016].

[58] John Langford, L Li, and A Strehl. Vowpal wabbit. *URL https://github.com/JohnLangford/vowpalwabbit/wiki*, 2011.

[59] C. Snijders, U. Matzat, and U.-D Reips. Big Data: Big Gaps of Knowledge in the Field of Internet Science. *International Journal of Internet Science*, 7, 1:1–5, 2012.

[60] A. Aamnitchi, S. Doraimani, and G. Garzoglio. Filecules in highenergy physics: Characteristics and impact on resource management. *High Performance Distributed Computing*, pages 69–80, 2016.

[61] D. Minoli. *A Networking Approach to Grid Computing*. John Wiley & Sons, Inc., 2015.

[62] C. Nicholson and et al. Dynamic data replication in LCG 2008. *Concurrency and Computation: Practice and Experience*, 20, 11:1259–1271, 2008.

[63] L. Magnoni, U. Suthakar, C. Cordeiro, M. Georgiou, J. Andreeva, A. Khan, and D.R Smith. Monitoring WLCG with lambda-architecture: a new scalable data store and analytics platform for monitoring at petabyte scale. *Journal of Physics: Conference Series*, 664(5):052023, 2015.

[64] J. Knobloch and L. Robertson. LHC Computing Grid:Technical Design Report-LCG-TDR-001, 2005.

[65] K. Grim. Tier-3 computing centers expand options for physicists, International Science Grid This Week (iSGTW), 2009.

[66] I. Foster and C. Kesselman. *The Grid 2: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2013.

[67] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the Grid: Enabling scalable virtual organizations. *The International Journal of High Performance Computing Applications*, 15, 3:200–222, 2011.

[68] E. Laure, S.M Fisher, A. Frohner, C. Grandi, P.Z Kunszt, A. Krenek, O. Mulmo, F. Pacini, F. Prelz, J. White, M. Barroso, P. Buncic, F. Hemmer, A. Di Meglio, and A. Edlund. Programming the Grid with gLite. *Computational Methods in Science and Technology*, 12, 1:33–45, 2006.

[69] I. Foster. What is the grid? A three point checklist, GRIDToday, 2011.

[70] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The Hadoop Distributed File System. *IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–10, 2010.

[71] S. Ghemawat, H. Gobioff, and S.T Leung. The google file system. *ACM SIGOPS Operating Systems Review*, 37, 5:29–43, 2003.

[72] Apache Flume project. https://flume.apache.org, 2012. [Online; accessed 02-01-2016].

[73] K. Skaburska. The Dirq project. http://dirq.readthedocs.org, 2013. [Online; accessed 27-12-2015].

[74] R. Gardner, S. Campana, G. Duckeck, J. Elmsheuser, A. Hanushevsky, F.G Hnig, J. Iven, F. Legger, I. Vukotic, W. Yang, and the Atlas Collaboration. Data federation strategies for ATLAS using XRootD. *Journal of Physics: Conference Series*, 513(4):042049, 2014.

[75] Jeffrey Dean and Sanjay Ghemawat. In *MapReduce: simplified data processing on large clusters*. OSDI04: Proceedings of the 6th conference on symposium on operating systems design and implementation - USENIX Association, 2004.

[76] Thomas Bernhardt. The Esper project, http://www.espertech.com/esper.

[77] Julia Andreeva, Benjamin Gaidioz, Juha Herrala, Gerhild Maier, Ricardo Rocha, Pablo Saiz, and Irina Sidorova. Dashboard for the LHC experiments. In *International Symposium on Grid Computing, ISGC 2007*, pages 131–139, 2009.

[78] Holden Karau, Andy Konwinski, Patrick Wendell, and Matei Zaharia. *Learning spark: lightning-fast big data analysis*. " O'Reilly Media, Inc.", 2015.

[79] Zaharia, Matei et all. In *Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing*, NSDI'12, pages 2–2, Berkeley, CA, USA, 2012. Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation - USENIX Association.

[80] Amazon. https://aws.amazon.com, 2012. [Online; accessed 12-04-2016].

[81] Databricks. https://databricks.com. [Online; accessed 12-04-2016].

[82] WLCG. WLCG Data acTivities dashboard. http://dashb-wdt-xrootd.cern.ch/ui, 2014. [Online; accessed 27-04-2016].

[83] T Beermann, P Maettig, G Stewart, M Lassnig, V Garonne, M Barisits, R Vigne, C Serfon, L Goossens, A Nairz, et al. Popularity prediction tool for atlas distributed data management. In *Journal of Physics: Conference Series*, volume 513, page 042004. IOP Publishing, 2014.

[84] Vincent Garonne, R Vigne, G Stewart, M Barisits, M Lassnig, C Serfon, L Goossens, A Nairz, Atlas Collaboration, et al. Rucio–the next generation of large scale distributed system for atlas data management. In *Journal of Physics: Conference Series*, volume 513, page 042021. IOP Publishing, 2014.

[85] Tadashi Maeno. Panda: distributed production and distributed analysis system for atlas. In *Journal of Physics: Conference Series*, volume 119, page 062036. IOP Publishing, 2008.

[86] Janez Demšar, Blaž Zupan, Gregor Leban, and Tomaz Curk. Orange: From experimental machine learning to interactive data mining. In *European Conference on Principles of Data Mining and Knowledge Discovery*, pages 537–539. Springer, 2004.

[87] Brian D Ripley. The r project in statistical computing. *MSOR Connections. The newsletter of the LTSN Maths, Stats & OR Network*, 1(1):23–25, 2001.

[88] Rich Caruana and Alexandru Niculescu-Mizil. An empirical comparison of supervised learning algorithms. In *Proceedings of the 23rd international conference on Machine learning*, pages 161–168. ACM, 2006.

[89] Florent Burba, Frédéric Ferraty, and Philippe Vieu. k-nearest neighbour method in functional nonparametric regression. *Journal of Nonparametric Statistics*, 21(4):453–469, 2009.

[90] Cort J Willmott. Some comments on the evaluation of model performance. *Bulletin of the American Meteorological Society*, 63(11):1309–1313, 1982.

[91] A Colin Cameron and Frank AG Windmeijer. An r-squared measure of goodness of fit for some common nonlinear regression models. *Journal of Econometrics*, 77(2):329–342, 1997.